Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Análisis, diseño e implementación de un sistema de equilibrado en un entorno PLM

Autor: Francisco Javier Sánchez Sabido

Tutor: Jesús Racero Moreno

Dep. Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Análisis, diseño e implementación de un sistema de equilibrado en un entorno PLM

Autor:
Francisco Javier Sánchez Sabido

Tutor:
Jesús Racero Moreno
Profesor titular

Dep. Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Trabajo Fin de Grado: Análisis, diseño e implementación de un sistema de equilibrado en un entorno PLM

Autor: Francisco Javier Sánchez Sabido

Tutor: Jesús Racero Moreno

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

A mis maestros

A Monesvol

Agradecimientos

Gracias a todos los que me habéis apoyado en esta última etapa de mi vida, especialmente a mi familia. Gracias mamá, te quiero con mi alma. Jesús, has sido un campeón y me has apoyado en todo. Te estoy eternamente agradecido. Y a mi Raquelilla, que es la más mejor en todo. Muchas gracias desde lo más profundo de mi ser.

Especial agradecimiento también a mis amigos, en especial a Rafa, que ha estado presente en los buenos y en los malos momentos.

Una mención especial a esos profesores que me han hecho más ameno el paso por la escuela, a auqellos que me las han hecho pasar canutas y no tanto a aquellos que no sienten pasión por su trabajo, pero que incluso de eso se aprende. Y aprendiendo se vive, y viviendo se aprende.

A todos. Gracias.

Francisco Javier Sánchez Sabido

Sevilla, 2017

Resumen

El presente trabajo tiene por objetivo la implementación de un sistema de equilibrado en un entorno PLM (Product Lifecycle Management). El sistema de equilibrado que implementaremos será una heurística de una sola pasada para solucionar el problema de equilibrado más simple, conocido como SALBP-1 (Simple Assembly Balancing Problem), basado en la regla LCR (Largest Candidate Rule). El entorno PLM con el que trabajaremos será el software OpenSource Aras Innovator. Sobre él trabajaremos, y realizaremos las modificaciones necesarias para poder introducir la estructura PPR (Parte Proceso Recurso) de fabricación de un helicóptero de Lego.También hemos desarrollado un módulo de interoperabilidad que nos permite introducir los datos desde un fichero de texto con un formato estandarizado, tipo Express-I. Una vez tengamos los datos introducidos en Aras ejecutaremos el algoritmo, implementado en lenguaje de programación orientado a objetos C#. Por último evaluaremos si los resultados obtenidos cumplen con los objetivos que nos hemos propuesto cumplir.

Abstract

This document pretends to solve a simple assembly line balancing problem from a PLM (ProductLifecycle Management) software. This problem will be solved by using LCR (Largest Candidate Rule) heuristic.

The PLM software with wich we are going to work is named Aras Innovator. Its an Open Source PLM system, and it allows us to modify its structure to implement a PPR (Product, Procsess, Resource) manufacturing structure of a LEGO helicopter.

Theremore we will develop a functionality which will allow us to introduce data in Aras from an external standardized file, in format Express-I.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

Nunca he visto una desorganización tan bien organizada.

- Henry Ford -

Con el objetivo de mejorar la competitividad dentro de un mercado cada vez más exigente y globaliazdo, la optimización de procesos de producción en la industria es un aspecto clave para reducir costes, mejorar la eficiencia de los procesos productivos y optimizar el consumo de recursos. En esta dirección las empresas deben preocuparse por conocer, estudiar y mejorar el comportamiento de las variables implicadas en sus procesos productivos, específicamente en el caso de las líneas de montaje ya que éstas presentan diferentes problemas: minimizar estaciones, minimizar el tiempo ocioso, maximizar el flujo de producción o minimizar tiempos de ciclo. Todo ello implica una disminución en los costes de producción y en consecuencia un aumento de la competitividad. (Restrepo, 2009).

Las líneas de montaje están compuestas por un conjunto finito de estaciones de trabajo y de tareas, que tienen asignado un tiempo de proceso, y un conjunto de relaciones de precedencias, las cuales especifican el orden de proceso permitido de las tareas. El problema de equilibrado de líneas consiste en asignar las tareas a realizar a las estaciones donde se vayan a ejecutar, de forma que se satisfagan las relaciones de precedencia y se optimize una función objetivo, como minimizar el número de estaciones, minimizar el tiempo de ciclo o maximizar la eficiencia. (Capacho Betancourt, 2004). La solución a un problema de equilibrado consiste en proporcionar una solución factible en la que cada operación esté asignada a una única estación, se respeten las relaciones de precedencia y los tiempos en las estaciones no excedan al tiempo de ciclo.

En el presente trabajo fin de grado resolveremos el equilibrado de una línea de montaje aplicado al caso práctico del ensamblaje de un helicóptero de Lego®. Para tal efecto haremos uso de la herramienta PLM Aras Innovator, gestor open source del ciclo de vida del producto, donde almacenaremos toda la información relevante a la estructura BOM (Bill of materials) y a la que accederemos mediante código para implementar un método de equilibrado. La clasificación de la problemática de equilibrado la veremos más adelante, así como algunos de los métodos de resolución existentes. Describiremos en qué consiste una herramienta PLM y en particular Aras Innovator, la cual customizaremos para adaptarla a nuestro problema de equilibrado. Desarrollaremos el método de equilibrado que vamos a implementar y mostraremos la implementación que hemos realizado mediante programación orientada a objetos en lenguaje C#. La parte del trabajo desarrollada en código la visualizaremos empleando el programa Enterprise Architect, donde elaboraremos los casos de uso de nuestro sistema, así como los diagramas de clase asociados al código desarrollado. Tendremos un diagrama asociado al diseño implementado en Aras, donde se verá reflejada la estructura interna y las relaciones entre los tipos de ítem (itemType), un diagrama dedicado a la implementación de la interoperabilidad, mediante el cual podremos introducir datos a Aras desde un fichero de texto y por último un diagrama asociado al código necesario para implementar el algoritmo de equilibrado de línea.

2 OBJETIVOS

2.1 Objetivo Global.

El objetivo global del presente trabajo fin de grado es implementar un algoritmo de equilibrado que resuelva el problema SALBP, (Simple Assembly Line Balancing Problem) de tipo 1 en un entorno PLM (Product Lifecycle Management), realizando previamente un análisis de los algoritmos existentes para la resolución de los distintos tipos de problemas de equilibrado.

Desarrollaremos un método que resuelva el problema implementándolo en Aras Innovator, de forma que una vez resuelto cree automáticamente las estaciones necesarias y se las asgne a las operaciones.

2.2 Objetivos parciales.

Asimismo contaremos entre los objetivos la customización de la herramienta para adaptarla a las necesidades de nuestro problema de equilibrado y el desarrollo de código en lenguaje C# que sirva para introducir la estructura BOM (Bill of Materials, o desglose de componentes) de nuestro helicóptero, las operaciones a realizar, los recursos empleados, las partes afectadas por cada operación, las operaciones que preceden a una operación las estaciones asociadas a las operaciones de nuestro modelo en Aras Innovator a partir de un fichero de texto, así como para extraer la información ya almacenada en Aras para su procesamiento en el código que implementa el algoritmo. También el código necesario para implementar en Aras una estructura BOM.

3 LA LÍNEA DE MONTAJE

3.1 Introducción

Una línea de montaje está compuesta por estaciones de trabajo y operaciones, las cuales tienen asignado un tiempo de proceso y guardan relaciones de precedencia entre sí que describen el orden en que se realiza el proceso que conduce a la obtención de un producto final.

El ensamblaje de productos se ha venido realizando a lo largo de toda la historia, sin embargo el objetivo de las líneas de ensamblaje modernas es producir productos de gran calidad a bajo coste.

Las primeras referencias a las líneas de ensamblaje se encuentran en en siglo XIII en el arsenal de Venecia, donde los barcos eran producidos utilizando partes pre-manufacturadas dispuestas en almacenes alrededor del canal, de forma que eran los barcos quienes se desplazaban hasta los almacenes, que se podrían considerar estaciones de trabajo. La flota veneciana era la más poderosa del Mediterráneo durante la Edad Media, y esto en parte fue debido al empleo de líneas de montaje en la producción de barcos. El Arsenal de Venecia llegaba a producir un barco al día empleando métodos que no se volverían a ver hasta la Revolución Industrial. Su empleo de la línea de montaje aumentó la producción de barcos acelerando el proceso y disminuyendo el consumo de madera respecto al sistema anterior romano, que consistía en la producción gradual de navíos desde la quilla. División y especialización del trabajo, concentración de todas las actividades en un único lugar, estandarización y control de calidad de los componentes y protección por parte del gobierno al acceso a abundantes recursos son las claves que explican el poder del Arsenal de Venecia. (Venecia)

No es hasta el desarrollo del taylorismo con la publicación de la obra de Taylor *Principles of Scientific Management* en el año 1911 y sus planteamientos de organización racional del trabajo que no se asientan las bases para el desarrollo de las líneas de montaje como las conocemos hoy en día. Se le atribuye el mérito del desarrollo de las líneas de montaje a Henry Ford, aunque previo a éste Ransom Eli Olds ya desarrolló y patentó la primera línea de montaje, donde ya estaban presentes características tales como estaciones de trabajo, piezas estandarizadas intercambiables y operarios que debían realizar un trabajo específico, sencillo y repetitivo. De esta forma fabricó su Oldsmobile Curved Dash, el que fuese el primer automóvil de gasolina producido en masa de la historia, del que entre 1901 y 1907 se produjeron más de 19.000 unidades. (López, 2009).

Sin embargo, el fordismo significó un paradigma en la producción industrial de la época, llevando la cadena de montaje hasta su máxima expresión. Entre 1908 y 1927 se produjeron 15 millones de unidades del modelo Ford T. En el año 1921 el modelo Ford T alcanzó el 57% de la producción mundial de automóviles. El precio inicial de éste modelo era de 825$ en 1908, reduciéndose hasta los 260$ en 1925. Esta considerable reducción en los costes fue posible gracias al progresivo perfeccionamiento de la cadena de montaje, que supuso un aumento en la productividad pasando de fabricar una unidad de 12.5 horas a 93 minutos. Como detalle adicional cabe añadir que para reducir costes laborales asociados a la alta rotación de personal asociada a su empresa, donde llegaron a contratar 52.000 personas para cubrir las necesidades de 14.000, Ford aumentó el salario a 5$ por una jornada laboral de 8 horas cuando lo normal en aquella época era cobrar en torno a 2.25$ por una jornada de 9 horas. Esto supuso que los empleados de Ford se sintieran más motivados, reduciendo la rotación de personal y con ello los costes laborales. (Worstall, 2012).

El inicio del interés por el estudio de las cadenas de montaje desde el punto de vista académico se produjo a mediados del siglo XX, cuando Helgeson plantea un análisis del problema en 1954, y Salveson propone porprimera vez una formulación matemática del mismo. (Camps, 2010).

3.2 La línea de montaje en su contexto.

Existen diferentes formas de organizar los procesos productivos atendiendo a las diferentes características que se pueden presentar, con el objetivo de optimizar el aprovechamiento del espacio, de los equipos y de los empleados, reducir el flujo de información y de movimientos de material y empleados, proporcionar unas buenas condiciones de trabajo, además de aumentar la flexibilidad y mejorar la relación con el cliente.

El objetivo de la estrategia de distribución de las instalaciones es desarrollar una distribución efectiva y eficiente que cumpla con los requerimientos competitivos de la empresa, ayudando así a la empresa a alcanzar sus objetivos competitivos de diferenciación, costo o respuesta. (Heizer J, 2009)

De esta forma nos encontramos con los siguientes tipos de organización de procesos:

- **Organización de posición fija:** Se adopta cuando se pretende ejecutar un proyecto voluminoso, como por ejemplo barcos, edificios, molinos de viento o puentes. El proyecto se lleva a cabo en un lugar determinado al que hay que transportar los materiales y la mano de obra. El proceso se desarrolla en torno al producto final.
- **Organización orientada al proceso:** En este tipo de organización el producto se mueve entre las diferentes áreas de trabajo, lugares diferenciados entre sí por el tipo de procesos que se realizan en ellos. Admite una gran variedad de productos, aunque un bajo volumen de fabricación. Indicada para la producción de piezas en pequeños grupos o lotes. Una avería en un puesto determinado no paraliza toda la producción, como en las organizaciones orientadas al producto. Ejemplos de este tipo de organización serían los hospitales, los restaurantes o los talleres de fabricación mecánica.
- **Organización de oficinas:** El objetivo de este tipo de organización es mejorar los flujos de información internos, reducir los desplazamientos de los trabajadores y aportarles seguridad y comodidad, agrupando trabajadores, espacios y equipos.
- **Organización de comercios:** La distribución de los productos se asigna en función de sus respectivas ventas. La idea principal de este tipo de distribución es que los beneficios varían con la exposición del cliente a los productos. Se emplean diferentes técnicas, como por ejemplo ubicar los artículos con mayores ventas en la periferia de la tienda o usar las mejores ubicaciones para artículos de alto margen.
- **Organización de almacenes:** Aquí lo que se desea es minimizar los costes por manejo de materiales (transporte de entrada, almacenamiento y transporte de salida) maximizando la utilización del espacio disponible.
- **Organización orientada al producto:** En este tipo de organización el producto tiene una demanda estable, y debe ser lo suficientemente elevada como para justificar la inversión en equipo especializado. Esta configuración no admite una gran variedad de referencias, aunque tiene asociado un gran volumen de producción. Se obtienen unos tiempos de fabricación reducidos, las tareas a realizar resultan simplificadas, lo que permite el empleo de mano de obra no cualificada.

Organización de posición fija	Orientada al proceso	Organización de oficinas	Organización de comercios	Organización de almacenes	Orientada al producto
- Barcos - Autopistas - Pozos petróleo - Aeropuertos	- Hospital - Restaurantes	- Seguros - Informática	- Supermercados - Tiendas	- Libros - Paquetería - Distribuidora	- Televisores - Vehículos - Vidrio - Cerveza
Trasladar materiales a las zonas limitadas de almacenaje	Gestionar el flujo variado de materiales para cada producto	Ubicar los trabajadores para reducir desplazamientos entre sí	Presentar al cliente artículos de alto margen	Combinar almacenaje de bajo coste con manejo de materiales de bajo coste	Igualar el tiempo de las tareas en cada estación de trabajo

Ilustración 3-1: Tipos de organizaciones existentes.

Vistas las diferentes formas de organizar los procesos productivos en relación a las características demandadas por cada uno, las líneas de ensamblaje se emplean en organizaciones orientadas al producto.

3.3 Clasificación de las líneas de ensamblaje.

Según Pascual García (Pascual García, 2015) existen diversos tipos de líneas de ensamblaje y pueden clasificar atendiendo a diversos factores, como son el tipo de producto que procesa la línea, si los tiempos de operación son estocásticos o deterministas, si el tipo de operador es manual o automatizado, el tipo de estación de que está compuesta la línea, así como su distribución, si la línea es síncrona o asíncrona y por último sila forma de entrada de material a la línea es fija o variable. A continuación se describirán los distintos tipos de líneas de ensamblaje atendiendo a los diferentes criterios de clasificación considerados.

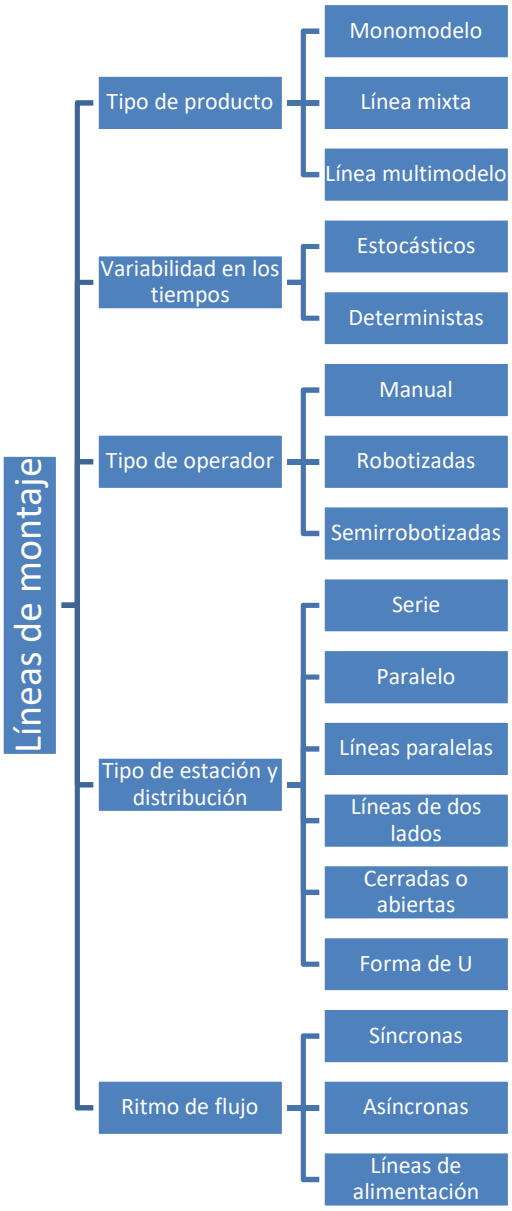


Ilustración 3-2: Clasificación de las líneas de montaje.

3.3.1 Tipo de producto.

Atendiendo al tipo de producto que producen, las líneas se pueden clasificar en tres grandes grupos: líneas monomodelo, líneas mixtas y líneas multimodelo.

a) Líneas monomodelo: Si solo se ensambla un producto, estamos ante una línea monomodelo. Ésta es la configuración más simple de todas. Debido a que la línea monta un único tipo de producto, los operarios desarrollan un alto grado de especialiación.

b) Líneas mixtas: Si varios productos o modelos son procesados en la misma línea estamos ante una línea de ensamblaje mixta, y se presenta además un problema de secuenciación de operaciones. Ésta secuencia es importante con respecto a la eficiencia de una línea, porque los task time pueden diferir considerablemente entre productos. En este tipo de modelos las unidades producidas siguen una secuencia aleatoria y no es necesaria la inclusión de tiempos de setup.

c) Líneas multimodelo: en una línea de montaje multimodelo se producen lotes de productos, cada uno de los cuales contienen unidades de un solo modelo o un grupo de modelos similares, requiriendo de operaciones intermedias de puesta a punto.

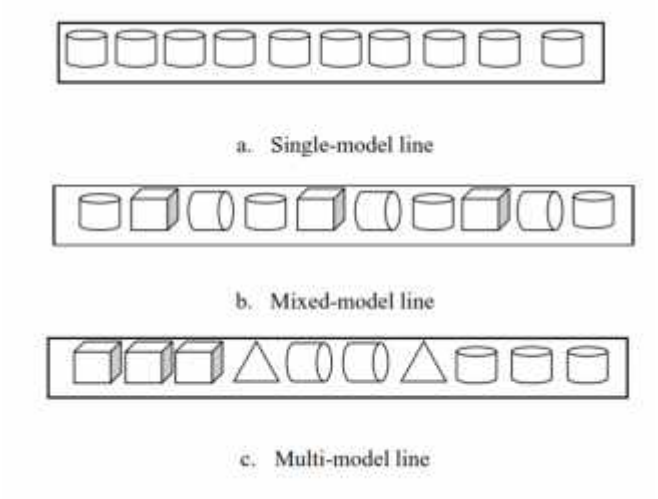


Ilustración 3-3: Tipos de líneas de ensamblaje.

3.3.2 Variabilidad en los tiempos.

Encontramos diferentes procesos según la variabilidad de sus tiempos, pudiendo ser éstos de tipo determinista, donde se conoce con exactitud la duración de las tareas. Éste tipo de tiempos se emplea en el método de ruta crítica (CPM), a diferencia de los tiempos empleados por la técnica de evaluavión y revisión de programas (PERT), donde los tiempos de operación son estocásticos. PERT emplea tres estimadores de tiempo para cada actividad, como son el tiempo optimista, el tiempo pesimista y el tiempo más probable, a partir de los cuales se obtiene la forma de la distribución de tipo beta que éste método supone que siguen los tiempos de operación.

3.3.3 Tipo de operador.

Nos encontramos aquí con tres posibilidades atendiendo al nivel de automatización de la línea, existiendo así líneas que son completamente manuales, donde los operarios son humanos, líneas semirrobotizadas, donde intervienen tanto operarios humanos como robots, y líneas robotizadas, donde los procesos están completamente automatizados.

3.3.4 Tipo de estación y distribución.

Encontramos diferentes distribuciones de las estaciones a lo largo de la línea, así como diferentes formas de línea de ensamblaje.

a) Líneas seriales: En este tipo de líneas las estaciones están colocadas sucesivamente, pasando las tareas

de una estación a la siguiente mediante el empleo de elementos de transporte tales como cintas transportadoras.

b) Líneas con estaciones en paralelo: Son líneas en las que se permiten estaciones en paralelo con vistas a solucionar los casos en que algunas de las tareas tengan un tiempo que exceda al tiempo de ciclo, dado que la duración de la tarea se verá reducida de forma proporcional al número de estaciones que se introduzcan en paralelo. Todas las estaciones que se instalen en paralelo deberán realizar las mismas tareas, por lo que deberán estar equipadas con las mismas máquinas y herramientas.

c) Líneas paralelas: En esta configuración se disponen varias líneas en paralelo. Para modelos múltiples se asocia cada línea a un modelo o a una familia de modelos.

d) Líneas de dos lados: Se trata de dos líneas de tipo serial dispuestas en paralelo, cada una de las cuales consta de estaciones opuestas entre sí que procesan simultáneamente una misma pieza. Este tipo de líneas se emplean en el ensamblaje de automóviles, donde se requiere realizar tareas a ambos lados del producto.

e) Líneas circulares: En este tipo de distribuciones las estaciones están dispuestas alrededor de una cinta transportadora circular, de la cual los operadores toman las piezas sobre las que van a realizar operaciones y, una vez terminadas, las depositan de nuevo en la cinta, excepto el operario que realiza la operación final, que retirará la pieza de la cinta transportadora.

f) Líneas en forma de U: Este tipo de configuración hace más flexible la producción, aumenta la interacción entre trabajadores. También requiere que éstos tengan múltiples habilidades para operar diferentes máquinas y procesos.

3.3.5 Ritmo de flujo.

Podemos encontrar tres tipos de líneas atendiendo a esta característica, siendo éstas las líneas síncronas, las asíncronas y las líneas de alimentación.

a) Líneas síncronas: En este tipo de líneas todas las estaciones tienen el mismo tiempo de ciclo, con lo que las piezas van pasando de una estación a la siguiente de forma ininterrumpida, no existiendo almacenes intermedios de piezas.

b) Líneas asíncronas: En este tipo de líneas las estaciones tienen diferentes tiempos de proceso, con lo que entre estaciones se hace necesario colocar almacenes de productos intermedios.

c) Líneas de alimentación: En este tipo de líneas se tienen líneas subordinadas que alimentan a la principal. Éstas líneas se denominan líneas de alimentación, *feeder lines*, y en ellas se realizan sub ensamblados.

4 EL PROBLEMA DE EQUILIBRADO DE LÍNEAS DE MONTAJE

Los problemas de equilibrado consisten en distribuir las tareas necesarias para realizar un ensamblado entre las diferentes estaciones que componen la línea de montaje, considerando las relaciones de precedencia y el tiempo de ciclo de la línea. Capacho Betancourt (Capacho Betancourt, 2004) cita a Baybars para decir que una línea se considera equilibrada cuando, utilizando los recursos al máximo, la suma de los tiempos ociosos de las estaciones es mínimo. También menciona el caso hipotético en que la línea se encuentra bajo condiciones de equilibrado perfecto, es decir, que no existen holguras en las estaciones. Cuando no se dan estas condiciones de equilibrado perfecto, la tasa de producción efectiva estará determinada por la estación más lenta: la estación cuello de botella.

4.1 Clasificación de los problemas de equilibrado de líneas de montaje.

En la literatura encontramos diversos tipos de problemas relacionados con las líneas de montaje. Así podemos encontrar problemas de balanceado de línea, de diseño o de desmontaje. Nuestro caso es un problema de equilibrado de línea de montaje.

Los problemas de equilibrado de línea de montaje se pueden clasificar a su vez según diferentes criterios.

Según Pascual García, encontramos principalmente dos clasificaciones de éstos problemas ampliamente aceptadas. (Pascual García, 2015). En base a éstas se realizan la mayoría de clasificaciones encontradas en la literatura.

- La primera sería la clasificación realizada por Baybars donde distingue entre dos tipos de problema atendiendo a su simplicidad. Aparecen aquí el problema de equilibrado simple y el general, *Simple Assembly Line Balancing Problem (SALBP)* y *General Assemble Line Balancing Problem (GALBP).*

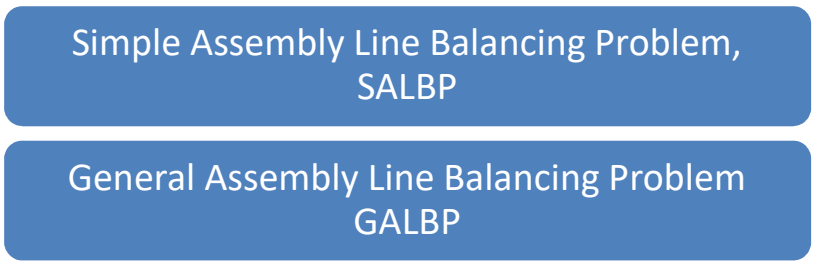


Ilustración 4-1: Clasificación de los problemas de equilibrado según Baybars.

- La segunda clasificación fue realizada por Ghosh y Gagnon, quienes basan su clasificación en el tipo de producto y la variabilidad de los tiempos de operación, de forma que tendríamos problemas monomodelo o de modelo mixto y múltiple, que a su vez se subdividirían según sus tiempos sean deterministas o estocásticos.

Ilustración 4-2: Clasificación de los problemas de equilibrado según Ghosh y Gagnon.

.

Atendiendo a la primera clasificación realizada de los problemas de equilibrado de línea, Capacho Betancourt (Capacho Betancourt, 2004) realiza una revisión de los problemas analizados en la literatura y plantea la siguiente clasificación de los problemas de equilibrado:

u

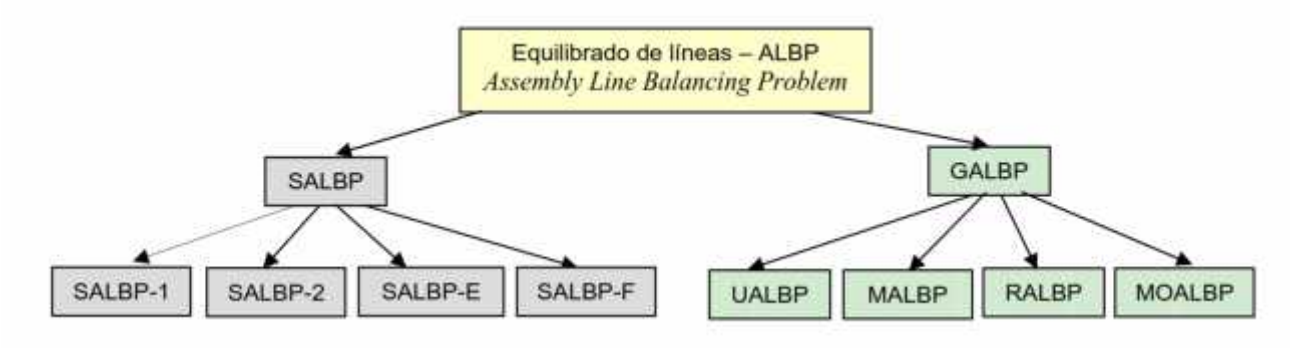


Ilustración 4-3: Clasificación de los problemas de equilibrado de líneas de montaje.

4.1.1 SALBP, Simple Assembly Line Balancing Problem.

Las características de los problemas simples de balanceo de línea han sido enumeradas por Boysen et al (Boysen, 2007) y quedan recogidas en la siguiente lista:

- Producción en masa de productos monomodelo.
- No existencia de formas alternativas de procesar las tareas.
- Línea síncrona conforme al tiempo de ciclo que se calcula en función de la cantidad que se quiere proucir.
- Línea de tipo serial sin elementos en paralelo ni líneas alimentadoras.
- Asignación de tareas limitada por relaciones de precedencia.
- Tiempos de operación determinísticos.
- Solo se le asignan a las tareas las restricciones de precedencia.
- Las tareas no se pueden dividir en más de una estación.
- Todas las estaciones están igualmente equipadas con respecto a maquinaria y trabajadores.

En función de la variable a optimizar por el problema de equilibrado encontramos cuatro categorías:

a) **SALBP-1:** Dado un tiempo de ciclo, minimizar el número de estaciones. Este tipo de problemas se da cuando se va a instalar una nueva línea siendo la demanda conocida.

b) **SALBP-2:** Dado el número de estaciones, minimizar el tiempo de ciclo. Este tipo de problemas se da cuando la línea ya está instalada y no se quiere modificar.

c) **SALBP-E:** Se pretende maximizar la eficiencia de la línea, calculada como el cociente entre la suma de los tiempos de las tareas y el producto del número de estaciones por el tiempo de ciclo. Dado que la suma de los tiempos de tareas se mantiene fijo, el problema se traduce en minimizar el producto entre el número de estaciones y el tiempo de ciclo.

d) **SALBP-F:** Este problema busca una solución factible dados un tiempo de ciclo y un número determinado de estaciones.

Esta clasificación queda recogida en la siguiente tabla:

	Cycle time *c*	
	Given	**Minimize**
No. *m* of stations		
Given	SALBP-F	SALBP-2
Minimize	SALBP-1	SALBP-E

Ilustración 4-4: Clasificación de los SALBP.

4.1.2 GALBP, General Assembly Line Balancing Problem.

Los problemas GALBP abarcan todos los problemas de equilibrado que no están contemplados en la categoría de SALBP, como aquellos que tienen estaciones en paralelo, modelos mixtos, tiempos de proceso variables, procesamientos alternativos, configuraciones en forma de U, etc. Estos problemas se acercan más a la realidad de la industria que los SALBP. De entre los problemas de este tipo destacan estas cuatro categorías:

a) **UALBP, U-Line Assembly Line Balancing Problem:** Este tipo de problemas están caracterizados de forma similar a los SALBP, con la salvedad de que en este tipo de problemas la línea no es lineal sino que tiene forma de U. En este tipo de configuraciones existe una mayor flexibilidad debido a que se pueden asignar tareas sin que hayan sido asignadas sus predecesoras. Además en este tipo de configuración una misma estación puede operar piezas que están en distintos puntos de la línea. Por ejemplo, una misma estación puede contener operaciones que se realizan al inicio y al final de la línea. Este tipo de estaciones se denominan *crossover stations*. Debido a esta característica de las líneas en U aumenta el número de posibilidades a la hora de equilibrar la línea, permitiendo así un equilibrado más eficiente con respecto a las líneas en serie. Al igual que en el caso de los SALBP encontramos tres tipos de UALBP en relación a los objetivos de optimización, existiendo así el UALBP-1, donde el objetivo es minimizar el número de estacionas para un tiempo de ciclo dado, el UALBP-2, cuyo objetivo es minimizar el tiempo de ciclo para un número de estaciones dado y el UALBP-E, que busca maximizar la eficiencia de la línea minimizando el producto entre el número de estaciones y el tiempo de ciclo. Cabe añadir que cualquier solución factible para un SALBP también lo es para un UALBP, dado que no es obligatorio incluir *crossover stations* en éstas líneas. Sin embargo, debido al citado aumento de posibilidades de emparejamiento entre operaciones y estaciones a menudo el óptimo de una solución al UALBP mejora al de las SALBP.

b) **MALBP, Mixed-Model Assembly Line Balancing Problem:** Se trata del problema de equilibrado de líneas de modelos mixtos. Según Becker y Scholl (Becker, 2006), este tipo de líneas producen diferentes modelos de un mismo producto, los cuales pueden variar entre sí con respecto al tamaño, color o material del producto, por lo que la producción requerirá de diferentes operaciones, tiempos de operación y/o relaciones de precedencia. En este tipo de problemas se introduce la necesidad de obtener, además de la distribución de tareas entre estaciones, la secuenciación de las mismas, presentando así lo que se denomina como mixel-model sequencing problem, MSP. En analogía con los SALBP, los MALBP consisten en encontrar un número de estaciones y tiempos de ciclo de forma que se optimice la capacidad o costo del sistema, existiendo así los siguientes tres tipos de problemas mixtos: MALBP-1, MALBP-2 y MALBP-E, cuyos objetivos ya hemos mencionado anteriormente. Este tipo de problemas se modela y soluciona mediante dos procedimientos diferentes: reducción a problemas de tipo simple, o balanceo horizontal bajo el contexto de los modelos múltiples.

c) **RALBP, Robotic Assembly Line Balancing Problem:** Según Levitin et al, (Levitin, 2006) debido al incremento de la variedad en los productos demandados y a los rápidos cambios tecnológicos, la flexibilidad en la producción se ha convertido en un requisito importante. Esto ha provocado el desarrollo de los sistemas de ensamblado flexible (FAS, por sus siglas en inglés), equipados por robots ensambladores. El balanceo de líneas de ensamblaje roboticas persigue dos objetivos principales: optimizar el balanceo de la línea para un número dado de estaciones o para una tasa de

producción dada, y asignar a cada estación el robot que mejor se adapte a las operaciones a realizar, dado que en este tipo de problemas la diración de las operaciones depende del robot que las acometa. A diferencia del balanceo de línea donde el trabajo de ensamblaje es manual y donde el balanceo óptimo es mas bien teórico debido a la variabilidad de los tiempos de operación (tiempos estocásticos), para el balanceo de línas robotizadas es crucial la calidad de la solución al problema de equilibrado, así como la asignación de robots a las diferentes estaciones. Los citados Levitin et al han desarrollado un algoritmo genético capaz de resolver problemas complejos de tipo RALBP, minimizando el tiempo de ciclo para un número de estaciones dado. Según afirman, esta solución no presenta los problemas de grandes requerimientos en tiempos de procesamiento computacional y de almacenamiento que sí presentan los métodos Branch and Bound.

d) MOALBP, Multi-Objective Assembly Line Balancing Problem: Este tipo de problemas buscan satisfacer varios objetivos simultáneamente. Malakooti, por ejemplo, (Malakooti, 1994) resuelve este problema buscando optimizar el número de estaciones, el tamaño de los buffers, el tiempo de ciclo y el coste total de operación con buffers. Otro ejemplo de solución a un problema de equilibrado multiobjetivo es aportado por Pastor et al (Pastor, 2002), quienes buscan maximizar la productividad, equilibrar el tiempo de ciclo para todos los modelos producidos, equilibrar la carga de trabajo de las estaciones y minimizar la dispersión del trabajo en los diferentes modelos, asignando las tareas comunes de los diferentes modelos a la misma estación.

