15. Design pattern II

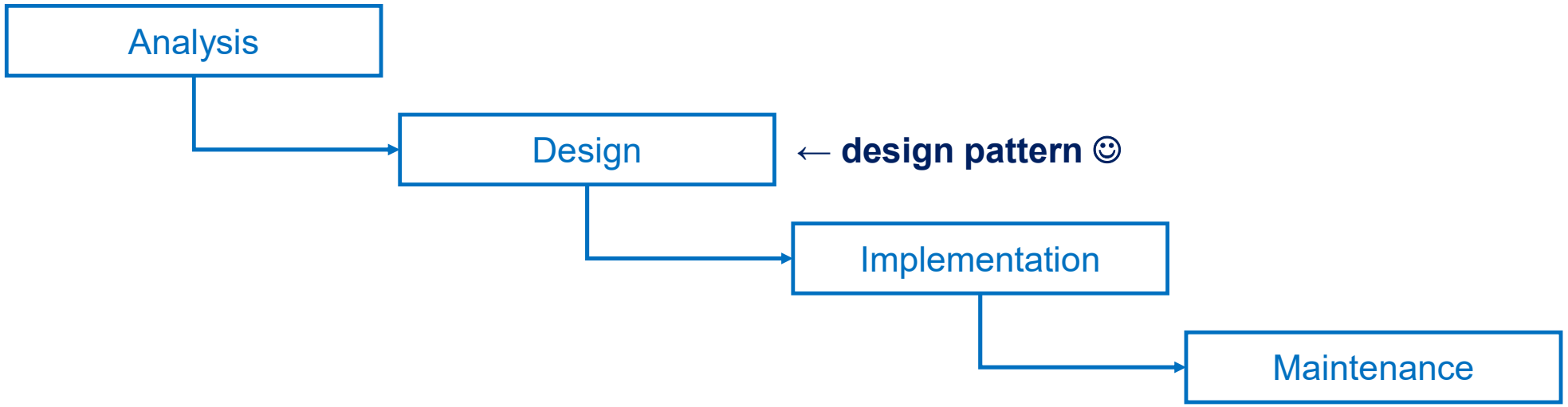
IS 2024-2025

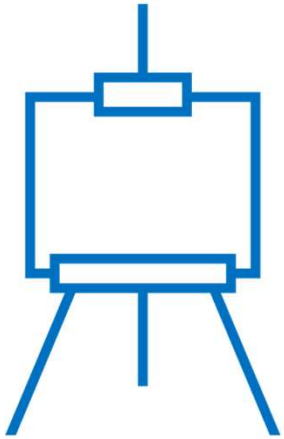
Laura Semini, Jacopo Soldani
Corso di Laurea in Informatica
Dipartimento di Informatica, Università of Pisa

REMINDER

«Each pattern **describes a problem** which occurs over and over again in our environment, and then **describes the core of the solution** to that problem, in such a way that **you can use this solution a million times over**, without ever doing it the same way twice»
(Christopher Alexander, A Pattern Language, 1977)

Pattern per ogni fase del processo software





(SIMPLE) FACTORY

FACTORY: CI SERVONO?

Si, nel laboratorio di reti

- metodi statici di una classe che restituiscono oggetti di quella classe
- i seguenti metodi contattano il DNS per la risoluzione di indirizzo/hostname

```
static InetAddress getLocalHost() throws UnknownHostEception
static InetAddress getByName (String hostname) throws UnknownHostException
static InetAddress [] getAllByName (String hostName)
       throws UnknownHostException
static InetAddress getLoopBackAddress()
```

- i seguenti metodi statici costruiscono oggetti di tipo `InetAddress`, ma non contattano il DNS (utile se DNS non disponibile e conosco indirizzo/host)
- nessuna garanzia sulla correttezza di hostname/IP, `UnknownHostException` sollevata solo se l'indirizzo è malformato

```
static InetAddress getByAddress(byte IPAddr[]) throws UnknownHostException
static InetAddress getByAddress (String hostName, byte IPAddr[])
                                                 throws UnknownHostException
```

Ma come si progettano?

PATTERN CREAZIONALI

I pattern creazionali **astraggono** il processo di **istanziazione** degli oggetti

- Nascondono l'effettiva creazione degli oggetti
- Rendono il sistema indipendente da come i suoi oggetti sono creati e composti
- Semplificano la costruzione di oggetti complessi (rispetto alla chiamata esplicita dei costruttori)

Una classe **factory** ha il solo compito di **creare e restituire istanze** di altre classi

Creare un oggetto mediante una classe «factory» riduce linee di codice e complessità, consentendo di creare/inizializzare rapidamente oggetti

Esempi (in Java):

- sequenze di tasti (KeyStroke)
- connessioni di rete (SocketFactory)
- log (LogFactory)

OSSERVAZIONI SU «NEW»

Invocando il comando **new** per creare un oggetto, violiamo il principio «code to an interface»

- Assegniamo a una variabile (con tipo interfaccia) un oggetto ottenuto da una classe concreta
- Esempio: `List list = new ArrayList()`

Inoltre, consideriamo una classe che controlla alcune variabili e istanzia una particolare classe basata su tali variabili, p.e.

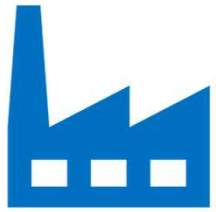
```
if (condition) { return new ArrayList(); }
else { return new LinkedList();}
```

La classe in questione:

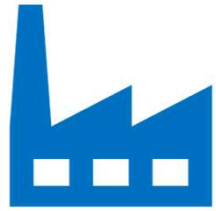
- dipende da ogni classe riferita al suo interno
- deve essere aggiornata e ricompilata se cambiano le classi riferite (aggiunta di nuove classi, rimozione di classi esistenti)

⇒ Violazione dei principi «open-closed» e «information hiding»

Questo si può evitare con le factory



TRE TIPI DI FACTORY

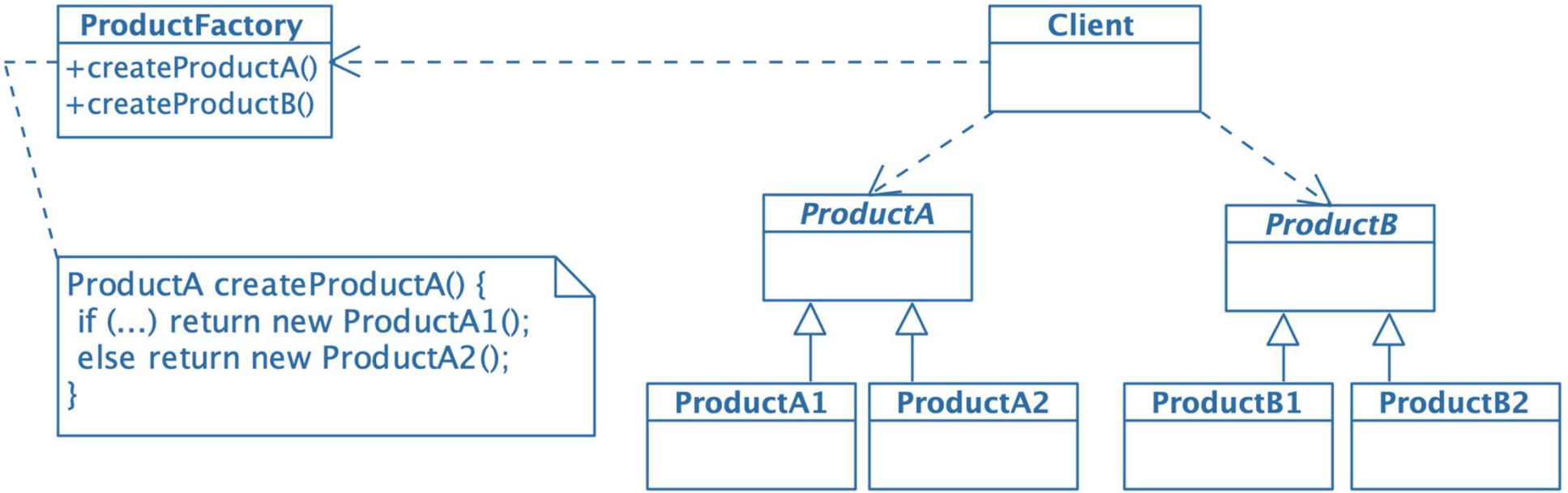


- **Simple Factory** (detto anche ConcreteFactory)
  - non è un pattern GoF
  - è una semplificazione molto diffusa di Abstract Factory
- **Factory Method**
- **Abstract Factory**

SIMPLE FACTORY (AKA. CONCRETE FACTORY)

Problema: Chi deve creare gli oggetti quando la logica di creazione è **complessa** e si vuole **separare la logica di creazione** dalle altre funzionalità di un oggetto?

