Esonero Pattern Creazionali

1. Quale/i tra i seguenti Design Pattern è un pattern di classe?

   a Abstract Factory
   b Factory Method
   c Abstract Class

Nel contesto dei design pattern in ingegneria del software, i pattern di classe si riferiscono a quei pattern che utilizzano l'ereditarietà per comporre interfacce o implementazioni. La chiave è l'uso della ereditarietà e delle classi per definire e implementare le parti variabili di un algoritmo.

Qui abbiamo tre opzioni:

- a) **Abstract Factory**: Questo è un design pattern creazionale che fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete. L'Abstract Factory è più riguardante la creazione di oggetti, ma non è definito come un pattern di classe perché non si basa principalmente sull'ereditarietà tra classi per la sua struttura.
- b) **Factory Method**: Questo è un pattern creazionale che definisce un'interfaccia per creare un oggetto, ma lascia che le sottoclassi decidano quale classe istanziare. Il Factory Method ritarda l'istanziazione a sottoclassi. Questo è un esempio classico di un pattern di classe perché utilizza l'ereditarietà per variare la classe dell'oggetto che viene creato.
- c) **Abstract Class**: Non è propriamente un design pattern, ma piuttosto un concetto fondamentale nella programmazione orientata agli oggetti. Le classi astratte sono usate come basi per sottoclassi, ma non possono essere istanziate da sole. Nonostante sia strettamente legata all'uso di classi, di per sé non costituisce un pattern di design.

Risposta corretta:

La **b) Factory Method**, perché è l'unico tra quelli elencati che è un pattern di classe nel vero senso del termine.

2. Quale/i tra i seguenti Design Patterns permette di configurare in modo dinamico (a tempo di esecuzione) e virtualmente senza limiti la tipologia di oggetti da creare?

   a Abstract Factory
   b Prototype
   c Builder

- a) **Abstract Factory**: Questo pattern fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete. È utilizzato quando il sistema deve essere indipendente da come i suoi prodotti sono creati, composti e rappresentati e quando le famiglie di prodotti devono essere configurate o quando si vuole fornire una libreria di prodotti senza esporre l'implementazione.
- b) **Prototype**: Questo pattern è utilizzato quando la creazione di un'istanza è più conveniente clonando un prototipo esistente piuttosto che creare un nuovo oggetto da zero, specialmente quando si tratta di istanze complesse o costose da creare. Il pattern Prototype permette di copiare oggetti esistenti senza rendere il codice dipendente dalle loro classi. Questo approccio consente di configurare dinamicamente la tipologia di oggetti da creare a runtime tramite la clonazione di oggetti.
- c) **Builder**: Questo pattern separa la costruzione di un oggetto complesso dalla sua rappresentazione, in modo che lo stesso processo di costruzione possa creare diverse rappresentazioni. Viene utilizzato quando un algoritmo dovrebbe essere indipendente dai componenti dell'oggetto che costruisce e quando i dettagli di costruzione di un oggetto devono essere isolati dalla sua rappresentazione.

Risposta corretta:

Tra le opzioni fornite, il pattern **b) Prototype** è quello che meglio si adatta alla descrizione di configurare dinamicamente e virtualmente senza limiti la tipologia di oggetti, grazie alla sua capacità di clonare oggetti esistenti. Questo permette di avere una grande flessibilità nel creare nuove istanze di oggetti a tempo di esecuzione.

