**CAPÍTULO 1: Concurrencia y Patrones de Diseño**

En este capítulo veremos una introducción a ExecutorService, una interfaz de Java para la gestión de tareas concurrentes y al patrón de diseño creacional Abstract Factory por medio del cual podremos crear familias de objetos.

Por último se presentará un caso de aplicación que combina la gestión de tareas concurrentes por medio de ExecutorService y la flexibilidad del patrón de diseño Abstract Factory para la creación de familias de objetos.

**ExecutorService**

La interfaz ExecutorService es un herramienta que facilita la gestión de tareas concurrentes, la misma permite ejecutar por medio de hilos dichas tareas de manera asíncrona, manejando la creación y gestión de los hilos.

ExecutorService es una extensión de Executor, ambas dentro del paquete ‘java.util.concurrent’. Executor es una interfaz simple que permite ejecutar tareas de tipo Runnable, cuenta con un único método ‘execute’, por otro lado ExecutorService añade características que ayudan a gestionar el ciclo de vida, tanto de las tareas individuales como del propio ejecutor.

Una de las implementación de ExecutorService presente en el paquete ‘java.util.concurrent’ es una implementación de tipo thread pool (ThreadPoolExecutor) donde se tiene un conjunto de hilos, los cuales al ejecutar y finalizar una tarea podrán ser reusados para nuevas ejecuciones, por lo que un mismo hilo podrá ejecutar múltiples tareas.

**Métodos de ExecutorService**

| execute(Runnable r) | Ejecuta la tarea dada (de tipo Runnable) en uno de los hilos disponibles. No retorna ningún valor. |
| --- | --- |
| shutdown() | Inicia un cierre ordenado, permitiendo que las tareas en ejecución terminen, pero no acepta nuevas tareas. |
| shutdownNow() | Intenta detener todas las tareas en ejecución, devuelve una lista de tareas pendientes y no ejecutadas, y no acepta nuevas tareas. |
| isShutdown() | Retorna true si el ExecutorService ha sido cerrado usando shutdown() o shutdownNow(). No indica si las tareas han finalizado. |
| isTerminated() | Retorna true si todas las tareas se han completado tras la llamada a shutdown(). |
| awaitTermination(long timeout, TimeUnit unit) | Bloquea la ejecución hasta que todas las tareas finalicen o el tiempo especificado expire. Retorna true si todas las tareas han terminado, false si se agotó el tiempo. |
| submit(Runnable task) | Envía una tarea de tipo Runnable para su ejecución. Retorna un Future que puede usarse para comprobar el estado de la tarea. |
| submit(Callable<V> task) | Envía una tarea de tipo Callable (con retorno de valor) para su ejecución y retorna un Future con el resultado de la tarea. |
| invokeAll(Collection<? extends Callable<T>> tasks) | Ejecuta una colección de tareas Callable y espera hasta que todas se completen, devolviendo una lista de Future con los resultados. |
| iinvokeAny(Collection<? extends Callable<T>> tasks) | Ejecuta una colección de tareas Callable y retorna el resultado de la primera tarea que completa con éxito, cancelando las demás. |

Comparaciones entre algunos métodos.

* submit() devuelve un Future, lo que permite verificar el resultado de la tarea o manejar excepciones, mientras que execute() simplemente ejecuta la tarea sin devolver nada.
* shutdown() permite que las tareas en curso terminen de manera ordenada, mientras que shutdownNow() intenta detener todas las tareas inmediatamente.

**Ventajas y desventajas de ExecutorService(Runnable) frente a Thread y Runnable**

Al comparar el uso de ExecutorService frente al trabajo con hilos (Thread) y tareas (Runnable), que fue visto en la materia de ‘Programación Concurrente’, podemos mencionar algunas ventajas y desventajas:

Ventajas:

* Como ExecutorService utiliza pools de hilos, se evita una posible sobrecarga al crear y destruir hilos constantemente. Esto mejora la eficiencia, especialmente en aplicaciones con muchas tareas concurrentes.

Con Thread y Runnable, cada tarea requiere la creación de un nuevo hilo.

* ExecutorService gestiona automáticamente el ciclo de vida de los hilos, permitiendo una terminación ordenada de los hilos y la posibilidad de reutilizarlos.