Soluzione: Delega a un oggetto **factory** che gestisce la creazione

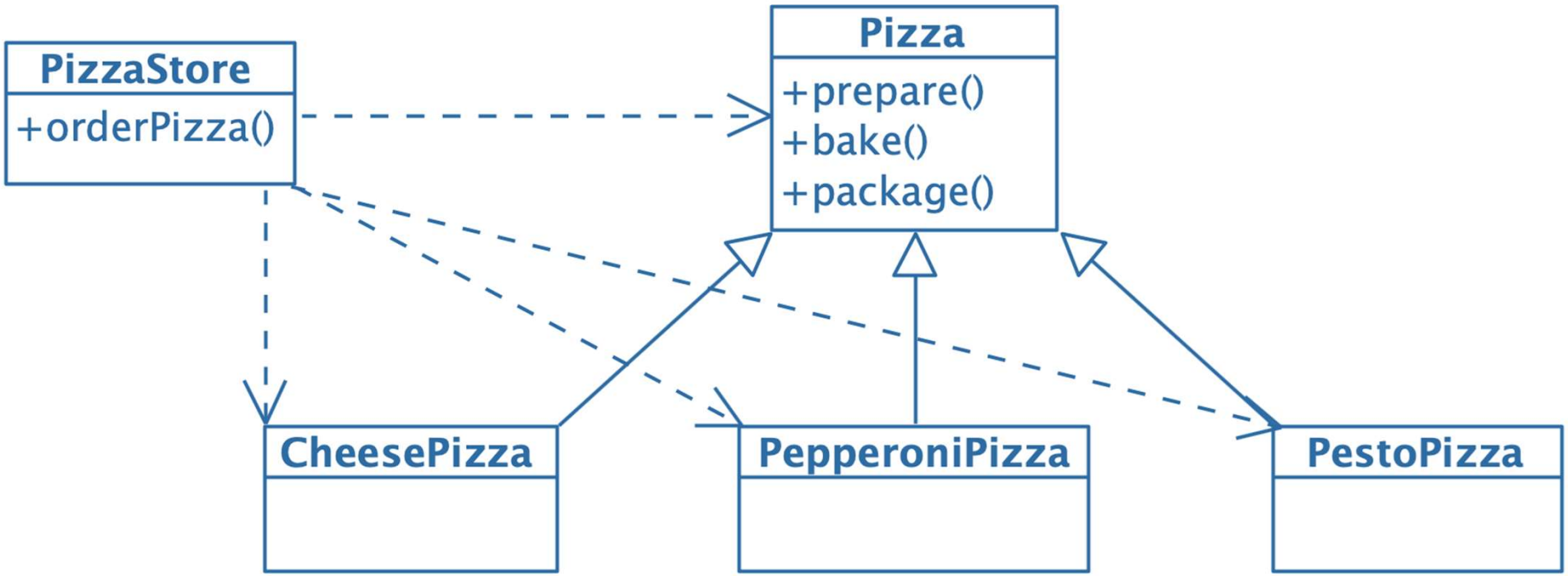


ESEMPIO 

```
public class PizzaStore {
  Pizza orderPizza(String type){
    Pizza pizza;

    if (type == "cheese")
      pizza = new CheesePizza();
    else if (type == "pepperoni")
      pizza = new PepperoniPizza();
    else if (type == "pesto")
      pizza = new PestoPizza();

    pizza.prepare();
    pizza.bake();
    pizza.package();
    return pizza
  }
}
```

```
// ---------------------------------
// Stays the same, independent from
// the actual pizza type
// ---------------------------------
//
// This is the creation code
// - dependent on the pizza classes
// - more complex as we add pizzas
//
//
// ---------------------------------
// This is the preparation code
// - stays the same
// - independent from pizza type
//
// ---------------------------------
```

ESEMPIO (CONT.)



Estrarre il **creation code** e metterlo in una classe che si occupi solamente di creare pizze (**PizzaSimpleFactory**)

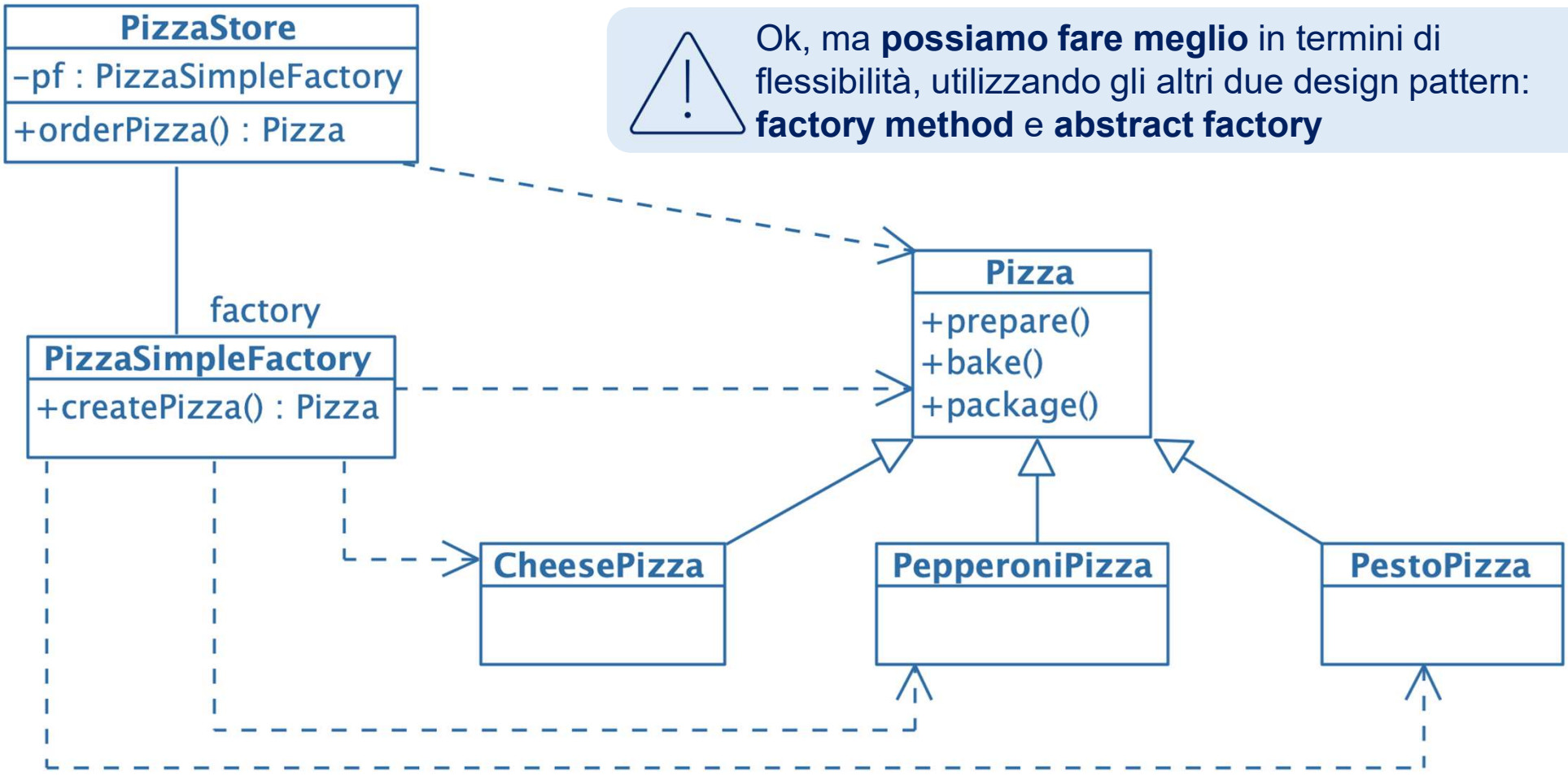
ESEMPIO  (CONT.)

```
public class PizzaStore {

  private PizzaFactory pf;

  PizzaStore(PizzaSimpleFactory pf) {
    this.pf = pf
  }

  Pizza orderPizza(String type){
    Pizza pizza = this.pf.createPizza(type)

    pizza.prepare();
    pizza.bake();
    pizza.package();
    return pizza
  }
}
```

```
public class PizzaSimpleFactory {
  public Pizza createPizza(String type) {
    if (type == "cheese")
      return new CheesePizza();
    else if (type == "pepperoni")
      return new PepperoniPizza();
    else if (type == "pesto")
      return new PestoPizza();
  }
}
```

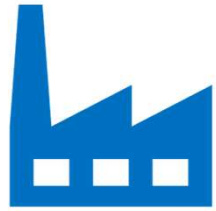
- Istanziazione concreta rimpiazzata da una **chiamata a factory**
- Non è necessario toccare `PizzaStore` se aggiungiamo nuove pizze!

ESEMPIO (CONT.)



Ok, ma **possiamo fare meglio** in termini di flessibilità, utilizzando gli altri due design pattern: **factory method** e **abstract factory**

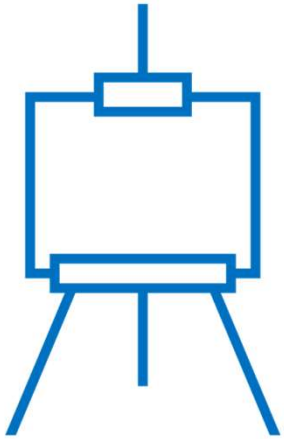
TRE TIPI DI FACTORY



- **Simple Factory** (detto anche ConcreteFactory)
  - non è un pattern GoF
  - è una semplificazione molto diffusa di Abstract Factory

- **Factory Method**
  - è un «**class** creational pattern»
  - usa **l'ereditarietà** per decidere l'oggetto da istanziare

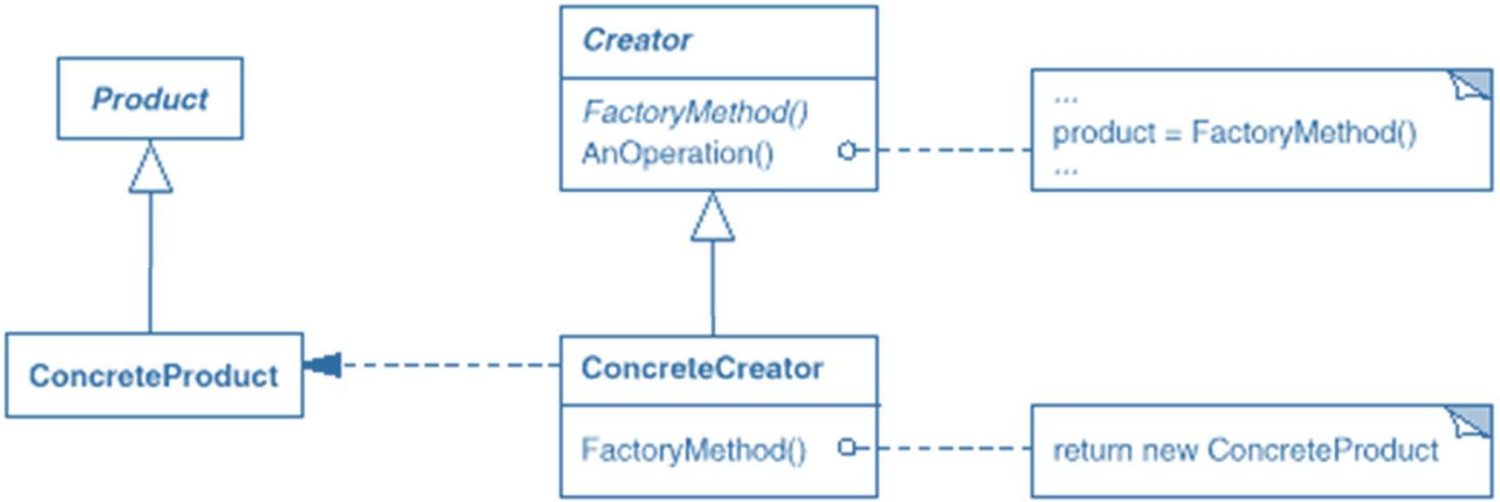
- **Abstract Factory**
  - è un «**object** creational pattern»
  - usa il meccanismo di delega (**delegation**) per istanziare altri oggetti