3. Quale/i tra i seguenti Design Patterns permette di gestire famiglie paral-lele di oggetti promuovendo la coerenza nell'appartenza degli oggetti alla famiglia scelta?

   a Abstract Factory
   b Prototype
   c Builder

- a) **Abstract Factory** è un design pattern creazionale che fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete. Questo pattern è utile quando un sistema deve essere indipendente dall'implementazione degli oggetti che utilizza, o quando il sistema deve essere configurato con una di più famiglie di oggetti. Promuove la coerenza tra i prodotti creando una famiglia di oggetti correlati senza dover dipendere dalle loro classi concrete.
- b) **Prototype** è un design pattern creazionale che viene utilizzato quando il tipo di oggetti da creare è determinato da un'istanza prototipica, che viene clonata per produrre nuovi oggetti. Questo pattern è particolarmente utile quando la creazione di un oggetto è più conveniente o più efficiente attraverso il processo di clonazione piuttosto che attraverso la creazione ex novo. Non si occupa direttamente della coerenza di una famiglia di prodotti.
- c) **Builder** è un altro design pattern creazionale che consente di costruire oggetti complessi passo dopo passo. Il pattern Builder separa la costruzione di un oggetto complesso dalla sua rappresentazione, in modo che lo stesso processo di costruzione possa creare diverse rappresentazioni. Questo pattern è utilizzato quando la costruzione di un oggetto deve consentire diverse rappresentazioni dell'oggetto che viene costruito. Non è specificamente pensato per mantenere la coerenza tra una famiglia di oggetti.

Risposta corretta:

La risposta corretta alla domanda fornita è quindi la **a) Abstract Factory**. Questo pattern è specificamente progettato per gestire e mantenere famiglie di oggetti correlati assicurando la loro coerenza e appartenenza alla famiglia scelta.

4. Quale/i tra i seguenti pattern è meno significativo in ambienti di programmazione dinamici come java?

   a Abstract Factory
   b Prototype
   c Singleton

- a) **Abstract Factory** è un design pattern creazionale molto significativo in Java, in quanto fornisce un modo per incapsulare un gruppo di singole factory che hanno un tema comune senza specificare le loro classi concrete. È utilizzato ampiamente per sostenere la coesione e il principio di inversione delle dipendenze, specialmente in applicazioni complesse dove si possono avere molteplici varianti di prodotti correlati.
- b) **Prototype** non è comunemente necessario in Java come in altri linguaggi che utilizzano la programmazione basata sui prototipi (come JavaScript). Java è un linguaggio che supporta la programmazione orientata agli oggetti con classi, e di solito, gli oggetti vengono creati tramite istanziazione di classi piuttosto che clonazione di oggetti. Tuttavia, può essere utile in situazioni in cui è richiesta la creazione di oggetti duplicati che sono costosi da creare.
- c) **Singleton** è utilizzato in Java per assicurarsi che una classe abbia una sola istanza e per fornire un punto di accesso globale a quella istanza. È un pattern significativo e ampiamente utilizzato in molte applicazioni Java, soprattutto quando si deve garantire che una risorsa condivisa (come una connessione al database) sia unica e accessibile globalmente.

Risposta corretta:

In ambienti di programmazione dinamici come Java, il pattern che tende ad essere meno significativo è il **b) Prototype**, a causa dell'incoraggiamento dell'uso di classi e della creazione di oggetti attraverso l'istanziazione anziché la clonazione. Mentre Abstract Factory e Singleton hanno applicazioni dirette e comuni in Java, Prototype è meno utilizzato in quanto Java non è basato sui prototipi e fornisce altri meccanismi per la creazione e gestione degli oggetti.

5. Quale/i dei seguenti pattern consente di separare il processo di creazione di un oggetto dalla sua rappresentazione?

   a Abstract Factory
   b Builder
   c Factory Method

- a) **Abstract Factory** è un pattern creazionale che fornisce un'interfaccia per creare famiglie di oggetti correlati senza specificare le loro classi concrete. L'Abstract Factory è utile per incapsulare un gruppo di singole factory che hanno un tema comune. Tuttavia, non separa il processo di creazione di un oggetto dalla sua rappresentazione; piuttosto, separa l'uso di un prodotto dalla sua implementazione.
- b) **Builder** è un design pattern che separa la costruzione di un oggetto complesso dalla sua rappresentazione, in modo tale che lo stesso processo di costruzione possa creare diverse rappresentazioni. Il cliente costruisce un oggetto complesso passo dopo passo, specificando il tipo e il contenuto, e il builder si occupa della costruzione e dell'assemblaggio. Questo è il pattern che consente la massima separazione tra il processo di creazione e la rappresentazione dell'oggetto.
- c) **Factory Method** è un pattern creazionale che fornisce un'interfaccia per la creazione di un oggetto, ma lascia che le sottoclassi decidano quale oggetto istanziare. Questo pattern è più incentrato sul permettere alle sottoclassi di estendere la creazione di oggetti senza cambiarne il codice. Non separa il processo di creazione dalla rappresentazione come fa il Builder.