Con Thread, el programador debe gestionar explícitamente la creación, ejecución y terminación de cada hilo.

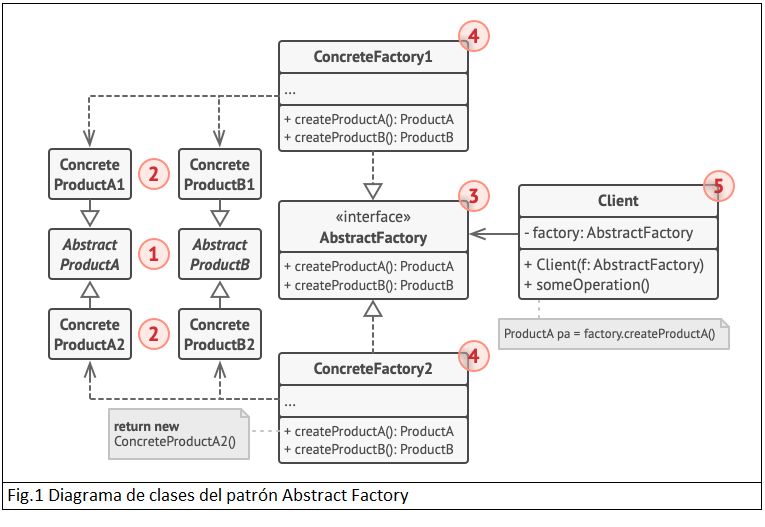
Desventajas:

* Para aplicaciones que solo necesitan ejecutar unas pocas tareas concurrentes, el uso de ExecutorService puede no ser tan conveniente, mientras que crear directamente hilos con Thread puede ser suficiente y más fácil de implementar.
* Con ExecutorService, el control sobre los hilos individuales es más limitado, ya que el pool gestiona automáticamente los hilos.

**Abstract Factory**

Abstract Factory es un patrón de diseño creacional que permite producir familias de objetos relacionados sin especificar sus clases concretas.

**Estructura**

****

En la Fig.1 se observa:

1. Los Productos Abstractos(Abstract Product) declaran interfaces para un grupo de productos diferentes pero relacionados que forman una familia de productos.
2. Los Productos Concretos(Concrete Product) son implementaciones distintas de productos abstractos agrupados por variantes. Cada producto abstracto debe implementarse en todas las variantes dadas.
3. La interfaz Fábrica Abstracta(Abstract Factory) declara un grupo de métodos para crear cada uno de los productos abstractos.
4. Las Fábricas Concretas implementan métodos de creación de la fábrica abstracta. Cada fábrica concreta se corresponde con una variante específica de los productos y crea tan sólo dichas variantes de los productos.
5. Aunque las fábricas concretas instancian productos concretos, sus métodos de creación deben devolver los productos abstractos correspondientes. De este modo, el código cliente que utiliza una fábrica no se acopla a la variante específica del producto que obtiene de una fábrica. El Cliente puede funcionar con cualquier variante fábrica/producto concreta, siempre y cuando se comunique con sus objetos a través de interfaces abstractas.