FACTORY METHOD

IL FACTORY METHOD PATTERN

Delega alle **sottoclassi** la **decisione** di quali classi istanziare



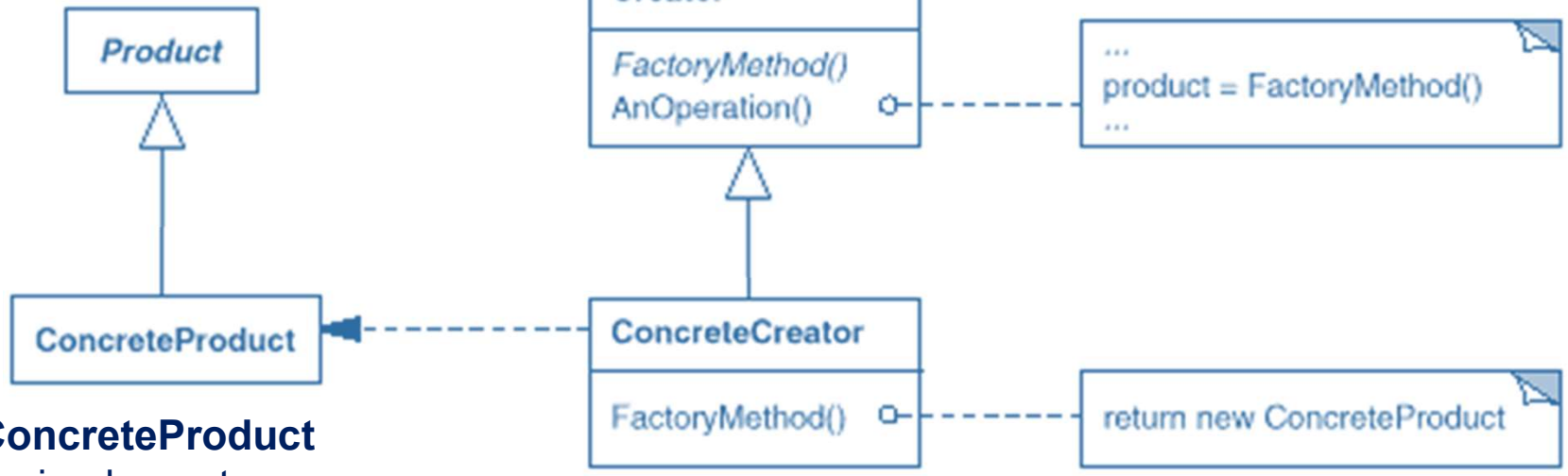
NB: `Creator` è scritto senza sapere quali siano i prodotti che saranno effettivamente creati

→ scelta dovuta all'effettiva sottoclasse (di `Creator`) che viene instanziata

IL FACTORY METHOD PATTERN (CONT.)

Product definisce l'**interfaccia** per il tipo di **oggetti** che sono **creati** dal **factory method**

Creator dichiara il **factory method** (usato da altri metodi)



ConcreteProduct implementa l'interfaccia di **Product**

ConcreteCreator sovrascrive il factory method per restituire un'istanza del **ConcreteProduct**

ESEMPIO

Consideriamo la seguente evoluzione del PizzaStore

- Differenti franchise per parti diverse del paese (es. California, New York, Chicago)
- Ogni franchise ha la propria *factory* per creare pizze in linea con i gusti dei clienti locali
- Mantenere il processo di preparazione (comune) che determina il successo di PizzaStore

Con il **factory method**, possiamo rendere PizzaStore una superclasse astratta

- «code to an interface» nella superclasse // processo di preparazione comune
- creazione degli oggetti nelle sottoclassi // pizze adatte ai gusti dei locali

ESEMPIO (CONT.)

Creator

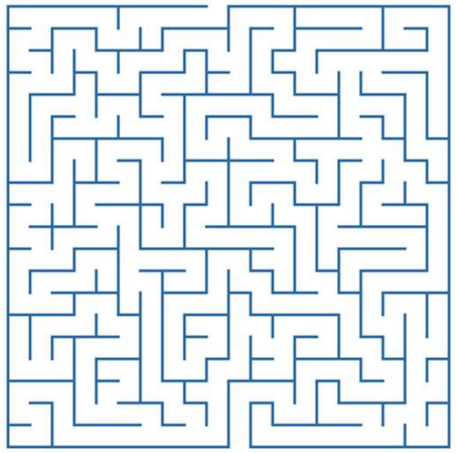
```
public abstract class PizzaStore {
  protected abstract createPizza(String type);
  public Pizza orderPizza(String type) {
    Pizza pizza = createPizza(type);
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
  }
}
```

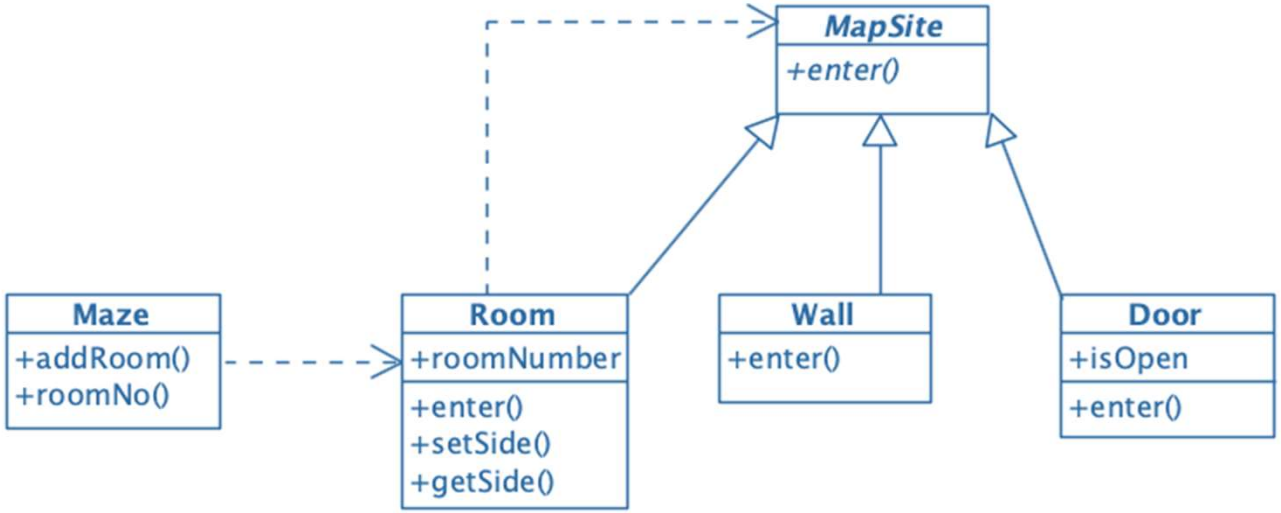
```
public class NYPizzaStore extends PizzaStore {
  public Pizza createPizza(String type) {
    if(type.equals("cheese")) return new NYCheesePizza();
    if(type.equals("greek")) return new NYGreekPizza();
    if(type.equals("pepperoni")) return new NYPepperoniPizza();
    return null;
  }
}
```

ConcreteCreator

UN ALTRO ESEMPIO: MAZE

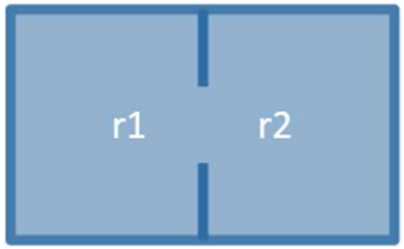
Vogliamo costruire un **labirinto** con **stanze**, **pareti** e **porte**





MAZE (CONT.)

```
public class MazeGame {
  // Create the maze
  public Maze createMaze() {
    Maze maze = new Maze();
    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door door = new Door(r1, r2);
    maze.addRoom(r1);
    maze.addRoom(r2);
    r1.setSide(MazeGame.North, new Wall());
    r1.setSide(MazeGame.East, door);
    r1.setSide(MazeGame.South, new Wall());
    r1.setSide(MazeGame.West, new Wall());
    // continues on next column
    r2.setSide(MazeGame.North, new Wall());
    r2.setSide(MazeGame.East, new Wall());
    r2.setSide(MazeGame.South, new Wall());
    r2.setSide(MazeGame.West, door);
    return maze;
  }
}
```



createMaze crea le componenti e le assembla (ha **due responsabilità**)

MAZE (CONT.)

Il **problema** di `createMaze` è la sua ***inflexibility***

- Come fare se vogliamo un labirinto magico con `EnchantedRoom` e `EnchantedDoor`? E se vogliamo elementi nascosti come `DoorWithLock` e `WallWithHiddenDoor`?

- Sarebbero necessari **cambiamenti significativi** a `createMaze` a causa delle istanziazioni esplicite (con **`new`**) per la creazione delle componenti del labirinto

- Come riprogettare il tutto in modo da rendere «facile» la creazione di nuovi tipi di componenti con `createMaze`?

MAZE (CONT.)

```
public class MazeGame {
  // Factory methods
  public Maze makeMaze() {
    return new Maze();
  }
  public Room makeRoom(int n) {
    return new Room(n);
  }
  public Wall makeWall() {
    return new Wall();
  }
  public Door makeDoor(Room r1, Room r2)  {
    return new Door(r1, r2);
  }
  ...
```

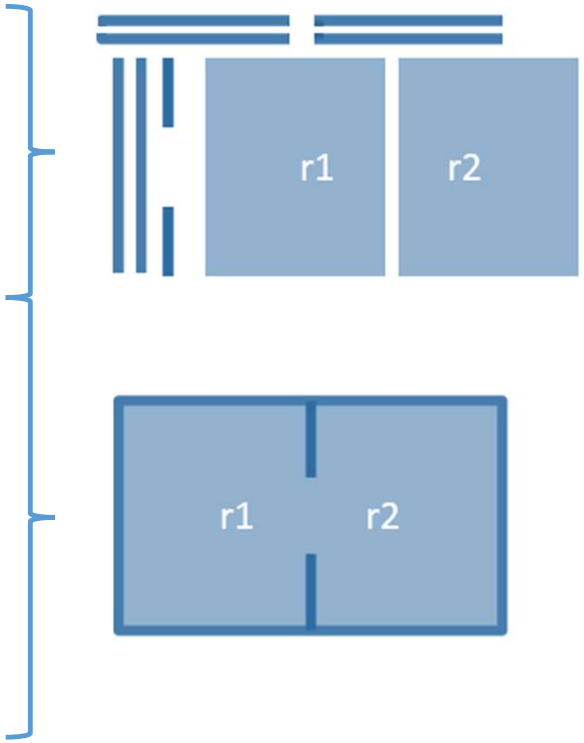
L'implementazione concreta del labirinto è **delegata** alle sottoclassi

I metodi `make*` possono essere

- astratti o
- concreti (implementazione di **default**)

MAZE (CONT.)

```
  ...
  public Maze createMaze() {
        Maze maze = makeMaze();
        Room r1 = makeRoom(1);
        Room r2 = makeRoom(2);
        Door door = makeDoor(r1, r2);
        maze.addRoom(r1);
        maze.addRoom(r2);
        r1.setSide(MazeGame.North, makeWall());
        r1.setSide(MazeGame.East, door);
        ...
        r2.setSide(MazeGame.West, door);
        return maze;
  }
}
```

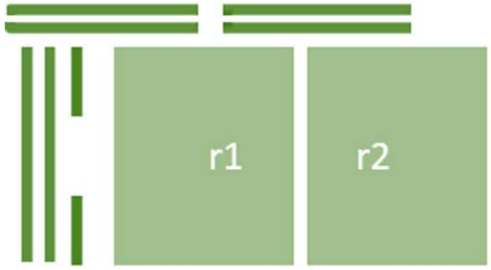


MAZE (CONT.)