Risposta corretta:

La risposta corretta è quindi **b) Builder**. Questo pattern è progettato per gestire costruzioni complesse e per separare la costruzione dell'oggetto dalla sua rappresentazione finale, consentendo così di produrre diverse rappresentazioni a partire da lo stesso processo di costruzione.

6. Quale/i dei seguenti pattern consente di fornire un punto di accesso univoco ad una struttura dati?

   a Abstract Factory
   b Singleton
   c Factory Method

La domanda richiede di identificare quale design pattern fornisce un punto di accesso univoco ad una struttura dati.

- a) **Abstract Factory** non è progettato per fornire un punto di accesso univoco ad una struttura dati. L'Abstract Factory consente di creare famiglie di oggetti correlati senza specificare la loro classe concreta. Si tratta di un insieme di Factory Methods che lavorano insieme per produrre un set di oggetti correlati.
- b) **Singleton** è il pattern che mira specificamente a garantire che una classe abbia una sola istanza e fornisce un punto di accesso globale a tale istanza. Viene comunemente utilizzato per gestire risorse condivise, come una connessione al database o un file di configurazione. In questo modo, tutte le parti del programma possono accedere alla stessa istanza della struttura dati attraverso il Singleton.
- c) **Factory Method** è un pattern che permette di definire un'interfaccia per creare un oggetto, ma lascia che le classi sottostanti decidano quale classe istanziare. Questo pattern è utilizzato per delegare la responsabilità di creazione degli oggetti alle sottoclassi. Non fornisce di per sé un punto di accesso univoco ad una struttura dati.

Risposta corretta:

La risposta corretta è **b) Singleton**. Questo design pattern è specificamente progettato per assicurare che una classe abbia una sola istanza e per fornire un punto di accesso globale a questa istanza, che può essere una struttura dati.

7. Quale/i dei seguenti pattern è basato principalmente sull'ereditarietà?

   a Factory Method
   b Abstract Factory
   c Prototype

- a) **Factory Method** è un pattern creazionale che utilizza l'ereditarietà e si basa sull'idea di utilizzare una funzione membro - il "factory method" - per creare oggetti. In questo pattern, una classe non crea istanze di classi direttamente, ma chiama un metodo factory che è destinato ad essere sovrascritto (overridden) dalle sottoclassi se è necessario modificare il tipo di oggetto che verrà creato. Pertanto, il Factory Method è strettamente legato all'ereditarietà poiché le sottoclassi implementano il metodo factory per creare diverse istanze di oggetti.
- b) **Abstract Factory** estende il concetto del Factory Method. È una fabbrica di fabbriche; un pattern che fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete. Anche se utilizza l'ereditarietà per permettere diverse implementazioni dell'interfaccia della fabbrica astratta, il focus principale non è l'ereditarietà in sé, ma piuttosto l'astrazione e la fornitura di un'interfaccia coerente per famiglie di prodotti.
- c) **Prototype** non si basa sull'ereditarietà, ma piuttosto sull'incapsulamento e sul clonare oggetti. Un oggetto prototipo viene creato e può essere clonato per produrre nuove istanze piuttosto che istanziare classi. Questo pattern è particolarmente utile quando la creazione di un'istanza di una classe è costosa o complessa. Con il Prototype, è possibile copiare un esistente oggetto invece di creare uno nuovo dal nulla, evitando così tutto il costo associato alla nuova istanziazione.