En lo sucesivo focalizaremos en la resolución de SALBP-1, donde dado el tiempo de ciclo el objetivo es minimizar el número de estaciones.

4.2 Definiciones

Los métodos de equilibrado que vamos a estudiar se basan en una serie de conceptos que se definen a continuación:

- **Operación (i=1…N):** Es la unidad de trabajo más pequeña e indivisible. Las operaciones se reaizan en las estaciones por los operarios, pudiendo ser éstos humanos o robotizados. Cada operación tiene asignado un tiempo de operación. Llamaremos N al número total de operaciones.
- **Estación (j=1…M):** Es la zona de trabajo perteneciente a la línea de montaje en la que se ejecutan las operaciones. Dispone de las herramientas y piezas necesarias para la ejecución de las operaciones. Llamaremos M al número total de estaciones.
- **Tiempo de operación (t_i):** Es el tiempo que tarda en ejecutarse la operación i.
- **Tiempo de producción diario:** Es el tiempo total disponible diariamente para producir. Se puede calcular conociendo el número de trabajadores y la duración de la jornada laboral, excluyendo los tiempos de descanso.
- **Tiempo de ciclo (t_c):** Es el tiempo que una pieza permanece en una estación de trabajo. También indica el tiempo que tarda una pieza completada (o producto terminado) en salir por el final de la línea de montaje.

$$t_c = \frac{T \quad d \; p \quad \acute{o} n \; d}{D \quad d}$$

- **Tiempo total de montaje (TT):** Equivale a la suma de todos los tiempos de operación.

$$T = \sum_{i=1}^{N} t_i$$

- **Tasa de producción:** Es el número de unidades producidas por unidad de tiempo. Se calcula como la inversa del tiempo de ciclo.

$$P = \frac{1}{t_c}$$

- **Carga de trabajo:** Se define como el conjunto de operaciones asignadas a una estación de trabajo.
- **Número mínimo de estaciones de trabajo (M_{mi}):** Es el mínimo número de estaciones necesarias para poder asignar todas las operaciones a las estaciones.

$$M_m = \left[\frac{S_i \quad d \quad t_i \quad d \quad o}{T \quad d \quad c_i}\right]^+$$

- **Tiempo de operación de una estación (TO_j):** Tiempo total empleado en una estación a realizar todas las operaciones que tiene asignadas.

$$TO_j = \sum_{i \in j} t_i$$

- **Tiempo ocioso de una estación (OI_j):** Es el tiempo que transcurre desde que finalizan todas las operaciones a realizar en una estación hasta que se alcanza el tiempo de ciclo.

$$OI_j = C - TO_j$$

- **Tiempo ocioso total (D):** Se define como la cantidad de tiempo ocioso existente en toda la línea de montaje. Se calcula como la diferencia entre el número de estaciones por el tiempo de ciclo menos el tiempo total de montaje:

$$D = C \cdot M - T$$

- **Relaciones de precedencia:** Indican el orden de ejecución de las operaciones, de forma que no se puede llevar a cabo una operación hasta que no se hayan completado todas las operaciones que la preceden.

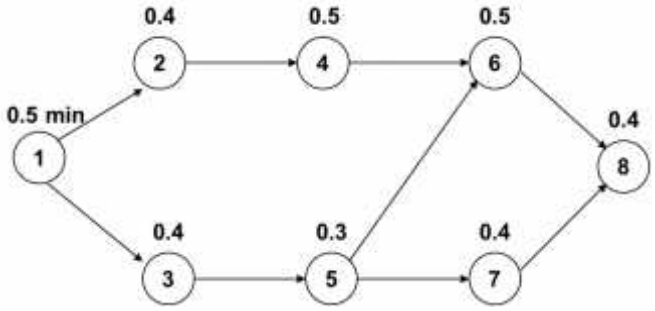


Ilustración 4-5: Ejemplo de diagrama de relaciones de precedencia.

- **Restricciones de zona:** Indican si las tareas se deben realizar forzosamente en la misma estación, prohibición de realizarlas en la misma estación u obligatoriedad de realizarlas en estaciones colindantes. Podemos clasificarlas en tres tipos:
  - **Zonificación Positiva:** Obliga a que varias tareas se realicen en la misma estación.
  - **Zonificación Negativa:** Impide que dos tareas se realicen en la misma estación.
  - **Zonificación Límite:** Obliga a que se asignen las tareas a estaciones anteriores o posteriores a una dada.

4.3 Métodos de resolución de los SALBP-1.

Como una primera clasificación podemos dividir los métodos de resolución de los SALBP en dos categorías, siendo éstas las conformadas por los algoritmos exactos y por los algoritmos heurísticos. Los métodos exactos proporcionan la solución óptima al problema, pero tienen el inconveniente de que solo pueden resolver problemas de un tamaño limitado, puesto que debido a la naturaleza combinatoria de los problemas el número de posibilidades a explorar aumenta de forma exponencial con el número de operaciones, lo que conlleva grandes esfuerzos computacionales. Por su parte los métodos heurísticos permiten aproximarse al óptimo, incluso a veces llegan a alcanzarlo.

La variedad de métodos desarrollados en la literatura es amplia. Scholl y Becker (Scholl, 2006) realizan una revisión exhaustiva de los métodos existentes, clasificándolos en soluciones exactas y heurísticas para los diferentes problemas SALBP.

4.3.1 Soluciones Exactas.

En su revisión del trabajo de investigación existente relacionado con la resolución de SALBP-1, los citados Scholl y Becker han sumarizado hasta dos docenas de procedimientos diferentes, que clasifican en procedimientos de branch and bound (B&B) y procedimientos de programación dinámica (DP, por sus siglas en inglés).

4.3.1.1 Formulación del problema.

Faura Cabañas (Faura Cabañas, 2003) expone el modelo matemático de programación lineal binaria desarrollado por White, quedando como sigue:

$$F\ :[M\quad]z = \sum_{i=1}^{N}\sum_{j=1}^{M} w_j \cdot x_{i,j}$$

$$s.a.:$$

$$\sum_{j=1}^{M} x_{i,j} = 1\ \forall\ i = 1 \dots N \quad (1)$$

$$\sum_{i=1}^{N} t_i \cdot x_{i,j} \leq C \cdot y_j\ \forall\ j = 1 \dots M \qquad (2)$$

$$\sum_{j=1}^{M} j \cdot x_k \ \leq \ \sum_{j=1}^{Mm} j \cdot x_i \ \ \forall\ k \ (3)$$

$$y_{j+1} \leq y_j\ \forall\ j = 1 \dots M \qquad -1 \ (4)$$

$$x_i \ \in \{0,1\}\ \forall\ (i,j); y_j = \{0,1\}\ \forall\ (j)$$

Donde:

i es el subíndice asociado a las tareas.

j es el subíndice asociado a las estaciones.

N es el número total de tareas a asignar.

Mmax es el número máximo de estaciones de trabajo consideradas en el modelo.

w_j es el coeficiente de penalización que garantiza que no se empleen más estaciones de las estrictamente

necesarias, obligando a su vez a que las tareas se asignen a las primeras estaciones mediante la penalización de la asignación de éstas a las últimas estaciones, puesto que el valor del coeficiente aumenta mucho conforme las estaciones van aumentando.

C es el tiempo de ciclo.

t_i es el tiempo de ejecución de la tarea i-ésima.

$x_{i,j}$ es uno cuando la tarea i se realiza en la estación j.

y_j es uno si existe la estación j.

Significado de las restricciones:

La restricción (1) impone que cada operación sea asignada a una sola estación.

La restricción (2) garantiza que la duración de las tareas asignadas a una estación no exceda el tiempo de ciclo.

La restricción (3) impone que se cumplan las relaciones de precedencia.

Por último, la restricción (4) establece que de no existir una estación tampoco existan las sucesivas.

4.3.1.2 Procedimientos branch and bound.

Estos procedimientos se caracterizan por considerar el árbol de soluciones, donde éstas se ordenan conforme a las ramas de un árbol, y detectar cuándo las soluciones comienzan a ser inferiores al óptimo, lo cual permite desprenderse de las ramas que se encuentran debajo del nodo en consideración ahorrando así recursos de computación.

Para reslver este tipo de problemas, Scholl (Scholl, 2006) distingue entre dos estrategias de resolución.

a. In depth-first search (DFS): Esta estrategia de búsqueda desarrolla completamente una rama del árbol de precedencias hasta llegar al nodo final, almacena la solución encontrada y vuelve hacia la raíz examinando la siguiente rama alternativa.
b. Minimal lower bound strategy (MLB): Esta estrategia siempre elige un nodo aún no desarrollado que tenga el mínimo valor del límite inferior de una lista de candidatos. Este nodo es desarrollado completamente construyendo todos sus nodos descendientes.

4.3.1.3 Procedimientos de progamación dinámica.

El algoritmo propuesto por Held et al (Held, 1963) fue una de las primeras aplicaciones de la programación dinámica al problema de líneas, lo cual sentó un precedente.

Este modelo minimiza el número de estaciones para un tiempo de ciclo dado.

Para definir el modelo vamos a describir una serie de conceptos:

- **Subconjunto admisible:** Formado por j operaciones que se pueden ejecutar respetando el orden de precedencias.

$$v = \{J_1 \dots J_j\}$$

- **Subsecuencia admisible:** Es una de las ordenaciones posibles de un subconjunto admisible.

$$(J_1 \dots J_j)$$

- **Tiempo del subconjunto:** Se define como el menor tiempo obtenido para todas las subsecuencias posibles para el subconjunto dado.

$$t\{v\} = t(J_1 \dots J_j) = \min_{J_k \in v}(t\{v - J_k\} + \Delta J_k)$$

- **Tiempo de la subsecuecia:** Es el tiempo que tarda en ejecutarse una subsecuencia, considerando el tiempo de ciclo. Se define como el tiempo de la subsecuencia anterior añadiendo el incremento de

tiempo considerando la inclusión de una nueva tarea para conformar la nueva subsecuencia. Si la nueva tarea no entra en la estación considerada actualmente el tiempo de la subsecuencia será el tiempo de todas las estaciones llenas hasta ahora añadiendo el incremento de tiempo que supone la inlclusión de la nueva tarea.

$$t(J_1 \dots J_{j-1}, J_j) = t(J_1 \dots J_{j-1}) + \Delta J_j = \begin{cases} t(J_1 \dots J_{j-1}) + t_j & s\ J_j\ e \quad e\ \ l\ \ e \quad \acute{o}n\ M_i. \\ M_i \cdot C + t_j & s\ J_j\ n\ \ e \quad e\ \ l\ \ e \quad \acute{o}n\ M_i. \end{cases}$$

Vamos a ilustrar el modelo con un ejemplo.

Considerando el siguiente diagrama de precedencias, con los tiempos dados:

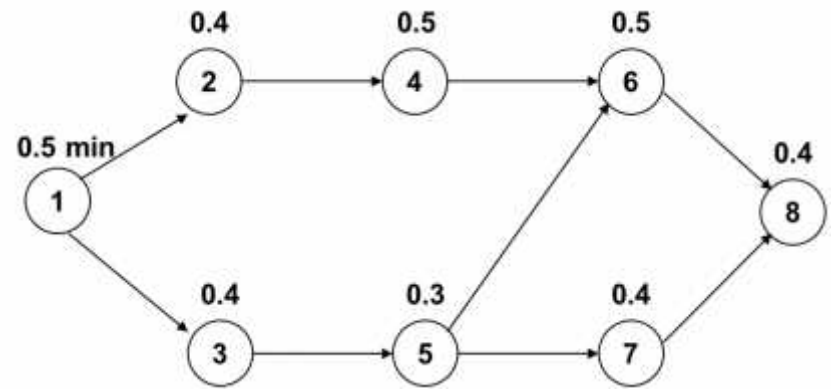


Ilustración 4-6: Ejemplo de Programación Dinámica.

Tarea	t_i	Predecesoras
1	0.5	-
2	0.4	1
3	0.4	1
4	0.5	2
5	0.3	3
6	0.5	4,5
7	0.4	5
8	0.4	6,7

Tabla 4-1: Datos del ejemplo de programación dinámica.

Para un tiempo de ciclo dado de 1 minuto por unidad la aplicación del método de programación dinámica propuesto por Held quedaría como sigue a continuación:

El primer subconjunto a considerar para iniciar el algoritmo es el subconjunto vacío.

$$v_0 = \{\} \rightarrow\ t\{\} = 0$$

Los siguientes subconjuntos a considerar incluyen a todos los que solo consideran una operación admisible según el diagrama de precedencias. En este caso es solo la operación 1.

$$v_1 = \{1\} \rightarrow\ t\{v_1\} = t\{v_0\} + \Delta t_1 = 0.5 \rightarrow |1|$$

Como resultado provisional tenemos la primera estación cargada con la operación uno |1|.

Pasamos a considerar los subconjuntos admisibles formados por dos operaciones. En este caso solo tenemos dos posibilidades:

$$v_{2_1} = \{1,2\} \rightarrow t\{v_{2_1}\} = t\{v_1\} + \Delta t_2 = 0.5 + 0.4 = 0.9 \rightarrow |1,2|$$

$$v_{2_2} = \{1,3\} \rightarrow t\{v_{2_2}\} = t\{v_1\} + \Delta t_3 = 0.5 + 0.4 = 0.9 \rightarrow |1,3|$$

Subconjuntos admisibles formados por tres operaciones:

$$v_{3_1} = \{1,2,4\} \rightarrow t\{v_{3_1}\} = t\{v_{2_1}\} + \Delta t_4 = 1 \cdot 1 + 0.5 = 1.5 \rightarrow |1,2|4|$$

$$v_{3_2} = \{1,2,3\} \rightarrow t\{v_{3_2}\} = M \quad (t\{v_{2_1}\} + \Delta t_3; t\{v_{2_2}\} + \Delta t_2) = 1.4 \rightarrow |1,2|3|; |1,3|2|$$

$$v_{3_3} = \{1,3,5\} \rightarrow t\{v_{3_3}\} = t\{v_{2_2}\} + \Delta t_5 = 1 \cdot 1 + 0.3 = 1.3$$

Subconjuntos admisibles formados por cuatro operaciones:

$$v_{4_1} = \{1,2,3,4\} \rightarrow t\{v_{4_1}\} = M \quad (t\{v_{3_1} + \Delta t_3;\ t\{v_{3_2}\} + \Delta t_4) = 1.9 \rightarrow |1,2|4,3|; |1,2|3,4|; |1,3|2,4|$$

$$v_{4_2} = \{1,2,3,5\} \rightarrow t\{v_{4_2}\} = M \quad (t\{v_{3_2}\} + \Delta t_5; t\{v_{3_2}\} + \Delta t_2) = 1.7$$

$$v_{4_3} = \{1,3,5,7\} \rightarrow t\{v_{4_3}\} = t\{v_{3_3}\} + \Delta t_7) = 1.7$$

Subconjuntos admisibles formados por cinco operaciones:

$$v_{5_1} = \{1,2,3,4,5\} \rightarrow t\{v_{5_1}\} = M \quad (t\{v_{4_1}\} + \Delta t_5; t\{v_{4_2}\} + \Delta t_4) = 2.3 \rightarrow |\ldots|\ldots|5|$$

$$v_{5_2} = \{1,2,3,5,7\} \rightarrow t\{v_{5_2}\} = M \quad (t\{v_{4_2}\} + \Delta t_7; t\{v_{4_3}\} + \Delta t_2) = 2.4$$

Subconjuntos admisibles formados por seis operaciones:

$$v_{6_1} = \{1,2,3,4,5,6\} \rightarrow t\{v_{6_1}\} = t\{v_{5_1}\} + \Delta t_6) = 2.8 \rightarrow |\ldots|\ldots|5,6|$$

$$v_{6_2} = \{1,2,3,4,5,7\} \rightarrow t\{v_{6_2}\} = M \quad (t\{v_{5_1}\} + \Delta t_7; t\{v_{5_2}\} + \Delta t_4) = 2.7$$

Subconjunto admisible formado por siete operaciones:

$$v_7 = \{1,2,3,4,5,6,7\} \rightarrow t\{v_7\} = M \quad (t\{v_{6_1}\} + \Delta t_7; t\{v_{6_2}\} + \Delta t_6) = 3.4 \rightarrow |\ldots|\ldots|5,6|7|$$

Y por último evaluamos el conjunto de todas las operaciones:

$$v_8 = \{1,2,3,4,5,6,7,8\} \rightarrow t\{v_8\} = t\{v_7\} + \Delta t_8 = 3.8 \rightarrow |\ldots|\ldots|5,6|7,8|$$

Para obtener la secuencia resultante del equilibrado por programación dinámica comenzamos desde el final y vamos construyendo la secuencia de operaciones de atrás adelante.

Las composiciones subrayadas son las que han ido obteniendo mejores tiempos de subconjunto, por lo tanto serán las que compongan la secuencia de montaje final. Algunos casos presentan soluciones alternativas, por lo que el resultado final quedaría recogido en la siguiente tabla:

Numero de secuencia	Composición de estaciones
1	|1,2|3,4|5,6|7,8|
2	|1,3|2,4|5,6|7,8|
3	|1,2|4,3|5,6|7,8|

Tabla 4-2 : Resultados del ejemplo de programación dinámica.

Todas las secuancias serían óptimas.

La eficiencia de la línea se calcula como el cociente entre la suma de tiempos de todas las operaciones y el tiempo total disponible en la línea:

$$E = \frac{\sum_{i=1}^{N} t_i}{M \cdot C} = \frac{3{,}4}{4 \cdot 1} = 0.85 = 85\%$$

4.3.1.4 Consideraciones para reducir el esfuerzo de cálculo.

En su trabajo presentan métodos para calcular **límites inferiores** en el número de estaciones a considerar, así como diferentes posibilidades para reducir el esfuerzo de enumeración de posibles soluciones mediante **reglas de dominancia** y reducir el esfuerzo computacional mediante **reglas de reducción**.

Entre los métodos para calcular el **límite inferior** de estaciones encontramos los siguientes:

a) Bin packing bounds.

b) One-machine scheduling bound.

c) Destructive improvement bounds.

Las **reglas de dominancia** permiten acotar las souciones estableciendo una dominancia de unas soluciones parciales sobre otras, reduciendo así el tamaño del problema en cuanto a la enumeración de posibles soluciones factibles, dado que dichas soluciones parciales quedan excluidas sin llegar a ser completadas.

Las reglas de dominancia enumeradas por Scholl y Becker son las siguientes:

a) Regla de la máxima carga.

b) Regla de Jackson.

c) Regla del set factible.

d) Reglas de ordenación de estaciones.

Además de las reglas de dominancia, también describen **reglas de reducción**, orientadas a disminuir el esfuerzo computacional derivado de la resolución de este tipo de problemas. Estas reglas de reducción modifican algunos datos del problema de forma que el número de posibles soluciones se vea reducido o el límite inferior del número de estaciones se vea incrementado, teniendo en cuenta que el problema reducido mantiene al menos una solución óptima en común con el problema original. Estas reglas son enumeradas a continuación:

a) Task time incrementing rule.

b) Prefixing.

c) Aditional precedence relations.

d) Task conjoining rule.

4.3.2 Soluciones Heurísticas.

Existen diversos métodos heurísticos de resolución de los SALBP. Pascual García (Pascual García, 2015) se basa en la clasificación realizada por Talbot, Patterson y Gehrlein para ofrecer una clasificación de las mismas.

Se pueden clasificar en algoritmos basados en reglas de backtracking, algoritmos de aproximación a partir de algoritmos exactos, algoritmos de una sola pasada y algoritmos de composición.

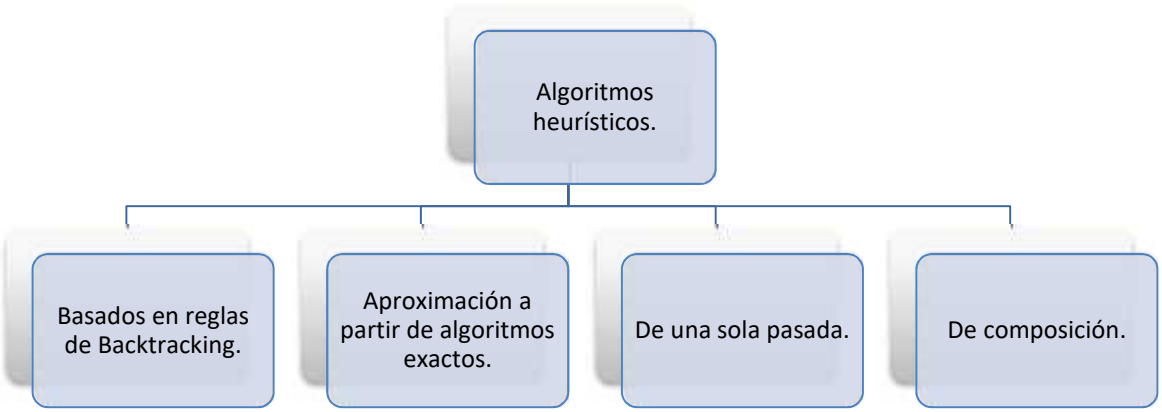


Ilustración 4-7: Clasificación de algoritmos heurísticos.

4.3.2.1 Algoritmos basados en reglas de backtracking.

El algoritmo de backtracking enumera una lista de posibles candidatos que, en principio, podrá ser completada de diferentes formas para dar todas las posibles soluciones. Esta forma de completar la lista se va haciendo de forma gradual mediante una secuencia de pasos de extensión de candidatos.

a) Eureka. Este algoritmo presentado por Hoffmann (Hoffmann, 1992) emplea el método de resolución DFS (in deep-first search). Es un método de búsqueda orientado a la estación que combina la aplicación de un algoritmo branch and bound que explora ambas direcciones del árbol de soluciones, junto con reglas heurísticas, de forma que si el procedimiento de branch and bound no encuentra una solución factible en un límite de tiempo dado yendo hacia delante, iniciará la búsqueda hacia atrás. Si tampoco lo encuentra, se aplicará la heurística propuesta por Hoffmann.

b) MALB. Propuesto por Dar-El (Dar-El, 1973), emplea cuatro heurísticas que controlan la cantidad de retroceso permitido. Es capaz de encontrar soluciones factibles en un tiempo computacional bajo para problemas de hasta 140 tareas empleando diferentes diagramas de precedencias.

4.3.2.2 Aproximacion partiendo de algoritmos exactos.

El algoritmo FABLE, acrónimo de Fast Algorithm for Balancing Lines Effectively y desarrollado por Johnson (Johnson, 1998), emplea un algoritmo branch and bound para recorrer las posibles soluciones e incluye reglas de dominancia para reducir el tamaño del árbol de enumeración, además de cuatro tipos de cotas que se emplean para descartar soluciones sub-óptimas.

Emplea una estrategia de búsqueda orientada a la tarea y permite obtener soluciones factibles para problemas de más de mil operaciones en menos de 20 segundos de tiempo de computación en una CPU 3033U de IBM.

4.3.2.3 Algoritmos de composición.

COMSOAL. Acrónimo de Computer Method of Sequencing Operations of Assembly Lines. Presentado por Arcus (ARCUS*, 1965), es un algoritmo que destaca por su sencillez de implementación en entornos de programación. Restrepo et al (Restrepo, 2009) aplica este algoritmo a un problema de 57 tareas, obteniendo un equilibrado con una eficiencia del 90.6%. El algoritmo COMSOAL consta de los siguientes 6 pasos:

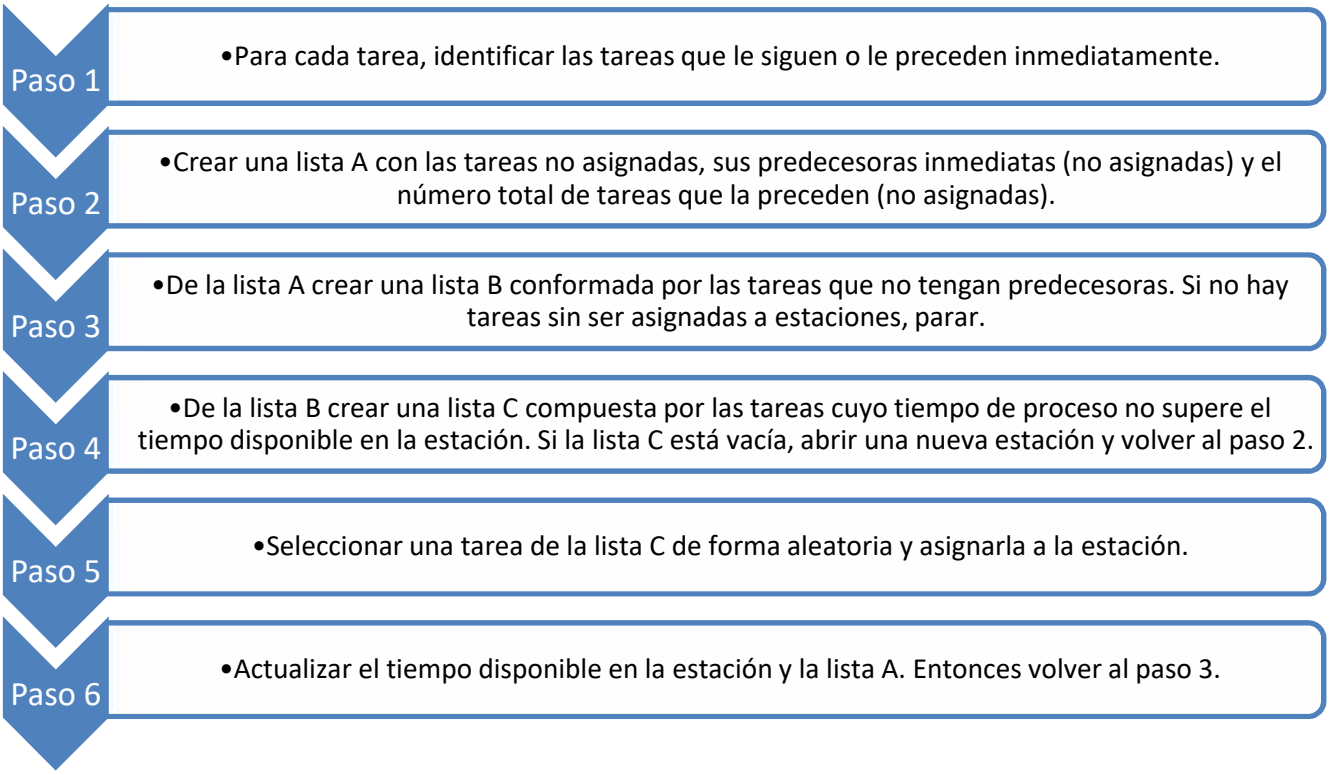


Ilustración 4-8: Algoritmo COMSOAL.

4.3.2.4 Algoritmos de una sola pasada.

4.3.2.4.1 Kilbridge and Wester.

También conocido como el método de las columnas, se trata de un método gráfico de equilibrado, por lo que el resultado dependerá en parte de la persona que ejecute el método.

Para implementar este método primero ha de construirse el diagrama de precedencias en forma de columnas, asignando a cada columna las operaciones que tienen el mismo nivel de precedencias. En la primera clumna se ubican todas las tareas que no tengan precedencias, asignando a las siguientes columnas las tareas cuyas predecesoras ya se encuentren en el diagrama. Las tareas pueden desplazarse de una columna a otra.

Como reglas heurísticas, se debe asignar cuanto antes las tareas de mayor duración, y de realizar desplazamientos entre columnas éstos deben ser los mínimos.

Para ejemplificar su uso resolveremos el mismo problema resuelto mediante programación dinámica en el apartado 4.3.1.3. La tabla recoge las precedencias y los tiempos de operación, y el tiempo de ciclo se mantiene a 1 minuto/unidad.

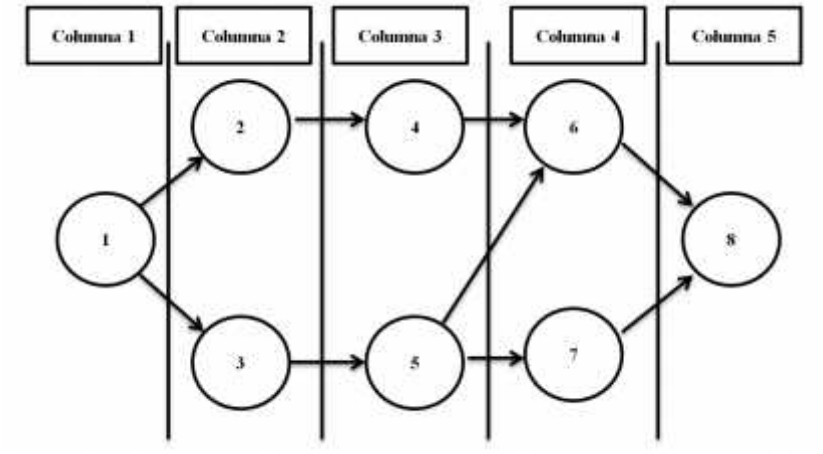


Ilustración 4-9: Método de las columnas. Situación de partida.

Tarea	t_i	Predecesoras
1	0.5	-
2	0.4	1
3	0.4	1
4	0.5	2
5	0.3	3
6	0.5	4,5
7	0.4	5
8	0.4	6,7

Tabla 4-3 : Datos del ejemplo de aplicación del método de las columnas.

Se trata de ir colocando las operaciones en las columnas de forma que el tiempo total empleado en la estación no supere el tiempo de ciclo.

Como primera iteración, movemos la operación 2 de la segunda columna a la primera. También podríamos haber seleccionado la tarea 3 puesto que tiene el mismo tiempo de operación que la tarea 2. Nuestro diagrama de precedencias quedaría como sigue:

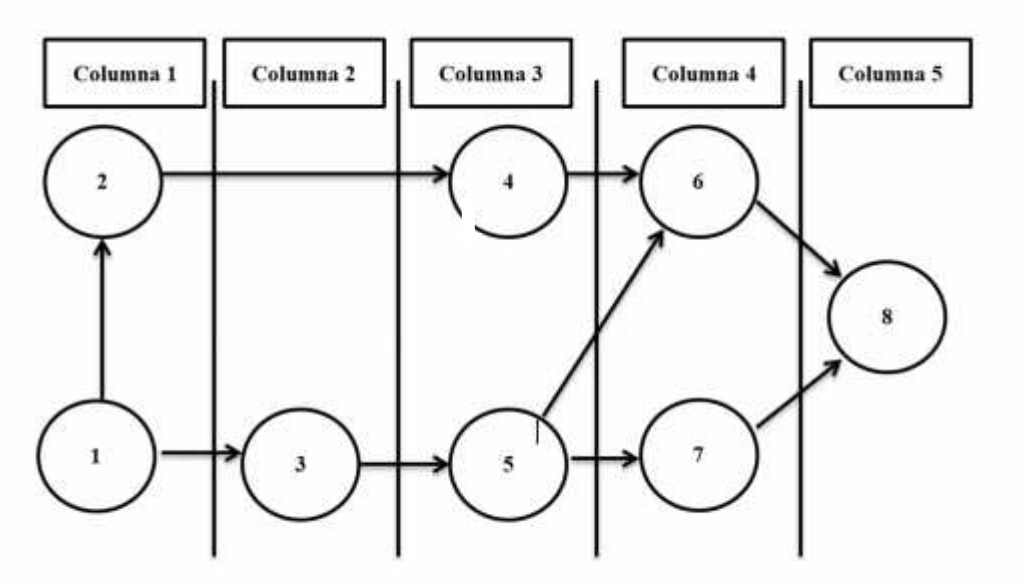


Ilustración 4-10: Método de las columnas. Primera iteración.

En la columna 1 hemos ocupado 0,9 minutos, por lo que ni la tarea 3 ni la 4 cabrían en esta columna. Pasamos a evaluar la segunda columna, donde ocurre lo mismo. Entre las tareas 3 y 4 suman 0,9 minutos, con lo que ya tendríamos completa la segunda columna.

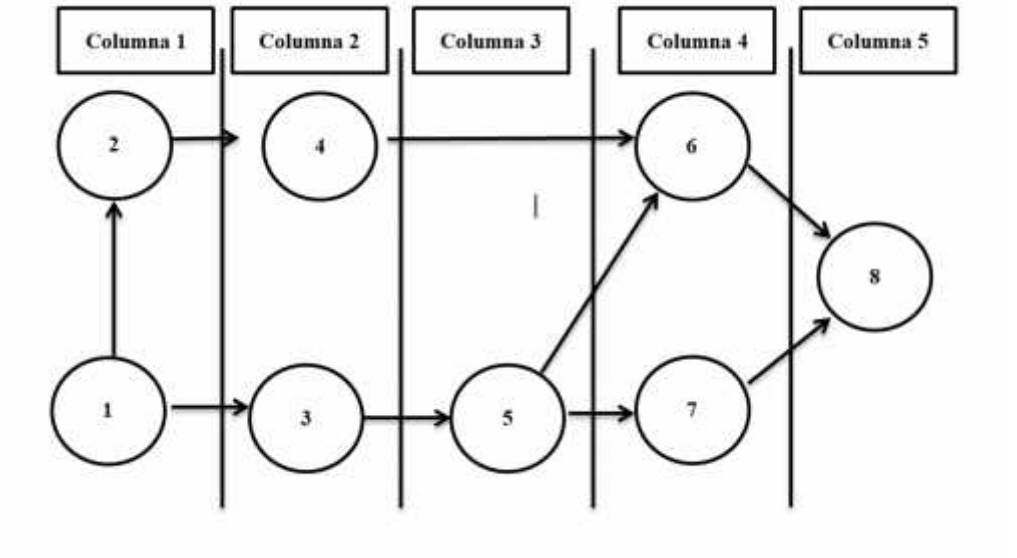


Ilustración 4-11: Método de las columnas. Segunda iteración.