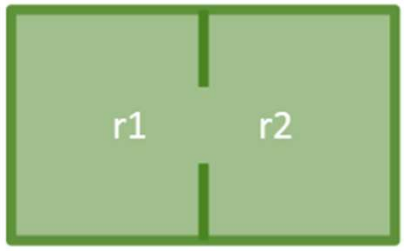
Abbiamo reso createMaze un po' più complessa, ma molto **più flessibile**!

Creiamo un **labirinto incantato**?

```
public class EnchantedMazeGame extends MazeGame {
  public Room makeRoom(int n) {
    return new EnchantedRoom(n);
  }
  public Wall makeWall() {
    return new EnchantedWall();
  }
  public Door makeDoor(Room r1, Room r2) {
    return new EnchantedDoor(r1, r2);
  }
}
```



Il metodo `createMaze` è ereditato da `MazeGame` e può essere usato per creare altri labirinti senza modifiche!



MAZE: OSSERVAZIONI

Funziona perché `createMaze` **delega la creazione** dei componenti del labirinto alle **sottoclassi**

Nell'esempio:

- Creator ⟹ `MazeGame`
- ConcreteCreator ⟹ `EnchantedMazeGame` // ma anche MazeGame per la versione default
- Product ⟹ `Wall, Room, Door`
- ConcreteProduct ⟹ `EnchantedWall`, `EnchantedRoom`, `EnchantedDoor`

NB: Maze è un ConcreteProduct (ma anche un Product)

SUL FACTORY METHOD PATTERN

Utilizzo

- Consente di disaccoppiare una classe dalle classi degli oggetti che crea e utilizza
- Delega alle sottoclassi la specifica degli oggetti da creare

Benefici

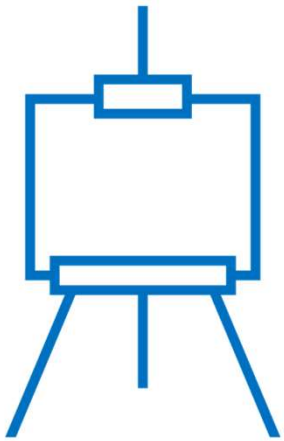
- Il codice è reso più flessibile e riusabile (senza l'istanziazione delle classi application-specific)
- Code to an interface (la classe si basa sull'interfaccia di Product e può funzionare con qualunque ConcreteProduct che supporti tale interfaccia)

Svantaggi

- Necessario estendere la classe Creator anche solo per istanziare un particolare ConcreteProduct

Implementazione

- I Creator possono essere astratti o concreti (con metodi di default)
- Se usato per creare diversi tipi di prodotto, il factory method deve avere un parametro per decidere quale tipo di oggetto creare (p.e., con `if-then-else`)



ABSTRACT FACTORY

L'ABSTRACT FACTORY PATTERN

Delega ad **altre classi** la **decisione** di quali classi istanziare

Definisce un'interfaccia

- per creare «famiglie di oggetti» (correlati e dipendenti fra loro)
- senza specificare le classi concrete

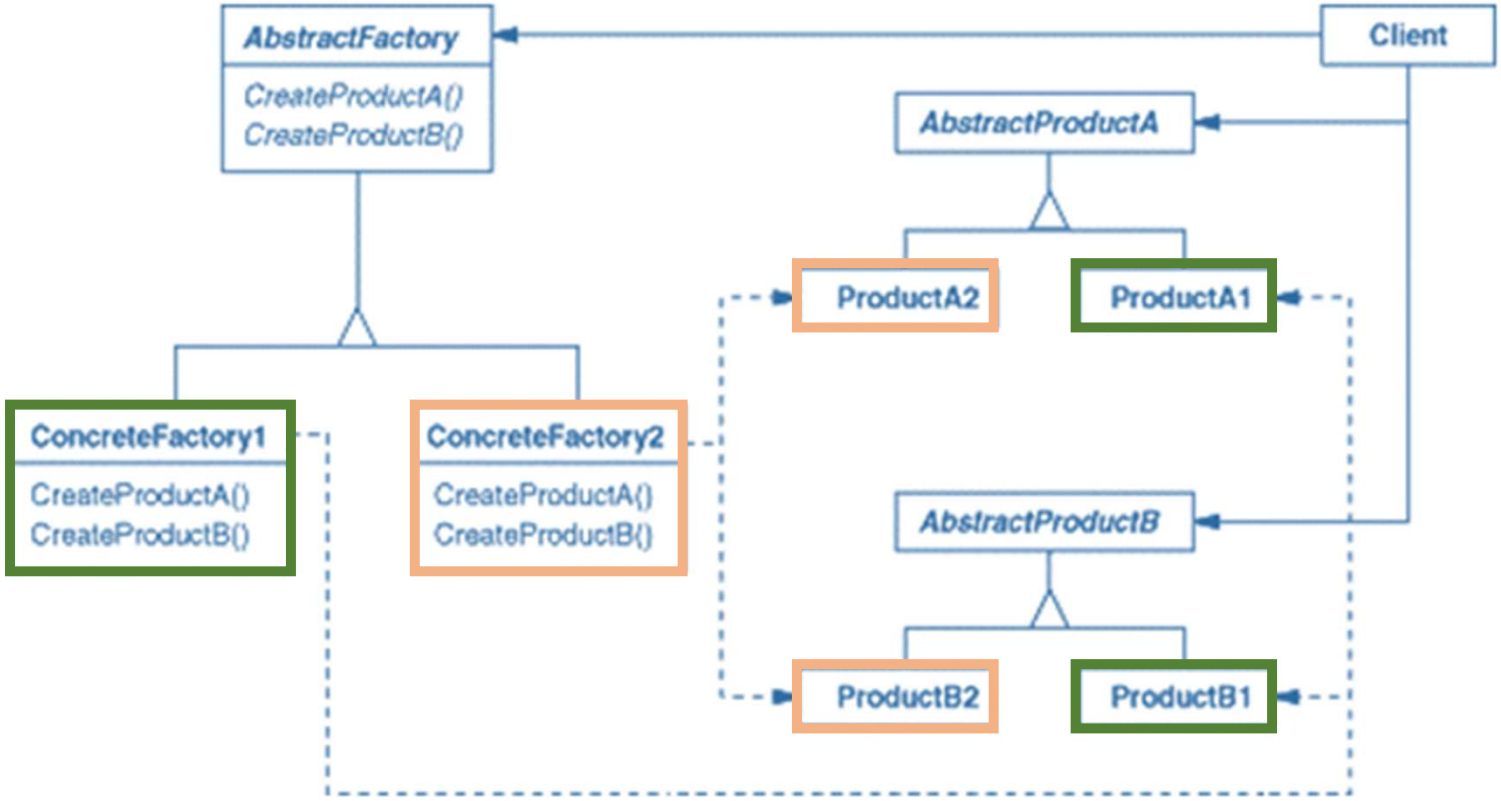
Come **cambia** l'istanziazione di oggetti tra **Abstract Factory** e **Factory Method**?

- **Factory Method** → Delegata a **sottoclassi** sfruttando l'**ereditarietà**
- **Abstract Factory** → Delegata ad **altri oggetti** mediante **composizione**

Composizione?

NB: Spesso l'oggetto delegato usa factory method per effettuare l'istanziazione, quindi si applicano entrambi i pattern

L'ABSTRACT FACTORY PATTERN (CONT.)



MAZE, WITH ABSTRACT FACTORY

```
// MazeFactory – Abstract Factory
public interface MazeFactory {
  public Maze makeMaze();
  public Room makeRoom(int n);
  public Wall makeWall();
  public Door makeDoor(Room r1, Room r2);
}
```

```
// BasicMazeFactory – Concrete factory for basic parts
public class BasicMazeFactory implements MazeFactory {
  public Maze makeMaze() { return new BasicMaze(); }
  public Room makeRoom(int n) { return new BasicRoom(n); }
  public Wall makeWall() { return new BasicWall(); }
  public Door makeDoor(Room r1, Room r2) { return new BasicDoor(r1, r2); }
}
```

MAZE, WITH ABSTRACT FACTORY (CONT.)

```
public class MazeGame {
  // createMaze now inputs a MazeFactory reference as parameter
  public Maze createMaze(MazeFactory factory) {
    Maze maze = factory.makeMaze();
    Room r1 = factory.makeRoom(1);
    Room r2 = factory.makeRoom(2);
    Door door = factory.makeDoor(r1, r2);
    maze.addRoom(r1);
    maze.addRoom(r2);
    r1.setSide(MazeGame.North, factory.makeWall());
    ...
    return maze;
  }
}
```

`createMaze` **delega** la responsabilità di creare le componenti del labirinto all'oggetto `MazeFactory`