**¿Cuándo utilizar Abstract Factory?**

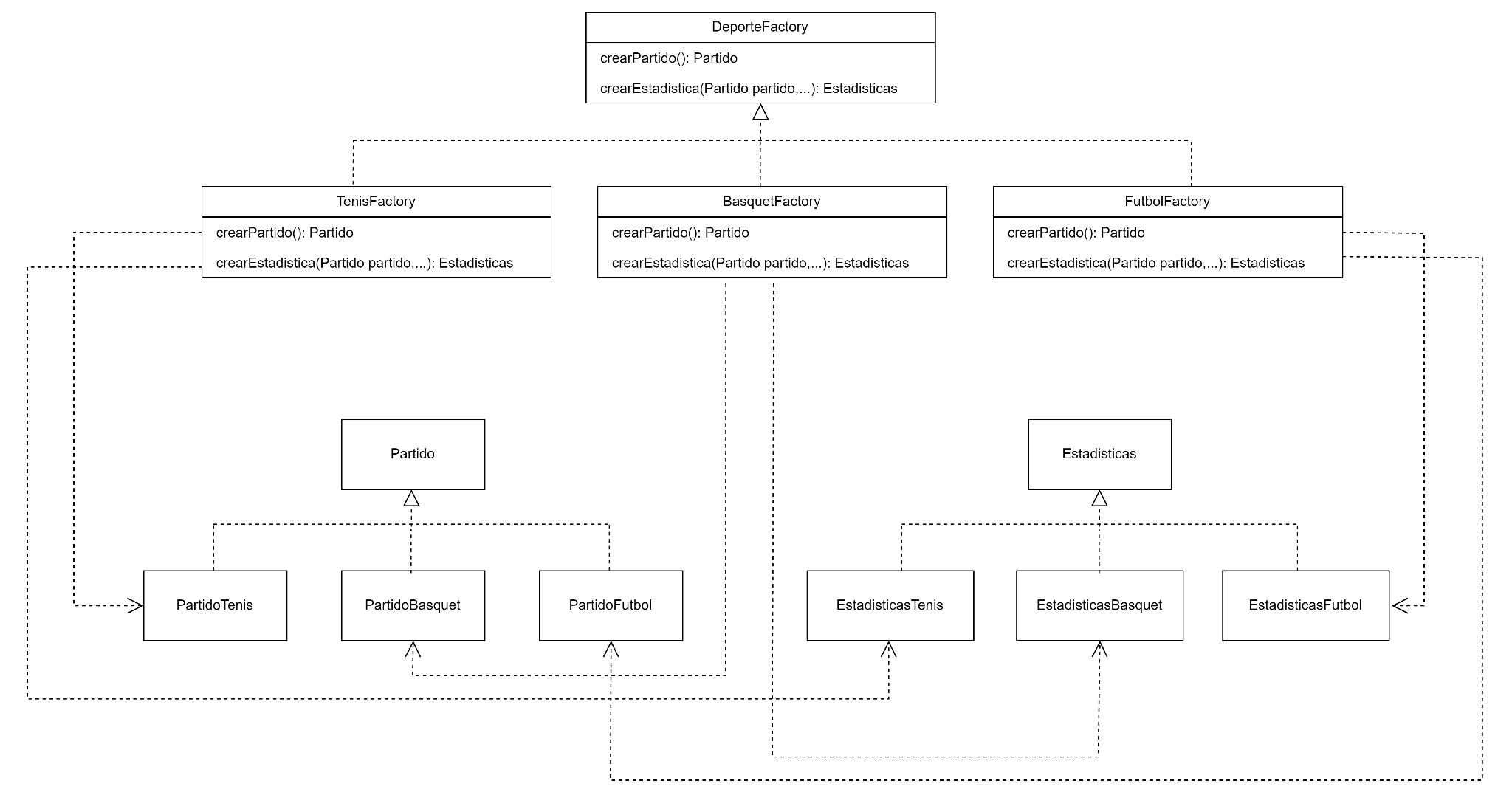
* Cuando se necesita trabajar con familias de productos relacionados y no se quiere que el código dependa de las clases concretas. En especial cuando las clases concretas no son conocidas con anterioridad o se busque escalabilidad.

**Caso de aplicación**

En el caso de aplicación que se presenta, se desarrolla un sistema para procesar y calcular estadísticas de distintos deportes. El objetivo es gestionar eficientemente la ejecución de estas tareas de cálculo de manera concurrente mediante la interfaz ExecutorService, lo que permite aprovechar los recursos del sistema a través de un pool de hilos. Cada tarea de procesamiento corresponde al cálculo de estadísticas de los partidos de un deporte específico, como fútbol, basquet o tenis.

El sistema gestiona múltiples deportes, donde cada deporte tiene datos propios, además por cada tipo de deporte existe una implementación lógica para el procesamiento de las estadísticas, dichas implementaciones son las tareas que ejecutarán los hilos.

El patrón Abstract Factory se aplicó para definir una "fábrica" de objetos que crea instancias de partidos e instancias de las clases encargadas del procesamiento de las estadísticas, dependiendo del tipo de deporte. Existen fábricas concretas para cada deporte, que son responsables de crear las instancias correspondientes.



repositorio: <https://github.com/gaston-cid/lab-prog-tp1>