Entre las operaciones 5 y 6 se emplean 0,8 minutos, por lo que en la columna 3 no cabrían las operaciones 5, 6 y 7. Otra opción para completar la columna 3 sería ocuparla con las tareas 5 y 7. Optamos por la primera solución.

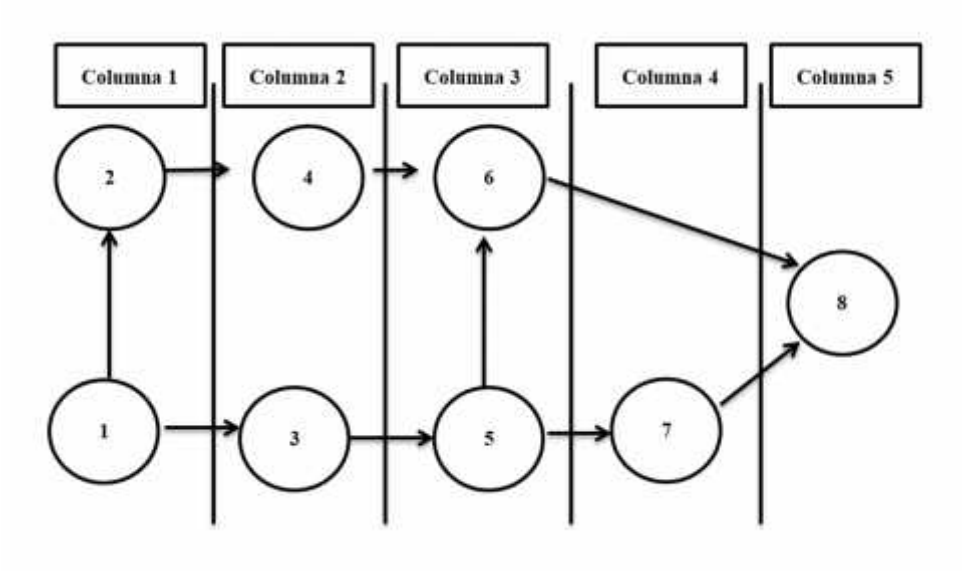


Ilustración 4-12: Método de las columnas. Tercera iteración

Por último, las operaciones 7 y 8 suman un total de 0,8 minutos, por lo que cabrían ambas en la columna 4.

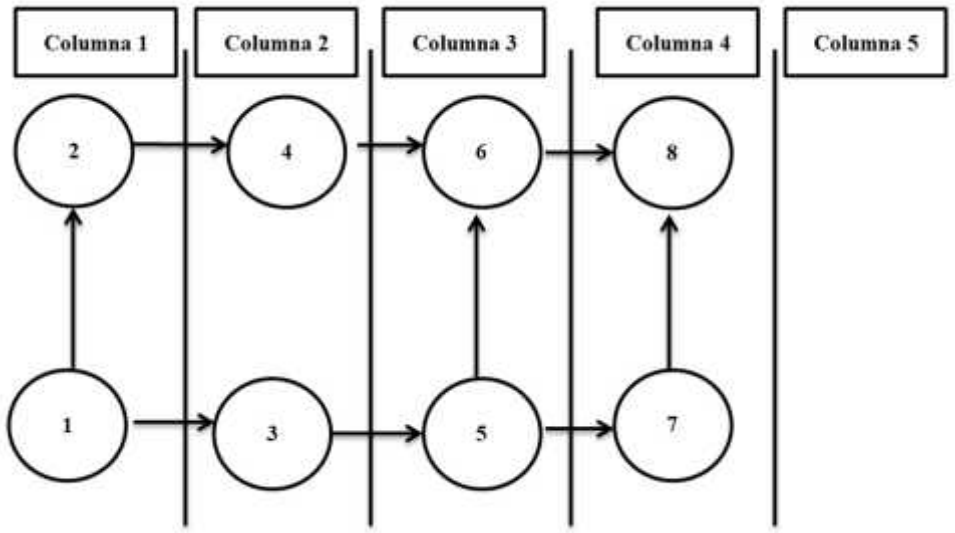


Ilustración 4-13: Método de las columnas. Cuarta iteración.

Como resultado hemos obtenido la secuencia de operaciones: |1,2|3,4|5,6|7,8|. Hemos obtenido una solución con 4 estaciones.

Estación	Operaciones	Tiempo ocioso (min)
1	1,2	0.1
2	3,4	0.1
3	5,6	0.2
4	7,8	0.2

Tabla 4-4: Tiepo ocioso por estación.

Como podemos observar, la secuenciación obtenida coincide con una de las soluciones aportadas por el método de programación dinámica, luego hemos alcanzado una solución óptima al equilibrado. La eficiencia de la línea se calcula como:

$$E = \frac{\sum_{i=1}^{N} t_i}{M \cdot C} = \frac{3{,}4}{4 \cdot 1} = 0.85 = 85\%$$

4.3.2.4.2 Helgenson & Birnie

Este método asigna pesos posicionales a las operaciones en función de su posición en el diagrama de precedencias, priorizando la asignación de aquellas operaciones cuya suma de tiempos de todas las operaciones que las suceden, incluyendo su propio tiempo de operación, sea mayor.

El peso posicional de cada tarea se calcula según la fórmula:

$$p_i = t_i + \sum_{k \in S(i)} t_k$$

Donde S(i) son todas las tareas que suceden a la tarea i.

El proceso de ejecución de éste método se describe en la Ilustración 4-14.

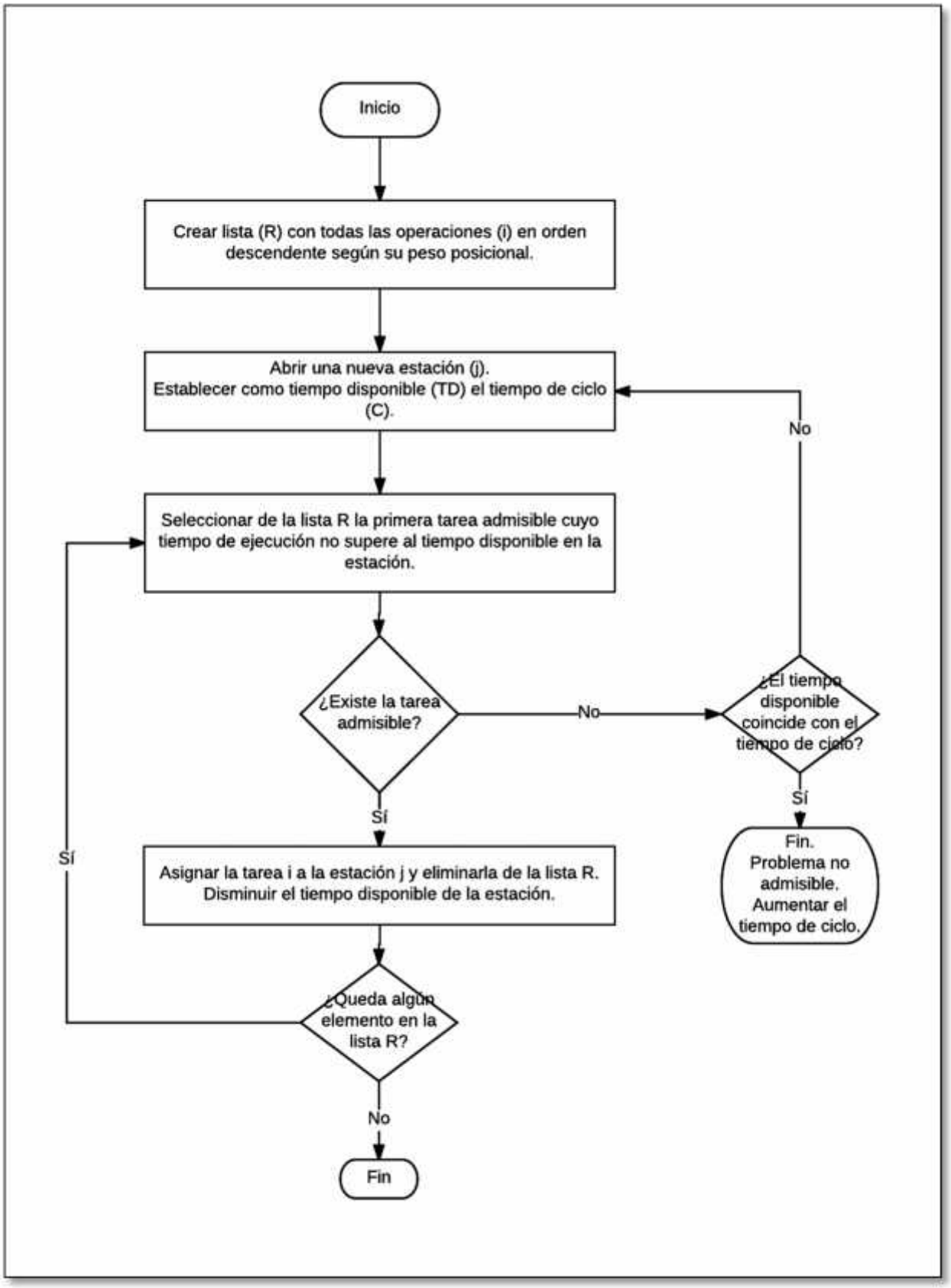


Ilustración 4-14: Diagrama de flujo del método Helgeson & Birnie.

Ilustremos su aplicación con un ejemplo:

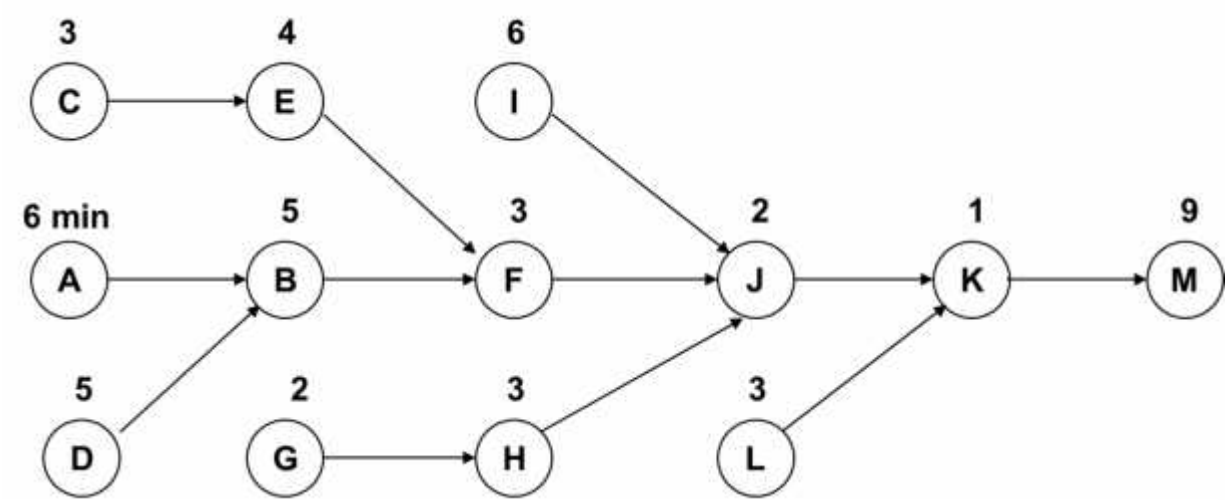


Ilustración 4-15: Ejemplo de equilibrado mediante pesos posicionales. Diagrama de precedencias.

El tiempo de ciclo es de 10 minutos por unidad y el objetivo es minimizar el número de estaciones (SALBP-1).

Comenzamos el método creando la lista R, que recoge las tareas ordenadas por su peso posicional.

Tarea	A	B	C	D	E	F	G	H	I	J	K	L	M
Tiempo	6	5	3	5	4	3	2	3	6	2	1	3	9
Peso	26	25	22	20	19	18	17	15	15	13	12	10	9

Tabla 4-5: Lista R.

Estación	Tareas admisibles	Tarea candidata	Tiempo de operación	Tiempo acumulado en la estación	Tiempo disponible	¿Asignar?
I	{A,D,C,I,G,L}	A	6	6	4	Sí
	{D,C,I,G,L}	D	5	11	-1	No
		C	3	9	1	Sí
II	{D,E,I,G,L}	D	5	5	5	Sí
	{B,E,I,G,L}	B	5	10	0	Sí
III	{E,I,G,L}	E	4	4	6	Sí
	{F,G,I,L}	F	3	7	3	Sí
	{I,G,L}	I	6	13	-3	No
		G	2	9	1	Sí
IV	{I,H,L}	I	6	6	4	Sí
	{H,L}	H	3	9	1	Sí
V	{J,L}	J	2	2	8	Sí
	{L}	L	3	5	5	Sí
VI	{K}	K	1	1	9	Sí
	{M}	M	9	10	0	Sí

Tabla 4-6: Equilibrado aplicando pesos posicionales.

Estación	Tareas asignadas	Tiempo ocioso
I	{A,C}	1
II	{D,B}	0
III	{E,F,G}	1
IV	{I,H}	1
V	{J,L}	5
VI	{K,M}	0

Tabla 4-7: Carga de las estaciones tras equilibrado.

La eficiencia de este balanceo de línea es:

$$E = \frac{\sum_{i=1}^{N} t_i}{M \cdot C} = \frac{52}{6 \cdot 10} = 0.867 = 86{,}7\%$$

4.3.2.4.3 Largest Candidate Rule.

En este método heurístico el algoritmo a aplicar es el mismo que en el caso de los pesos posicionales, con la diferencia de que el criterio de ordenación de la lista R varía. En este caso las tareas con mayor preferencia a la hora de ser asignadas serán aquellas cuyo tiempo de operación sea mayor.

Éste será el método que implementaremos en Aras Innovator para balancear nuestra línea de montaje.

5 INTRODUCCIÓN A LOS SISTEMAS PLM

5.1 Introducción.

Un sistema PLM (Product Lifecycle Management) es una herramienta para la **gestión del ciclo de vida del producto**.

El ciclo de vida del producto es el recorrido que transita un producto a lo largo de su desarrollo desde las fases de ideación, definición y realización, hasta cuando es usado y finaliza su vida útil (Stark, 2015). **Las etapas del ciclo de vida** se pueden describir en siete puntos:

- Definición estratégica.
- Diseño conceptual o preliminar.
- Diseño de detalle y desarrollo.
- Verificación y testeo.
- Producción y/o construcción.
- Utilización del producto.
- Retirada, reciclaje y fin de vida del producto.

Ilustración 5-1: Etapas del ciclo de vida de un producto.

Durante este trayecto **se genera una gran cantidad de información** tal como: requerimientos, bocetos, planos técnicos, modelos, prototipos, resultados de pruebas, indicadores de sostenibilidad, etc. Tener control sobre los documentos de diseño, administrar esta información e integrarla con otros procesos organizacionales es una tarea compleja. Hoy en día las empresas utilizan los **sistemas PLM** como estrategia para el

almacenamiento, gestión y compartición eficiente de los datos técnicos del producto, ahorrando así en costos y tiempo.

El uso de un ambiente PLM como estrategia organizacional trae algunas **ventajas** tales como la reducción del tiempo de lanzamiento al mercado del producto, la optimización de los tiempos de desarrollo, la reducción de errores, eliminación de reprocesos, optimización en la utilización de recursos, aumento de la productividad, mejora de la calidad del producto, disminución de los costos de introducción al mercado, además del cumplimiento de normativas y el aumento de la rentabilidad. (Gómez).

Este efecto sobre los costes del empleo de herramientas colaborativas de gestión del ciclo de vida del producto se explica atendiendo a cómo se organizaba tradicionalmente este proceso. El enfoque tradicional emplea un desarrollo de producto conocido como "**Comuniación sobre la pared**", donde cada área de la empresa, después de ejecutar la parte que le corresponde, transfiere su resultado a la siguiente área, la cual detectará fallas desde la perspectiva de su propia especialidad, lo que conllevará que se devuelva el trabajo realizado al área inicial para realizar los correspondientes ajustes. Esto genera muchos cambios y retroalimentaciones entre las diferentes áreas encargadas del desarrollo del producto, dado que ciertas características necesarias en etapas posteriores no se hayan tenido en cuenta en etapas previas.

Por su parte, el enfoque moderno de **ingeniería concurrente** se basa en el empleo de equipos multidisciplinares, integrados por profesionales en sus respectivas áreas, que trabajan conjuntamente (concurrentemente) desde el inicio del proyecto, de forma que todas las áreas implicadas en el ciclo de vida disponen de igual información acerca de la evolución del concepto de producto desde el inicio, de forma que puedan ir planificando su actuación en las etapas siguientes a la de definición detallada y planificación. Estas etapas exigen mayor inversión de tiempo que en el modelo tradicional de comunicación sobre la pared, pues ahí radica precisamente el éxito de la ingeniería concurrente: evitar futuros errores y necesidades de reprocesado enfocándose en las etapas tempranas del desarrollo del producto.

5.2 Los sistemas PLM.

Los sistemas PLM son sistemas informáticos que incluyen artículos, documentos, lista de materiales, resultados de análisis, especificaciones de pruebas, estándares de calidad, información sobre el rendimiento del producto, datos de proveedores, etc. En cuanto a la gestión de la información, los sistemas PLM permiten la recuperación de datos, el intercambio de información, la modificación de datos asociados al produto, dispone de capacidades automatizadas y proporciona seguridad en los datos.

Un sistema PLM sirve para:

- Centralizar y organizar todos los datos del producto.
- Gestionar formalmente los proyectos de diseño y desarrollo de productos.
- Integrar los procesos de diseño con los de industrialización y producción.

A grandes rasgos se trata de una plataforma común sobre la que todos los actores implicados en el desarrollo de un producto pueden trabajar a la vez, aportando información relevante desde su área de especialidad al desarrollo del producto y pudiendo acceder a su vez a la información aportada por el recto de actores.

Ilustración 5-2: Entornos colaborativos en sistemas PLM.

Como se muestra en la figura, los sistemas PLM se emplean en la empresa junto con sistemas de gestión de recursos de la empresa (ERP, Enterprise Resource Planning) sistemas de gestión de la cadena de suministros (SCM, Supply Chain Management) y sistemas de gestión de las relaciones con el cliente (Customer Relationship Management).

Los problemas más frecuentes que se encuentran día a día en la gestión del ciclo de vida del producto son los siguientes:

- Falta de claridad y definición de conceptos, términos y acrónimos existentes dentro de las empresas.
- Variedad de formatos en los que se almacena la información.
- Integración y actualización de la información producida por los diferentes departamentos.

5.3 Funciones de un sistema PLM.

Liñán Alfaro realiza una descripción de las principales funcionalidades de un sistema PLM, que desarrollaremos a continuación. (Liñán Alfaro, 2016).

5.3.1 Almacenar, organizar y proteger datos.

Uno de los mayores problemas que soluciona la aplicación de sistemas PLM en la empresa es el del acceso a la información. Mediante el uso de un sistema PLM los archivos quedan guardados en una misma base de datos, y los miembros de la empresa que producen la información pueden almacenarla de forma lógica, por ejemplo asociándola al producto, al cliente o al proyecto al que pertenezca. De esta forma no se generan archivos duplicados almacenados de forma caótica, ahorrando con ello el tiempo necesario para acceder a la información, dado que el usuario que quiera acceder a ella podrá hacerlo de forma lógia mediante una sencilla búsqueda.

Por otro lado, también permite proteger la información ante posibles pérdidas, con lo que también mejora la seguridad de la información.

5.3.2 Gestionar los documentos y sus cambios.

A partir del momento en que algún usuario de la aplicación crea un documento y lo almacena en el sistema, cualquier otro usuario (con los pertinentes permisos) puede acceder a dicho documento y realizar acciones sobre el, tales como modificaciones para crear una nueva versión, verificaciones, confirmación de revisiones o visualización del documento. El sistema permite realizar una trazabilidad sobre los cambios realizados en el documento, permitiendo saber a qué etapa del ciclo de vida pertenece cada versión del documento. También registra los cambios realizados sobre el documento, requiriendo al usuario datos referentes a quién ha realizado

la modificación, cuándo, por qué motivo y, además, que detalle los cambios realizados.

5.3.3 Buscar y recuperar la información.

Los sistemas PLM ofrecen una potente capacidad de búsqueda a sus usuarios. No solo permite localizar rápidamente cualquier documento, sino que además permite navegar en la estructura de la documentación, por lo que si este documento se tratase de un plano, por ejemplo, se podría saber fácilmente a qué pieza estaría asociado, y a su vez a qué conjunto pertenece dicha pieza.

5.3.4 Compartir datos con otros usuarios de forma controlada.

Esta función permite que varios usuarios puedan estar realizando modificaciones de forma simultánea sobre un mismo documento sin que exista riesgo de pérdida de información o solapamiento.

5.3.5 Ejecutar procesos y flujos de trabajo.

Un sistema PLM ayuda a definir y coordinar los procesos a realizar, junto con las acciones en las que se descomponen dichos procesos, estableciendo los actores responsables de llevar a cabo las diferentes tareas y la normativa a aplicar en cada caso. Permite plasmar de forma gráfica los flujos de trabajo que se llevarán a cabo durante la fabricación, incluyendo todos los detalles relevantes.

5.3.6 Crear, clasificar y gestionar ítems.

En un sistema PLM, los ítems están vinculados a documentos, de forma que cada vez que un ítem sufra un amodificación también lo hara el documento a través de dicho vínculo de unión entre ambos. De esta forma, la estructura representada en los documentos reflejará a la estructura del producto real, aunque los ítems se utilicen en nuevos proyectos o nuevos conjuntos.

5.3.7 Crear estructuras y listas de materiales.

Los sistemas PLM permiten crear relaciones entre los ítems para realizar la estructura de los productos. Un ejemplo de ello son las listas de materiales, o BOM (Bill of Materials).

5.3.8 Integrar la información de la ingeniería con otros sistemas y procesos informáticos empresariales.

Esta función permite que la información generada durante la fase de ingeniería se pueda exportar a otros sistemas de información de la empresa, como pueden ser los correspondientes al área de compras y de producción. De esta forma se consigue que los ítems, estructuras y listas de materiales creados se transfieran por completo al sistema ERP, encargado de tareas relacionadas con estos departamentos.

Sin el empleo de los sistemas PLM, dicha transferencia de información se tendría que realizar a mano, con las posibles pérdidas de información que ello conllevaría y el tiempo invertido en realizarla, por no hablar de las pérdidas que podría ocasionar un fallo en este punto aguas abajo del proceso.

5.3.9 Gestionar proyectos de diseño y desarrollo de productos.

Los sistemas PLM ofrecen funciones específicas en forma de aplicaciones que permiten gestionar proyectos mediante el control sobre los recursos, las tareas, los costes, los tiempos y los entregables.

5.4 Estructura de los sistemas PLM.

La estructura de un sistema PLM indica cómo se relacionan los elementos que componen el sistema entre sí. Todo sistema PLM se divide en tres componentes principales: el servidor, los clientes y el hardware.

5.4.1 Servidor.

Se apoya en una base de datos relacional, donde las varibles y sus valores se relacionan entre sí de forma bilateral, para almacenar toda la información generada y gestionar su posterior uso.

5.4.2 Clientes.

Los clientes son los usuarios. Se conectan al servidor a través de una aplicación propia del sistema instalada en sus ordenadores. Se puede acceder al sistema mediante acceso web o por red local

5.4.3 Hardware.

Aquí se distinguen dos hardwares: el del servidor y el de los clientes. Las características de hardware del servidor dependen de las dimensiones requeridas por el uso que se le vaya a dar, dependiendo del volumen de datos a manejar y del número de usuarios que vayan a tener acceso a ellos. Por su parte, los usuarios no requieren más que ordenadores personales.

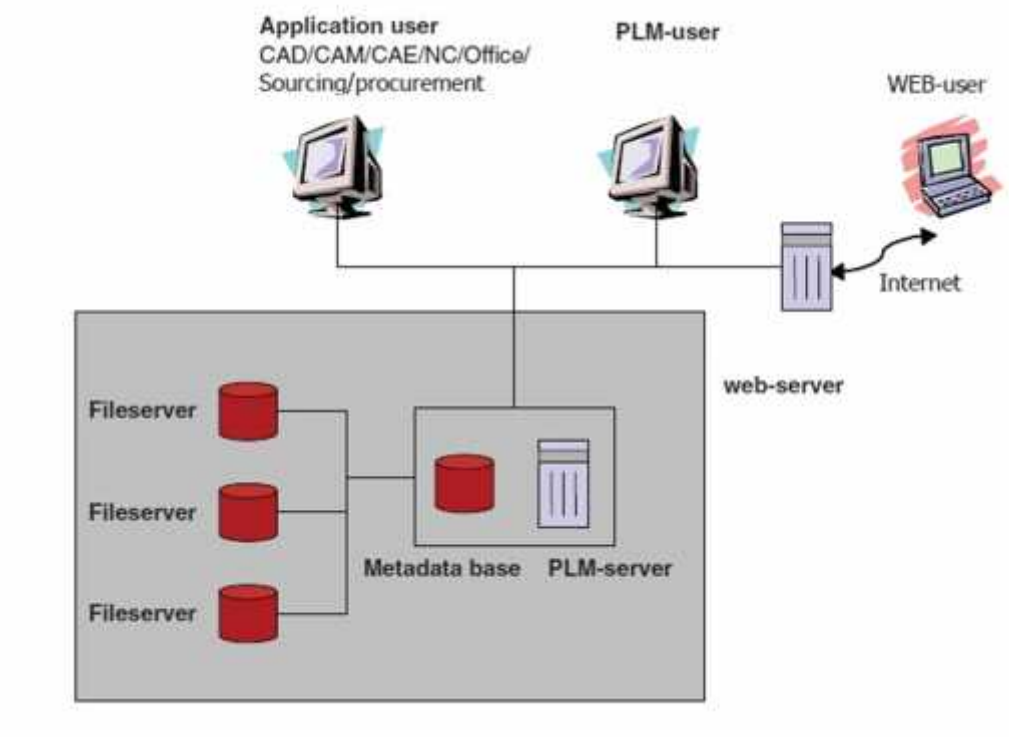


Ilustración 5-3: Estructura básica de un sistema PLM.

5.5 Aras Innovator.

El sistema PLM que hemos empleado para la realización de este proyecto es Aras Innovator. Se trata de un sistema PLM open source, de la compañía ARAS CORPORATION, de origen estadounidense y que ofrece servicios de integración de soluciones PLM y PDM en una sola plataforma.

Es una aplicación Open Source, lo que hace que el código sea abierto, peromitiendo así que el software pueda ser distribuido y modificado libremente sin restricciones de licencia. Esto permite a las empresas adaptar el software a sus necesidades mediante su customización.

Por otra parte, la compañía ARAS CORPORATION ofrece la posibilidad de poder comprar paquetes de licencias con módulos desarrollados por ella.

Para lograr esta flexibilidad, Aras está basado en una arquitectura orientada al servicio, SOA (Service-Oriented Architecture). Se trata de un marco de trabajo conceptual que establece una estructura de diseño para la integración de aplicaciones que permite a las organizaciones unir los objetivos de negocio, en cuanto a flexibilidad de integración, con los sistemas legados y la alineación directa a los procesos de negocio con la infraestructura de las tecnologías de la información de la empresa.

En la Ilustraión 5-4 se describen distintas funcionalidades de Aras asociadas a las fases de concepto, desarrollo y producción, así como su relación con otras aplicaciones.

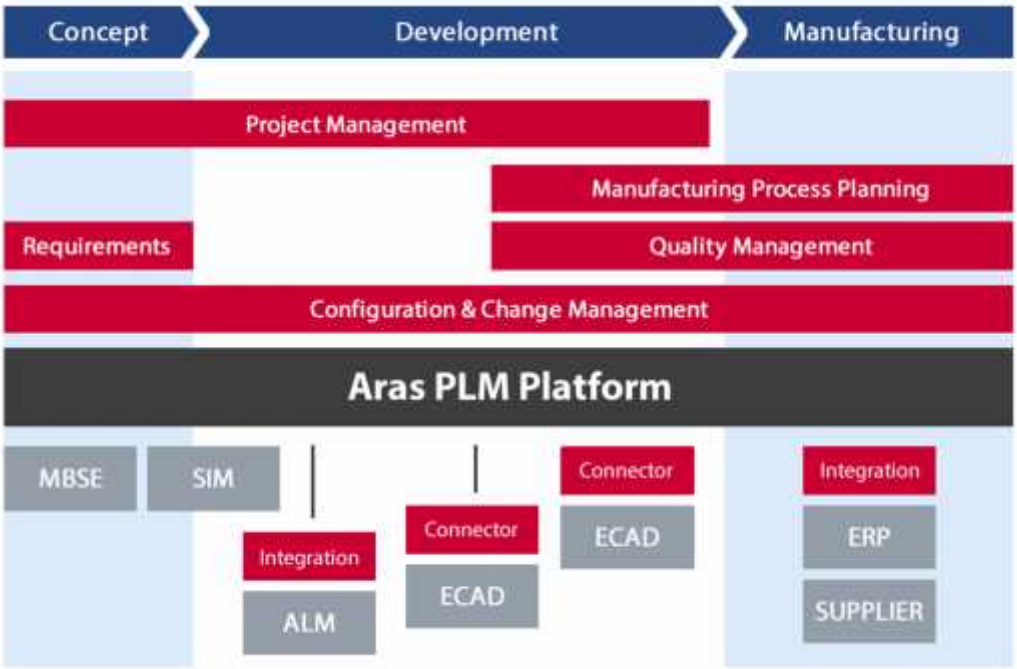


Ilustración 5-4: Relación de Aras Innovator con las fases del ciclo de vida y su integración con otras aplicaciones.

Las soluciones Aras están divididas en tres grandes bloques:

- **PLM:** para gestionar de manera colaborativa los procesos a través del ciclo de vida.
- **PDM:** para gestionar los datos de CAD (Computer Aided Design) en referencia al producto, las partes y la documentación asociada.
- **PLATAFORMA:** para desarrollar aplicaciones adaptadas a las necesidades de la organización.

El sistema está compuesto por los clientes web, un servidor de aplicación, una base de datos SQL server y un servidor de archivos, todo ello basado en protocolos estándar de Internet, incluyendo HTTP/HTTPS, XLM y SOAP (Simple Object Access Protocol).

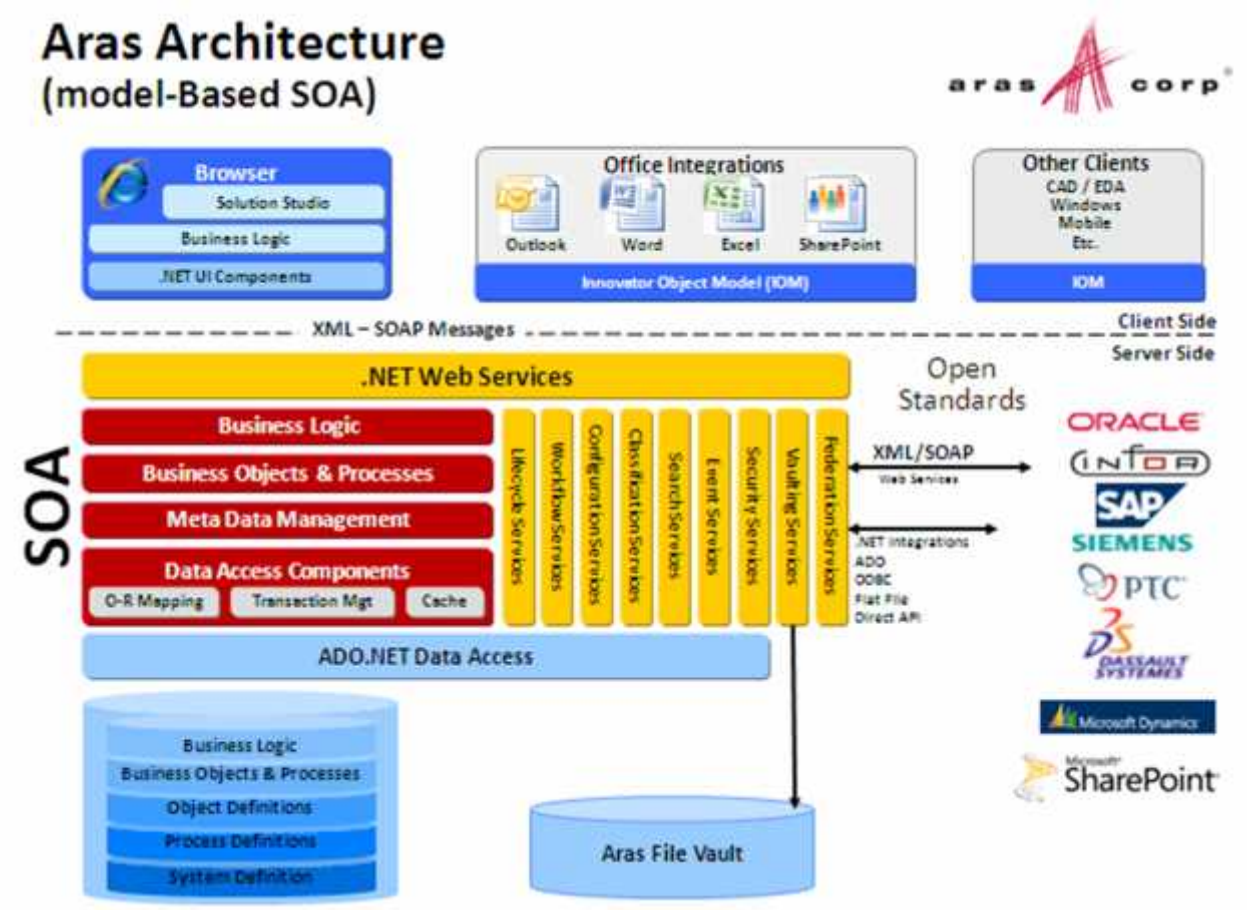


Ilustración 5-5: Arquitectura SOA de Aras Innovator.