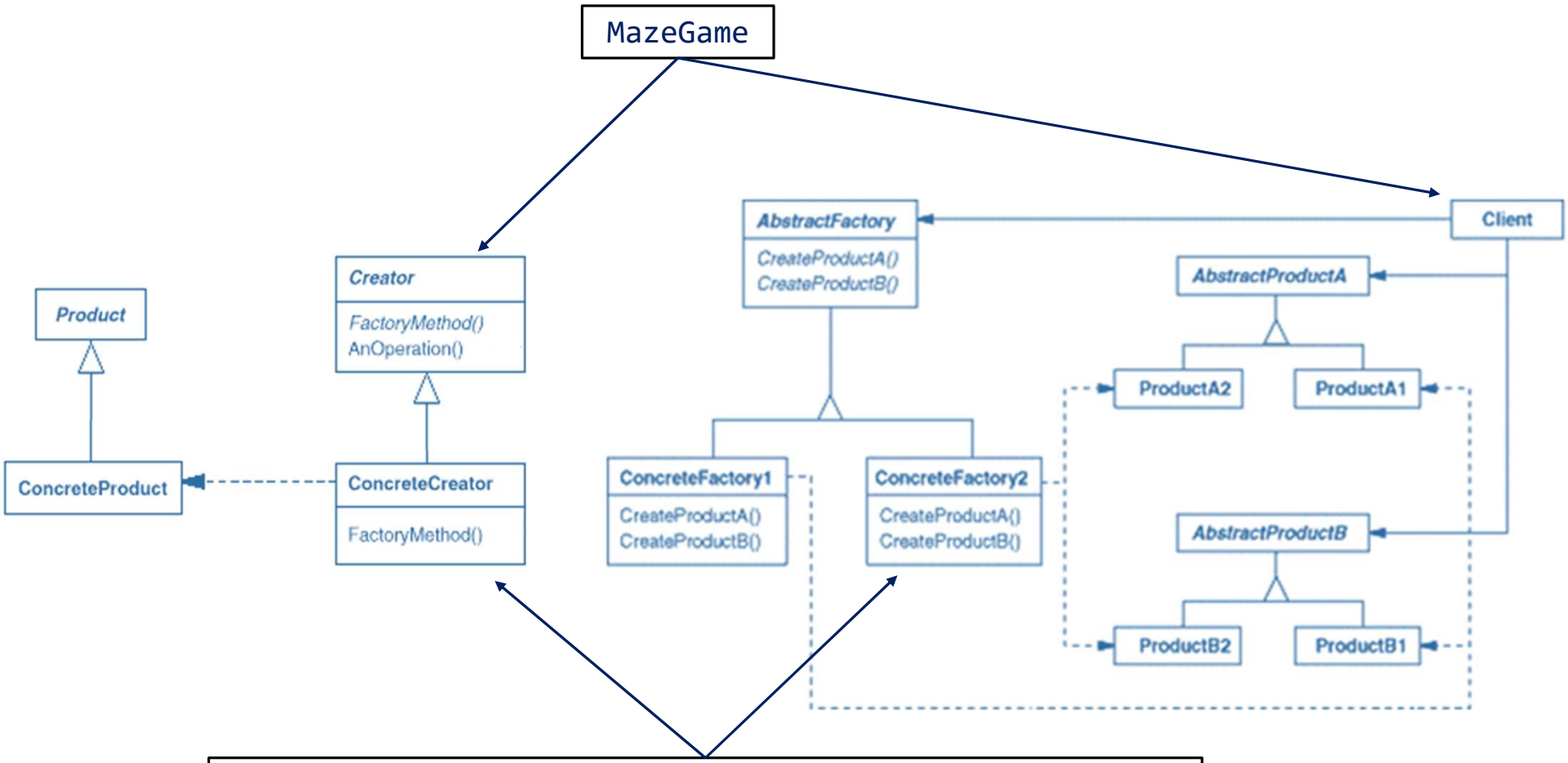
MAZE, WITH ABSTRACT FACTORY (CONT.)

```
// Another concrete factory
public class EnchantedMazeFactory implements MazeFactory {
  public Room makeRoom(int n) { return new EnchantedRoom(n); }
  public Wall makeWall() {return new EnchantedWall(); }
  public Door makeDoor(Room r1, Room r2) { return new EnchantedDoor(r1, r2); }
}
```

In questo esempio abbiamo:

- AbstractFactory → `MazeFactory`
- ConcreteFactory → `BasicMazeFactory`, `EnchantedMazeFactory`
- AbstractProduct → `Wall`, `Room`, `Door`
- ConcreteProduct → `BasicWall`, `BasicRoom`, `BasicDoor`, `EnchantedWall`, `EnchantedRoom`, `EnchantedDoor`

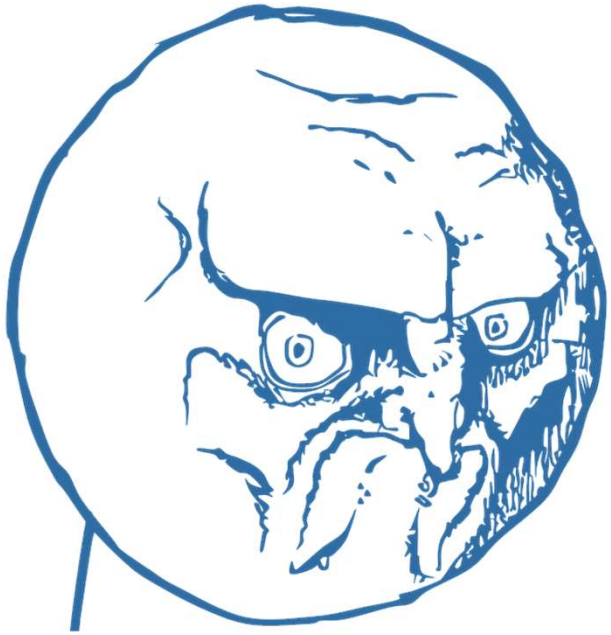
MAZE: FACTORY METHOD VS ABSTRACT FACTORY



E IL PIZZA STORE?

Grande successo del factory method per la realizzazione di franchise diversi, ma abbiamo scoperto che alcuni franchise

- seguono le nostre procedure (forzatamente, grazie al codice astratto in PizzaStore)
- risparmiano sulla qualità degli ingredienti per ridurre i costi e aumentare gli utili



NO! QUESTO È INACCETTABILE!

Il successo della nostra compagnia è dovuto anche all'utilizzo di ingredienti freschi e di qualità

PIZZA STORE V2

Usiamo un **abstract factory** per reperire gli ingredienti utilizzati per preparare le pizze

- Regioni diverse usano ingredienti diversi ⇒ creiamo sottoclassi «region-specific» del factory
- Indipendentemente dai requisiti «region-specific», ci assicureremo che i factory forniscano ingredienti in linea con gli standard di qualità di PizzaStore

(I franchise dovranno trovare un altro modo per ridurre i costi ☺)

PIZZA STORE V2 (CONT.)

```
// Abstract factory for pizza ingredients
public interface PizzaIngredientFactory {
  public Dough createDough();
  public Sauce createSauce();
  public Cheese createCheese();
  public Veggies[] createVeggies();
  public Pepperoni createPepperoni();
  public Clams createClams();
}
```

Richiede l'introduzione di nuove classi astratte come `Dough`, `Sauce`, `Cheese`, etc.

PIZZA STORE V2 (CONT.)

```
// Concrete factory for Chicago pizzas
public class ChicagoPizzaIngredientFactory implements PizzaIngredientFactory {
  public Dough createDough() { return new ThickCrustDough(); }
  public Sauce createSauce() { return new PlumTomatoSauche(); }
  public Cheese createCheese() { return new MozzarellaCheese(); }
  public Veggies[] createVeggies() {
    Veggies veggies[] = { new BlackOlives(), new Spinach, new Eggplant() };
    return veggies;
  }
  public Pepperoni createPepperoni() { return new SlicedPepperoni(); }
  public Clams createClams() { return new FrozenClams(); }
}
```

Consente di garantire la qualità degli ingredienti usati durante la preparazione delle pizze, tenendo anche in considerazione i gusti dei clienti di Chicago

Ok, ma dove si usa?

PIZZA STORE V2 (CONT.)

```
// Pizza base class, now with an abstract "prepare" method
public abstract class Pizza {
  String name;
  Dough dough;
  Sauce sauce;
  Veggies veggies[];
  Cheese cheese;
  Pepperoni pepperoni;
  Clams clams;
  abstract void prepare(); // now this is abstract!
  void bake() { System.out.println("Bake for 25 minutes at 350"); }
  void cut() { ... }
  ...
```

Ora basta **aggiornare le sottoclassi** in modo che utilizzino `IngredientFactory`

PIZZA STORE V2 (CONT.)

```
// A concrete (subclass of) Pizza
public class CheesePizza extends Pizza {
  PizzaIngredientFactory ingrFactory;
  public CheesePizza(PizzaIngredientFactory ingrFactory) {
    this.ingrFactory = ingrFactory;
  }
  void prepare() {
    dough = ingrFactory.createDough();
    sauce = ingrFactory.createSauce();
    cheese = ingrFactory.createCheese();
  }
}
```

Non ci servono più sottoclassi tipo `NYCheesePizza` o `ChicagoCheesePizza`, perché `PizzaIngredientFactory` tiene già conto delle differenze regionali

Dobbiamo però aggiornare i vari `PizzaStore` in modo che creino le pizze usando `PizzaIngredientFactory` appropriati

PIZZA STORE V2 (CONT.)

```
// A concrete (subclass of) PizzaStore
public class ChicagoPizzaStore extends PizzaStore {
  protected Pizza createPizza(String item) {
    Pizza pizza = null;
    PizzaIngredientFactory ingrFactory = new ChicagoPizzaIngredientFactory();
    if (item.equals("cheese")) {
      pizza = new CheesePizza(ingrFactory);
      pizza.setName("Chicago Style Cheese Pizza");
    } else if (item.equals("veggie")) {
      pizza = new VeggiePizza(ingrFactory);
      pizza.setName("Chicago Style Veggie Pizza");ù
    } else ...
```

PIZZA STORE V2: RECAP

- Abbiamo creato un **abstract factory** per gli ingredienti (aka. `PizzaIngredientFactory`)

- L'abstract factory fornisce un'interfaccia per la creazione di «famiglie di prodotti»

  ⇒ Disaccoppia il codice client dall'implementazione del factory che genera i prodotti concreti

- Consente al codice cliente (aka. `PizzaStore` e sottoclassi)
  - Selezione e utilizzo del factory più appropriato alla regione
  - Creazione del giusto stile di pizza (con il **factory method**)
  - Utilizzo del giusto set di ingredienti (con **abstract factory**)

ESERCIZIO

Si consideri la produzione di due prodotti di due brand:

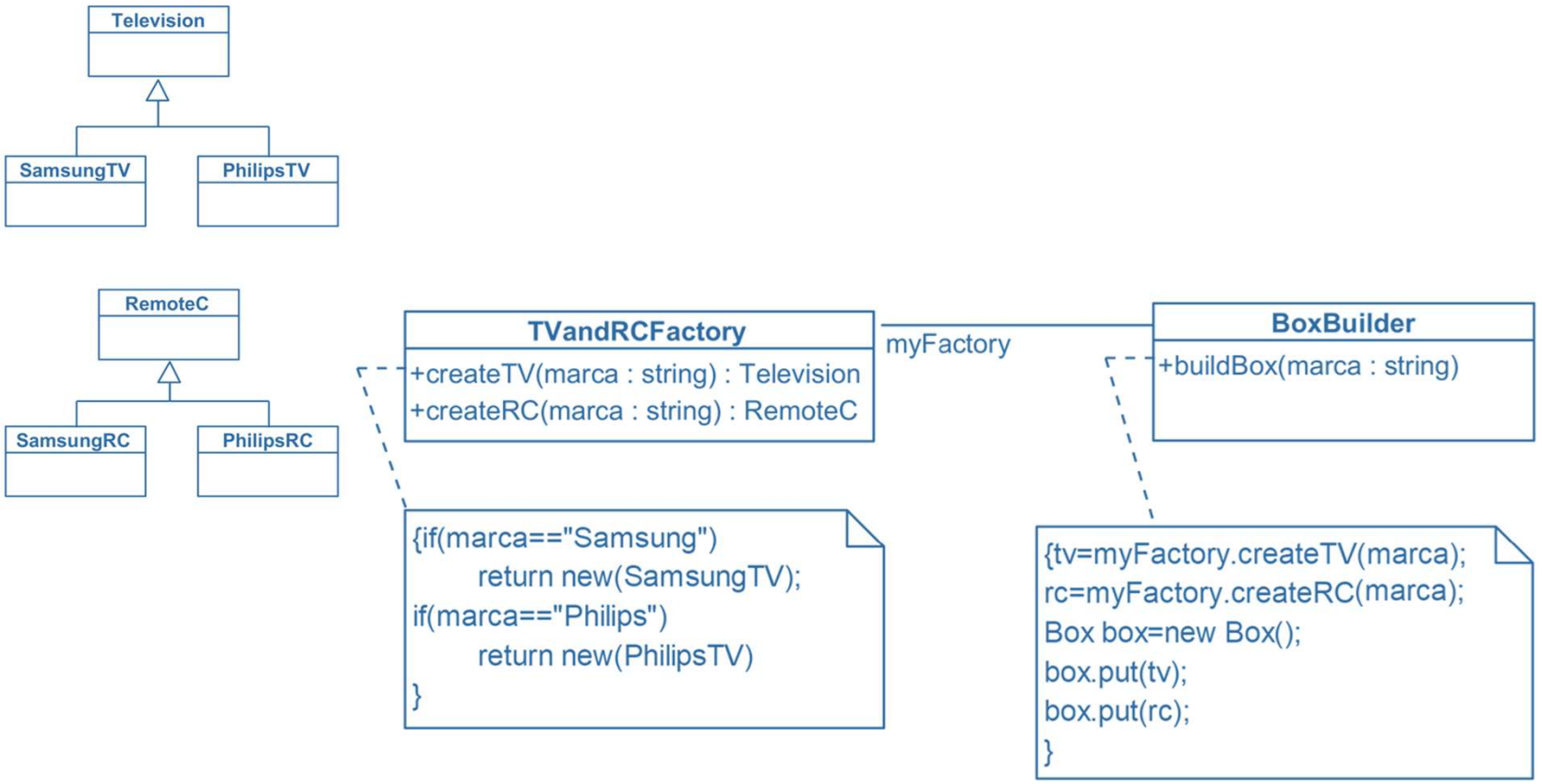
- Prodotti: TV e Remote Control (RC)
- Brand: Samsung e Philips

(NB: Una TV Samsung usa un Samsung RC, mentre una TV Philips usa un Philips RC)

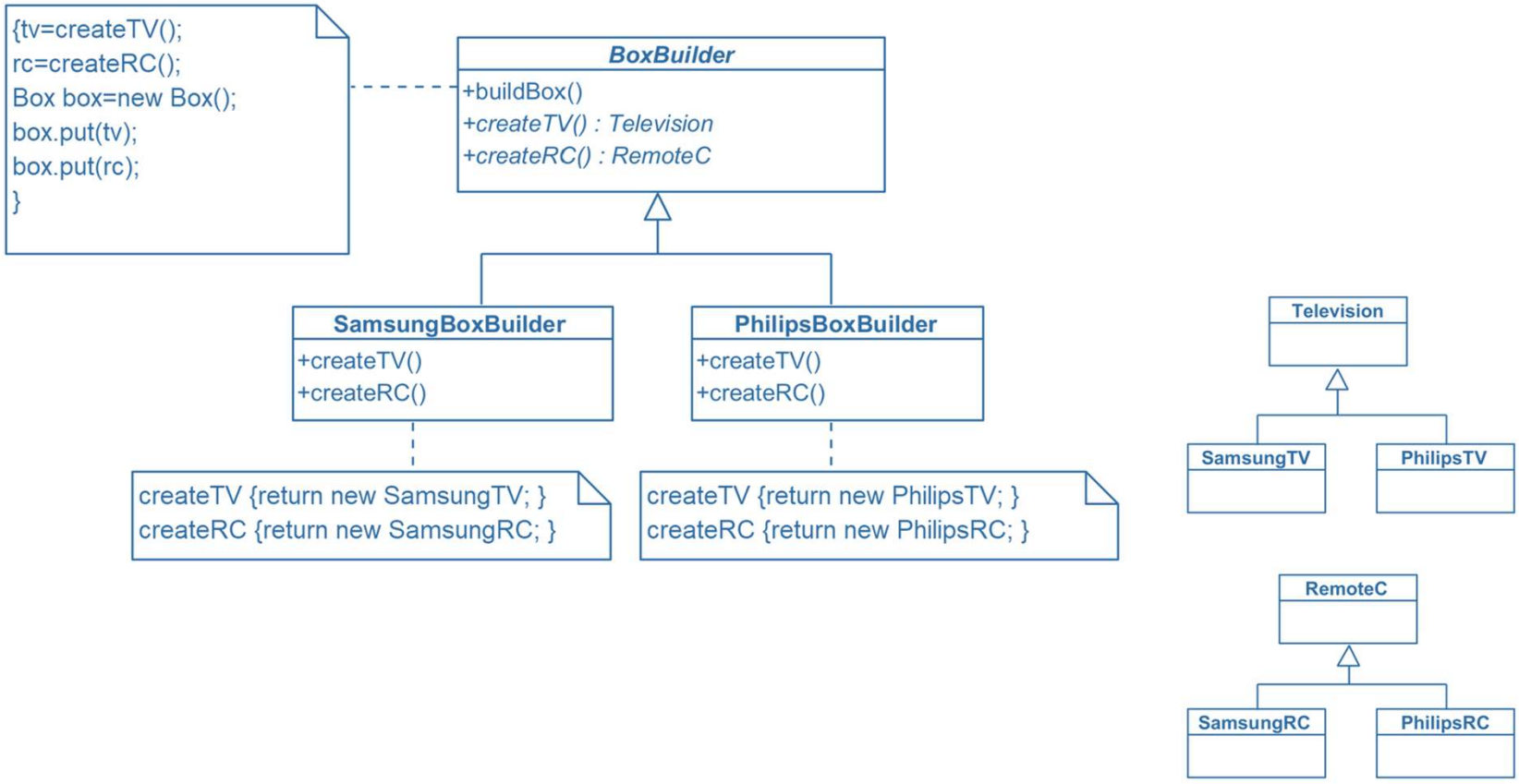
Fornire quanto segue:

1. Implementazione con Simple Factory (unico factory con parametri)
2. Implementazione con Factory Method (un creator astratto costruito per avere una TV ed il suo RC, che poi vengono impacchettati in un box)
3. Implementazione con Abstract Factory (un cliente sceglie quale factory usare e richiede la creazione dei prodotti che servono, per poi impacchettarli in un box)

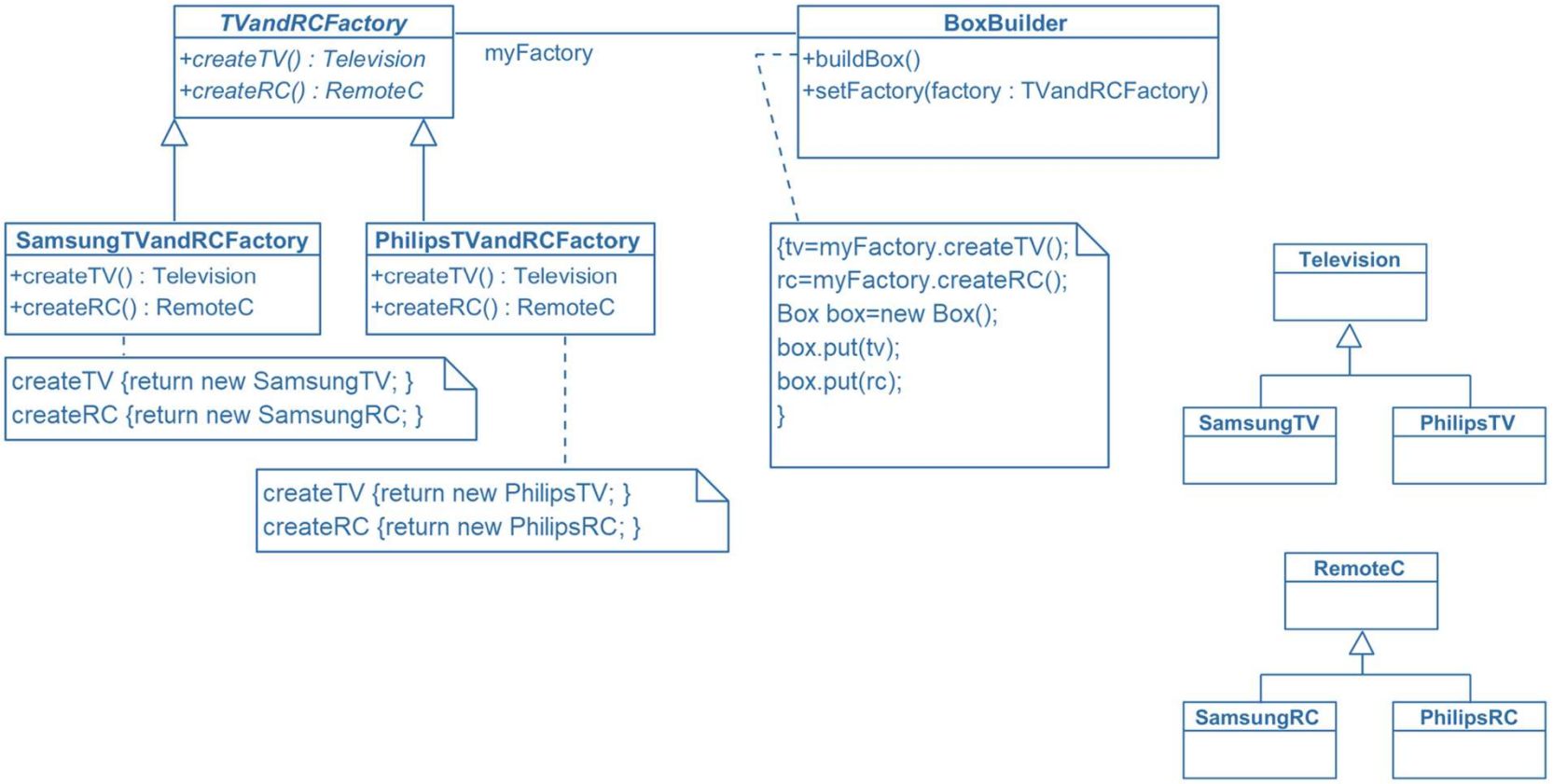
POSSIBILE SOLUZIONE: SIMPLE FACTORY



POSSIBILE SOLUZIONE: FACTORY METHOD



POSSIBILE SOLUZIONE: ABSTRACT FACTORY



PURE FABRICATION

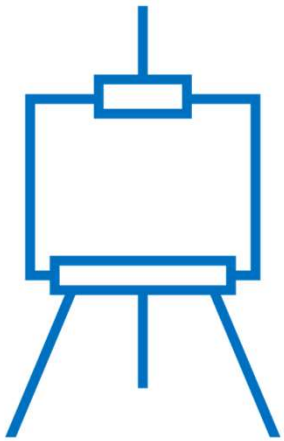
Si tratta di un **pattern GRASP** che sostanzialmente assegna **responsabilità** fortemente **correlate** ad una **classe artificiale**

