Design Pattern

- I design pattern sono strutture software (ovvero micro-architetture) per un piccolo numero di classi che descrivono soluzioni di successo per problemi ricorrenti
  - Tali micro-architetture specificano le diverse classi ed oggetti coinvolti e le loro interazioni
- Si mira a riusare un insieme di classi, ovvero la soluzione ad un certo problema ricorrente, che spesso è costituita da più di una classe
- Durante la progettazione, le conseguenze sulle classi di varie scelte potrebbero non essere note, e le classi potrebbero diventare difficili da riusare o non esibire alcune proprietà
- Esistono tanti cataloghi di design pattern, per vari contesti
- Sistemi centralizzati, concorrenti, distribuiti, real-time, etc.

Design Pattern

- Un design pattern descrive un problema di progettazione ricorrente che si incontra in specifici contesti e presenta una soluzione collaudata generica ma specializzabile
- Documentano soluzioni già applicate che si sono rivelate di successo per certi problemi e che si sono evolute nel tempo
- Aiutano i principianti ad agire come se fossero esperti
- Supportano gli esperti nella progettazione su grande scala
- Evitano di re-inventare concetti e soluzioni, riducendo il costo
- Forniscono un vocabolario comune e permettono una comprensione dei principi del design
- Analizzano le loro proprietà non-funzionali: ovvero, come una funzionalità è realizzata, es. affidabilità, modificabilità, sicurezza, testabilità, riuso

Descrizione Di Un Pattern

- Un design pattern nomina, astrae ed identifica gli aspetti chiave di un problema di progettazione, le classi e le istanze che vi partecipano, i loro ruoli e come collaborano, ovvero la distribuzione delle responsabilità
- La descrizione include cinque parti fondamentali
- **Nome**: permette di identificare il design pattern con una parola e di lavorare con un alto livello di astrazione, indica lo scopo del pattern
- **Intento**: descrive brevemente le funzionalità e lo scopo
- **Problema** (Motivazione + Applicabilità): descrive il problema a cui il pattern è applicato e le condizioni necessarie per applicarlo
- **Soluzione**: descrive gli elementi (classi) che costituiscono il design pattern, le loro responsabilità e le loro relazioni
- **Conseguenze**: indicano risultati, compromessi, vantaggi e svantaggi nell'uso del design pattern

Descrizione

- Nella sezione Problema della descrizione di un design pattern si parla di forze (come in fisica). Ovvero, le forze sono obiettivi e vincoli, spesso contrastanti, che si incontrano nel contesto di quel design pattern
- Altre parti della descrizione di un design pattern possono essere
- Esempi di utilizzo: illustrano dove il design pattern è stato usato
- Codice: fornisce porzioni di codice che lo implementano

Organizzazione

- I design pattern sono organizzati sul catalogo (libro GoF) in base allo scopo
- **Creazionali**: riguardano la creazione di istanze
  - Singleton, Factory Method, Abstract Factory, Builder, Prototype
- **Strutturali**: riguardano la scelta della struttura
  - Adapter, Facade, Composite, Decorator, Bridge, Flyweight, Proxy
- **Comportamentali**: riguardano la scelta dell'incapsulamento di algoritmi
  - Iterator, Template Method, Mediator, Observer, State, Strategy, Chain of Responsibility, Command, Interpreter, Memento, Visitor

Design Pattern Creazionali

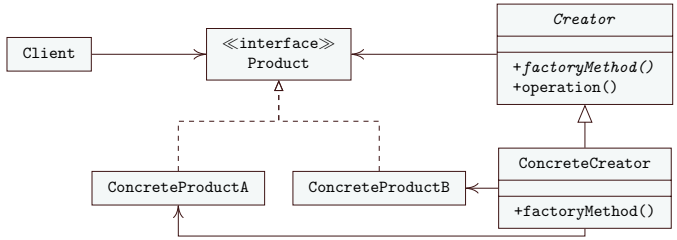
- Permettono di astrarre il processo di creazione oggetti: rendono un sistema indipendente da come i suoi oggetti sono creati, composti, e rappresentati
- Sono importanti se i sistemi evolvono per dipendere più su composizioni di oggetti che su ereditarietà tra classi
  - L'enfasi va dal codificare un insieme fissato di comportamenti verso un più piccolo insieme di comportamenti fondamentali componibili
- Incapsulano conoscenza sulle classi concrete che un sistema usa
- Nascondono come le istanze delle classi sono create e composte