6 IMPLEMENTACIÓN E INTEGRACIÓN EN UN ENTORNO PLM

El objetivo de nuestro proyecto es adaptar la herramienta Aras Innovator para incluir en ella la estructura PPR (Producto, Proceso, Recurso) de un helicóptero de Lego, desarrollar un módulo mediante código que permita la interoperabilidad entre Aras y un fichero de texto externo que recoja información a introducir en Aras en formato Express-I, y posteriormente desarrollar otra aplicación que obtenga información referente a los procesos (operaciones), como los tiempos de proceso y las suboperaciones, que en términos de equilibrado serían las precedencias del proceso en cuestión, para aplicar el algoritmo heurístico de equilibrado basado en la regla LCR (Largest Candidate Rule).

Ilustración 6-1: Helicóptero de Lego objeto del Proyecto.

6.1 Casos de uso.

En el diagrama de casos de uso representado en la Ilustración 6-2 están representados los actores que interactúan con el sistema y los módulos desarrollados en Aras.

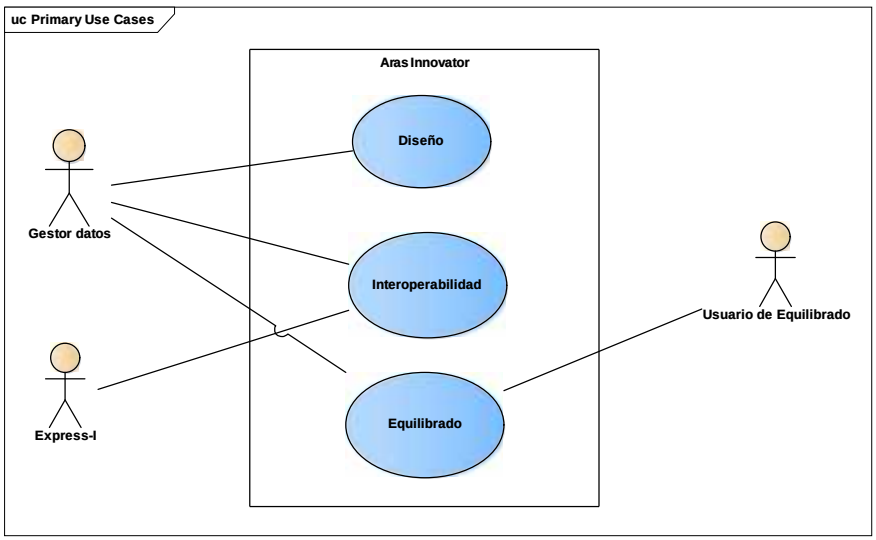


Ilustración 6-2: Diagrama de casos de uso.

6.1.1 Actores.

- **Gestor de datos**: Es el encargado de diseñar la estructura interna necasaria para implementar la estructura PPR en Aras, creando los ItemTypes asociados a las operaciones, las estaciones y los reursos, definiendo sus atributos, así como sus formularios asociados para intrducir los datos, las relaciones entre los elementos, la estructura de los usuarios asociados al proyecto y sus privilegios de acceso a la información. También es el agente encargado de desarrollar los módulos de interoperabilidad y equilibrado.
- **Express-I:** Es un archivo de texto que recoge información referente a las operaciones, estaciones, partes y recursos, así como sus relaciones. Será procesado por el módulo de interoperabilidad para introducir la información en Aras.
- **Usuario de equilibrado:** Usuario que accede al sistema solo para consultar los resultados de la ejecución del módulo de equilibrado.

6.1.2 Módulos.

El trabajo desarrollado con Aras consta de tres etapas, listadas a continuación.

Serán desarrollados en detalle más adelante, con sus diagramas de clase correspondientes.

- **Diseño:** Para poder implementar el algoritmo de equilibrado objetivo de nuestro proyecto necesitaremos trabajar con la información asociada a nuestro prototipo de helicóptero de Lego, y para introducir dichos datos previamente tenemos que diseñar la estructura interna de datos, que responde a la pregunta de qué elementos necesitaremos y cómo se relacionan éstos entre sí. Así pues, el módulo de diseño recoge la estructura interna implementada en Aras. Está compuesto por los ItemTypes asociados a las partes, las operaciones, las estaciones y los recursos, así como sus atributos y los ItemTypes que las relacionan entre sí.
- **Interoperabilidad:** Desarrollaremos un programa que nos facilite la introducción de datos en Aras. El módulo de interoperabilidad nos permitirá leer los datos recogidos en un fichero externo con formato Express-I y la almacenarlos de forma estructurada en Aras.
- **Equilibrado:** Módulo que implementa el algoritmo heurístico de equilibrado basado en la regla LCR (Largest Candidate Rule), devuelve por pantalla las estaciones asociadas a cada operación y asocia en aras las estaciones resultantes a las operaciones implicadas en el balanceo.

6.2 Diagramas de Clases.

6.2.1 Diseño.

6.2.1.1 Estructura del diseño implementado.

Para integrar nuestra estructura PPR en Aras tenemos que acondicionar la herramienta para los tipos de datos que vamos a manejar. Nuestro helicóptero está compuesto por una serie de componentes, que se agrupan entre sí para formar componentes agregados pasando por etapas. A dichos componentes los llamaremos **partes**. Cada vez que agrupamos un conjunto de partes formamos una parte resultante, y este proceso se enmarca dentro de las **operaciones**. A su vez, las operaciones son realizadas en **estaciones**, y requieren de ciertos **recursos** para llevarse a cabo. Un **plan de proceso** tiene asignada una operación, que sería la última operación realizada en el proceso (ésta operación contendrá como suboperaciones el resto de operaciones necesarias) y un modelo resultado, que será el producto final. Veremos en esta sección como implementar todo esto en Aras.

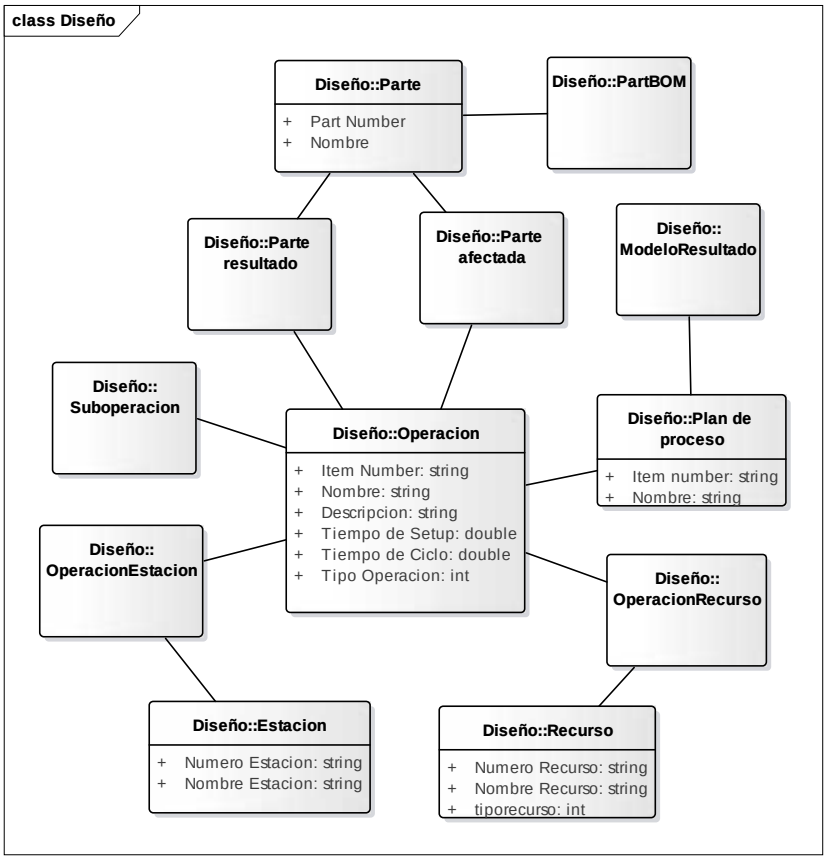


Ilustración 6-3: Diagrama de clase asociado al Diseño.

6.2.1.2 Partes y PartBOM.

Por un lado, las partes se agrupan de forma jerárquica conformando el BOM, que quedará estructurado en Aras. Debe existir, pues, un ItemType de tipo Parte. Éste ya está definido por defecto en Aras. A su vez, debe existir un tipo de ItemType que relacione las partes entre sí para formar la estructura BOM. También existe por defecto este ItemType en Aras. En la Ilustración 6-4 puede observarse, acudiendo a la categoría Administration, elemtento ItemTypes (1) y realizando la búsqueda "part" (2), que ya están definidos los ItemTypes Part (3) y PartBOM (4). Además se observa que el ItemType PartBOM tiene marcada la pestaña "Relationship" (5).

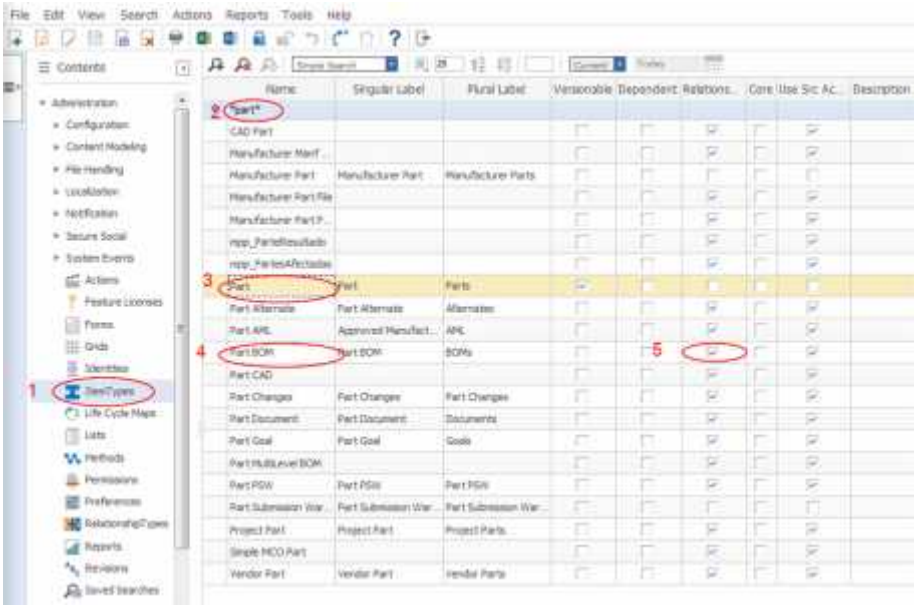


Ilustración 6-4: ItemTypes Part y PartBOM

Dado que el ItemType PartBOM es de tipo relacional, si acudimos al apartado de RelationshipTypes, ubicado en la categoría Administration, encontraremos allí el ItemType PartBOM, como se muestra en la Ilustración 6-5. En esta ventana se puede observar qué elementos relaciona el ItemType PartBOM. En este caso el Source ItemType sería lo que llamaremos el "padre", que en este caso es una parte, y el Related ItemType sería lo que llamaremos el tipo de ítem "hijo". También es una parte en este caso.

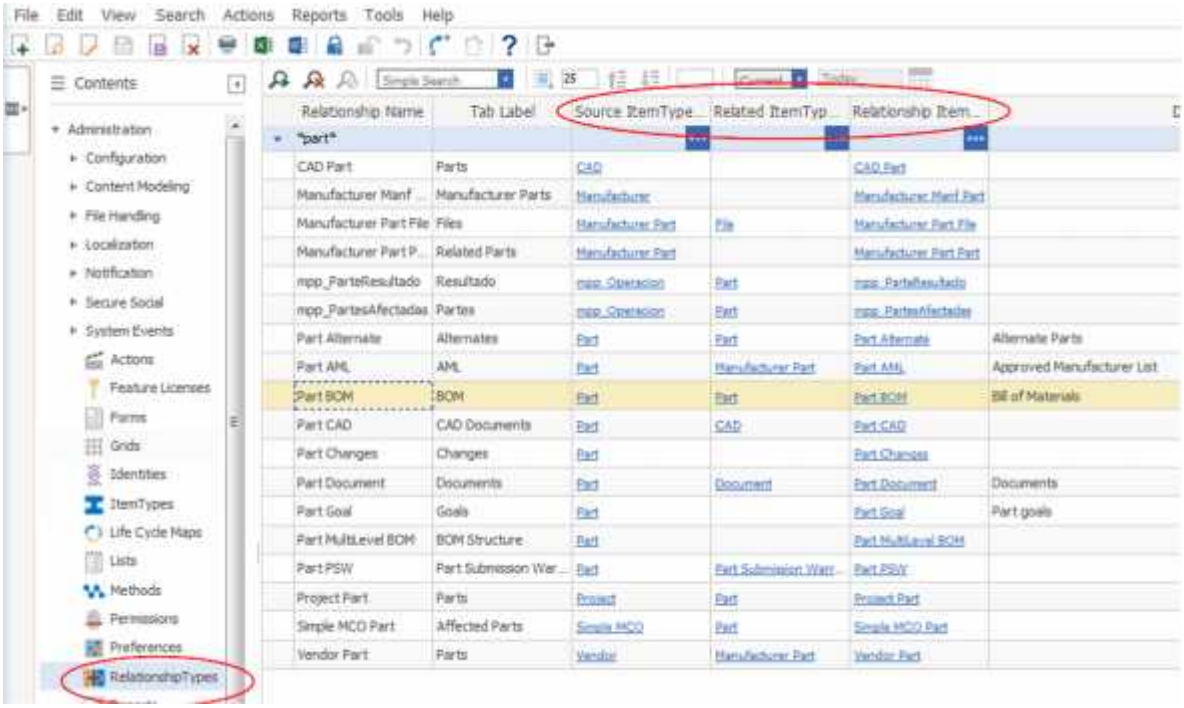


Ilustración 6-5: Ubicación del ItemType PartBOM.

Si desplegamos el elemento PartBOM accederemos a un formulario donde se recogen datos esenciales asociados a este ItemType de tipo relacional (Véase la Ilustración 6-6), como son (1) su nombre y su etiqueta, que será el nombre con el que será representado cuando aparezca en los elementos relacionados de las partes (Véase el resalto número 3 en la Ilustración 6-7), los elementos que relaciona (2), y la relación de cardinalidad entre estos elementos relacionados (3).

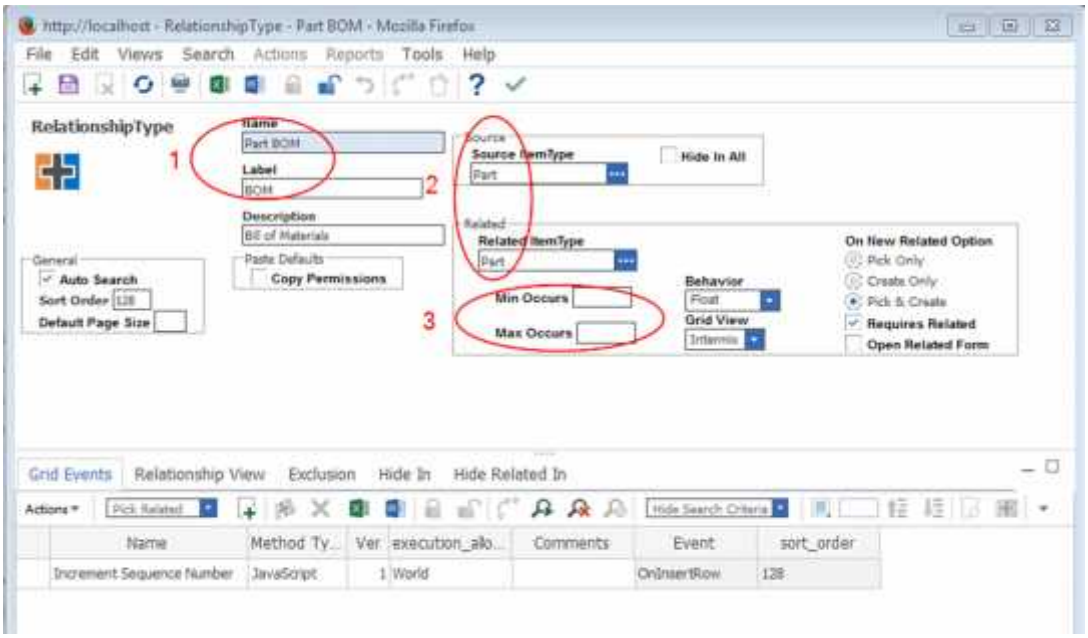


Ilustración 6-6: Formulario asociado a los ItemType de tipo relacional.

Como se muestra en la Ilustración 6-7, accediendo a los elementos de tipo Parts, ubicados en la sección Design (1), podemos añadir nuevas partes (2). No todos los campos a rellenar son obligatorios, y éstos se pueden modificar según nuestras necesidades como se explicará más adelante. Como ya se dijo con anterioridad, este tipo de ítem ya venía definido por Aras. A cada parte se le puede asociar un número de parte, un nombre, un tipo de parte, su creador, etc. Además se le pueden asignar documentos de tipo CAD o documentos de ofimática. A cada parte se le pueden asociar las subpartes que la componen accediendo a la pestaña BOM (3) y clicando en el botón de añadir (4). Esto abrirá un cuadro de diálogo que nos permitirá realizar la búsqueda de la subparte que queramos asociarle, la cual deberá estar añadida con anterioridad. Realizando esto con todos los elementos que componen nuestro helicóptero llegaremos a la estructura BOM que quedaría recogida en la pestaña BOM Structure. Esto lo veremos con más adelante en el apartado dedicado a la estructura BOM.

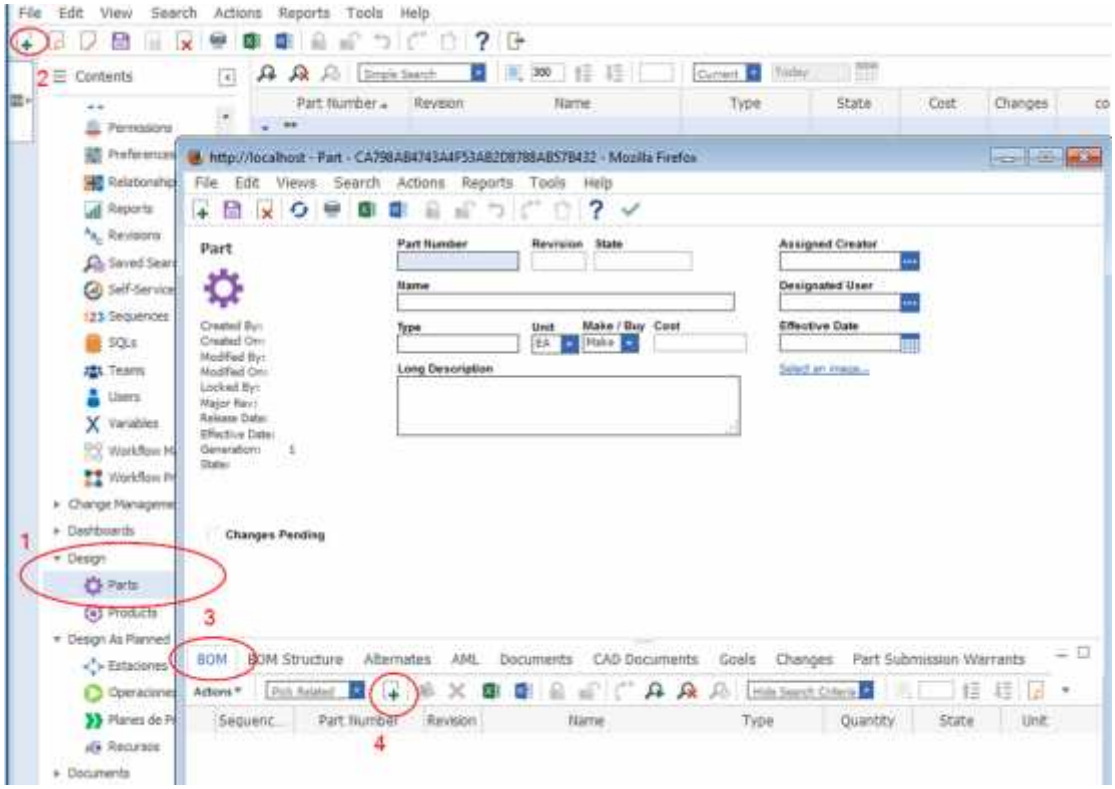


Ilustración 6-7: Ventana asociada a las partes.

6.2.1.3 Operaciones.

Por otro lado, las partes son procesadas por las operaciones, por lo que tendremos que crear este ItemType, que lo llamaremos **mpp_Operacion.** Para crear un nuevo ItemType nos dirigimos a la barra de contenidos, desplegamos la categoría de administrador y en la categoría ItemTypes (1, Ilustración 6-8) le damos al botón de añadir (2). Rellenamos los campos de nombre y etiquetas (3) y le asignamos las propiedades (4), que vendrían a ser el equivalente a los **atributos** para una clase en la programación orientada a objetos. Una vez definidas las propiedades tendremos que diseñar el **formulario** asociado a las operaciones, mediante el cual introduciremos los datos cuando queramos añadir una nueva operación. Este formulario se crea a parte y, una vez creado, se le asocia al ItemType mpp_Operacion mediante la pestaña de Views (6). También veremos como incluir **relaciones** entre las operaciones y las partes, estaciones, recursos y suboperaciones, que se hará creando nuevos ItemTypes de tipo relacional e incluyéndolos en la pestaña RelationshipTypes (5). Por último, tenemos que configurar también el **acceso** a este ItemType, para determinar qué usuarios pueden acceder a esta información, si solo pueden consultarla o también pueden añadir operaciones, y desde qué categoría accederán al ítem. Esto se hace desde las pestañas TOC Access, Can Add y Permissions.

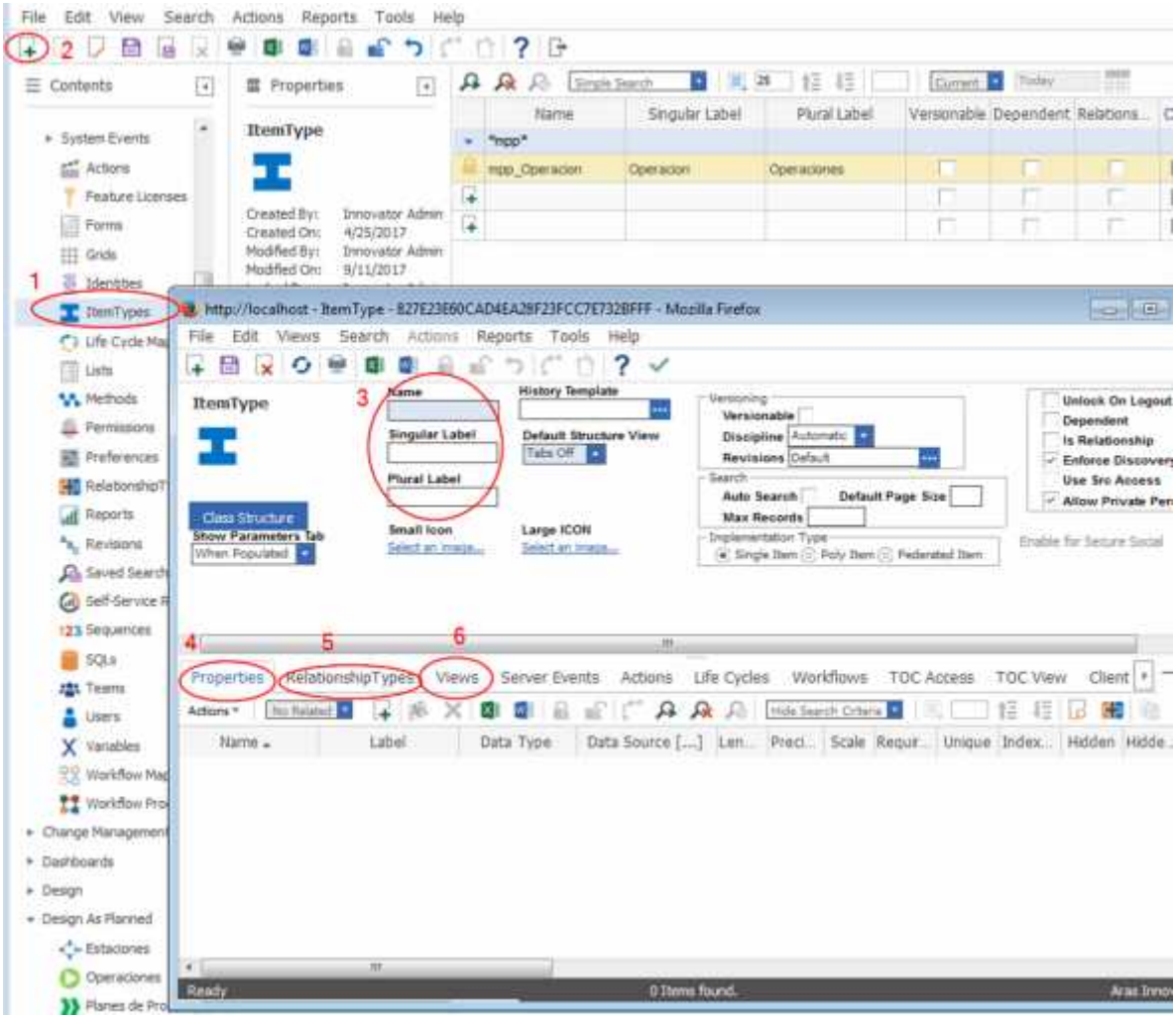


Ilustración 6-8: Creación de un nuevo ItemType.

6.2.1.3.1 Atributos.

Para una operación los atributos que le asignaremos serán los de número de ítem, nombre, descripción, tiempo de setup, tiempo de ciclo y tipo de operación. Los atributos de número de ítem, nombre y descripción serán de tipo string, los de tiempo de ciclo y tiempo de setup serán de tipo double y el de tipo de operación será de tipo lista.

Esta lista la podremos definir nosotros (Ilustración 6-9), y en ella incluiremos los tipos de proceso que se pueden dar en una operación, como podrían ser: provide, transporte, ensamblado, transformación u otros

definidos por el usuario.

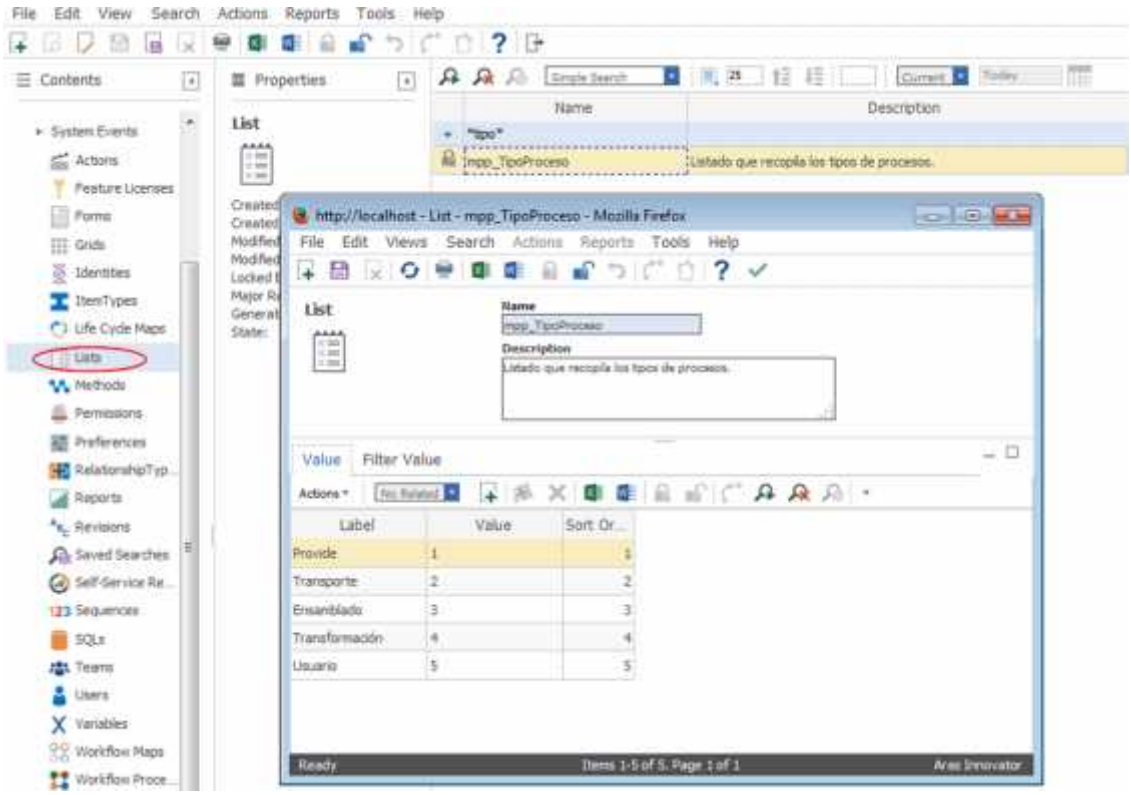


Ilustración 6-9: Definición de elemento tipo Lista.

Una vez creada la lista ya podemos incluir el atributo “tipo de operación” a nuestro ItemType mpp_Operacion en el apartado de Properties.

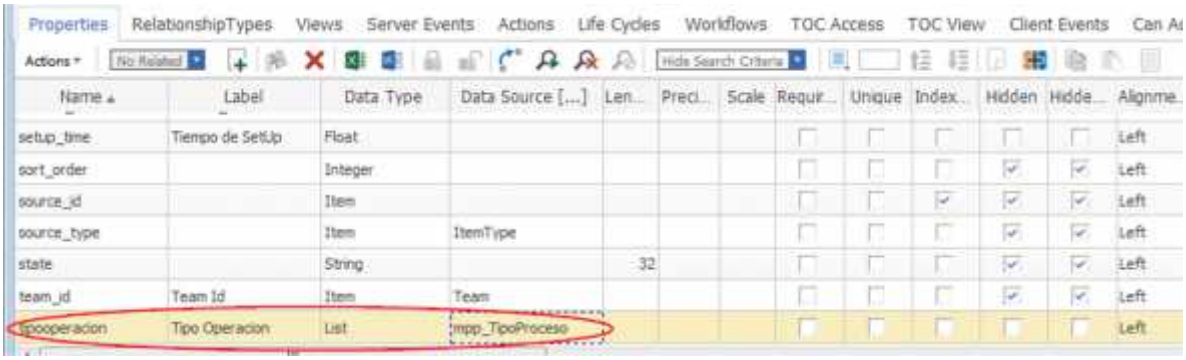


Ilustración 6-10: Inclusión del atributo Tipo de Operación, de tipo List.

6.2.1.3.2 Relaciones.

Cada operación procesa partes, lo que llamaremos como **partes afectadas**, y como resultado de cada operación tendremos una **parte resultante.** Deberemos crear, pues, dos ItemTypes que relacionen las partes con las operaciones. Estos son el ItemType **mpp_PartesAfectadas** y el ItemType **mpp_ParteResultado.**

Las relaciones de cardinalidad entre los ItemType de las partes con los ItemType de las operaciones son de tipo (n…1), de forma que a una operación le pueden corresponder varias partes afectadas, y la relación de cardinalidad entre los ItemType mpp_Operacion y Part son de tipo (1…1), pues a cada operación le corresponde solo una parte resultado.

Las operaciones precisan de recursos y son realizadas en estaciones, por lo que también tendremos que crear los ItemTypes **mpp_Estacion** y **mpp_Recurso6745**, así como los ItemType de tipo relacional **mpp_OpeEstacion** y **mpp_Recurso6745NI.**

Por otra parte, las operaciones tienen operaciones previas a realizar, por lo que también existirá un tipo de ítem

que relacione operaciones con operaciones: mpp_SubOperaciones.

Una vez creados estos ItemTypes de tipo relacional los añadimos al ItemType mpp_Operacion en la pestaña de RelationshipTypes.

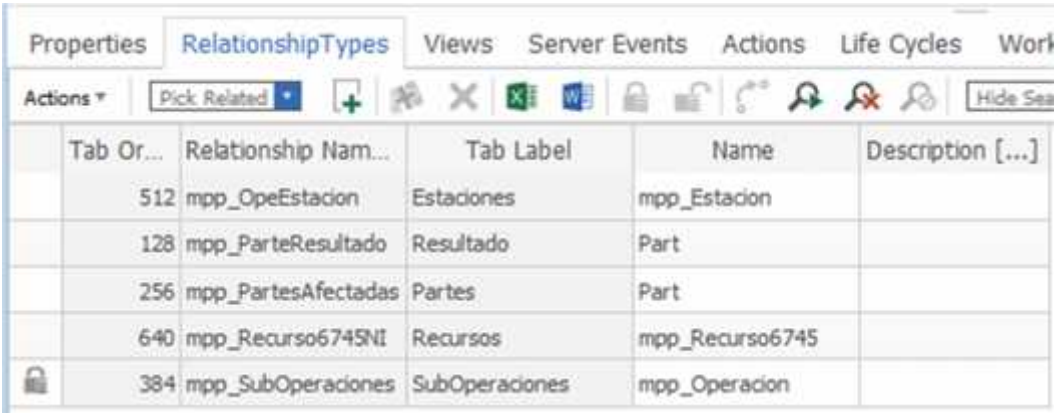


Ilustración 6-11: Inclusión de las relaciones entre Items asociadas a las operaciones.

Esto añadirá pestañas al formulario de las operaciones desde las que podremos incluir los elementos relacionados con cada operación que precisemos incluir en nuestro modelo.