- Implementa **behavioural decomposition** con l'obiettivo di supportare **high cohesion**, **low coupling** e **riuso**
- La classe artificiale **non rappresenta niente** nel dominio del problema
- La classe artificiale è introdotta **solo** per **convenienza** del **designer**

(NB: Il nome **pure fabrication** deriva proprio da questi ultimi due punti)

In generale, un **factory** è un **pure fabrication** con l'obiettivo di

- Confinare la responsabilità di creazioni complesse in oggetti coesi
- Incapsulare la complessità della logica di creazione



SINGLETON

UNA FABBRICA DI CIOCCOLATO

Consideriamo una fabbrica di cioccolato, in cui un sistema informatico controlla la fase riguardante i «chocolate boiler», il cui compito è quello di

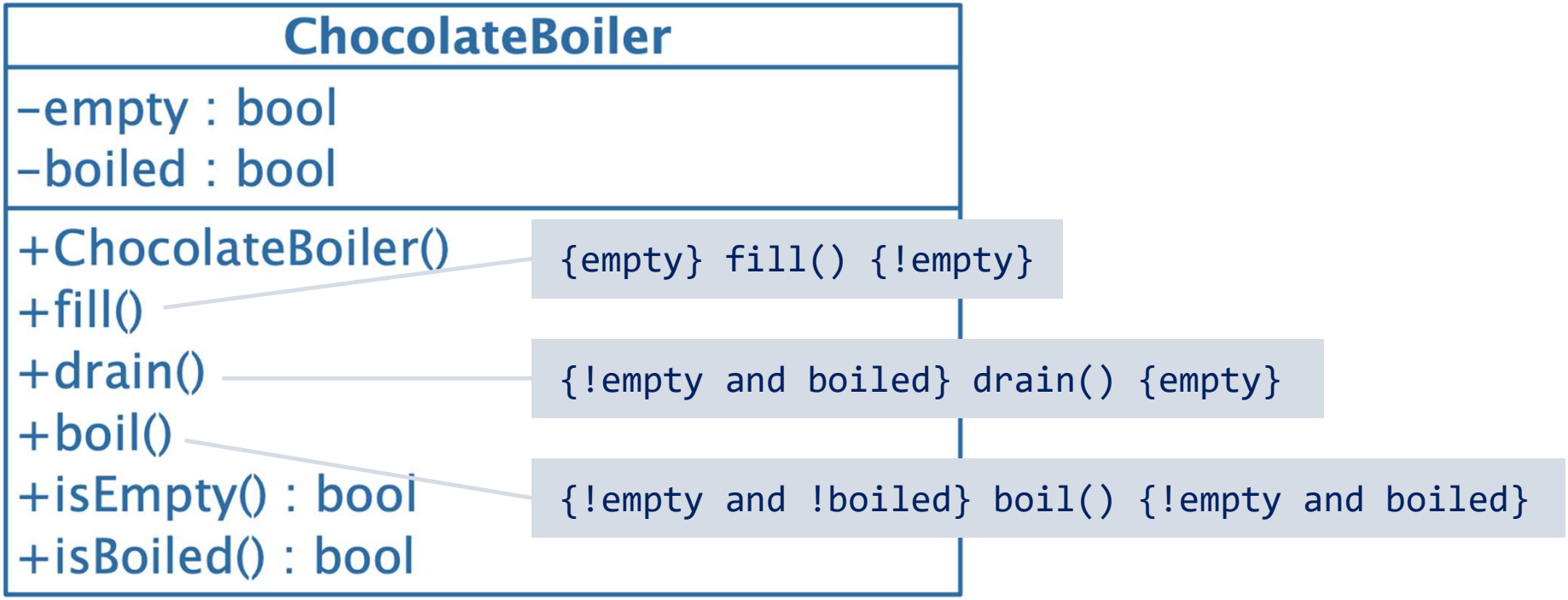
- mescolare latte e cioccolato,
- metterli in un bollitore e
- passarli fase successiva (creazione delle barrette di cioccolato)

Una delle principali funzioni del sistema è quella di prevenire incidenti, ad esempio

- disperdere 500 litri di prodotto non ancora bollito
- riempire il bollitore quando è già pieno
- accendere il bollitore quanto è vuoto

CONTROLLARE CHOCOLATE BOILER

I boiler sono controllati dalla classe ChocolateBoiler



PROBLEMA

Il cioccolato è strabordato dal boiler! È stato aggiunto altro latte anche se il boiler era già pieno!

Come mai?

- Suggerimento: cosa succede se creiamo più istanze di `ChocolateBoiler`?
- Due istanze che controllano lo stesso boiler potrebbero avviare la procedura fill contemporaneamente

IL SINGLETON PATTERN

Garantisce che una classe abbia **una sola istanza**
fornendo un punto di **accesso globale** a tale istanza

Attenzione

- La necessità di avere **oggetti unici** è abbastanza comune
- La maggior parte degli oggetti in un'applicazione ha un'unica responsabilità assegnata
- Le **classi singleton** però sono relativamente **rare**

  → Il fatto che un oggetto/classe sia unico non significa che usi il **singleton** pattern

IL SINGLETON PATTERN (CONT.)

- Risponde alla necessità di avere una singola istanza di una classe in un sistema
  (p.e. window manager, o un unico factory per un insieme di prodotti)
- Rende tale istanza facilmente accessibile
- Garantisce che non vengano create altre istanze

Come?

IL SINGLETON PATTERN (CONT.)

- Risponde alla necessità di avere una singola istanza di una classe in un sistema
- Rende tale istanza facilmente accessibile
- Garantisce che non vengano create altre istanze

1) Si rende il costruttore della classe privato

```
private ChocolateBoiler() = { ... }
```

2) Si aggiunge un oggetto statico privato (che conterrà l'unica istanza disponibile)

```
private static ChocolateBoiler _chocolateBoiler = new ChocolateBoiler()
```

3) Si rende l'unica istanza disponibile accessibile solo attraverso un metodo statico

```
public static ChocolateBoiler getChocolateBoiler() { return _chocolateBoiler; }
```

INIZIALIZZAZIONE LAZY

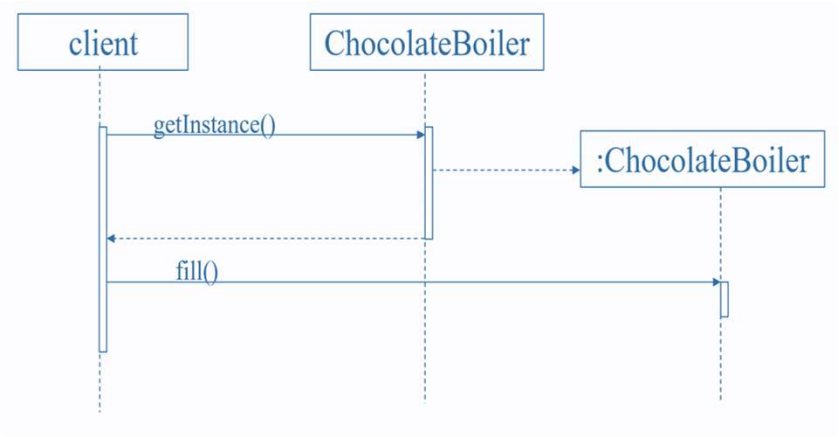
Invece di creare ogni oggetto singleton dall'inizio, meglio **aspettare che sia necessario**

```
private static ChocolateBoiler _chocolateBoiler = null;

public static ChocolateBoiler GetChocolateBoiler() {
  if (_chocolateboiler == null) {
    _chocolateboiler = new ChocolateBoiler();
  }
  return _chocolateboiler
}
```

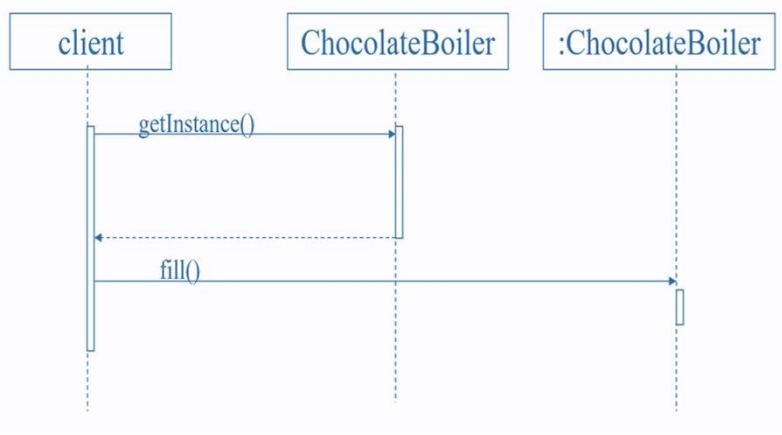
INIZIALIZZAZIONE LAZY: PERCHÉ?