Risposta corretta:

La risposta corretta è **a) Factory Method** perché è il pattern che si basa maggiormente sull'ereditarietà, permettendo alle sottoclassi di alterare il tipo di oggetti che verranno creati.

8. Quale/i dei seguenti pattern non richiede necessariamente l'esistenza di sotto-classi?

   a Factory Method
   b Abstract Factory
   c Prototype

- La domanda si riferisce ai design pattern e alla loro relazione con l'ereditarietà o l'uso di sottoclassi.
- a) **Factory Method** richiede l'uso di sottoclassi. Il Factory Method funziona attraverso l'ereditarietà: una classe astratta fornisce l'interfaccia con un metodo di fabbrica astratto, e le sottoclassi concrete implementano questo metodo per creare oggetti. L'idea è di permettere alle sottoclassi di modificare il tipo di oggetti che verranno creati.
- b) **Abstract Factory** può implicare l'uso di sottoclassi nel senso che diverse fabbriche concrete possono ereditare da una interfaccia o classe astratta di fabbrica. Tuttavia, il pattern Abstract Factory si focalizza più sull'offrire un'interfaccia per creare famiglie di oggetti correlati senza specificare le loro classi concrete. L'ereditarietà qui non è usata per creare oggetti, ma per creare famiglie di oggetti attraverso l'interfaccia comune.
- c) **Prototype** non si basa sull'ereditarietà o sulle sottoclassi. Invece, crea nuovi oggetti clonando un'istanza esistente. Con il pattern Prototype, un oggetto è in grado di produrre copie di se stesso senza che sia necessaria alcuna ereditarietà. È particolarmente utile in ambienti di programmazione dinamica dove gli oggetti possono non essere creati attraverso l'ereditarietà, ma attraverso il processo di clonazione.

Risposta corretta:

La risposta corretta è **c) Prototype**. Questo design pattern consente la creazione di nuovi oggetti senza la necessità di sottoclassi, utilizzando la clonazione di un oggetto esistente come meccanismo per la creazione di nuove istanze.

9. Quale/i dei seguenti pattern può essere utilizzato per realizzare una fabbrica?

   a Factory Method
   b Builder
   c Prototype

- a) **Factory Method** è un design pattern creazionale che fornisce un'interfaccia per la creazione di oggetti, ma lascia che le sottoclassi alterino il tipo di oggetti che verranno creati. In questo modo, le sottoclassi possono estendere la classe base e fornire l'oggetto concreto da creare. È un modo per creare un "framework" che deleghi la decisione di quali oggetti creare alle sottoclassi. Questo pattern può essere effettivamente considerato una fabbrica perché è specializzato nella creazione di oggetti.
- b) **Builder** è un pattern utilizzato per costruire un oggetto complesso passo dopo passo. A differenza di una fabbrica che produce prodotti finiti (oggetti), il Builder si concentra sulla costruzione di un oggetto complesso seguendo alcuni passaggi. Non è una fabbrica nel senso convenzionale perché non astrae e incapsula la creazione di una serie di oggetti intercambiabili.
- c) **Prototype** è un design pattern che viene utilizzato quando la creazione di un oggetto è più conveniente o più efficiente attraverso il processo di clonazione di un oggetto esistente. Questo pattern non stabilisce una fabbrica in quanto non fornisce un'interfaccia per la creazione di diversi tipi di oggetti. Invece, si basa su un'istanza esistente per produrre nuovi oggetti clonati.

Risposta corretta:

La risposta corretta è **a) Factory Method**. Questo pattern è il più vicino al concetto di una "fabbrica" nel contesto dei design pattern, poiché astrae il processo di creazione degli oggetti e permette che la creazione di oggetti specifici sia gestita dalle sottoclassi.