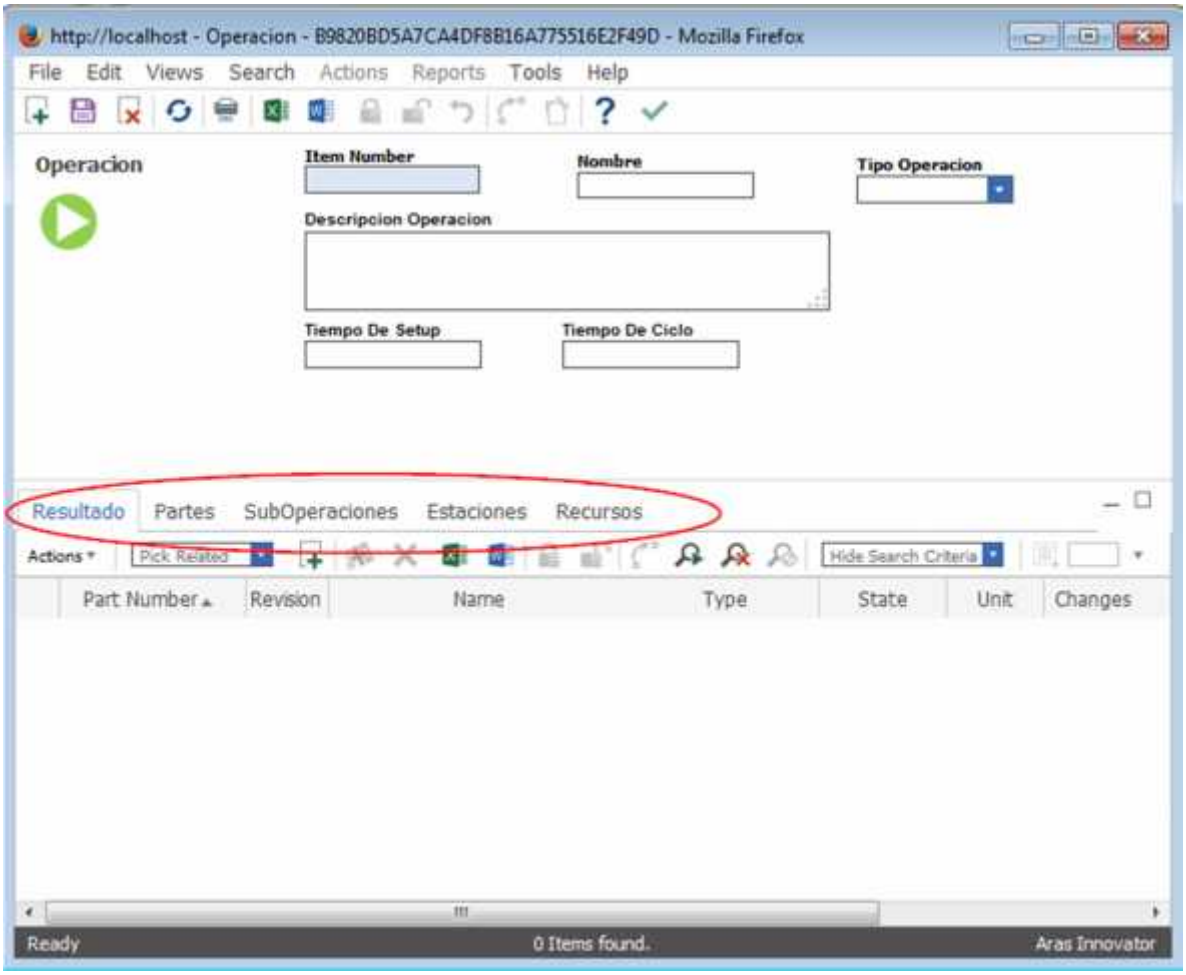


Ilustración 6-12: Vista de la ventana para añadir operaciones a Aras.

6.2.1.3.3 Formulario.

Desde Aras es posible diseñar el formulario que emplearemos para añadir operaciones (lo mismo para las estaciones, los recursos y las partes). Para hacerlo primero tendremos que crear un formulario. Esto se hace desde el elemento Forms, en la categoría Administration.

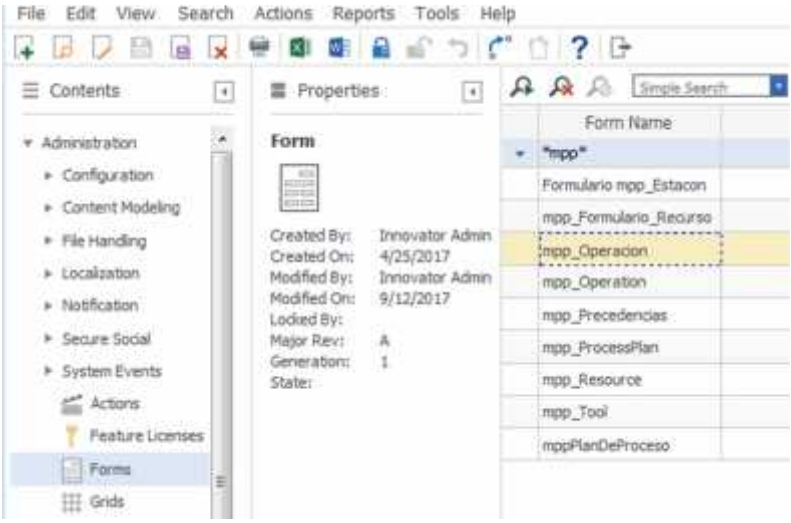


Ilustración 6-13: Añadir Formulario.

Una vez creado el formulario, lo asociamos al ItemType mpp_Operacion desde la pestaña Views.

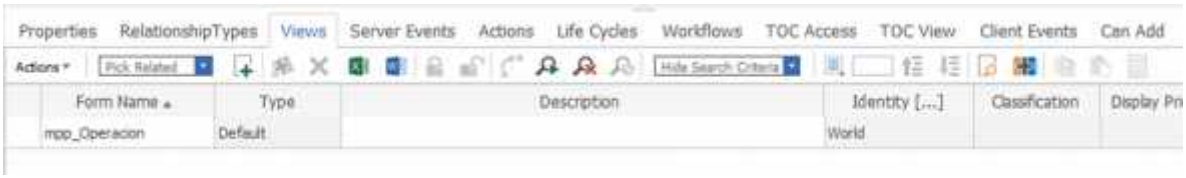


Ilustración 6-14: Pestaña Views del ItemType mpp_Operacion.

Ahora ya podemos configurar nuestro formulario. Abriendo el elemento de tipo Form nombrado como mpp_Operacion (Ilustracion 6-13) podemos acceder a la customización de nuestro formulario, añadiendo las propiedades que queramos que se vean incluidas y ubicándolas a nuestro gusto.

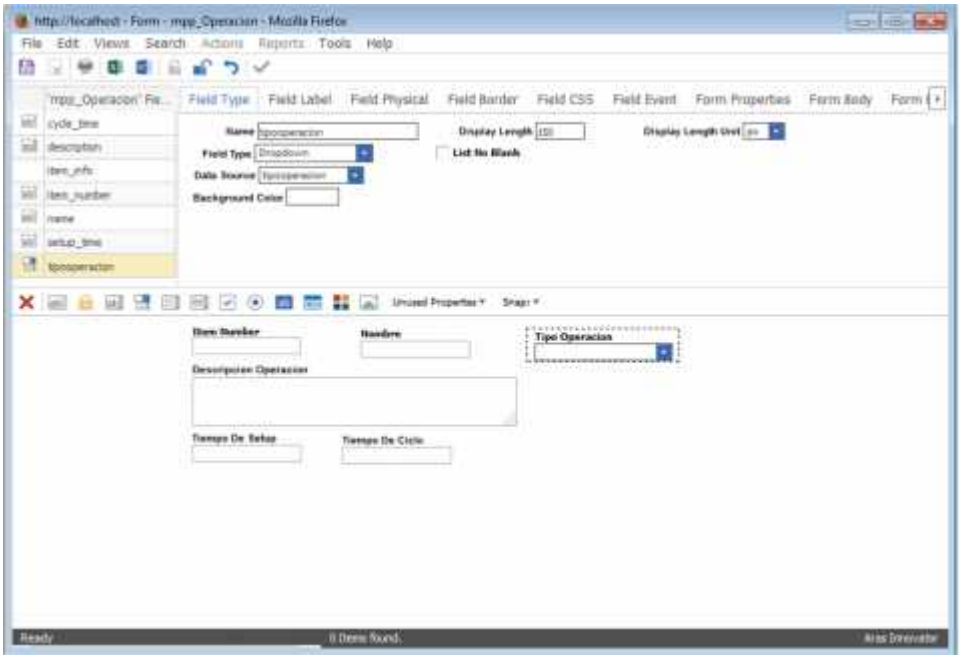


Ilustración 6-15: Ventana de diseño de formularios.

6.2.1.3.4 Acceso.

Por último nos quedaría por definir el acceso al Item Operación. Para que nuestro recién creado ítem sea accesible primero tendremos que ubicarlo en alguna categoría. Estas categorías se pueden modificiar,

añadiendo las que sean necesarias. Nosotros hemos creado la categoría Design as Planned, donde incluiremos las operaciones, los recursos, las estaciones y los planes de proceso.

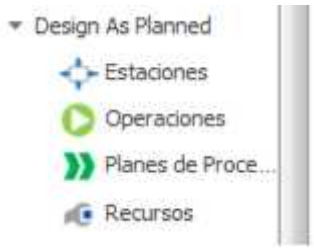


Ilustración 6-16: Categoría Design As Planned, creada por nosotros.

Para crear una nueva categoría basta con añadir un nuevo elemento a la lista "Categories", que se puede encontrar en los elementos de tipo lista.

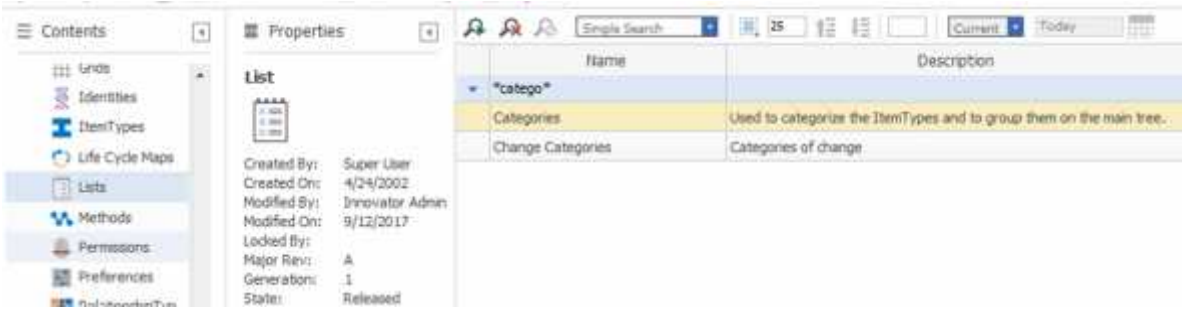


Ilustración 6-17: Ubicación de la lista de Categorías.

Una vez añadida nuestra nueva categoría a esta lista, pasamos a la pestaña TOC Access de nuestro ItemType mpp_Operacion. Aquí podremos añadir la **identidad** que tendrá acceso al ítem y asignarle la categoría donde éste estará ubicado.

Ilustración 6-18: Pestaña TOC Access del ítem mpp_Operacion.

Una identidad es una agrupación de **usuarios**, a las cuales se les pueden asignar **permisos** como acceder a la información, añadir nuevos elementos o realizar modificaciones.

Los **usuarios** se crean desde la categoría Administration, en el elemento Users. Cada usuario queda registrado y podrá acceder desde su propio equipo al sistema Aras y realizar sus aportaciones. A modo de ejemplo hemos creado los usuarios Responsable COLA, Responsable FUSELAJE y Responsable ROTOR. Cada uno de estos usuarios es el responsable del hipotético equipo de diseño o fabricación del conjunto asignado. Por ejemplo, hemos creado tres miembros del equipo COLA, lamados MEC1, MEC2 y MEC3. De esta forma es como se define en aras la OBS (Organitational Breakdown Structure).

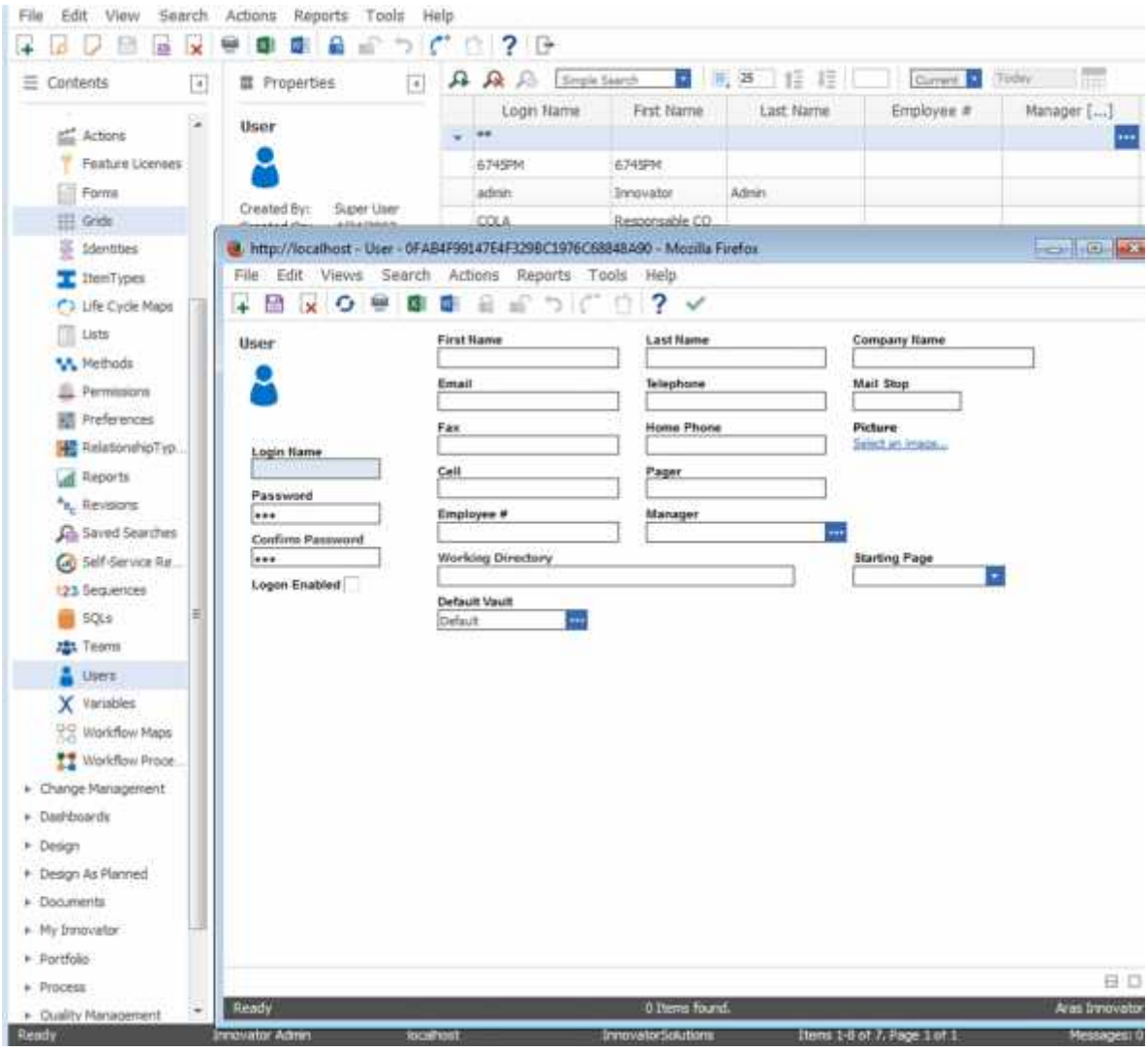


Ilustración 6-19: Añadir nuevo Usuario.

A la hora de crear un usuario se le debe asignar un Login Name y una contraseña, y habilitar la opción Logon Enabled. De esta forma podrán acceder a Aras con su propio usuario y contraseña.

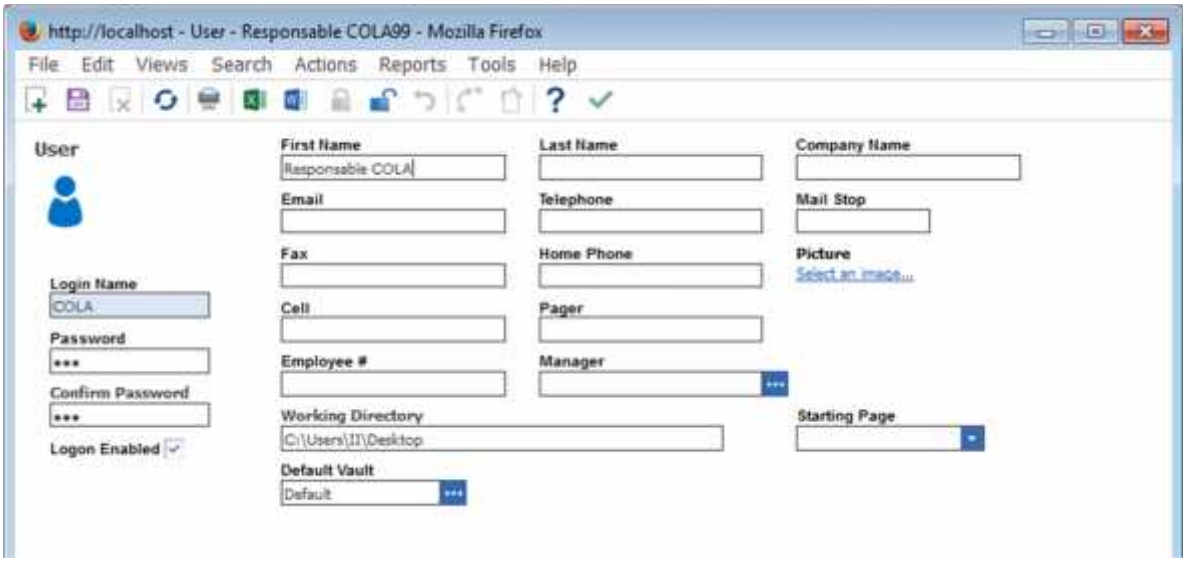


Ilustración 6-20: Ejemplo de Usuario.

Ilustración 6-21: Acceso a Aras del Usuario “Responsable COLA”.

A su vez, los usuarios son agrupados en identidades, a las cuales se les pueden asignar determinados permisos.

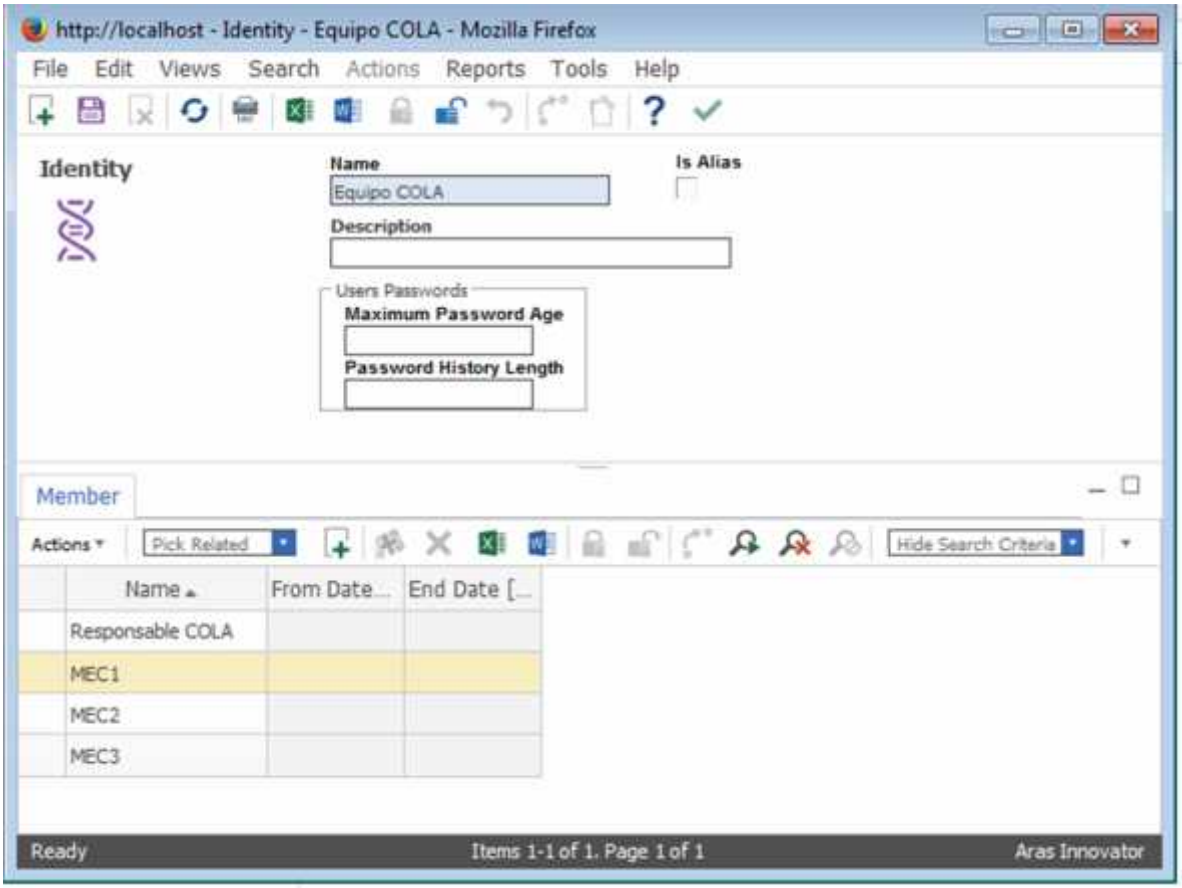


Ilustración 6-22: Entidad que agrupa a los miembros del Equipo COLA.

Ilustración 6-23: Permisos otorgados a la identidad Equipo COLA.

Volviendo a la ventana del Item mpp_Operacion y acudiendo a la pestaña Can Add seleccionaremos las identidades que podrán añadir elementos de tipo operación.

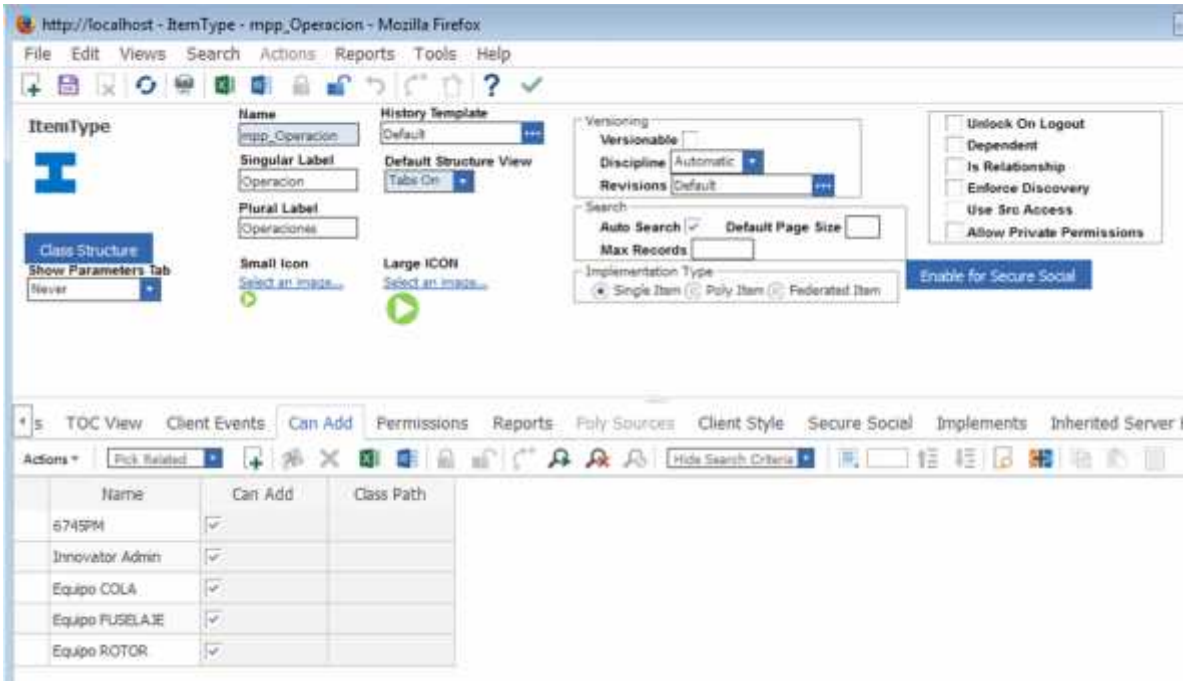


Ilustración 6-24: Configuración de identidades que podrán añadir nuevos elementos de tipo operación.

Para que efectivamente estas entidades puedan añadir nuevas operacioens tendremos que añadir los permisos pertinentes en la pestaña Permissions.

6.2.1.4 Estaciones, Recursos y Planes de proceso.

Para crear las estaciones, los recursos y los planes de proceso seguiremos el mismo procedimiento descrito para las operaciones. El diagrama de clases de la Ilustración 6-3 recoge los ItemTypes que necesitamos crear y los atributos que tenemos que asignarles.

6.2.2 Interoperabilidad.

Una vez creada la estructura de datos que usaremos para implementar la maqueta digital pasamos a desarrollar un programa mediante código que permita introducir datos desde un fichero externo.

6.2.2.1 Desarrollo de métodos en Aras.

Para desarrollar código en Aras accederemos a la pestaña Methods ubicada en la categoría Administration (1). Una vez ahí creamos un nuevo método (2). Los métodos se pueden desarrollar orientados al servidor u orientados al cliente. Los métodos orientados al cliente se desarrollan en JavaScript, y los orientados al servidor se pueden desarrollar tanto en C# como en VisualBasic. Elegimos la opción ServerSide y C# (3).

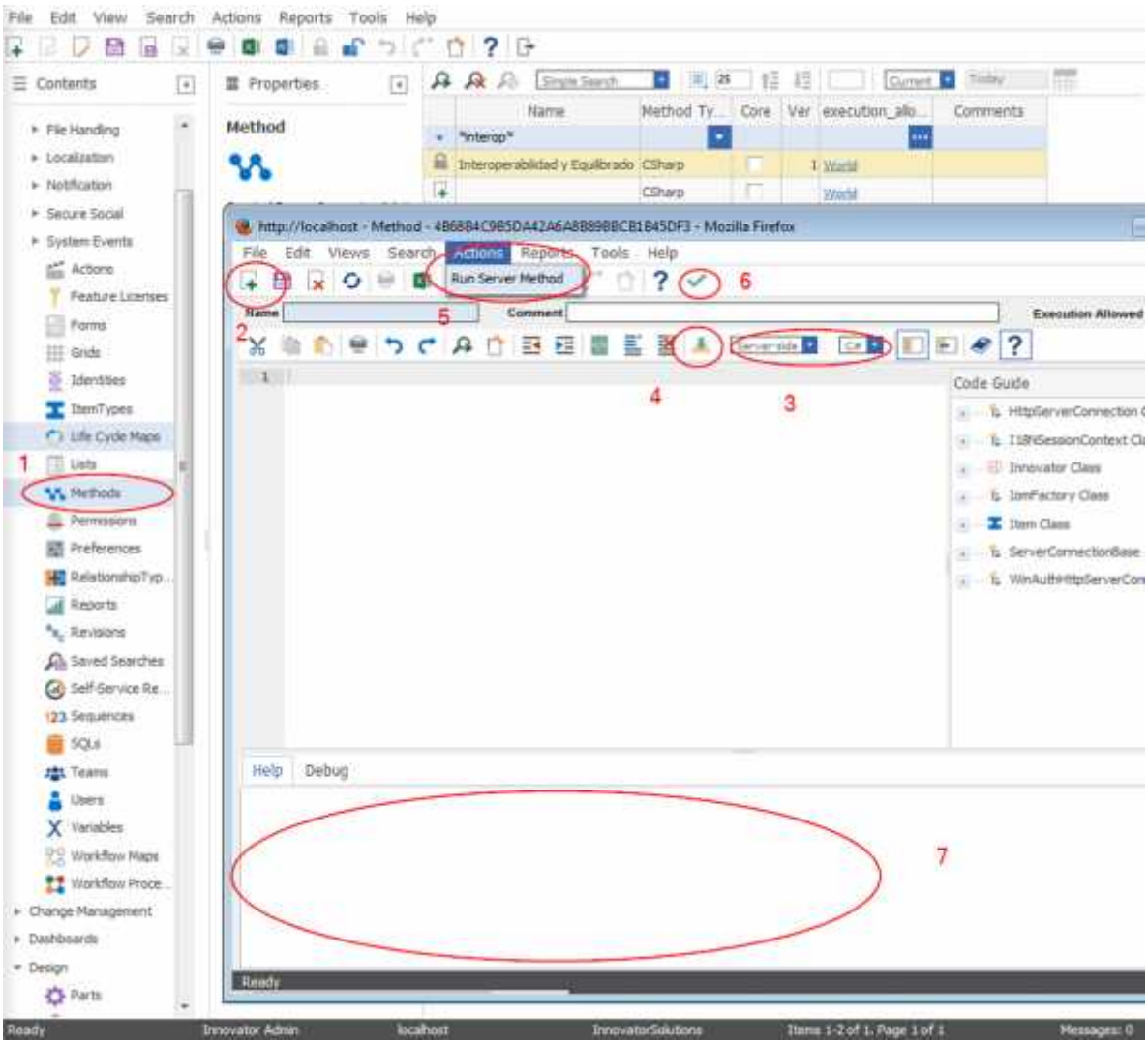


Ilustración 6-25: Acceso a la interfaz de desarrollo de programas en Aras.

Para revisar el código empleamos el botón (4). En caso de que exista algún tipo de error, éste quedará reflejado en (7), donde se notificará la línea donde se encuentra el error y una descripción del tipo de error. Una vez que hayamos chequeado la sintaxis del código y esté todo correcto pasamos a ejecutarlo (5). Para cerrar la ventana de métodos debemos clicar en (6).

6.2.2.2 Formato EXPRESS-I.

La información estará estructurada conforme al formato EXPRESS-I. Cada línea debe seguir una estructura dependiendo del tipo de dato que queramos introducir. Si se trata de una **parte**, consideraremos su numero de parte y su nombre, y la línea en formato EXPRESS-I quedaría así:

#NumeroLinea=Part(NumeroParte, “NombreParte”);

El número de línea se emplea para identificar la entidad dentro del propio fichero. A continuación se escribe el tipo de entidad al que se refiere la línea y entre paréntesis los atributos que le corresponden a ésta.

Tras procesar esta línea tendremos que haber creado an Aras una nueva parte a la que le asignaremos los atributos NumeroParte y NombreParte.

Agregarle una **subparte** a la parte equivale a crear un nuevo PartBOM. Emplearemos la siguiente estructura:

#NumeroLinea=PartBOM(#NumeroLineaPadre, #NumeroLineaHijo);

Entre paréntesis indicamos los números de las partes que relaciona el ItemType relacional PartBOM.

Para agregar una estación o un recurso emplearemos el siguiente tipo de línea:

#NumeroLinea=mpp_Estacion(NumeroEstacion, “NombreEstacion”);
#NumeroLinea=mpp_Recurso6745(NumeroRecurso, “NombreRecurso”);

De forma análoga, para las **operaciones y sus relaciones** emplearemos la siguiente estructura:

#NumeroLinea=mpp_Operacion(NumeroOperacion, “NombreOperacion”, TiempoCiclo);
#NumeroLinea=mpp_OpeEstacion(#NumeroLineaPadre, #NumeroLineaHijo);
#NumeroLinea=mpp_ParteResultado(#NumeroLineaPadre, #NumeroLineaHijo);
#NumeroLinea=mpp_PartesAfectadas(#NumeroLineaPadre, #NumeroLineaHijo);
#NumeroLinea=mpp_Recurso6745NI(#NumeroLineaPadre, #NumeroLineaHijo);
#NumeroLinea=mpp_SubOperaciones(#NumeroLineaPadre, #NumeroLineaHijo);

6.2.2.3 Estructura del código.

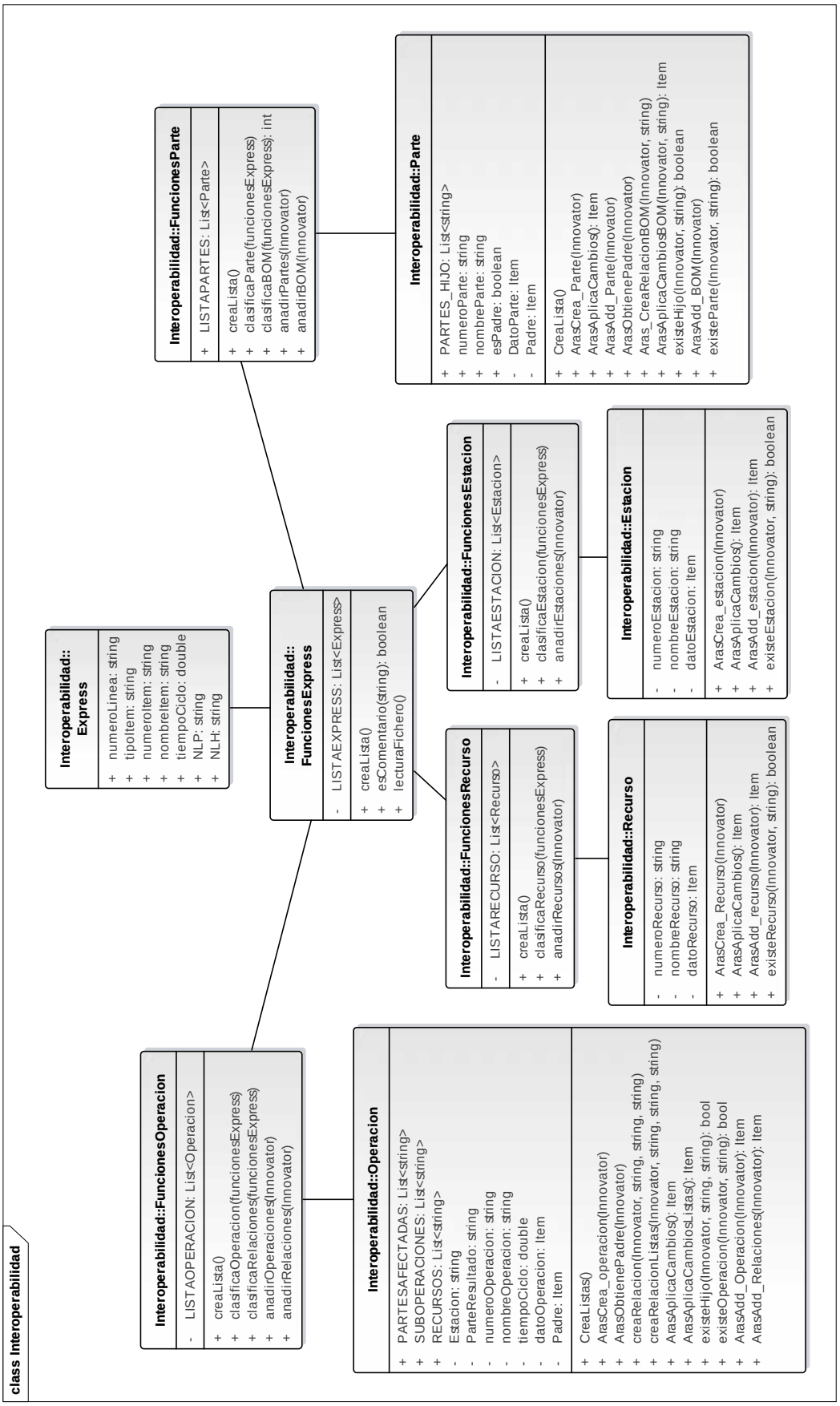


Ilustración 6-26: Diagrama de clase del código implementado para la interoperabilidad.