- Potremmo non avere informazioni sufficienti per istanziare il singleton staticamente
  - p.e. un ChocolateBoiler potrebbe dover attendere che siano stati stabiliti i canali di comunicazione con gli altri macchinari della fabbrica di cioccolato

- Il singleton potrebbe essere «resource intensive» e non necessario
  - p.e. un programma che si connette ad un database ed esegue query opzionali

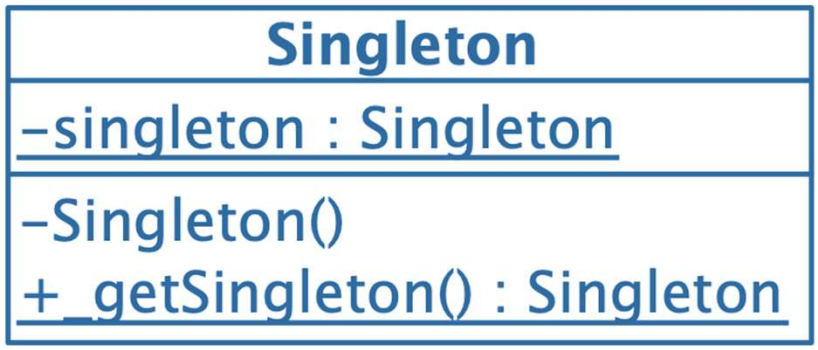


`_chocolateBoiler == null`

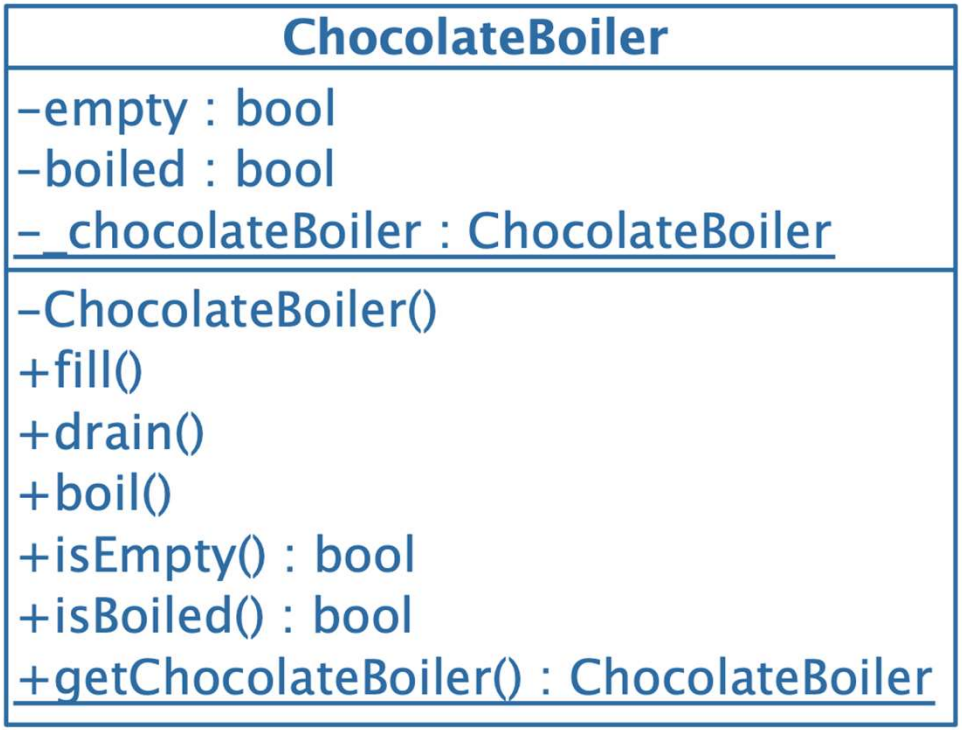


`_chocolateBoiler != null`

IL SINGLETON PATTERN, IN UML



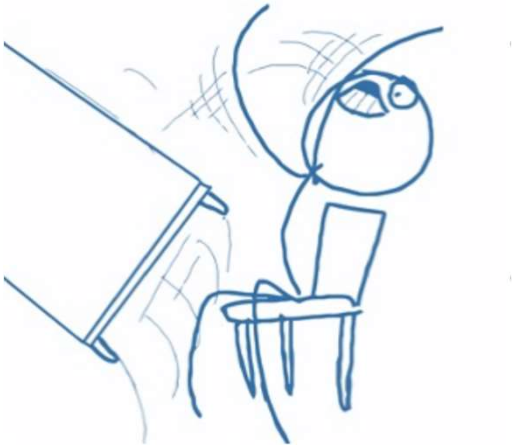
nel nostro esempio



Funzionerà anche in ambienti **multi-thread**?

Si.

Cos'è successo?
Il codice del singleton torna
Sarà mica l'ottimizzazione multi-thread?



SINGLETON E MULTI-THREAD

Thread diversi potrebbero inizializzare **più istanze** del singleton (quasi) **simultaneamente**

Thread 1	Thread 2
`public static ChocolateBoiler getInstance()`	
	`public static ChocolateBoiler getInstance()`
`if (_chocolateBoiler == null)`	
	`if (_chocolateBoiler == null)`
`_chocolateBoiler = new ChocolateBoiler()`	
	`_chocolateBoiler = new ChocolateBoiler()`
`return _chocolateBoiler`	
	`return _chocolateBoiler`

SINGLETON E MULTI-THREAD

Thread diversi potrebbero inizializzare **più istanze** del singleton (quasi) **simultaneamente**

Possibili soluzioni:

- Inizializzare **eager** (invece di inizializzazione lazy)

  → OK, ma alloca memoria anche per istanze che «non servono»

- **Sincronizzazione** del metodo «`getInstance`» (dichiarandolo **`synchronized`**)

  → OK, ma può ridurre le performance del sistema

Come si fa?

- **Double-checked locking**

  → Evita sincronizzazioni costose per tutte le invocazioni eccetto la prima

DOUBLE-CHECKED LOCKING

```
public class Singleton {
  private volatile static Singleton uniqueInstance;
  private Singleton() { ; }
  public static Singleton getInstance() {
    if (uniqueInstance == null) {
      synchronized(Singleton.class) {
        if (uniqueInstance == null) {
          uniqueInstance = new Singleton();
        }
      }
    }
    return uniqueInstance;
  }
}
```

Una variabile **volatile** garantisce che **ogni thread** acceda all'**ultimo valore** assegnatole (quando la referenzia)

Solo un thread alla volta può eseguire un blocco di codice **synchronized**

Servono due controlli sulla variabile **uniqueInstance** per garantire che sia ancora **null**

SINGLETON E SOTTOCLASSI

Cosa succede se **estendiamo** una classe **singleton**? Come possiamo consentire che l'unica istanza sia istanza di una **sottoclasse**?

Ad esempio:

- `MazeFactory` ha due sottoclassi `EnchantedMazeFactory` and `AgentMazeFactory`
- Vogliamo istanziarne solo una

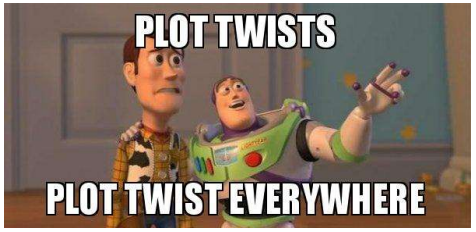
Come fare?

- Un metodo statico `getInstance` in `MazeFactory` che decide quale sottoclasse istanziare (es. in base a parametri o variabili d'ambiente)

ESEMPIO

```
public abstract class MazeFactory {
  // The private reference to the one and only instance
  private static MazeFactory uniqueInstance = null;

  // Create the instance using the specified String name
  public static MazeFactory getInstance(String name) {
    if(uniqueInstance == null) {
      if(name.equals("enchanted"))
        uniqueInstance = new EnchantedMazeFactory();
      else if(name.equals("agent"))
        uniqueInstance = new AgentMazeFactory();
    }
    return uniqueInstance;    // this may not be the one specified by the parameter
  }
}
```

I **costruttori** delle sottoclassi devono essere **pubblici** per consentire a `MazeFactory` di istanziarle (ma anche **altre classi** possono farlo)



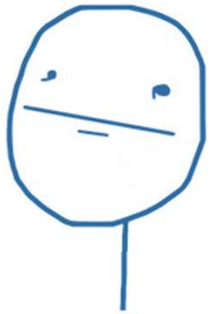
ESEMPIO (CONT.)

Inoltre, `getInstance` **viola** il principio **open-closed** (richiede modifiche per nuove sottoclassi)

Possiamo però usare i nomi delle classi Java come argomenti del metodo `getInstance`

```
public static MazeFactory getInstance(String name) {
  if (uniqueInstance == null)
  // Usa la reflection per creare un'istanza della classe specificata da nome
    uniqueInstance = Class.forName(name).newInstance();
  return uniqueInstance;
}
```

Ma questo comunque non risolve il nostro problema



SINGLETON E SOTTOCLASSI

Cosa succede se **estendiamo** una classe **singleton**? Come possiamo consentire che l'unica istanza sia istanza di una **sottoclasse**?

Ad esempio:

- `MazeFactory` ha due sottoclassi `EnchantedMazeFactory` and `AgentMazeFactory`
- Vogliamo istanziarne solo una

Come fare?

- Un metodo statico `getInstance` in `MazeFactory` che decide quale sottoclasse istanziare (es. in base a parametri o variabili d'ambiente)
  → I costruttori delle sottoclassi devono essere pubblici
  → Altri clienti possono creare altre istanze delle sottoclassi ☹
- Ogni sottoclasse fornisce un metodo statico `getInstance`
  →I costrtuttori delle sottoclassi possono essere privati

ESEMPIO

```
public abstract class MazeFactory {
  // The protected reference to the one and only instance
  protected static MazeFactory uniqueInstance = null;
  // The MazeFactory constructor (a default constructor cannot be private)
  protected MazeFactory() {}
  // Returns a reference to the single instance
  public static MazeFactory getInstance() { return uniqueInstance; }
}
```

```
public class EnchantedMazeFactory extends MazeFactory {
  // Return a reference to the single instance.
  public static MazeFactory getInstance() {
    if(uniqueInstance == null) uniqueInstance = new EnchantedMazeFactory();
    return uniqueInstance;
  }
  // Private subclass constructor
  private EnchantedMazeFactory() {}
}
```

ESEMPIO (CONT.)

Codice per la **creazione** della factory (lato client)

```
MazeFactory factory = EnchantedMazeFactory.getInstance()
```

Codice per l'**accesso** alla factory (lato client)

```
MazeFactory factory = MazeFactory.getInstance()
```

Osservazioni

- Se un client accede alla factory prima che sia creata, ottiene **`null`**
- La `uniqueInstance` ora è **`protected`**
- I costruttori delle sottoclassi ora sono **`private`**

  ⇒ se ne può creare **solo un'istanza**

MEMENTO: ATTRIBUTI STATICI

(Ogni oggetto di una classe ha la propria copia di tutte le variabili di istanza di quella classe)

In alcuni casi, **tutti gli oggetti** di una classe devono condividere **una sola copia** di una variabile

⇒ Variabile **statica**

- Rappresenta informazioni relative all'intera classe
- Se ne tiene in memoria una sola copia (indipendentemente dal numero di oggetti)
- Accessibile attraverso il nome della classe e l'operatore punto (es. Math.PI)
- Accessibile solo attraverso i metodi e le proprietà della classe

SINGLETON VS. CLASSI STATICHE

Pro

- Un singleton può essere passato come **parametro** a un altro metodo
- Un singleton può essere esteso da una o più **sottoclassi**
- Un singleton può essere istanziato mediante **fabric** (scegliendo anche quale classe istanziare)

Cons

- Quelli appena visti
- https://www.oracle.com/technical-resources/articles/java/singleton.html

In entrambi i casi, **attenzione** al **multi-threading**!


Q
A

RIFERIMENTI

Contenuti

- **Capitoli 16 e 17** di "Software Engineering" (G. C. Kung, 2023)

Approfondimenti