10. Indicare il/i pattern più adatto/i a modellare il seguente scenario indicandone il ruolo: nell'ambito di realizzazione di un sistema per l'editing di diagrammi si vuole fornire la possibilità di salvare un diagramma, stampare il diagramma, inviare il diagramma via email, esportare il diagramma come immagine generalizzando il processo che ottiene l'output desiderato a partire dalla modellazione astratta del diagramma stesso:

    a Abstract Factory
    b Factory Method
    c Builder
    d Singleton
    e Prototype

- a) **Abstract Factory** è utilizzato per creare famiglie di oggetti correlati senza specificarne la classe concreta. Potrebbe essere utilizzato in un editor di diagrammi per creare vari tipi di elementi di diagramma che hanno un'interfaccia comune. Tuttavia, non gestisce direttamente operazioni di I/O come il salvataggio o l'invio via email.
- b) **Factory Method** consente di creare oggetti lasciando che le sottoclassi decidano quali oggetti creare. Potrebbe essere utile per definire un'interfaccia di esportazione per i diagrammi e lasciare che sottoclassi specifiche implementino i vari formati di esportazione. Tuttavia, non gestisce per sé il flusso completo di operazioni richieste.
- c) **Builder** è adatto alla costruzione di oggetti complessi passo dopo passo. Potrebbe essere utilizzato per costruire o assemblare un diagramma complesso da vari elementi. Potrebbe anche essere esteso per "costruire" diverse rappresentazioni del diagramma, come l'output per la stampa o per l'esportazione, ma non è specificamente progettato per le operazioni di I/O o per la manipolazione di entità dopo la loro creazione.
- d) **Singleton** è utilizzato per garantire che una classe abbia una sola istanza e per fornire un punto di accesso globale a tale istanza. Questo potrebbe essere utile per gestire una configurazione globale del sistema di editing, o per gestire un servizio di esportazione centralizzato, ma non modella il flusso completo di operazioni.
- e) **Prototype** consente di creare nuovi oggetti clonando un oggetto esistente. Questo potrebbe essere utilizzato per duplicare diagrammi o parti di essi rapidamente. Tuttavia, non gestisce il processo di I/O o le varie operazioni richieste nello scenario descritto.

Risposta corretta:

Nessuno dei pattern elencati si adatta perfettamente da solo a modellare l'intero scenario descritto, che richiede una combinazione di creazione, gestione e operazioni di I/O. Tuttavia, il **Builder** potrebbe essere il più vicino in termini di modellare la creazione di diverse rappresentazioni di un diagramma, se si considera "costruire" l'output come parte del processo. Il Builder potrebbe essere utilizzato per definire il processo di costruzione delle rappresentazioni dei diagrammi e quindi estenderlo per includere diverse operazioni come salvataggio, stampa, invio via email e esportazione. Per le operazioni di gestione singola istanza, come il mantenimento di una configurazione globale o di un servizio di esportazione, potrebbe essere utilizzato in combinazione con il Singleton.

11. Implementare in Java o in C++ una fabbrica dei prototipi con quattro prodotti di cui esistono diverse varianti (dei prodotti riportare le sole interfacce), la fabbrica deve essere un singleton.

```
import java.util.HashMap;
import java.util.Map;

// Interfaccia per i Prototipi
interface Prototype {
    Prototype clone();
    void setType(String type);
    String getType();
}

// Classe concreta che implementa l'interfaccia Prototype
class ConcretePrototype implements Prototype {
    private String type;

    public ConcretePrototype(String type) {
        this.type = type;
    }

    @Override
    public Prototype clone() {
        return new ConcretePrototype(this.type);
    }

    @Override
    public void setType(String type) {
        this.type = type;
    }

    @Override
    public String getType() {
        return type;
    }
}

// Singleton Factory
class PrototypeFactory {
    private static PrototypeFactory instance;
    private Map<String, Prototype> prototypes = new HashMap<>();

    private PrototypeFactory() {
        // Inizializza i prototipi
        prototypes.put("Type1", new ConcretePrototype("Type1"));
        prototypes.put("Type2", new ConcretePrototype("Type2"));
        prototypes.put("Type3", new ConcretePrototype("Type3"));
        prototypes.put("Type4", new ConcretePrototype("Type4"));
    }

    public static synchronized PrototypeFactory getInstance() {
        if (instance == null) {
            instance = new PrototypeFactory();
        }
        return instance;
    }

    public Prototype getPrototype(String type) {
        try {
            return prototypes.get(type).clone();
        } catch (NullPointerException e) {
            System.out.println("Prototype with type: " + type + " doesn't exist");
            return null;
        }
    }
}

public class PrototypePatternExample {
    public static void main(String[] args) {
        PrototypeFactory factory = PrototypeFactory.getInstance();

        Prototype prototype1 = factory.getPrototype("Type1");
        System.out.println(prototype1.getType());

        Prototype prototype2 = factory.getPrototype("Type2");
        System.out.println(prototype2.getType());

        // Il client può continuare a creare prototipi come desiderato
    }
}
```