Para operar con los datos necesitamos diferentes clases. Las relaciones entre las clases, sus atributos y sus

funciones vienen recogidas por el diagrama de clases mostrado en la página anterior.

6.2.2.3.1 Clase Express.

Esta clase está destinada a almacenar los datos introducidos en el fichero de texto y leídos por la función **lecturaFichero()** perteneciente a la clase **FuncionesExpress.**

A cada línea del fichero de texto procesado le corresponderá un elemento de esta clase.

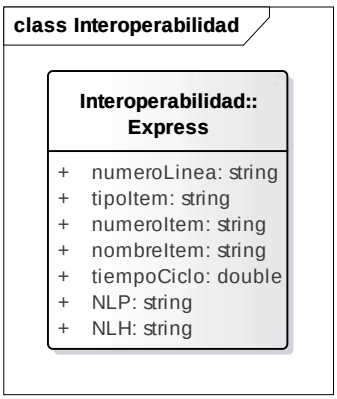


Ilustración 6-27: Clase Express.

Para proteger los datos de las clases hacemos uso de las funciones get/set:

```
public string numeroLinea;
public string getNumeroLinea () {return numeroLinea;}
public void setNumeroLinea (string parametro) {numeroLinea=parametro;}
```

6.2.2.3.2 Clase FuncionesExpress.

Esta clase contiene una lista de elementos tipo Express.

Para declarar en C# un **atributo de tipo lista** se requiere del siguiente código:

```
public List<Express> LISTAEXPRESS;
```

Primero definimos el tipo de acceso del atributo de la clase. Si no se pone nada, por defecto quedará definido como de acceso privado. A continuación escribimos "List" acompañado del tipo de dato de elementos de la lista "<Express>", el nombre de la lista "LISTAEXPRESS" y por último cerrar la instancia con ';'.

Las funciones contenidas en esta lista son:

1- **public void creaLista()**. Crea la LISTAEXPRESS declarada con anterioridad.

2- **public bool esComentario(string line).** Esta function recibirá por argumento una línea extraida del fichero Express. Si la línea comienza por '//' será catalogada como comentario y esta función devolverá 'true' allí donde sea llamada. La usaremos en la función de lectura de fichero, descrita a continuación, para ignorar las líneas que sean de tipo comentario.

3- **public void lecturaFichero ().** Recorre línea a línea todo el fichero Express y para cada línea crea un nuevo elemento de clase EXPRESS que, una vez rellenado con la información procedente del fichero Express y procesada mediante esta función, es añadido a la LISTAEXPRESS.

Para acceder a un fichero de texto empleamos esta línea de código:

```
System.IO.StreamReader file = new System.IO.StreamReader(@"\\VBOXSVR\aras\Express.txt");
```

Para obtener una línea del texto usamos la instrucción:

```
file.ReadLine()
```

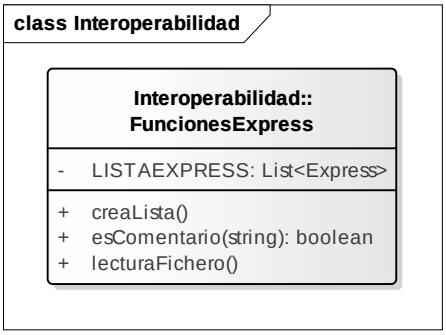


Ilustración 6-28: Clase FuncionesExpress.

6.2.2.3.3 Clase Operacion.

Esta clase contiene como atributos tres listas de tipo string, asociadas a las partes afectadas, las suboperaciones y los recursos, de forma que cada elemento de tipo Operación contenga los números de ítem de los elementos que cada operación lleva asocados. A parte tiene otros dos atributos de tipo string que marcan la relación entre cada operación con la parte resultado y con la estación. Estos elementos no son de tipo lista puesto que cada operación solo puede contener una parte resultado y estar asociada a una única estación.

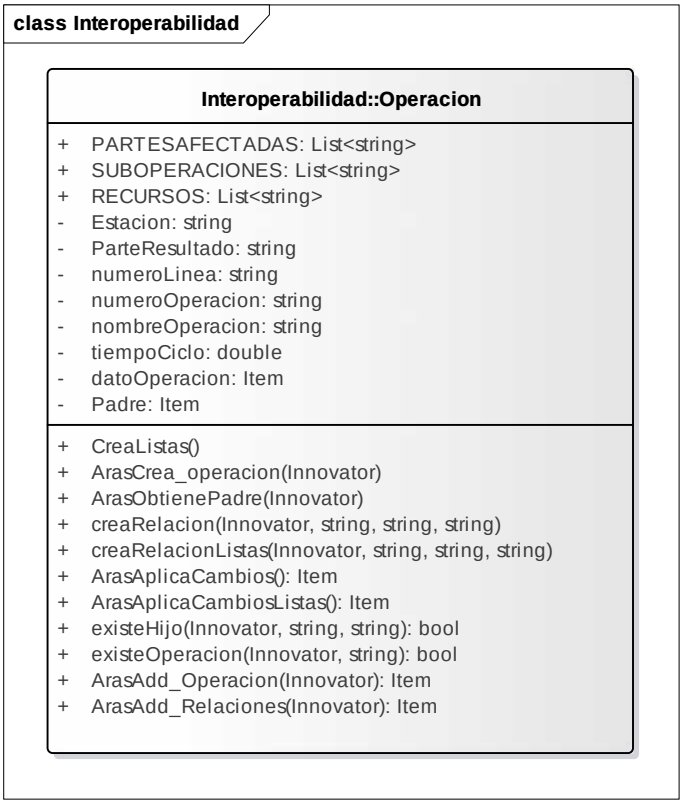


Ilustración 6-29: Clase Operacion.

Además tiene definidos los atributos que caracterizan a cada operación, como son su número, su nombre y el tiempo de ciclo.

A parte de lo anterior también necesitamos crear atributos de tipo Item que serán empleados por las funciones que realizan cambios y/o consultas sobre Aras. A estos atributos los llamaremos Padre y DatoOperacion.

Las funciones definidas en la clase Operacion son:

1- **public void CreaListas().** Esta función crea las listas declaradas como atributos. Cada vez que llamamos a esta función el contenido de las listas queda borrado.

2- **public void ArasCrea_operacion (Innovator innovator).** Crea una nueva operación en Aras con los datos contenidos en esta clase. Recibe como argumento un elemento de clase Innovator, que es una clase definida en Aras que tiene integradas funciones que sirven para relacionarse con Aras desde el código, permitiendo realizar funciones como consultar, crear o modificar elementos.

Para crear un nuevo Item en Aras empleamos el siguiente código:

```
public void ArasCrea_operacion (Innovator innovator)
  {
     DatoOperacion=innovator.newItem("mpp_Operacion","add");
     DatoOperacion.setProperty("item_number",getNumero());
     DatoOperacion.setProperty("name",getNombre());
     DatoOperacion.setProperty("cycle_time",Convert.ToString(getTiempoCiclo()));
  }
```

La primera línea crea un nuevo ItemType con el nombre pasado como primer argumento. El nombre que le dimos al ItemType de las operaciones cuando lo creamos fue "mpp_operacion". Como segundo argumento indicamos la acción a realizar sobre el ítem. Una vez hecho esto ya tenemos nuestro ítem asignado a la variable DatoOperacion, que es de tipo Item. Al ser de tipo Item, Aras nos permite editar sus propiedades. Como argumentos indicamos el nombre de la propiedad a editar y el valor a introducir.

3- **public void ArasObtienePadre(Innovator innovator).** Obtiene de Aras el Item cuyo itemNumber coincida con el almacenado en el atributo 'numeroOperacion' de esta misma clase Operación y lo almacena en el elemento 'Padre', de tipo Item.

4- **public static void creaRelacion(Innovator innovator, string nombreItemRelacion, string nombreItem, string itemNumber).** Esta función crea un nuevo Item de tipo relacional, cuyo nombre pasaremos por argumento. También le pasamos por argumento el tipo de ItemType que vamos a relacionar con la Operación, así como el identificador del elemento en concreto que queramos asociar a la operación. Como resultado esta función realiza una modificación sobre el atributo DatoOperacion, en concreto le añade un elemento relacionado, ya sea una estación, otra operación, una parte afectada, una parte resultado o un recurso.

   Por ejemplo, para asociarle una estación a una operación, primero hacemos uso de la función definida en 3, lo que obtendrá el Item de la operación que vamos a manipular y lo almacenará en 'Padre'. Posteriormente llamamos a esta función y le pasamos como argumentos el nombre del RelationshipType que relaciona las operaciones con las estaciones, en este caso 'mpp_OpeEstacion", el nombre del ItemType estación, 'mpp_Estacion' y por último el identificador (numeroEstacion) de la estación en particular que vamos a relacionar con la operación. Es muy importante tener en cuenta que la estación que le vamos a asociar a la operación ya debe existir en Aras, por lo que si tratamos de usar esta función antes de tener creadas las estaciones (o los elementos que vayamos a relacionar con la operación en cuestión) obtendremos un error que nos dirá que no existe el elemento al que estamos intentando acceder.

5- **public static void creaRelacionListas(Innovator innovator, string nombreItemRelacion, string nombreItem, string itemNumber).** Realiza la misma función que la función anterior con la única diferencia de que el elemento sobre el que recaen los cambios no es 'DatoOperacion' sino 'Padre'. El Item 'Padre' se obtiene en la función definida en el punto 3.

6- **public Item ArasAplicaCambios ().** Aplica los cambios realizados sobre el Item 'DatoOperacion', con lo que tras llamar a esta función se efectuarióan los cambios realizados sobre la operación que creamos empleando la función definida en el punto 2 de esta lista.

7- **public Item ArasAplicaCambiosListas ().** Exactamente igual que la anterior, solo que aplica los cambios sobre el Item 'Padre'.

8- **public bool existeOperacion(Innovator innovator, string Numero).** Realiza una consulta a Aras en busca de la operación cuyo numero de operación la pasamos como argumento, y devuelve 'true' en caso de que dicha operación ya exista en la base de datos.

9- **public bool existeHijo(Innovator innovator, string tipoItemRelacion, string hijoABuscar).** Esta función nos permite determinar si una operación contiene un determinado elemento relacionado. Nos será útil para no introducir elementos a las operaciones cada vez que ejecutemos el código, de forma que solo añadiremos el elemento relacionado en caso de que éste aún no exista.

10- **public bool existeOperacion(Innovator innovator).** Determina si la operación contenida por esta clase (clase Operacion) existe ya en Aras o no. De la misma forma que la función 9, esto nos servirá para evitar crear operaciones duplicadas cada vez que ejecutemos el código.

11- **public Item ArasAdd_Operacion (Innovator innovator).** Esta función crea una nueva operación y le asigna la estación y la parte resultado que le corresponda (definidos en los propios atributos de esta clase Operación). Hacemos uso de las funciones definidas en los puntos 2, 4 y 6 de esta lista.

12- **public Item ArasAdd_Relaciones(Innovator innovator).** Esta función le asigna a la operación contenida en esta clase (clase Operacion) todos los elementos contenidos en las listas PARTESAFECTADAS, SUBOPERACIONES y RECURSOS. Hace uso de las funciones 3, 5 y 7, además de la función 9 para evitar introducir relaciones duplicadas.

6.2.2.3.4 Clase FuncionesOperacion.

Esta clase se relaciona con la clase de tipo Operación a través de la lista LISTAOPERACION, que está formada por elementos de la clase 'Operacion', y con la clase FuncionesExpress mediante su uso como argumento de las funciones 'clasificaOperacion' y 'clasificaRelaciones', que describiremos más adelante.

Ilustración 6-30: Clase FuncionesOperacion.

Las funciones desarrolladas en esta clase son las siguientes:

1- **public void creaLista ().** Crea la lista LISTAOPERACION.

2- **public string clasificaOperacion(funcionesExpress FE).** Recorre todos los elementos de la lista LISTAEXPRESS contenida en la clase 'funcionesExpress' que se le pasa por argumento, hasta encontrar un elemento que sea de tipo operación. Entonces extrae la información almacenada en dicho elemento de clase 'Express' y la guarda en un elemento de clase 'Operacion' creado. A continuación añade este elemento de clase 'Operacion' a LISTAOPERACION.

3- **public void clasificaRelaciones(funcionesExpress FE).** Recorre todos los elementos de la lista LISTAEXPRESS contenida en la clase 'funcionesExpress' que se le pasa por argumento hasta encontrar alguno que sea un ItemType de tipo relacional que implique a las operaciones (mpp_ParteResultado, mpp_OpeEstacion, mpp_partesAfectadas, mpp_Recurso6745NI o mpp_SubOperaciones). Entonces extrae la información de los elementos de LISTAEXPRESS que guardan relación entre sí. Uno de ellos será de tipo operación, que además ya deberá estar incluido en la lista LISTAOPERACION, puesto que la función 2 debe ejecutarse antes que ésta, y el otro será de tipo estación, operación, recurso o parte. Finalmente añadirá al elemento de tipo 'Operacion' implicado la información referente a el elemento relacionado encontrado, quedando éste definido

unívocamente por su numero (numeroParte, numeroEstacion, numeroOperacion, numeroRecurso. Véase el punto 6.2.2.2: Formato EXPRESS-I), bien sea en las listas PARTESAFECTADAS, SUBOPERACIONES, RECURSOS o en los elementos 'ParteResultado' o 'Estacion'.

4- **public void anadirOperaciones(Innovator innovator).** Añade a Aras todas las operaciones de la LISTAOPERACIONES haciendo uso de la función 11 definida en la clase 'Operacion', y siempre y cuando ésta no exista. Para ello esta función hace uso de la función 8 de la lista de funciones de la clase 'Operacion'.

5- **public void anadirRelaciones(Innovator innovator).** Para cada operación de la LISTAOPERACION hace uso de la función 12 de las funciones definidas en la clase 'Operacion'. Dicha función ya está diseñada para impedir duplicar información.

6.2.2.3.5 Clase Parte.

Los atributos de esta clase están compuestos por el número de la parte y su nombre, que son de tipo string, así como de una lista de tipo string donde se almacenarán los números de ítem de aquellas partes que sean hijas de la que contenga esta clase, PARTES_HIJO. Además incluimos dos atributos de tipo Item que usaremos para crear una nueva parte e introducirla en Aras ('DatoParte') y para añadir una parte hijo a una parte ya existente ('Padre'). Por último una variable de tipo booleano que determinará si la parte almacenada en esta clase tiene hijos o no.

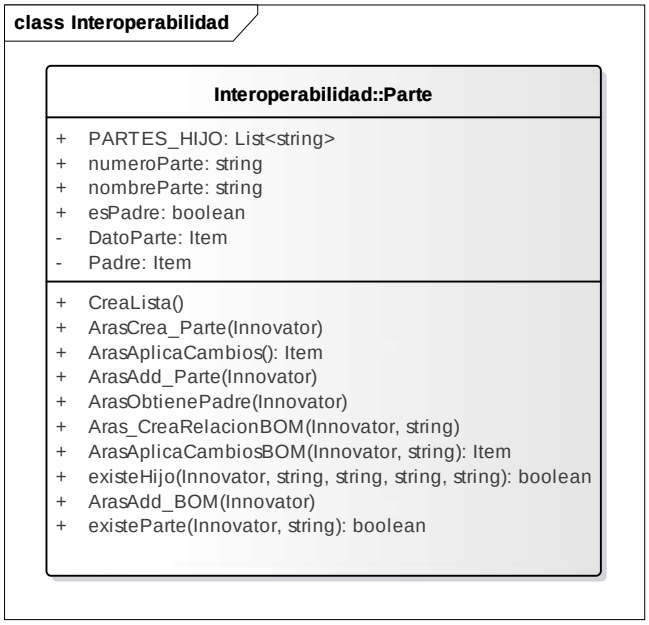


Ilustración 6-31: Clase Parte.

Las funciones desarrolladas en esta clase son:

1- **public void creaLista ().** Crea la lista PARTES_HIJO.

2- **public void ArasCrea_parte (Innovator innovator).** Crea un Item de tipo Part y le asigna los atributos almacenados en esta clase. Queda guardado en el atributo 'DatoParte'.

3- **public Item ArasAplicaCambios().** Hace efectivos los cambios efectuados sobre 'DatoPadre', creando la parte en Aras.

4- **public Item ArasAdd_parte (Innovator innovator).** Combina las dos funciones anteriores, con lo que el resultado de aplicar esta función es que se añade a Aras una nueva parte con los datos contenidos en esta clase.

5- **public void ArasObtienePadre(Innovator innovator).** Obtiene de Aras el Item de tipo parte que se corresponde con la parte almacenada en esta clase y se lo asigna al atributo de tipo Item 'Padre'.

6- **public static void creaRelacionBOM(Innovator innovator,string itemNumber).** Modifica el Item

‘Padre’ añadiéndole como item relacionado aquella parte cuyo número de parte se corresponde con el que se le pase por argumento. Ese ítem de tipo parte ya debe existir en Aras, ésta función simplemente realiza una consulta a Aras para obtener dicha parte y asignársela a ‘Padre’.

7- **public Item ArasAplicaCambiosBOM ().** Aplica los cambios realizados sobre ‘Padre’.

8- **public bool existeHijo(Innovator innovator, string hijoABuscar).** Evalúa si la parte contenida en esta clase ‘Parte’ ya tiene asignada una parte cuyo numero de parte se corresponde con el que se le pasa por argumento. Nos será de utilidad para evitar asignar varias veces el mismo hijo a una parte.

9- **public void ArasAdd_BOM(Innovator innovator).** Por cada elemento de la lista PARTES_HIJO añade una parte a la parte contenida en esta clase ‘Parte’, haciendo uso de las funciones 5, 6 y 7. Además emplea la función 8 para evitar crear duplicidades.

10- **public bool existeParte(Innovator innovator, string Numero).** Realiza una consulta a Aras buscando, entre las partes existentes, aquella cuyo numero de parte se le pasa por argumento. En caso de existir devuelve ‘true’.

6.2.2.3.6 Clase funcionesParte.

Esta clase emplea como único atributo una lista compuesta por elementos de tipo parte, LISTAPARTES.

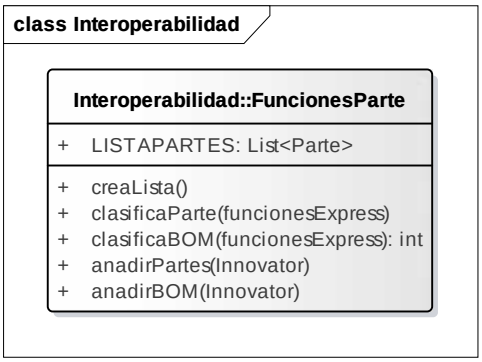


Ilustración 6-32: Clase funcionesParte.

Las funciones implementadas en esta clase son:

1- **public void creaLista ().** Crea la lista LISTAPARTES.

2- **public string clasificaParte(funcionesExpress FE).** Recorre todos los elementos de la lista LISTAEXPRESS contenida en la clase ‘funcionesExpress’ que se le pasa por argumento, hasta encontrar un elemento que sea de tipo parte. Entonces extrae la información almacenada en dicho elemento de clase ‘Express’ y la guarda en un elemento de clase ‘Parte’ creado. A continuación añade este elemento de clase ‘Parte’ a LISTAPARTES.

3- **public void clasificaBOM(funcionesExpress FE).** Recorre todos los elementos de la lista LISTAEXPRESS contenida en la clase ‘funcionesExpress’ que se le pasa por argumento hasta encontrar alguno cuyo ItemType coincida con ‘Part BOM’. Entonces extrae la información de los elementos de LISTAEXPRESS que guardan relación entre sí, que en este caso ambos son de tipo parte: uno es la parte padre y el otro es la parte hijo. El padre ya deberá estar incluido en la lista LISTAPARTES, puesto que la función 2 debe ejecutarse antes que ésta. Recorremos los elementos de dicha lista hasta encontrar aquél que se corresponda con la parte padre, y entonces le añadimos la otra parte a la lista PARTES_HIJO correspondiente al elemento de LISTAPARTES que se corresponda con la parte padre.

4- **public void anadirPartes(Innovator innovator).** Para cada elemento tipo ‘Parte’ de la lista LISTAPARTES crea en Aras una parte, haciendo uso de la función 4 de la clase ‘Parte’. Además empleamos en esta función a la función 10 de la clase ‘Parte’, evitando así duplicar información.

5- **public void anadirBOM(Innovator innovator).** Para cada elemento tipo ‘Parte’ de la lista LISTAPARTES creamos en Aras sus partes hijo, hacienda uso de la función 9 de la clase ‘Parte’.

6.2.2.3.7 Clase Recurso.

Los atributos de la clase 'Recurso' son su numero y su nombre, de tipo string. También necesitaremos, al igual que hicimos con las clases 'Operacion' y 'Parte', un atributo de tipo Item que llamaremos 'datoRecurso'.

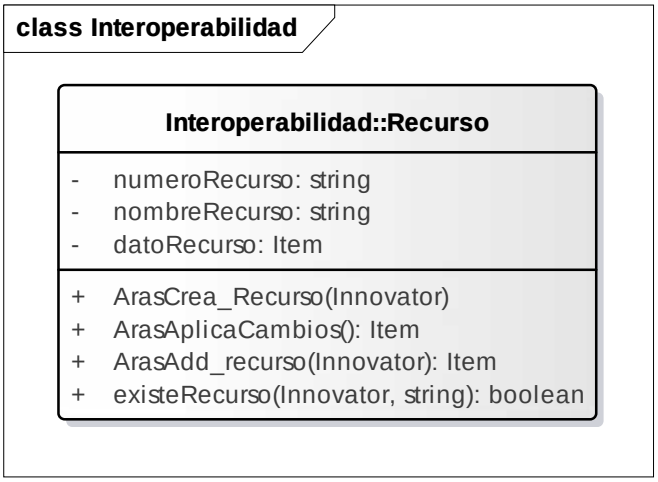


Ilustración 6-33: Clase Recurso.

Las funciones que emplea esta clase son:

1- **public void ArasCrea_recurso (Innovator innovator).** Crea un Item del tipo 'mpp_Recurso6745' definido en Aras, le asigna los datos contenidos en los atributos de esta clase y lo guarda en 'datoRecurso'.

2- **public Item ArasAplicaCambios ().** Hace efectivos los cambios aplicados sobre el ítem 'datoRecurso', quedando añadido éste a Aras.

3- **public Item ArasAdd_recurso (Innovator innovator).** Hace uso de las funciones 1 y 2 para crear un item de tipo recurso en Aras con los datos de esta clase.

4- **public bool existeRecurso (Innovator innovator, string Numero).** Realiza una consulta a Aras para comprobar si existe el recurso cuyo número de recurso se le pasa por argumento. Devuelve 'true' en caso de que exista.

6.2.2.3.8 Clase funcionesRecurso.

Esta clase tiene como único atributo una lista compuesta por elementos de la clase 'Recurso', que hemos llamado LISTARECURSO.

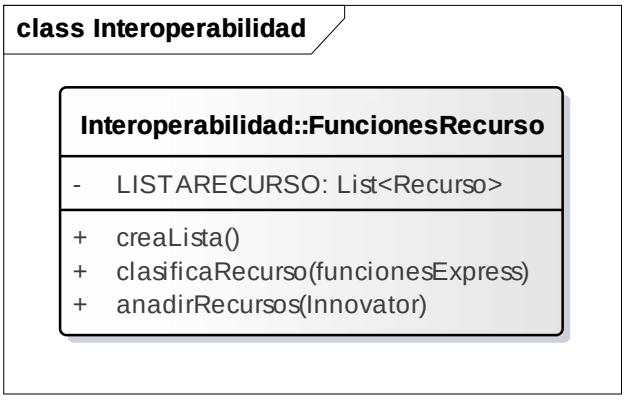


Ilustración 6-34: Clase funcionesRecurso.

Las funciones que forman arte de esta clase son:

1- **public void creaLista ().** Crea la lista LISTARECURSO.

2- **public void clasificaRecurso(funcionesExpress FE).** Añade a LISTARECURSO los elementos de

LISTAEXPRESS que sean de tipo recurso.

3- **public void anadirRecursos(Innovator innovator).** Añade a Aras un recurso por cada elemento contenido en la lista LISTARECURSO. Hace uso de las funciones 3 y 4 de la clase 'Recurso'.

6.2.2.3.9 Clase Estacion.

Análogamente a la clase 'Recurso' en esta clase definimos los atributos de tipo string asociados al numero de estación y a su nombre. También necesitaremos un atributo tipo Item que llamaremos 'datoEstacion'.

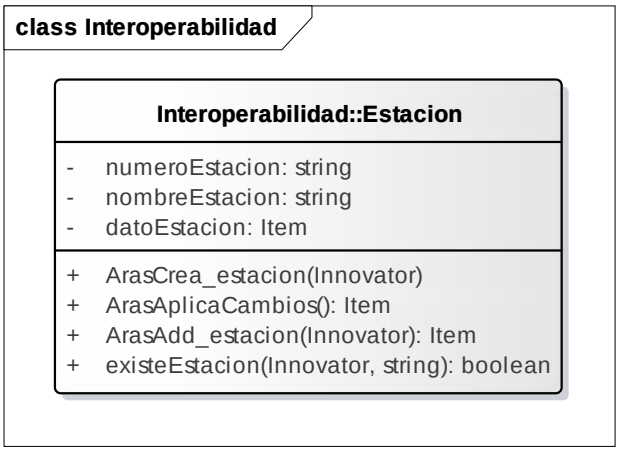


Ilustración 6-35: Clase Estacion.

Las funciones que emplea esta clase son:

1- **public void ArasCrea_estacion (Innovator innovator).** Crea un Item del tipo 'mpp_Estacion' definido en Aras, le asigna los datos contenidos en los atributos de esta clase y lo guarda en 'datoEstacion'.

2- **public Item ArasAplicaCambios ().** Hace efectivos los cambios aplicados sobre el ítem 'datoEstacion', quedando añadido éste a Aras.

3- **public Item ArasAdd_recurso (Innovator innovator).** Hace uso de las funciones 1 y 2 para crear un item de tipo estacion en Aras con los datos de esta clase.

4- **public bool existeRecurso (Innovator innovator, string Numero).** Realiza una consulta a Aras para comprobar si existe la estación cuyo número de estacion se le pasa por argumento. Devuelve 'true' en caso de que exista.

6.2.2.3.10 Clase funcionesEstacion.

Esta clase tiene como único atributo una lista compuesta por elementos de la clase 'Estacion', que hemos llamado LISTAESTACION.

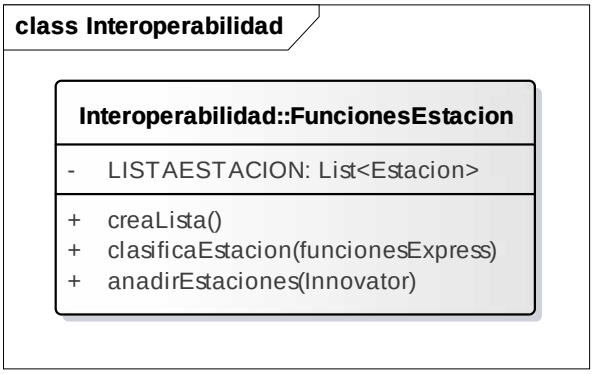


Ilustración 6-36: Clase funcionesEstacion.

Las funciones que forman arte de esta clase son:

1- **public void creaLista ().** Crea la lista LISTAESTACION.

2- **public void clasificaEstacion(funcionesExpress FE).** Añade a LISTAESTACION los elementos de LISTAEXPRESS que sean de tipo estacion.

3- **public void anadirEstaciones(Innovator innovator).** Añade a Aras una estación por cada elemento contenido en la lista LISTAESTACION. Hace uso de las funciones 3 y 4 de la clase 'Estacion'.

6.2.3 Equilibrado.

En éste módulo hemos implementado el algoritmo de eqilibrado basado en la regla LCR, Largest Candidate Rule. Los resultados de la ejecución de éste código para el caso de nuestro helicóptero los veremos en la sección 7 del presente trabajo, dedicada al caso de uso de todo lo aquí desarrollado.

La clase Equilibrado consta de tres clases, aunque está conectada al resto de clases mediante la clase 'Operacion'. Esto nos permitirá integrar una función en la clase 'Equilibrado' para, una vez realizado el equilibrado, asignar a cada operación la estación donde ésta se desarrollará. Esta función se llama 'asignarEstacion'.

La estructura de datos desarrollada consta de las tres clases recogidas en el siguiente diagrama de clases:

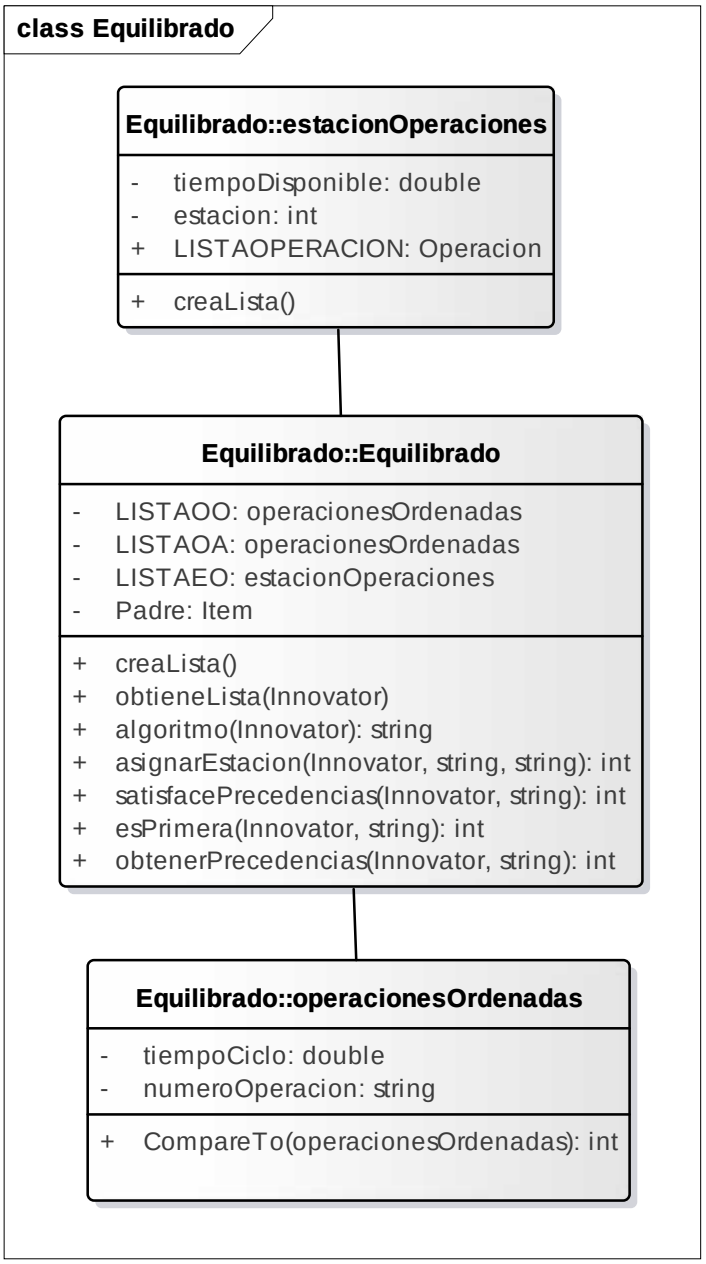


Ilustración 6-37: Diagrama de clase del código desarrollado para implementar el equilibrado.

Explicaremos con qué propósito hemos definido cada caso, atributo y función, y explicaremos con mayor detalle la función 'algoritmo' de la clase 'Equilibrado'.

6.2.3.1 Clase operacionesOrdenadas.

Esta clase recogerá las operaciones que forman parte del diagrama de precedencias de la estructura MBOM (Manufacturing Bill of Materials), también conocida como estructura As Planned, la cual desarrollaremos con detalle en el siguiente capítulo.

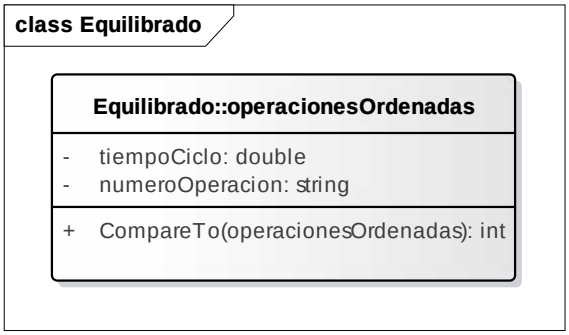


Ilustración 6-38: Clase operacionesOrdenadas.

Los atributos son el tiempo de ciclo, tipo double, y el número de operación, tipo string.

La función 'public int CompareTo(operacionesOrdenadas OO)' nos permitirá ordenar las listas LISTAOO (Lista de Operaciones Ordenadas, definida en la clase 'Equilibrado') y LISTAOA (Lista de Operaciones Asignadas, también definida en la clase 'Equilibrado') atendiendo a la variable tiempo de ciclo definida en esta clase.

6.2.3.2 Clase estacionOperaciones.

Esta clase nos permitirá asociar a cada estación varias operaciones, mediante la inclusión como atributo de una lista de operaciones, LISTAOPERACION. Cada estación tendrá un tiempo disponible y un número de estación.

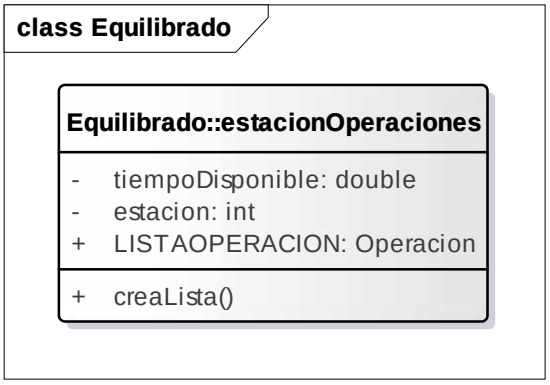


Ilustración 6-39: Clase estacionOperaciones.