Factory Method

- Intento
  - Definire una interfaccia per creare un oggetto, ma lasciare che le sottoclassi decidano quale classe istanziare. Factory Method permette ad una classe di rimandare l'istanziazione alle sottoclassi
- Problema
  - Un framework usa classi astratte per definire e mantenere relazioni tra oggetti. Il framework deve creare oggetti ma conosce solo classi astratte che non può istanziare
  - Un metodo responsabile per l'istanziazione (detto factory, ovvero fabbricatore) incapsula la conoscenza su quale classe creare

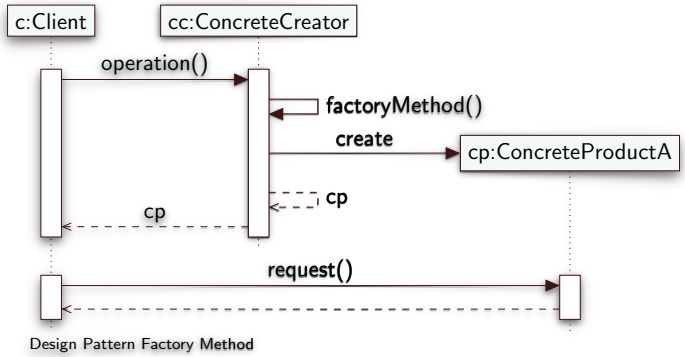
Factory Method

- Soluzione
  - **Product** è l'interfaccia comune degli oggetti creati da factoryMethod()
  - **ConcreteProduct** è un'implementazione di Product
  - **Creator** dichiara il factoryMethod(), quest'ultimo ritorna un oggetto di tipo Product. Creator può avere un'implementazione si default del factoryMethod() che ritorna un certo ConcreteProduct
  - **ConcreteCreator** implementa il factoryMethod(), o ne fa override, sceglie quale ConcreteProduct istanziare e ritorna tale istanza



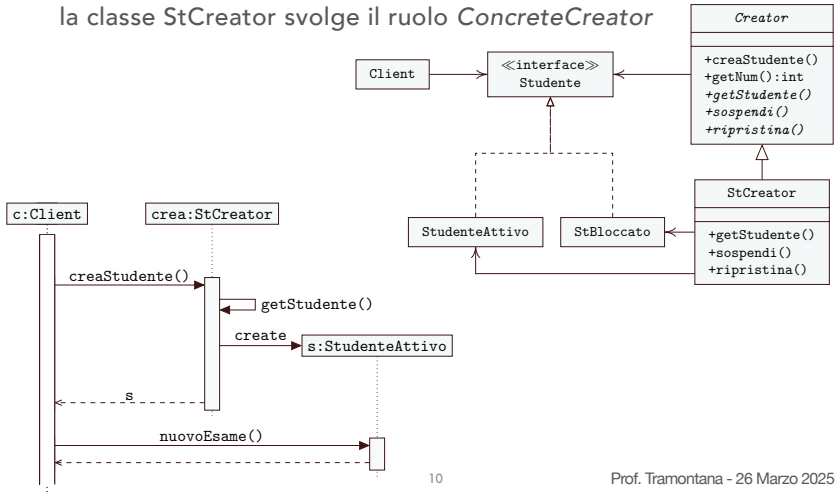
Factory Method

- Soluzione: diagramma UML di sequenza, che illustra le interazioni fra i vari ruoli

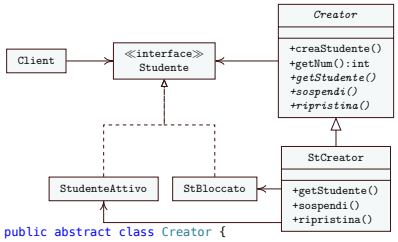


Esempio Di Factory Method

- L'interfaccia Studente svolge il ruolo *Product*, le classi StudenteAttivo e StBloccato svolgono il ruolo *ConcreteProduct*, e la classe StCreator svolge il ruolo *ConcreteCreator*




```java
public interface Studente {
    public boolean nuovoEsame(String m, int v);
    public float getMedia();
}

public class StudenteAttivo implements Studente {
    private List<Esame> esami = new ArrayList<>();
    @Override public boolean nuovoEsame(String m, int v) {
        esami.add(new Esame(m, v));
        System.out.println("Esame " + m + " Registrato");
        return true;
    }
    @Override public float getMedia() {
        System.out.println("num esami: " + esami.size());
        if (esami.isEmpty()) return 0;
        float sum = 0;
        for (Esame e : esami) sum += e.voto();
        float media = sum / esami.size();
        System.out.println("media: " + media);
        return media;
    }
}

public class StBloccato implements Studente {
    private float media;
    public StBloccato(float m) {
        media = m;
    }
    @Override public boolean nuovoEsame(String m, int v) {
        System.out.println("Non può sostenere esami");
        return false;
    }

    @Override public float getMedia() {
        System.out.println("media: " + media);
        return media;
    }
}
```