- L'interfaccia Prototype definisce le operazioni necessarie per i prototipi, ovvero clone() e getter/setter per il tipo.
- ConcretePrototype è una classe concreta che implementa Prototype, fornendo una copia superficiale di se stessa tramite il metodo clone().
- PrototypeFactory è un singleton che contiene una mappa di prototipi pre-inizializzati. Usa un blocco di inizializzazione privato per inserire i prototipi con tipi predefiniti.
- Il metodo getInstance() è sincronizzato per evitare problemi di multithreading e assicura che solo una istanza di PrototypeFactory sia creata.
- Il metodo getPrototype() cerca nella mappa un prototipo del tipo richiesto e restituisce una clonazione di quell'oggetto.

Questo design soddisfa il requisito di avere una fabbrica di prototipi come singleton che può produrre diverse varianti di prodotti (prototipi). Ogni volta che un client richiede un prototipo, ottiene una copia fresca da utilizzare, garantendo che l'oggetto originale rimanga inalterato.

12) Dato il seguente codice, identificare quale/i patterns sono stati utilizzati:

```
    public interface Battleship {
        String getDescription();
    }

    public interface Leader {
        String getDescription();
    }

    public interface Army {
        String getDescription();
    }
    // Terran implementations
    public class TerranBattleship implements Battleship

    @Override
    public String getDescription() {
    ...
    }

    public class TerranLeader implements Leader{
        @Override
        public String getDescription() {
        ...
        }

        public class TerranArmy implements Army{
        @Override
        public String getDescription() {
        ...
        }
        public class ProtossBattleship implements Battleship {}
        public class ZergBattleship implements Battleship {}
        public class ProtossLeaderfimplements Leader {}
        public class ZergLeader implements Leader {}
        public class ProtossArmy implements Army {}
        public class ZergArmy implements Army {}
        }
    }
```

```
    public interface SpeciesCreator {
        Battleship createBattleship();
        Leader createLeadero();
        Army createArmy();
    }

    public class TerranSpecieCreator implements SpeciesCreator {
        @Override
        public Battleship createBattleship(){
            return new TerranBattleship();
        }
        @Override
        public Leader createleader(){
            return new TerranLeader();
        }
        @Override
        public Army createArmy(){
            return new TerranArmy();
        }

        public class ProtossSpecieCreator implements SpeciesCreator{
            //Protoss implementation, like Terran implementation above
        }

        public class ZergSpecieCreator implements SpeciesCreator {
            // Zerg implementation, like Terran implementation above
        }
```

```
    SpeciesCreator creator = new ProtossSpecieCreator();
    Battleship ship = creator.createBattleship();
    Leader boss = creator.createLeader();
    Army army = creator.createArmy();
```

Il codice presentato illustra l'uso del pattern di design Abstract Factory. Vediamo come questo si applichi ai diversi segmenti del codice:

Analisi del Pattern di Design

- **Abstract Factory:**

Interfacce di Prodotti Astratti: Le interfacce Battleship, Leader e Army agiscono come prodotti astratti, definendo le operazioni comuni che le varie implementazioni devono fornire.