La función 'public void creaLista ()' crea la lista LISTAOPERACION.

6.2.3.3 Clase equilibrado.

Esta clase tiene por atributos dos listas de tipo 'operacionesOrdenadas', LISTAOO y LISTAOA, una lista de tipo 'estacionOperaciones', LISTAEO y una lista de tipo string, LISTAPRECEDENCIAS.

- LISTAOO. Es una lista de operaciones, que contendrá todas las operaciones implicadas en nuestro diagrama de precedencias.
- LISTAOA. También es una lista de operaciones, en este caso recogerá las operaciones asignadas a una determinada estación. Una vez que hayamos completado una estación tendremos que borrar esta lista después de haberla copiado a la LISTAOPERACION de la clase 'estacionOperaciones'.
- LISTAEO. En esta lista iremos añadiendo estaciones (elementos de tipo 'estacionOperaciones') conforme se vayan completando a lo largo de la ejecución de la función 'algoritmo'. A su vez, cada elmento de esta lista contendrá una lista de las operaciones que se le han asignado a la estación durante el equilibrado.
- LISTAPRECEDENCIAS. Recoge las operaciones que preceden a una operación dada. Será empleada en el algoritmo para evaluar si una operación es candidata o no para ser añadida a una estación.

Para, una vez completada la ejecución del algoritmo, asignar a cada operación la estación en la cual se llevará a cabo necesitaremos introducir datos en Aras, por lo que necesitamos un atributo de tipo Item que llamaremos 'Padre'.

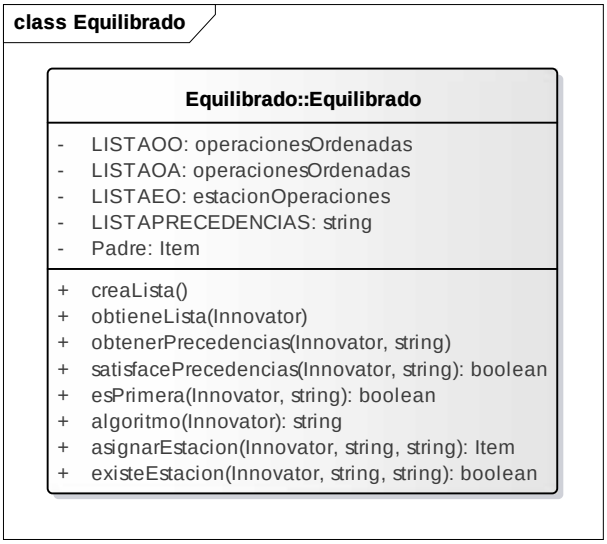


Ilustración 6-40: Clase equilibrado.

Las funciones que hemos necesitado desarrollar para implementar el algortmo son las siguientes:

1- **public void creaLista ().** Crea las listas LISTAOO, LISTAOA y LISTAEO.

2- **public void obtieneLista(Innovator innovator).** Esta función realiza una consulta a Aras para obtener todos los ítems de tipo operación, seleccionar aquellas operaciones que corresponden con nuestro problema y añadirlas a la LISTAOO. A continuación ordena la lista.

3- **public string obtenerPrecedencias(Innovator innovator, string numeroOperacion).** Para una operación pasada por argumento construye la lista LISTAPRECEDENCIAS, que contiene todas las operaciones que la preceden.

4- **public bool satisfacePrecedencias(Innovator innovator, string numeroOperacion).** Evalúa si la operación que se le pasa por argumento satisface el requisito de que todas sus predecesoras (sus suboperaciones) ya estén asignadas a alguna estación. En caso de ser así, devuelve 'true'.

5- **public bool esPrimera(Innovator innovator, string numeroOperacion).** Determina si la operación que se le pasa por argumento es primera, es decir, no tiene ninguna suboperación asignada.

6- **public string algoritmo (Innovator innovator).** Implementa el algoritmo de equilibrado y muestra el resultado por pantalla. Esta función será analizada en detalle en el siguiente apartado.

7- **public Item asignarEstacion(Innovator innovator, string numeroOperacion, string numeroEstacion).** Esta función crea una relación entre la operación y la estación que recibe por argumentos, asignándole la estación a la operación.

8- **public bool existeEstacion(Innovator innovator, string operacion, string estacion).** Determina si la operación que se le pasa por argumento ya tiene asignada la estación que se le pasa por argumento. En caso de ser así devuelve 'true'. La usaremos para evitar duplicidades.

6.2.3.3.1 Análisis detallado de la función 'algoritmo'.

A continuación se muestra el código que desarrolla el algoritmo:

```
public string algoritmo(Innovator innovator)
{
    //Inicializamos nuestras variables.
    int estacion=1;
    string resultado="\nOPERACION\tESTACION\n";
    //Incluimos límites a las iteraciones para prevenir bucles infinitos.
    int LI=0;
    int LI1=0;

    while(LISTAOO.Count>0)//Mientras existan operaciones no asignadas...
    {
 1  //Apertura de estación:
    estacionOperaciones EO=new estacionOperaciones();
    EO.setTiempoDisponible(20);
    EO.setEstacion(estacion);
    EO.creaLista();
    int interruptor=0;
    do
    {
        interruptor=0;
        for(int i=0;i<LISTAOO.Count;i++)
 2      {//n-1
            if( (satisfacePrecedencias(innovator,LISTAOO[i].getNumero())||
                esPrimera(innovator,LISTAOO[i].getNumero()))&
                LISTAOO[i].getTiempoCiclo()<=EO.getTiempoDisponible())
 3          {//n
                LISTAOA.Add(LISTAOO[i]);
                EO.setTiempoDisponible(EO.getTiempoDisponible()-LISTAOO[i].getTiempoCiclo());
                LISTAOO.Remove(LISTAOO[i]);
                interruptor=1;
                i=0;
            }
        }
        LI=LI+1;
        if (LI>50)break;

    }
    while(interruptor==1);

    for(int j=0;j<LISTAOA.Count;j++)
 4  {
        Operacion OPERACION=new Operacion();
        OPERACION.setNumero(LISTAOA[j].getNumero());
        EO.LISTAOPERACION.Add(OPERACION);
    }
 5  LISTAEO.Add(EO);
    estacion=estacion+1;
    LISTAOA.Clear();
    LI1=LI1+1;
    if(LI1>50)break;
    }
 6  for(int i=0;i<LISTAEO.Count;i++)
    {
        for(int j=0; j<LISTAEO[i].LISTAOPERACION.Count;j++)
        {
            resultado=resultado+LISTAEO[i].LISTAOPERACION[j].getNumero()+"\t\t";
            resultado=resultado+LISTAEO[i].getEstacion()+"\n";
        }
    }
 7  for(int i=0;i<LISTAEO.Count;i++)
    {
        for(int j=0; j<LISTAEO[i].LISTAOPERACION.Count;j++)
        {
            if(!existeEstacion(innovator,LISTAEO[i].LISTAOPERACION[j].getNumero(),Convert.ToString(LISTAEO[i].getEstacion()))
            asignarEstacion(innovator,LISTAEO[i].LISTAOPERACION[j].getNumero(),Convert.ToString(LISTAEO[i].getEstacion()));
        }
    }
    return resultado;
}//Implementa el algoritmo del Largest Candidate Rule. Solucion almacenada en LISTAEO.
```

Ilustración 6-41: Código que implementa el algoritmo.

A continuación comentamos los puntos destacados del código:

1. Abrimos estación. Creamos una nueva estación de tipo 'estacionOperacion', que llamamos EO. Le asignamos como tiempo disponible 20 unidades de tiempo y le asignamos su número de estación. Creamos la lista LISTAOPERACION, atributo de la clase 'estacionOperacion'.

2. Para cada operación de la lista de operaciones, ordenadas según el criterio LCR, evaluamos si es o no candidata a entrar en la estación. Para ello debe cumplir que satisface las relaciones de precedencia y que su tiempo de ciclo no exceda al tiempo disponible en la estación.
3. Si la operación resulta ser candidata la añadimos a la lista de operaciones asignadas, LISTAOA, descontamos al tiempo disponible de la estación el tiempo de ciclo de la operación recién incluida y eliminamos de la lista de operaciones a la operación recién asignada. Establecemos el interruptor a uno, puesto que dado que hemos podido asignar una estación a una operación y esto modificará la relación de operaciones candidatas a entrar en una estación, puesto que una vez asignada una operación puede que las que la suceden, que antes no eran candidatas puesto que ésta aún no estaba asignada, ahora pasen a serlo. Por esta misma razón reseteamos el contador 'i', para volver a recorrer la lista de operaciones.
4. El bucle ‘do…while’ termina cuando no ha encontrado ninguna operación que asignar a la estación que está abierta. Entonces pasamos a un bucle ‘for’ que coge cada elemento de la lista LISTAOA y se lo asigna a la lista LISTAOPERACIONES de la estación ‘EO’ que acaba de ser completada.
5. Añadimos la estación ‘EO’ a la lista de estaciones ‘LISTAEO’, incrementamos el número de la estación y borramos la lista de operaciones asignadas, ‘LISTAOA’.

Los pasos descritos hasta aquí se repiten hasta que no queden elementos por asignar en la lista ‘LISTAOO’, que recogía todas las operaciones de nuestro diagrama de precedencias.

6. Empleamos este bucle ‘for’ para construir los resultados que mostraremos por pantalla.
7. Una vez completado todo el proceso solo nos queda añadir a cada operación la estación a la que ha sido asignada, y así terminamos nuestro algoritmo de equilibrado.

6.3 Lista de componentes.

Tenemos dos posibles vistas de los componentes de nuestro helicóptero. Éstas son el **as Design** y el **as Planned.**

6.3.1 As Design.

La lista de materiales **as Designed**, también conocida como la EBOM, Engineering Bill of Materials, contiene los componentes que definen completamente un producto, agrupados según la lógica del diseño. Está compuesto de un desglose por funciones y contiene todos los comonentes que unidos forman el producto. Corresponde al área de ingeniería del diseño. Es una estructura cuya madurez es elevada, tanto que, al menos en el ámbito aeronáutico, se basa en gran medida en estándares internacionales. Está compuesta por tres niveles.

El alto nivel permanece invariante y se define previo al diseño de los componentes, pues así se evita diseñar algún componente cuya función no esté definida con anterioridad. A este nivel corresponden el Programa, la Serie, los Mayor Components y las Secciones.

En el siguiente nivel están los ATA Sections y ATA Zone, que establecen una clasificación por funcionalidad.

El último nivel es el nivel configurable, conformado por los ConfigurationItem, los LinkObjects y las Design Solutions.

En los anexos incluimos la clasificación que hemos realizado del EBOM atendiendo a estos criterios.

6.3.2 As Planned.

La lista de materiales **as Planned**, también conocida como MBOM, Manufacturing Bill of Materials, define los materiales necesarios para la fabricación en un determinado orden, representando la estructura en que se fabrica el producto. Corresponde al área de ingeniería de fabricación. Esta lista de materiales está orientada a la fabricación, de forma que las partes empleadas no tienen que llegar al nivel de detalle del EBOM.

En el anexo se puede encontrar la clasificación que hemos realizado atendiendo al criterio As Planned.

6.4 Diagrama de Precedencias.

El resultado de la estructura de fabricación que hemos implementado es el siguiente diagrama de precedencias.

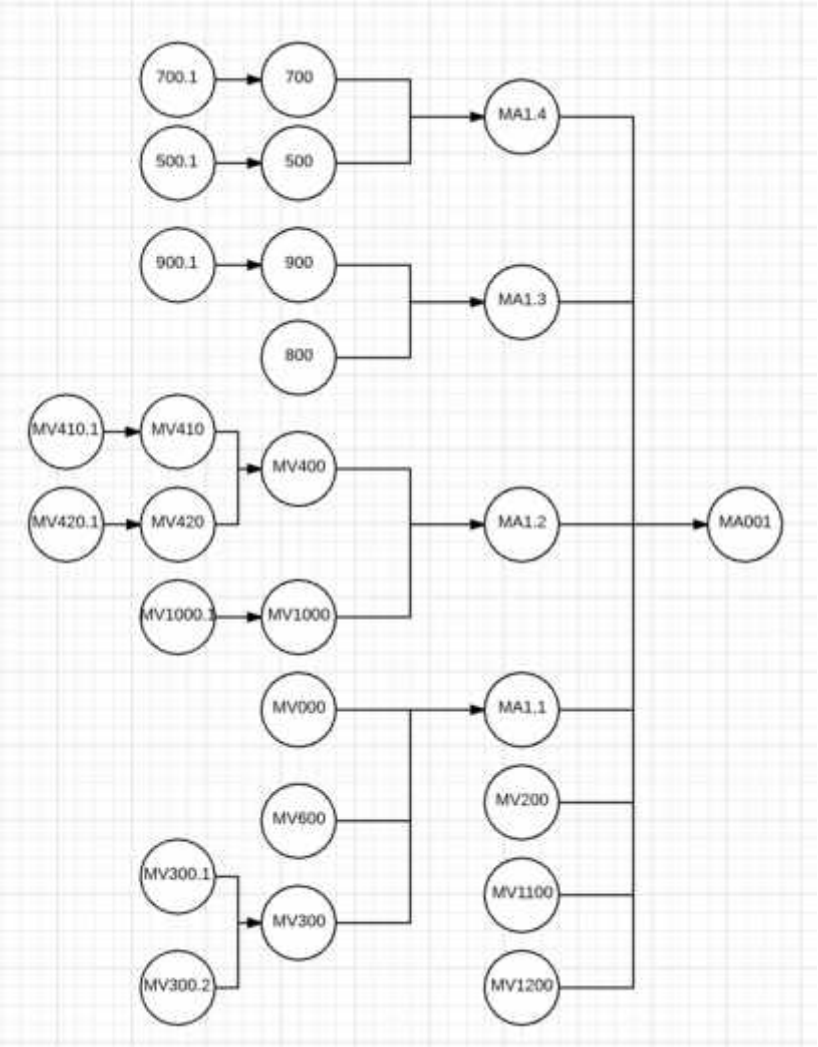


Ilustración 6-42: Diagrama de precedencias de nuestro MBOM.

6.5 Resultados.

Haciendo uso del modulo de interoperabilidad hemos introducido la estructura de fabricación, es decir, el MBOM en Aras, para posteriormente ejecutar el algoritmo de equilibrado y obtener las estaciones que le corresponden a cada operación.

Hemos establecido tiempos de operación aleatorios, estableciendo que sean mayores que uno y menores que diez unidades de tiempo, teniendo en cuenta que el tiempo de operaciones no puede superar el tiempo de ciclo, que hemos fijado en 20 unidades de tiempo.

Ésta es la vista final de nuestra estructura de fabricación una vez introducida en Aras:

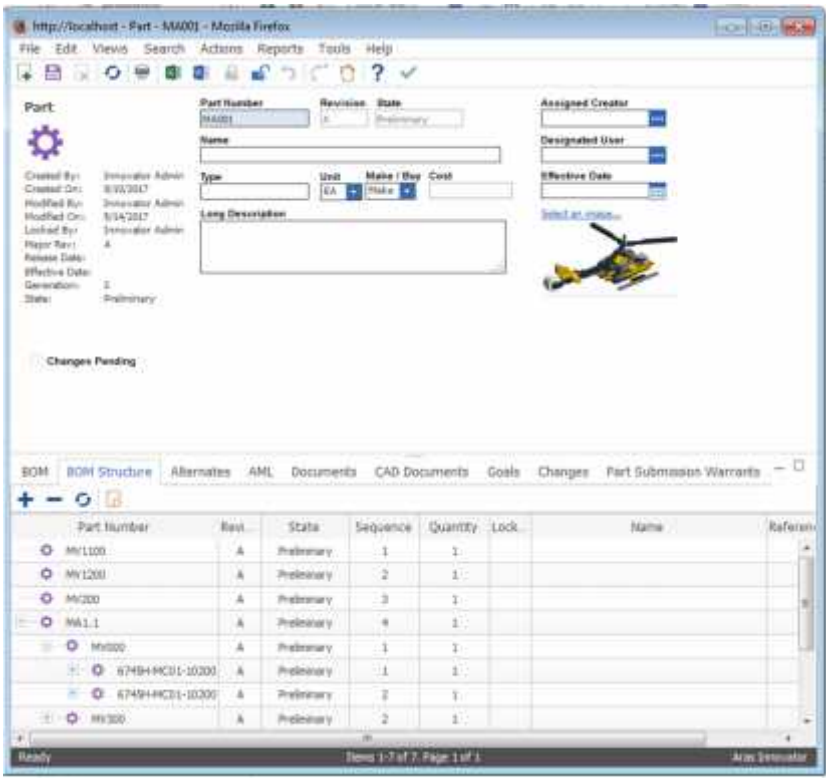


Ilustración 6-43: MBOM de nuestro helicóptero introducido en Aras.

La estructura de fabricación también queda recogida en las operaciones y sus relaciones con las suboperaciones, como se muestr a continuación:

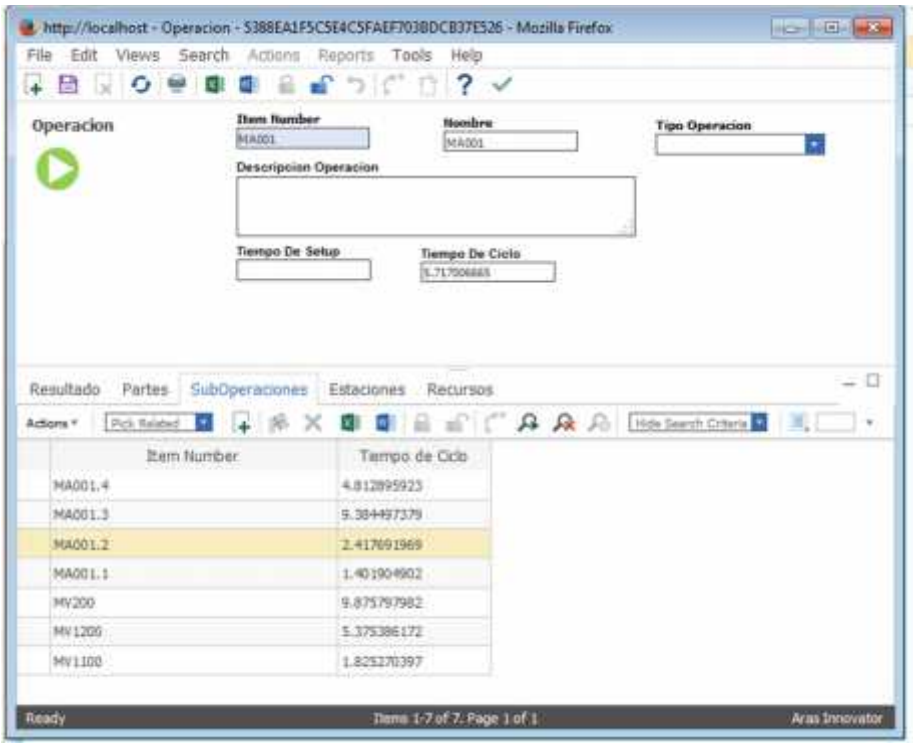


Ilustración 6-44: Suboperaciones de la operación MA001, que es la que cierra el ensamblaje del helicóptero.

El resultado del equilibrado que devuelve nuestro programa es el siguiente:

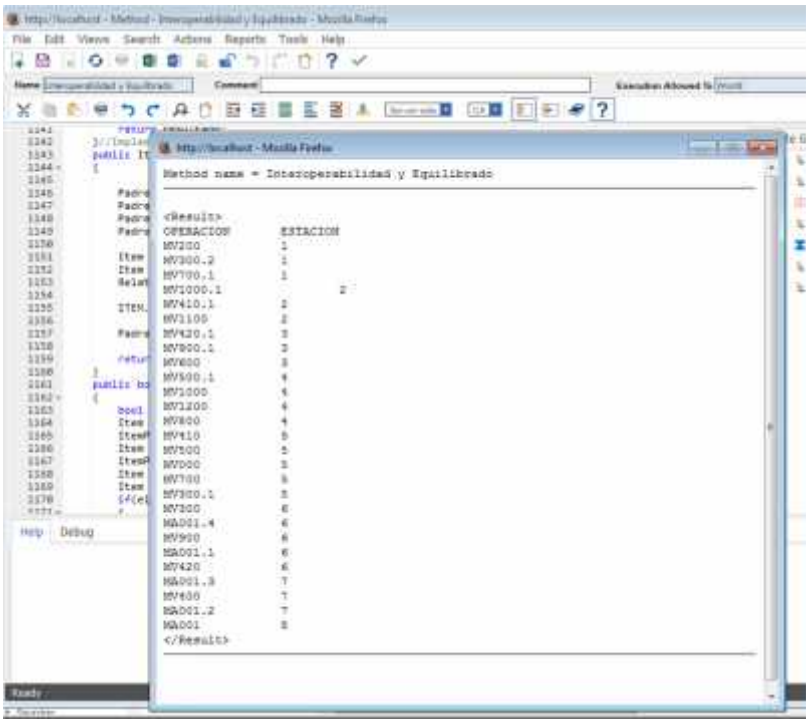


Ilustración 6-45:Resultado mostrado por pantalla del equilibrado.

El mismo resultado trasladado al diagrama de precedencias nos permite ver con claridad que se cumplen las precedencias a la hora de asignar las operaciones a estaciones.

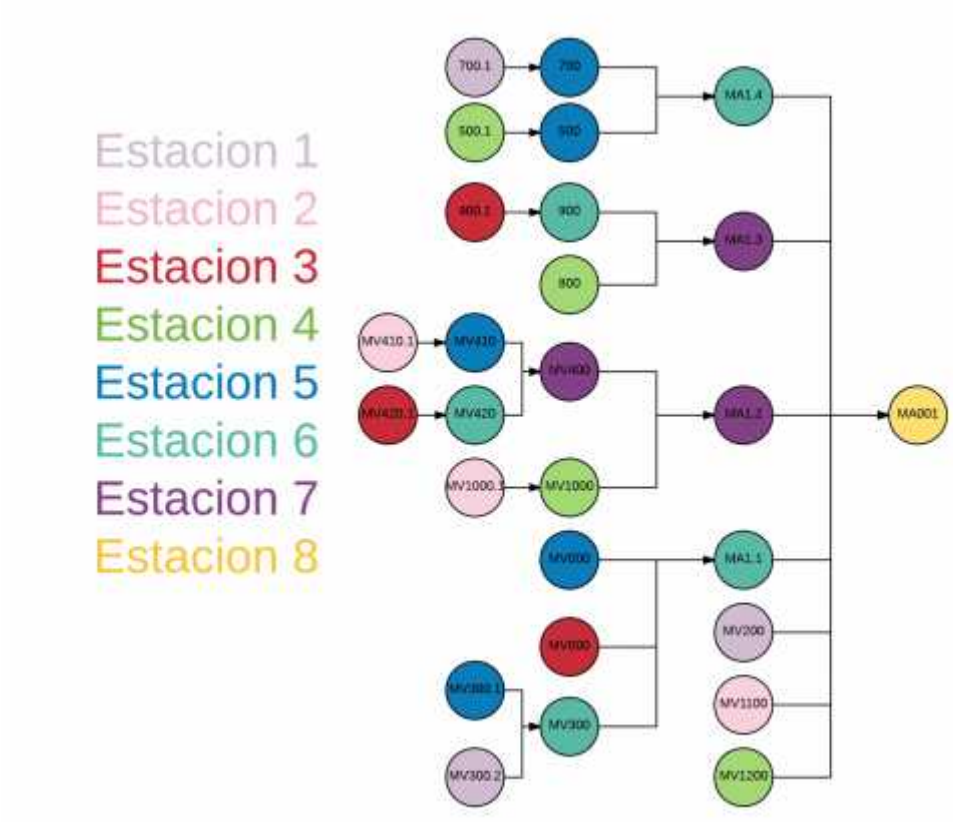


Ilustración 6-46: Resultado del equilibrado trasladado al diagrama de precedencias.

Por último nos queda comprobar que se han añadido las estaciones a las operaciones en Aras.

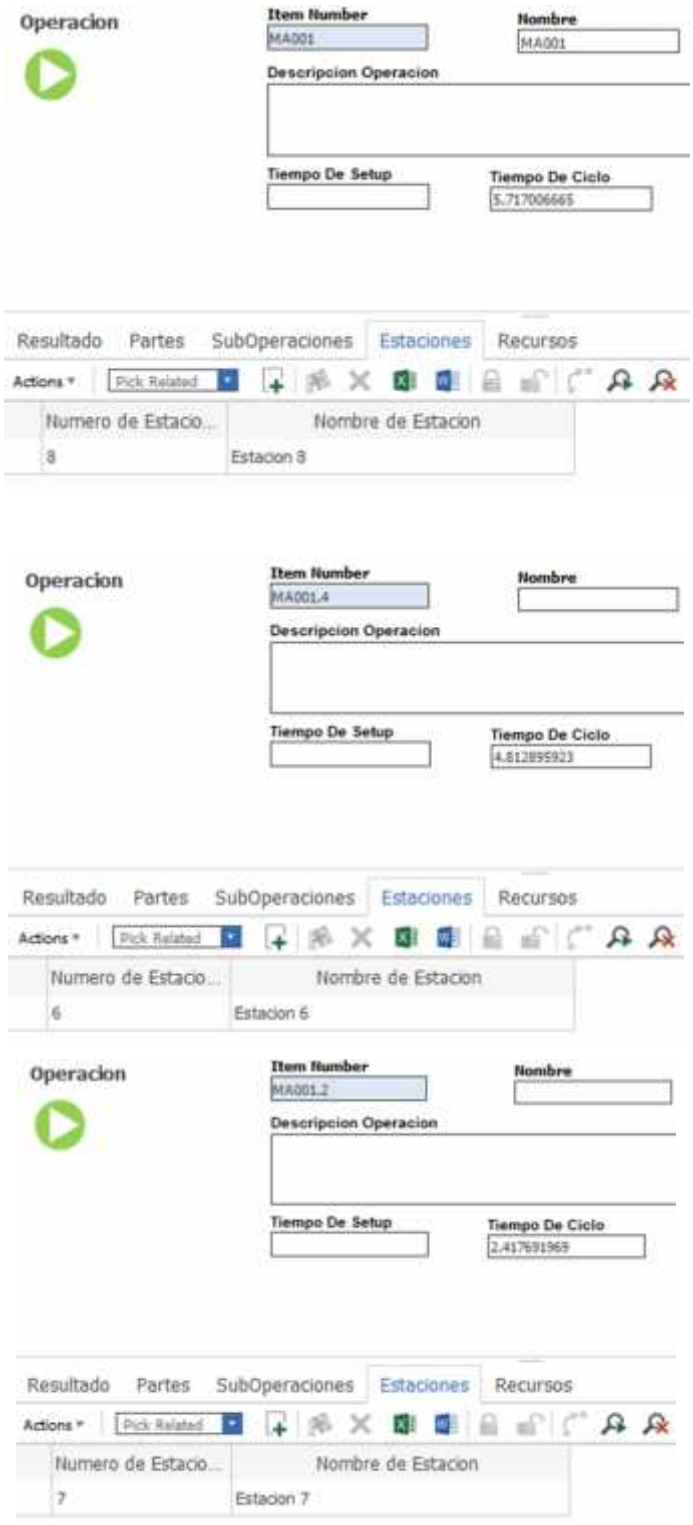


Ilustración 6-47: Estaciones asociadas a varias operaciones implicadas en el equilibrado.

7 CONCLUSIONES Y EXTENSION

Durante el desarrollo del presente proyecto hemos aprendido a customizar una herramienta PLM como es Aras Innovator, además de entender el propósito general de los sistemas de gestión del ciclo de vida del producto. Hemos aprendido también sobre los diferentes tipos de línea de montaje que existen, así como de los diferentes problemas asociados a éstas y los más destacados métodos de resolución existentes. Hemos aplicado métodos de equilibrado como el de programación dinámica, el método de las columnas o el de los pesos posicionales previamente a implementar nuestro algoritmo en código. Ha sido un desafío familiarizarnos con el lenguaje de programación orientada a objetos, a la vez que un proceso de descubrimiento y refuerzo de los conocimientos adquiridos durante la carrera.

Como conclusión podemos decir que hemos desarrollado dos funcionalidades prácticas en relación con Aras Innovator.

Una de ellas es la posibilidad de introducir datos desde un fichero de texto con un formato estandarizado, que uniéndolo con herramientas como Excel y su función de concatenar agiliza mucho el proceso de introducción de datos en Aras.

La otra funcionalidad nos ha permitido realizar un balanceo de línea de una forma limpia y eficaz. Éste balanceo se podría aplicar con diferentes tiempos de ciclo y diferentes tiempos de operación sin necesidad de modificar apenas el código. Sólo necesitaríamos modificar el tiempo de ciclo.

Como extensión podríamos decir que cabría la posibilidad de facilitar aún más la interoperabilidad desarrollando métodos de introducción de datos mediante formularios.

Respecto al equilibrado podríamos implementar el algoritmo Helgenson & Birnie de los pesos posicionales modificando solamente la forma de obtener la lista de operaciones a procesar por el algoritmo. No sería necesario modificar éste.

REFERENCIAS

ARCUS*, A. L. (1965). A computer method of sequencing operations for assembly lines. . *International Journal of Production Research, 4(4),* 259-277.

Becker, C. &. (2006). A survey on problems and methods in generalized assembly line balancing. *European journal of operational research, 168(3),* 694-715.

Berger, I. B. (1992). Branch-and-bound algorithms for the multi-product assembly line balancing problem. *European Journal of Operational Research, 58(2),* 215-222.

Boysen, N. F. (2007). A classification of assembly line balancing problems. *European journal of operational research, 183(2),* 674-693.

Camps, G. C. (2010). Equilibrado de líneas de montaje de productos voluminosos.

Capacho Betancourt, L. &. (2004). Generación de secuencias de montaje y equilibrado de líneas.

Dar-El, E. M. (1973). MALB—a heuristic technique for balancing large single-model assembly lines. *AIIE Transactions, 5(4),* 343-356.

Faura Cabañas, V. (2003). Diferentes formas de modelización para la resolución del problema del equilibrado

de líneas de montaje.

Gómez, J. M. (n.d.). *Trabajo colaborativo en ambientes PLM (Product Lifecycle Management).*

Heizer J, R. B. (2009). *Principios de Administración de Operaciones.* México.: Pearson.

Held, M. K. (1963). Assembly-line balancing—dynamic programming with precedence constraints. . *Operations Research, 11(3)*, 442-459.

Hoffmann, T. R. (1992). EUREKA: A hybrid system for assembly line balancing. *Management Science, 38(1)*, 39-47.

https://admedieval.wikispaces.com/El+arsenal+de+Venecia. (n.d.).

Johnson, R. V. (1998). Optimally balancing large assembly lines with "FABLE". *Management science, 34(2)*, 240-253.

Levitin, G. R. (2006). A genetic algorithm for robotic assembly line balancing. *European Journal of Operational Research, 168(3)*, 811-825.

Liñán Alfaro, F. J. (2016). Implantación de un sistema PLM para automatizar el proceso APQP (ADVANCED PRODUCT QUALITY PLANNING).

López, F. (2009). *www.diariomotor.com.* Retrieved from http://www.diariomotor.com/2009/12/14/henry-ford-no-invento-la-cadena-de-montaje/

Malakooti, B. B. (1994). Assembly line balancing with buffers by multiple criteria optimization. . *THE INTERNATIONAL JOURNAL OF PRODUCTION RESEARCH, 32(9)*, 2159-2178.

Pascual García, I. (2015). Metodologías de resolución para el problema simple de equilibrado de líneas de montaje.

Pastor, R. A. (2002). Tabu search algorithms for an industrial multi-product and multi-objective assembly line balancing problem, with reduction of the task dispersion. *Journal of the Operational Research Society*, 1317-1323.

Restrepo, J. H. (2009). Problemas de balanceo de línea salbp-1 y salbp-2: un caso de estudio. *Scientia et technica*.

Scholl, A. &. (2006). State-of-the-art exact and heuristic solution procedures for simple assembly line balancing. *European Journal of Operational Research, 168(3)*, 666-693.

Servín Ochoa, D. (2008). Equilibrado de líneas de ensamble en la industria del vestido: un enfoque mediante algoritmos genéticos híbridos (Doctoral dissertation).