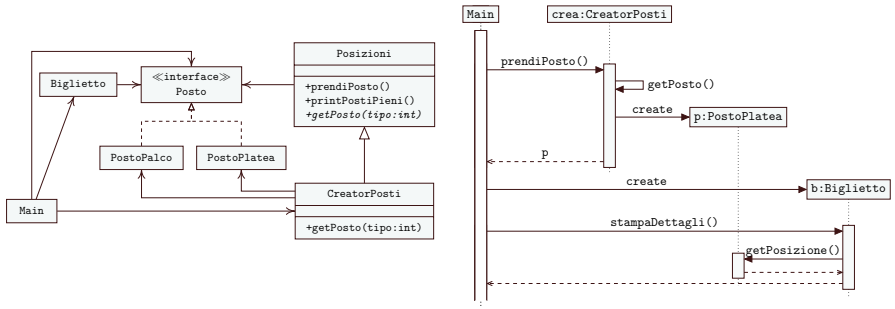

```java
public abstract class Creator {
    private int numIstanze = 0;
    public abstract Studente getStudente();
    final public Studente creaStudente() {
        numIstanze++; return getStudente();
    }
    public int getNum() { return numIstanze; }
    public abstract Studente ripristina(Studente s);
    public abstract Studente sospendi(Studente s);
}

public class StCreator extends Creator {
    private boolean attivo = true;
    private Map<Studente, Studente> bloccati
                                    = new HashMap<>();
    @Override public Studente getStudente() {
        if (attivo) return new StudenteAttivo();
        return new StBloccato(0);
    }
    @Override public Studente sospendi(Studente s) {
        Studente sosp = new StBloccato(s.getMedia());
        bloccati.put(sosp, s);
        return sosp;
    }
    @Override public Studente ripristina(Studente s) {
        return bloccati.remove(s);
    }
}
```


Factory Method

- Conseguenze
  - Il codice delle classi dell'applicazione conosce solo l'interfaccia `Product` e può lavorare con qualsiasi `ConcreteProduct`. I `ConcreteProduct` sono facilmente intercambiabili
  - Se si implementa una sottoclasse di `Creator` per ciascun `ConcreteProduct` da istanziare si ha una proliferazione di classi

Factory Method Per App Biglietti

- La classe Biglietto ha informazioni su un biglietto per un certo posto a teatro, il posto può essere palco o platea
- L'interfaccia Posto svolge il ruolo di Product, le classi PostoPalco e Platea svolgono il ruolo di ConcreteProduct. La classe Posizioni svolge il ruolo di Creator, e la classe CreatorPosti il ruolo di ConcreteCreator




```java
public interface Posto { // Posto è un Product
    public String getPosizione();
    public int getCosto();
    public String getSettore();
}

// PostoPalco è un ConcreteProduct
public class PostoPalco implements Posto {
    private final int numero;
    public PostoPalco() {
        numero = new Random().nextInt(20) + 1;
    }
    @Override public int getCosto() {
        if (numero > 10) return 50;
        return 40;
    }
    @Override public String getPosizione() {
        return Integer.toString(numero);
    }
    @Override public String getSettore() {
        if (numero == 20) return "Centrale";
        if (numero > 10) return "Verde";
        return "Blu";
    }
}

// Biglietto conosce solo l'interfaccia Posto
public record Biglietto(String nome, Posto posto) {
    public void stampaDettagli() {
        System.out.println(
          "Biglietto intestato a " + nome +
          " per il posto " + posto.getPosizione() +
          " settore " + posto.getSettore() +
          " costo " + posto.getCosto());
    }
}
```

```java
public abstract class Posizioni {
    private int MAXPOSTI = 100;
    private Set<String> postiPieni = new TreeSet<>();
    private List<Posto> posti = new ArrayList<>();

    // tipo indica se il posto è platea o palco
    public Posto prendiPosto(int tipo) {
        if (postiPieni.size() == MAXPOSTI) return null;
        Posto p;
        do {
            p = getPosto(tipo);
        } while (postiPieni.contains(p.getPosizione()));
        postiPieni.add(p.getPosizione());
        posti.add(p);
        return p;
    }

    public void printPostiPieni() {
        for (Posto p : posti)
            System.out.println(
              "Posto " + p.getPosizione() + " " +
              p.getSettore() + " " + p.getCosto());
    }

    public abstract Posto getPosto(int tipo);
}

// CreatorPosti è un ConcreteCreator
public class CreatorPosti extends Posizioni {
    @Override public Posto getPosto(final int tipo) {
        if (1 == tipo) return new PostoPalco();
        return new PostoPlatea();
    }
}
```


