Práctica 03: Academia Ninja

Equipo: SQLazo
César Becerra Valencia (322064287)
Victor Abraham Sánchez Morgado (322606003)
José Luis Cortes Nava (322115437)

21 de septiembre de 2025

Este proyecto es una solución a la práctica 03 de **Modelado y Programación** de la Facultad de Ciencias (UNAM). El objetivo es implementar un sistema para organizar grupos de entrenamiento para aspirantes a ninja, utilizando y combinando correctamente los patrones de diseño **Iterator**, **Builder** y **Factory**.

1. Descripción del Problema

La Academia Ninja necesita un sistema para formar equipos compuestos por un líder **Ninja** voluntario y un número variable de **Aspirantes**. A cada grupo formado se le debe asignar un **paquete de herramientas** (predefinido o personalizado) y un **campo de entrenamiento** específico, determinado por la suma de los niveles de habilidad de sus miembros. El sistema debe manejar dos colecciones de datos distintas: un `Array` para los ninjas y una `HashTable` para los aspirantes.

2. Prerrequisitos

Para compilar y ejecutar este proyecto, necesitas tener instalado:

- **JDK (Java Development Kit) versión 17 o superior.**

3. Cómo Compilar y Ejecutar

Puedes compilar y ejecutar el programa directamente desde la terminal usando los comandos `javac` y `java`.

Asegúrate de estar en el directorio raíz del proyecto (`Practica03_SQLazo/`) antes de ejecutar los siguientes comandos.

3.1. Compilación

El siguiente comando compilará todos los archivos `.java` que se encuentran en el directorio `src/` y dejará los archivos `.class` compilados en un nuevo directorio llamado `out/`.

```
javac -d out --source-path src src/mx/unam/ciencias/myp/Main.java
```

- `javac`: Es el compilador de Java.
- `-d out`: Le indica al compilador que coloque los archivos compilados (`.class`) en una carpeta llamada `out`.
- `--source-path src`: Especifica que el código fuente (`.java`) se encuentra en la carpeta `src`, para que pueda encontrar todas las clases necesarias.
- `src/mx/unam/ciencias/myp/Main.java`: Es el archivo que sirve como punto de entrada para la compilación.

3.2. Ejecución

Una vez compilado, puedes ejecutar el programa con el siguiente comando:

```
java -cp out mx.unam.ciencias.myp.Main
```

- `java`: Es la Máquina Virtual de Java (JVM) que ejecuta el código.
- `-cp out` (`-cp` es una abreviatura de `--class-path`): Le indica a la JVM que busque los archivos `.class` en el directorio `out`.
- `mx.unam.ciencias.myp.Main`: Es el nombre completamente calificado de la clase que contiene el método `main` que queremos ejecutar.

4. Anotaciones sobre la Implementación de Patrones

La elección y aplicación de los patrones de diseño no fue arbitraria. Cada uno resuelve un problema específico del requerimiento, buscando un diseño desacoplado, mantenible y extensible, acorde a los principios de *Clean Code* y SOLID.

4.1. Patrón Iterator

- **Problema**: Teníamos dos colecciones de participantes con estructuras internas completamente diferentes: un `Array` para los Ninjas y una `HashTable` para los Aspirantes. La lógica de formación de grupos necesitaba recorrer ambas colecciones de manera uniforme.
- **Solución y Argumento**: Se aplicó el patrón **Iterator** para **abstraer el proceso de recorrido**. Se crearon las clases `ColeccionNinjas` y `ColeccionAspirantes`, que devuelven un objeto `Iterator` compatible. Gracias a esto, la clase principal `AcademiaNinja` no necesita saber *cómo* están guardados los datos. Simplemente pide un iterador y lo usa, lo que la **desacopla de las colecciones**. Si en el futuro los aspirantes se guardaran en una base de datos, solo tendríamos que cambiar `ColeccionAspirantes`, sin tocar la lógica de negocio. Esto respeta el **Principio de Responsabilidad Única (SRP)** y el **Principio Abierto/Cerrado (OCP)**.

4.2. Patrón Builder

- **Problema**: La creación de un `PaqueteHerramientas` podía ser simple (paquetes prefabricados) o compleja (personalizada), involucrando múltiples atributos opcionales (kunais, shurikens, etc.). Usar un constructor con muchos parámetros sería confuso y propenso a errores (un Çonstructor Telescópico").
- **Solución y Argumento**: El patrón **Builder** se eligió para **simplificar la creación de objetos complejos**. Proporciona una API fluida (`.agregarKunais(2).agregarShurikens(3).build()`) que hace el proceso de construcción explícito y legible. Además, facilitó la creación de los paquetes predefinidos (`Básico`, `Avanzado`) mediante una clase `Director` (`EncargadoPaquetes`), que reutiliza el mismo proceso de construcción. Esto separa la lógica de construcción del objeto de su representación final, mejorando la modularidad.

4.3. Patrón Factory

- **Problema**: El tipo de `CampoEntrenamiento` a crear dependía de una lógica condicional basada en el nivel de habilidad total del grupo (< 7, $8 - 11$, > 12). Poner esta lógica de `if-else` o `switch` directamente en la clase `AcademiaNinja` la haría más compleja y difícil de mantener si se añadieran nuevos campos en el futuro.
- **Solución y Argumento**: Se utilizó el patrón **Factory Method** (implementado en `CampoEntrenamientoFactory`) para **encapsular la lógica de creación de los campos de entrenamiento**. La clase `AcademiaNinja` ya no es responsable de saber qué clase concreta instanciar (`new ValleDelDragon()`, etc.). Simplemente le pide a la fábrica: "dame el campo para un nivel de habilidad 10", y la fábrica se encarga de los detalles. Esto **desacopla el cliente del proceso de creación**, cumpliendo con el **SRP** y facilitando la adición de nuevos tipos de campos sin modificar el código cliente.