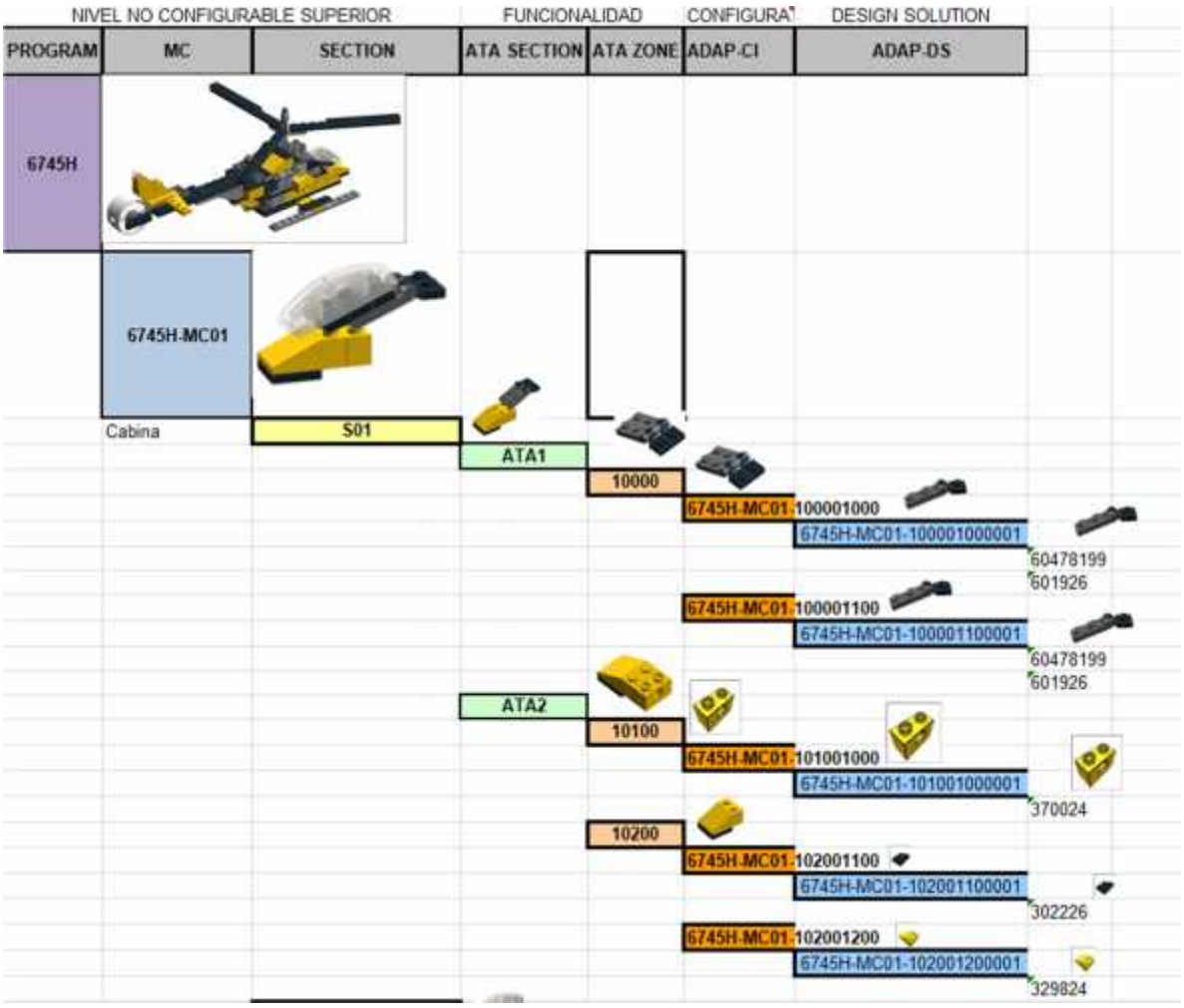
Stark, J. (2015). *Product lifecycle management. In Product Lifecycle Management (Volume 1) (pp. 1-29).* . Springer International Publishing.

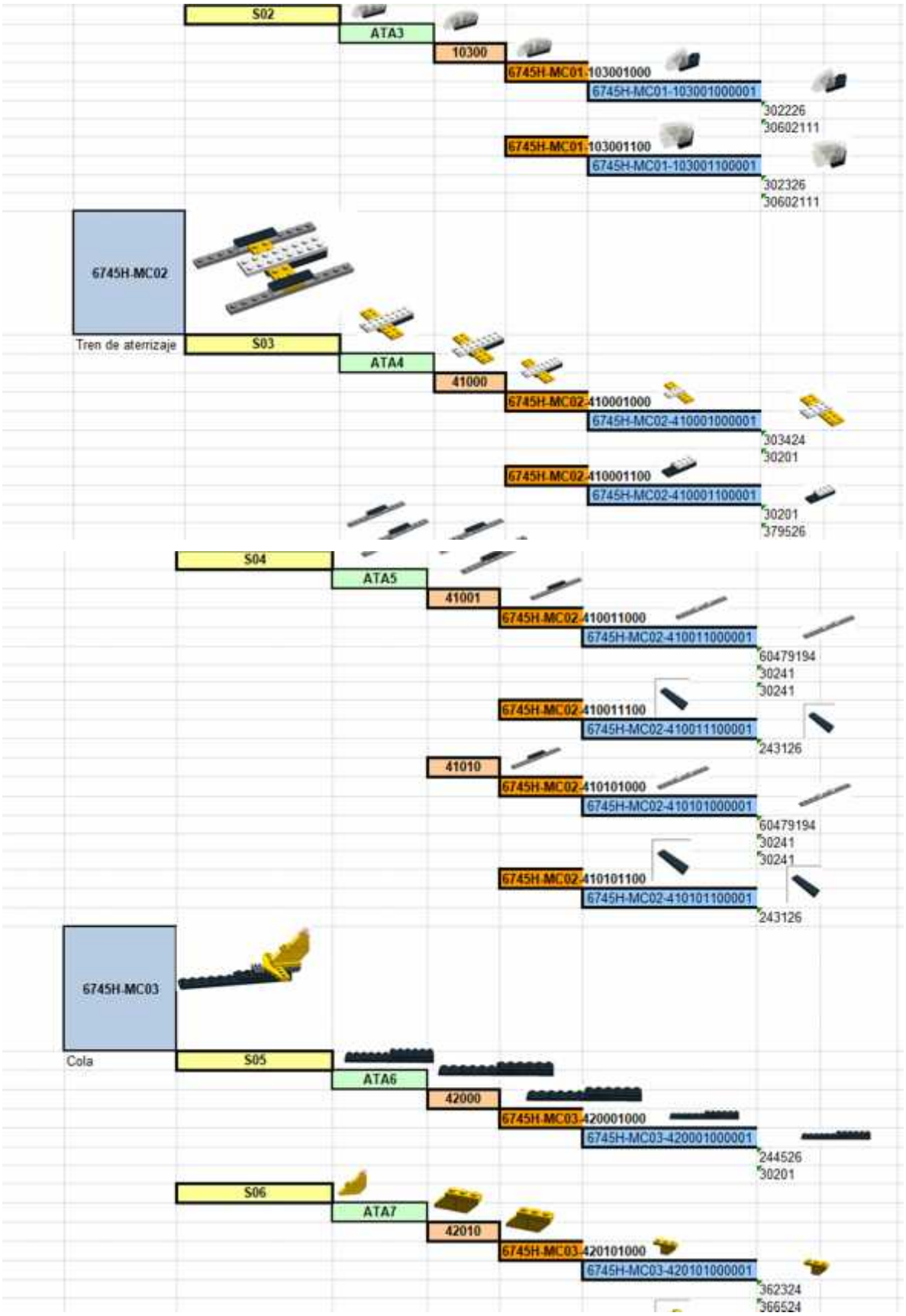
Sumichrast, R. T. (1990). Evaluating mixed-model assembly line sequencing heuristics for just-in-time production systems. Volume 9, Issue 3. *Journal of Operations Management*, 371-390.

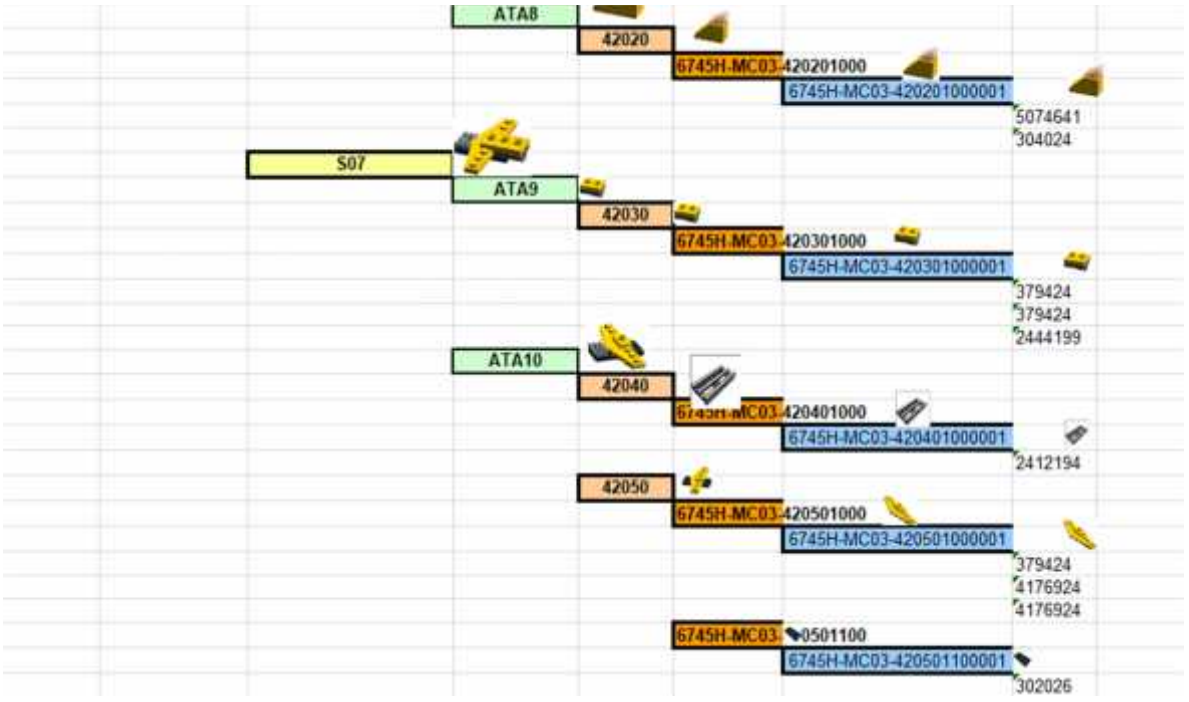
Venecia, E. A. (n.d.). *https://admedieval.wikispaces.com/El+arsenal+de+Venecia.*

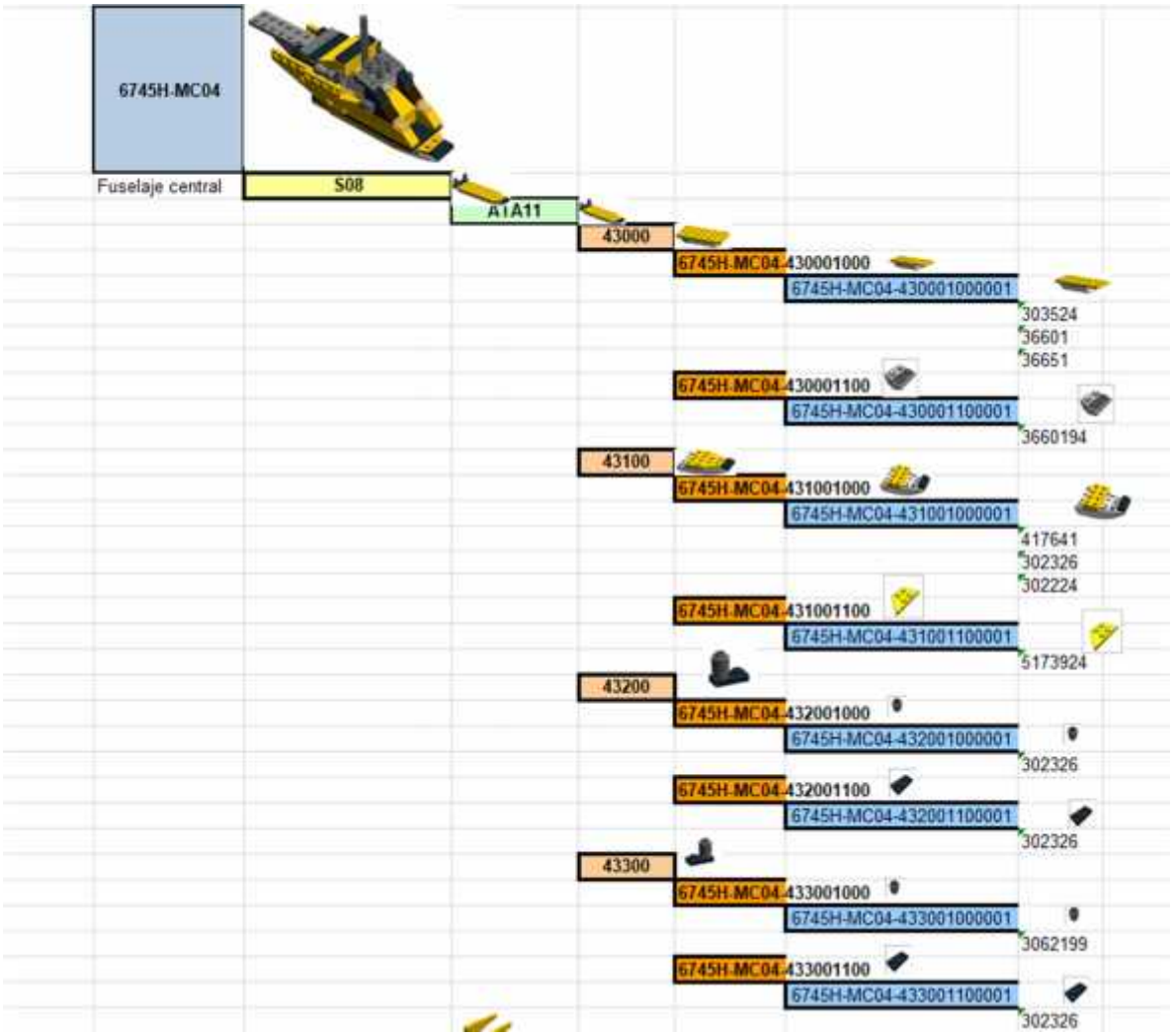
Worstall, T. (2012, Mar 4). *www.forbes.com.* Retrieved from https://www.forbes.com/sites/timworstall/2012/03/04/the-story-of-henry-fords-5-a-day-wages-its-not-what-you-think/#8abfb89766d2

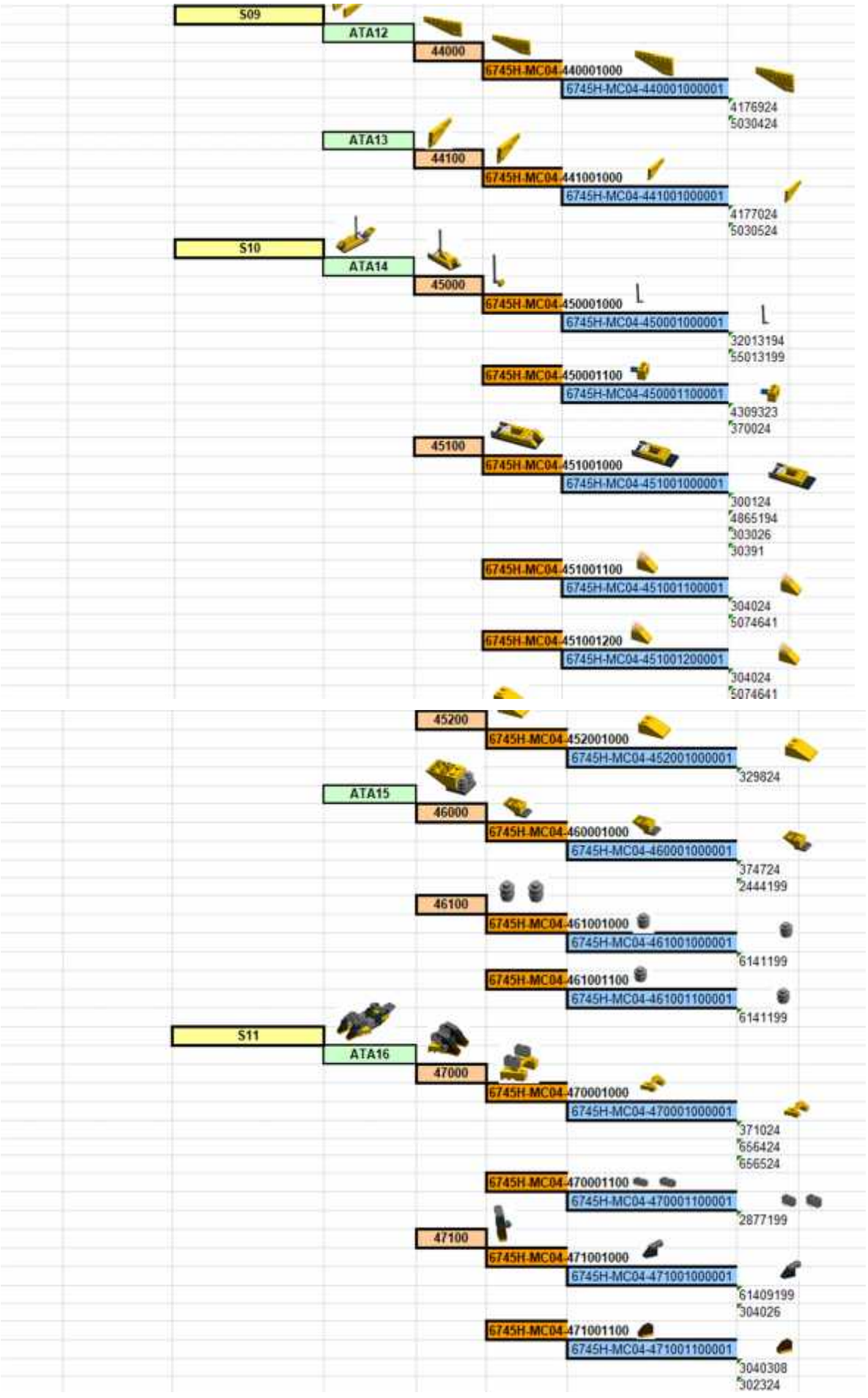
Estructura EBOM.

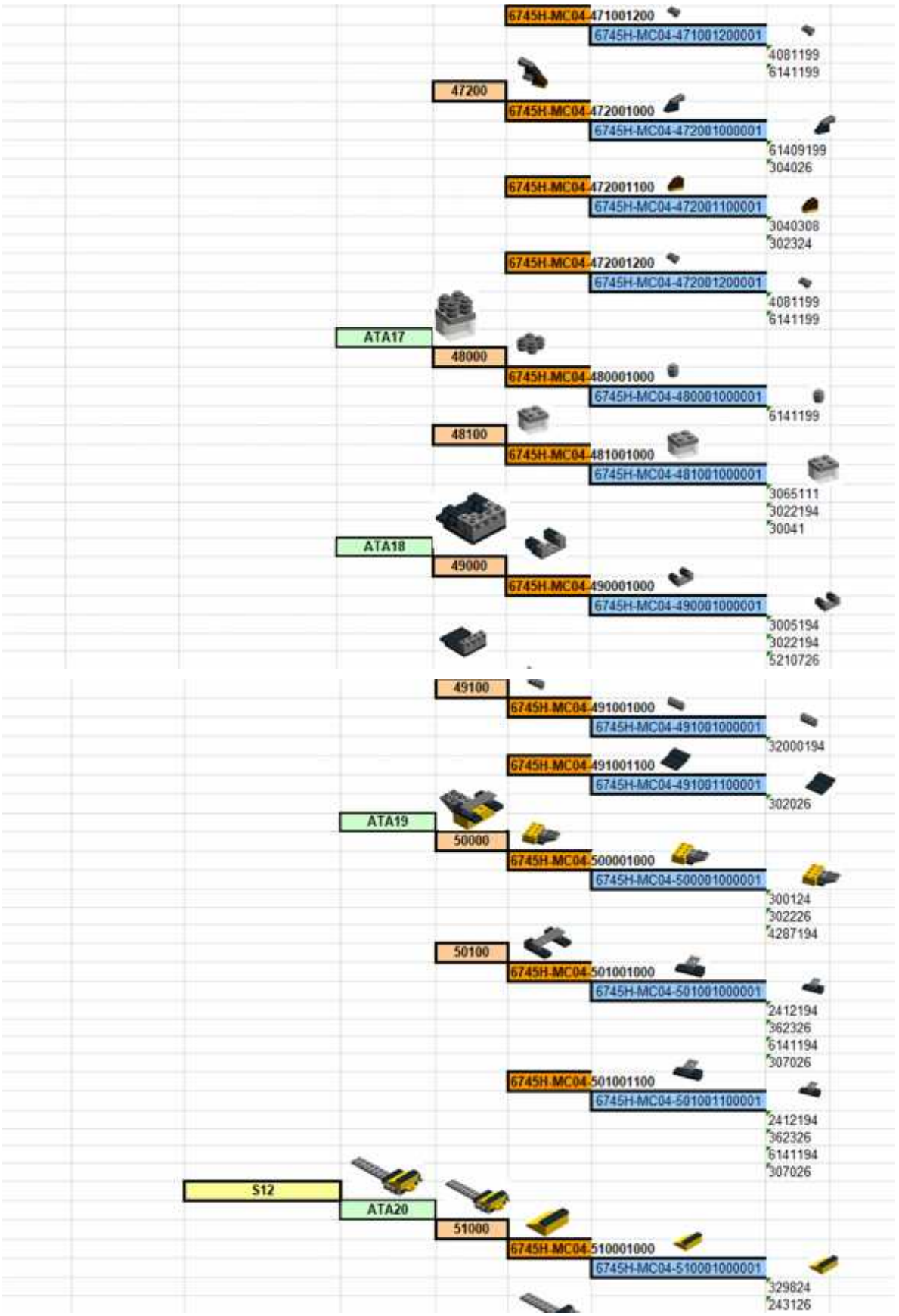


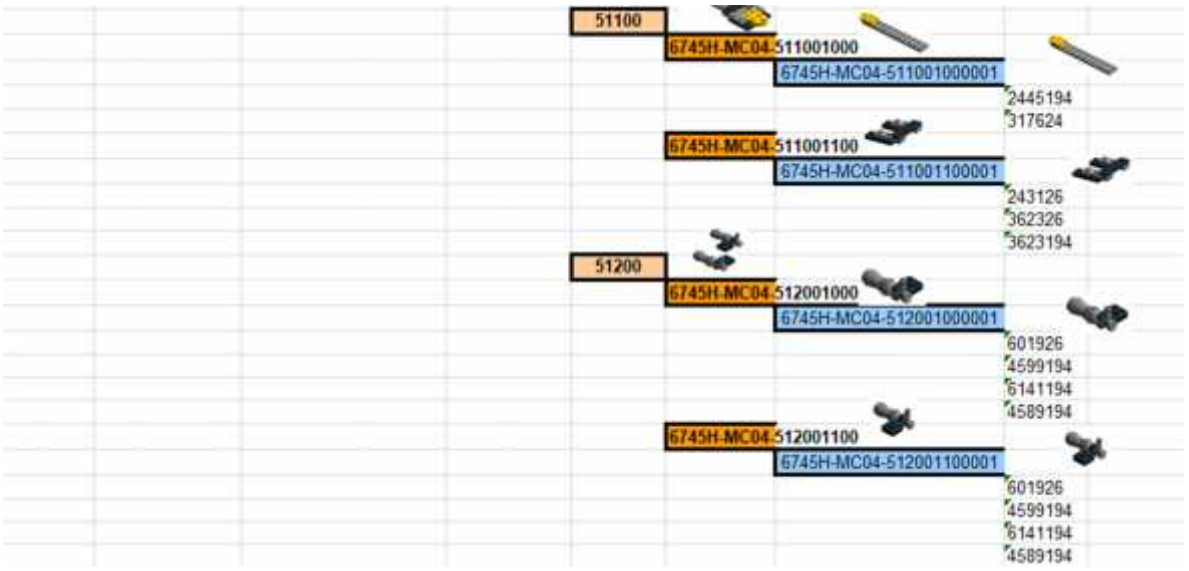

S02
ATA3
10300
6745H-MC01-103001000
6745H-MC01-103001000001
302226
30602111
6745H-MC01-103001100
6745H-MC01-103001100001
302326
30602111
6745H-MC02
Tren de aterrizaje
S03
ATA4
41000
6745H-MC02-410001000
6745H-MC02-410001000001
303424
30201
6745H-MC02-410001100
6745H-MC02-410001100001
30201
379526
S04
ATA5
41001
6745H-MC02-410011000
6745H-MC02-410011000001
60479194
30241
30241
6745H-MC02-410011100
6745H-MC02-410011100001
243126
41010
6745H-MC02-410101000
6745H-MC02-410101000001
60479194
30241
30241
6745H-MC02-410101100
6745H-MC02-410101100001
243126
6745H-MC03
Cola
S05
ATA6
42000
6745H-MC03-420001000
6745H-MC03-420001000001
244526
30201
S06
ATA7
42010
6745H-MC03-420101000
6745H-MC03-420101000001
362324
366524

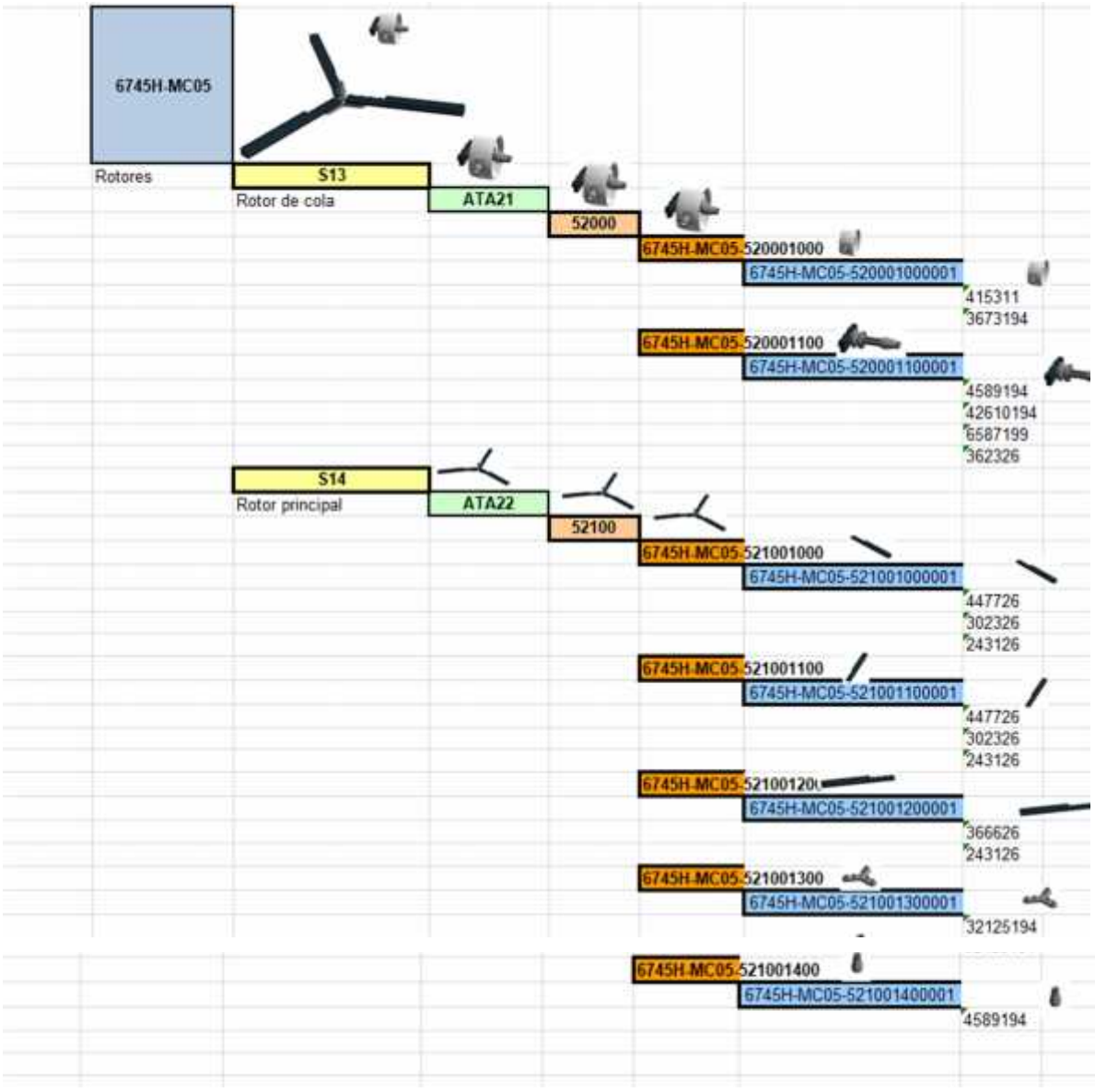

ATA8
42020
6745H-MC03-420201000
6745H-MC03-420201000001
5074641
304024
S07
ATA9
42030
6745H-MC03-420301000
6745H-MC03-420301000001
379424
379424
2444199
ATA10
42040
6745H-MC03-420401000
6745H-MC03-420401000001
2412194
42050
6745H-MC03-420501000
6745H-MC03-420501000001
379424
4176924
4176924
6745H-MC03-0501100
6745H-MC03-420501100001
302026


6745H-MC04
Fuselaje central
S08
A1A11
43000
6745H-MC04-430001000
6745H-MC04-430001000001
303524
36601
36651
6745H-MC04-430001100
6745H-MC04-430001100001
3660194
43100
6745H-MC04-431001000
6745H-MC04-431001000001
417641
302326
302224
6745H-MC04-431001100
6745H-MC04-431001100001
5173924
43200
6745H-MC04-432001000
6745H-MC04-432001000001
302326
6745H-MC04-432001100
6745H-MC04-432001100001
302326
43300
6745H-MC04-433001000
6745H-MC04-433001000001
3062199
6745H-MC04-433001100
6745H-MC04-433001100001
302326


S09
ATA12
44000
6745H-MC04-440001000
6745H-MC04-440001000001
4176924
5030424
ATA13
44100
6745H-MC04-441001000
6745H-MC04-441001000001
4177024
5030524
S10
ATA14
45000
6745H-MC04-450001000
6745H-MC04-450001000001
32013194
55013199
6745H-MC04-450001100
6745H-MC04-450001100001
4309323
370024
45100
6745H-MC04-451001000
6745H-MC04-451001000001
300124
4865194
303026
30391
6745H-MC04-451001100
6745H-MC04-451001100001
304024
5074641
6745H-MC04-451001200
6745H-MC04-451001200001
304024
5074641
45200
6745H-MC04-452001000
6745H-MC04-452001000001
329824
ATA15
46000
6745H-MC04-460001000
6745H-MC04-460001000001
374724
2444199
46100
6745H-MC04-461001000
6745H-MC04-461001000001
6141199
6745H-MC04-461001100
6745H-MC04-461001100001
6141199
S11
ATA16
47000
6745H-MC04-470001000
6745H-MC04-470001000001
371024
656424
656524
6745H-MC04-470001100
6745H-MC04-470001100001
2877199
47100
6745H-MC04-471001000
6745H-MC04-471001000001
61409199
304026
6745H-MC04-471001100
6745H-MC04-471001100001
3040308
302324


6745H-MC04-471001200
6745H-MC04-471001200001
4081199
6141199
47200
6745H-MC04-472001000
6745H-MC04-472001000001
61409199
304026
6745H-MC04-472001100
6745H-MC04-472001100001
3040308
302324
6745H-MC04-472001200
6745H-MC04-472001200001
4081199
6141199
ATA17
48000
6745H-MC04-480001000
6745H-MC04-480001000001
6141199
48100
6745H-MC04-481001000
6745H-MC04-481001000001
3065111
3022194
30041
ATA18
49000
6745H-MC04-490001000
6745H-MC04-490001000001
3005194
3022194
5210726
49100
6745H-MC04-491001000
6745H-MC04-491001000001
32000194
6745H-MC04-491001100
6745H-MC04-491001100001
302026
ATA19
50000
6745H-MC04-500001000
6745H-MC04-500001000001
300124
302226
4287194
50100
6745H-MC04-501001000
6745H-MC04-501001000001
2412194
362326
6141194
307026
6745H-MC04-501001100
6745H-MC04-501001100001
2412194
362326
6141194
307026
S12
ATA20
51000
6745H-MC04-510001000
6745H-MC04-510001000001
329824
243126


51100
6745H-MC04-511001000
6745H-MC04-511001000001
2445194
317624
6745H-MC04-511001100
6745H-MC04-511001100001
243126
362326
3623194
51200
6745H-MC04-512001000
6745H-MC04-512001000001
601926
4599194
6141194
4589194
6745H-MC04-512001100
6745H-MC04-512001100001
601926
4599194
6141194
4589194


6745H-MC05
Rotores
S13
Rotor de cola
ATA21
52000
6745H-MC05-520001000
6745H-MC05-520001000001
415311
3673194
6745H-MC05-520001100
6745H-MC05-520001100001
4589194
42610194
6587199
362326
S14
Rotor principal
ATA22
52100
6745H-MC05-521001000
6745H-MC05-521001000001
447726
302326
243126
6745H-MC05-521001100
6745H-MC05-521001100001
447726
302326
243126
6745H-MC05-52100120(
6745H-MC05-521001200001
366626
243126
6745H-MC05-521001300
6745H-MC05-521001300001
32125194
6745H-MC05-521001400
6745H-MC05-521001400001
4589194

Estructura MBOM.

MBOM	
Proceso	**Partes**
MA001	**6745H-MC05-521001400**
	MA001.1
	MA001.2
	MA001.3
	MA001.4
	MV1100
	MV1200
	MV200
MA001.1	**MV000**
	MV600
	MV300
MV000	6745H-MC01-102001100
	6745H-MC01-102001200
MV600	**6745H-MC04-440001000**
	6745H-MC04-441001000
MV300	**MV300.1**
	MV300.2
MV300.1	**6745H-MC02-410011000**
	6745H-MC02-410011100
MV300.2	**6745H-MC02-410101000**
	6745H-MC02-410101100
MA001.2	**MV400**
	MV1000
MV400	**6745H-MC03-420001000**
	MV410
	MV420
MV1000	**6745H-MC04-510001000**
	MV1000.1
MV1000.1	**6745H-MC04-511001000**
	6745H-MC04-511001100

MA001.3	MV900
	MV800
MV900	6745H-MC04-501001000
	6745H-MC04-501001100
	MV900.1
MV900.1	6745H-MC04-500001000
	6745H-MC04-491001000
	6745H-MC04-490001000
	6745H-MC04-491001100
MV800	6745H-MC04-460001000
	6745H-MC04-461001000
	6745H-MC04-461001100

MA001.4	MV700
	MV500
MV700	6745H-MC04-452001000
	MV700.1
MV700.1	6745H-MC04-451001100
	6745H-MC04-451001200
	6745H-MC04-451001000
	6745H-MC01-101001000
MV500	6745H-MC04-432001000
	6745H-MC04-432001100
	6745H-MC04-433001000
	6745H-MC04-433001100
	MV500.1
MV500.1	6745H-MC04-430001000
	6745H-MC04-431001000
	6745H-MC04-431001100
	6745H-MC04-430001100

MV1100	6745H-MC05-520001000
	6745H-MC05-520001100

MV1200	6745H-MC05-521001000
	6745H-MC05-521001100
	6745H-MC05-521001200
	6745H-MC05-521001300

MV200	6745H-MC01-103001000
	6745H-MC01-103001100
	6745H-MC01-100001000
	6745H-MC01-100001100