FCyT
Facultad de Ciencia y Tecnología

Ingeniería de Software II - FCyT

Patrones de diseño

Enunciados para Ejercicios en Aula sobre Patrones de Diseño

1. Patrón Singleton

Basándonos en los materiales proporcionados, el **Patrón Singleton** tiene como objetivo fundamental **asegurar que exista una única instancia de una clase concreta** y proveer una **accesibilidad global** a dicha instancia desde cualquier parte del programa. Para lograr esto, se implementa un **constructor privado** que impide la instanciación desde fuera de la clase, y se utiliza una **función estática** `getInstance()` que se encarga de construir la instancia internamente si no existe, o de devolver la ya creada. Un caso de uso típico es la gestión de un **archivo de logs** para asegurar que no haya múltiples instancias escribiendo simultáneamente en el mismo archivo.

Enunciado para ejercitar: Considerando la necesidad de gestionar un recurso único y centralizado en una aplicación (como un archivo de logs o una configuración global), **explica por qué el Patrón Singleton es la solución adecuada para garantizar que solo exista una instancia de la clase** que maneja dicho recurso. Detalla cómo la implementación de un **constructor privado y un método** `getInstance()` contribuyen a alcanzar este objetivo, y qué problemas se evitarían al aplicar este patrón. Representa en UML el enunciado

2. Patrón Factory

El **Patrón Factory** es un patrón creacional que permite **encapsular la lógica de creación de objetos**, especialmente cuando se necesita generar diferentes tipos de objetos de forma flexible y polimórfica. Se utiliza una **clase abstracta (o interfaz)** `Creator` con un método de creación (por ejemplo, `CreateEnemy()` en el contexto de un videojuego) del cual heredan **clases** `ConcreteCreator`. Esto permite que el código que solicita los objetos trabaje siempre con la **interfaz o clase padre**, y no con las clases concretas, facilitando el uso del **polimorfismo**. Así, se pueden generar objetos de forma aleatoria o bajo diferentes estrategias (ej., `RandomEnemyFactory` o `GoombaFactory`) **sin afectar otras partes del código**.

Enunciado para ejercitar: Imagina que estás desarrollando un videojuego donde necesitas generar diferentes tipos de enemigos con diversas dificultades que sorprendan al jugador. **Explica cómo el Patrón Factory te permitiría generar estos enemigos de forma aleatoria o específica** (por ejemplo, solo enemigos poderosos) **sin la necesidad de instanciar directamente cada clase concreta de enemigo** (ej. `new Koopa()`, `new Goomba()`) en el código principal. Argumenta por qué esta encapsulación de la lógica de creación mejora la

flexibilidad y la mantenibilidad de tu código, especialmente al añadir nuevos tipos de enemigos o estrategias de creación. Representa en UML el enunciado

3. Patrón Abstract Factory

El **Patrón Abstract Factory** es un patrón de diseño creacional que se distingue por permitir **producir familias de objetos relacionados sin especificar sus clases concretas**. A diferencia del Patrón Factory simple que suele tener un único método de creación, la Abstract Factory cuenta con **varios métodos que crean diferentes tipos de objetos relacionados**. Este patrón resuelve el problema de la **proliferación de fábricas individuales** por cada elemento (como bloques y monedas en un juego), y previene que se utilicen objetos que no deben ir juntos (ej. un enemigo 2D en un estilo 3D). Permite trabajar siempre con **clases abstractas**, independientemente de las implementaciones concretas, aplicando el **polimorfismo en su máxima expresión**. Un ejemplo claro es la creación de elementos de juego (monedas, bloques) que comparten un mismo "estilo" o "tema" (ej. `GameBoyItemFactory` o `NintendoDSItemFactory`).

Enunciado para ejercitar: En un juego como Super Mario Maker, donde los jugadores pueden construir niveles con diferentes estilos visuales (como GameBoy o NintendoDS), es fundamental que todos los elementos (monedas, bloques, enemigos) del nivel **correspondan coherentemente al estilo seleccionado**. **Explica por qué el Patrón Abstract Factory sería la elección más adecuada** para gestionar la creación de estos elementos en lugar de utilizar múltiples Patrones Factory individuales para cada tipo de objeto (una Factory para monedas, otra para bloques, etc.). Detalla cómo Abstract Factory **garantiza que siempre se creen "familias de objetos relacionados"** y cómo esto **protege de utilizar objetos incompatibles** o de un estilo incorrecto, aplicando el polimorfismo en su máxima expresión. Representa en UML el enunciado