Object Pool

- Un **object pool** è un deposito di istanze già create, una istanza sarà estratta dal pool quando una classe client ne fa richiesta
  - Il pool può **crescere** o può avere **dimensioni fisse**. Dimensioni fisse: se non ci sono oggetti disponibili al momento della richiesta, non ne creo di nuovi
  - Il client restituisce al pool l'istanza usata quando non più utile
- Il design pattern Factory Method può implementare un object pool
  - I client richiedono istanze, come visto per il Factory Method
  - I client dovranno indicare quando l'istanza non è più in uso, quindi riusabile
  - Lo stato dell'istanza da riusare potrebbe dover essere riscritto
  - L'object pool dovrebbe essere unico: potrei usare un Singleton

Esempio Di Object Pool

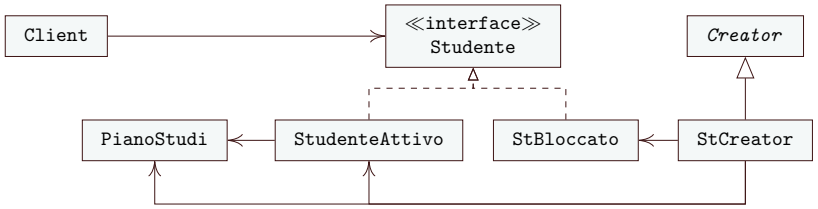
```java
import java.util.ArrayList;
import java.util.List;

// CreatorPool è un ConcreteCreator e implementa un Object Pool
public class CreatorPool {
    private List<Shape> pool = new ArrayList<>();

    // metodo factory che ritorna un oggetto prelevato dal pool
    public Shape getShape() {
        if (pool.size() > 0)
            return pool.remove(0);
        return new Circle();
    }

    // inserisce un oggetto nel pool
    public void releaseShape(Shape s) {
        pool.add(s);
    }
}
```

Dependency Injection

- Il design pattern Factory Method può essere usato per inserire le dipendenze (dependency injection) necessarie alle istanze di ConcreteProduct
- Tramite la Dependency Injection un oggetto (client) riceve altri oggetti da cui dipende, questi altri oggetti sono detti dipendenze
- La tecnica di Dependency Injection permette di **separare la costruzione delle istanze dal loro uso**
- **Il client non crea l'istanza** di cui ha bisogno
- Le dipendenze sono iniettate al client per mezzo di parametri nel suo costruttore. Questo permette di evitare complicazioni derivanti da metodi setter e da controlli per verificare che le dipendenze non siano null, di conseguenza il codice è più semplice
- L'oggetto che fa Dependency Injection si occupa di connettere (fa **wiring** di) varie istanze. In un unico posto vediamo le connessioni fra gli oggetti

Esempio Di Dependency Injection

- Si abbia una classe `PianoStudi` che è una dipendenza per `Studente`
- `Studente` riceve nel suo costruttore l'istanza di `PianoStudi`
- `Studente` conosce il tipo `PianoStudi` non i suoi eventuali sottotipi
- `Studente` non creare l'istanza di `PianoStudi`