Concrete Products: Le classi come TerranBattleship, ProtossBattleship, ZergBattleship, e analoghe implementazioni per Leader e Army per le diverse specie, sono i prodotti concreti. Queste classi implementano le interfacce di prodotto astratto.

Abstract Factory Interface: SpeciesCreator funge da interfaccia per la factory astratta, definendo i metodi per la creazione dei vari prodotti (navi da battaglia, leader, eserciti).

Concrete Factories: Classi come TerranSpecieCreator, ProtossSpecieCreator, e ZergSpecieCreator sono le factory concrete, implementando l'interfaccia SpeciesCreator e producendo i prodotti concreti specifici per ciascuna specie.

- **Singleton:**

  Non viene utilizzato in questo codice. Il pattern Singleton riguarda la creazione di una singola istanza di una classe in tutto il programma, che non è presente qui.

- **Factory Method:**

  Anche se presenta similitudini con il pattern Factory Method, in quanto entrambi coinvolgono la creazione di oggetti, il codice esemplifica più chiaramente l'Abstract Factory. Il Factory Method di solito coinvolge una singola interfaccia per la creazione di un tipo di oggetto, non un insieme di prodotti correlati come nel caso dell'Abstract Factory.

- **Builder:**

  Non viene utilizzato. Il Builder è focalizzato sulla costruzione di oggetti complessi passo dopo passo, e non è evidente in questo codice.

- **Prototype:**

  Non è presente nel codice. Il Prototype pattern riguarda la creazione di nuovi oggetti tramite la clonazione di un'istanza prototipo, che non è illustrato qui.

Conclusione

Il codice utilizza il pattern Abstract Factory per creare famiglie di oggetti correlati (come Battleship, Leader, Army) senza specificare le loro classi concrete, permettendo flessibilità e mantenendo separati la creazione degli oggetti dalla loro utilizzazione.

13) Dato il seguente codice, identificare quale/i patterns sono stati utilizzati:

```
    public final class SpringFrameworkController {
        private static SpringFrameworkController controller,

        private SpringFrameworkController() {

        public static SpringFrameworkController getlnstance(){
            if(controller == null) {
                controller==new SpringFrameworkController():}
            return controller;
        }
        public Response createResponse(){
            return new Response();
        }
        // getters and setters
        }
    }
```

Singleton

Istanza Unica: Il campo private static SpringFrameworkController controller è un esempio di una variabile statica usata per tenere traccia dell'unica istanza della classe SpringFrameworkController.

Costruttore Privato: Il costruttore private SpringFrameworkController() impedisce l'istanziazione diretta della classe da parte di altre classi, che è un tratto distintivo del pattern Singleton.

Metodo di Accesso Globale: Il metodo public static SpringFrameworkController getInstance() fornisce un modo globale per accedere all'istanza unica della classe. Se l'istanza non esiste, ne viene creata una nuova; altrimenti, viene restituita l'istanza esistente.

- **Abstract Factory:**

  Non è presente. L'Abstract Factory è un pattern per creare famiglie di oggetti correlati senza specificarne le classi concrete. Il codice non mostra questo comportamento.

- **Factory Method:**

  Il metodo createResponse() potrebbe sembrare un Factory Method, ma il suo scopo principale non è quello di creare oggetti attraverso un'interfaccia in una sottoclasse. È più un semplice metodo di istanza che crea e ritorna un nuovo oggetto Response.

- **Builder:**

  Non è utilizzato. Il Builder pattern aiuta nella costruzione di oggetti complessi passo dopo passo, che non è dimostrato in questo codice.

- **Prototype:**

  Non è presente nel codice. Il Prototype pattern riguarda la clonazione di oggetti, che non viene mostrato qui.

Conclusione

Quindi, il codice fornito è un esempio del pattern Singleton, utilizzato per garantire che ci sia una sola istanza di una classe in un'applicazione con un punto di accesso globale a tale istanza.