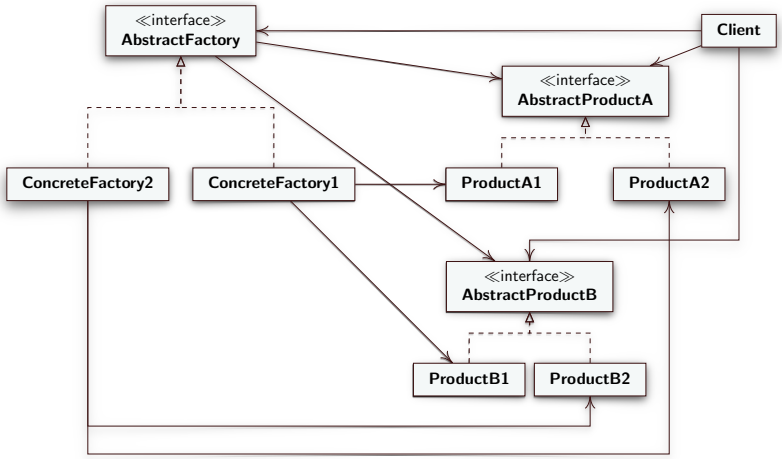
Abstract Factory

- Intento
  - Fornire un'interfaccia per creare famiglie di oggetti che hanno qualche relazione senza specificare le loro classi concrete
- Problema
  - Il sistema complessivo dovrebbe essere **indipendente dalle classi usate**, così da essere configurabile con una di varie famiglie di classi. Le classi di una famiglia dovrebbero essere usate in modo consistente
  - Es. lo strato di interfaccia utente permette vari tipi di look-and-feel, così differenti comportamenti per accessori (widget) dell'interfaccia utente sono possibili

Abstract Factory

- Soluzione
  - **AbstractFactory**: interfaccia astratta per creare famiglie di oggetti
  - **ConcreteFactory**: classi che implementano operazioni per creare ciascuna famiglia di oggetti
  - **AbstractProduct** è l'interfaccia per una famiglia di oggetti
  - **Product** definisce un oggetto, creato da un ConcreteFactory, che implementa l'interfaccia AbstractProduct
  - Il **client** usa solo interfacce dichiarate da AbstractFactory e AbstractProduct

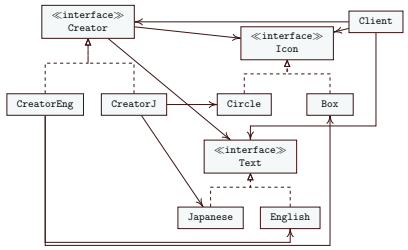
Abstract Factory



Abstract Factory

- Conseguenze
  - Permette di usare classi consistentemente (per famiglie)
  - Le famiglie di classi sono facilmente intercambiabili
  - Non è immediato supportare nuove classi Product, poiché bisogna aggiungere un metodo su AbstractFactory e su ogni ConcreteFactory


```java
public interface Icon { // AbstractProductA
  public void sad();
  public void happy();
  public void surprise();
}

public class Circle implements Icon { // ProductA1
    @Override
    public void sad() {
        System.out.print(" (> _ <) ");
    }
    @Override
    public void happy() {
        System.out.print(" (^ _ ^) ");
    }
    @Override
    public void surprise() {
        System.out.print(" (o _ o) ");
    }
}
public class Box implements Icon { // ProductA2
    @Override
    public void sad() {
        System.out.print(":-[ ");
    }
    @Override
    public void happy() {
        System.out.print(":-] ");
    }
    @Override
    public void surprise() {
        System.out.print(":-O ");
    }
}
```




```java
public interface Creator { // AbstractFactory
    public Icon getIcon();
    public Text getText();
}

// ConcreteFactory
public class CreatorEng implements Creator {
    public Icon getIcon() {
        return new Box();
    }
    public Text getText() {
        return new English();
    }
}
public class CreatorJ implements Creator {
    public Icon getIcon() {
        return new Circle();
    }
    public Text getText() {
        return new Japanese();
    }
}
```



```java
public interface Text { // AbstractProductB
    public void welcome();
    public void attention();
    public void surprise();
}
public class English implements Text { // ProductB2
    @Override
    public void welcome() {
        System.out.println(": : : Welcome. Nice to meet you. :::::");
    }
    @Override
    public void attention() {
        System.out.println(": : : Be quiet please! :::::");
    }
    @Override
    public void surprise() {
        System.out.println(": : : Oh, my dear! :::::");
    }
}
public class Japanese implements Text { // ProductB1
    @Override
    public void welcome() {
        System.out.println(" Youkoso. Konnichiwa! Hajimemashite.");
    }
    @Override
    public void attention() {
        System.out.println(" Shizuka ni shite kudasai!");
    }
    @Override
    public void surprise() {
        System.out.println(" Bikkuri shita!");
    }
}
```

```java
public class Client {
    public static void main(String args[]) {
        Creator c = new CreatorEng();
        Icon ic = c.getIcon();
        Text t = c.getText();
        ic.happy();
        t.welcome();
        ic.surprise();
        t.surprise();
        ic.sad();
        t.attention();
    }
}
```