

Introduzione

- Abstract Factory
 - è *creazionale* e *basato su oggetti*
 - ovvero, la *creazione* degli oggetti è *delegata* ad apposite classi
- Scopo:
 - « *fornire un'interfaccia per la **creazione** di **famiglie di oggetti correlati o dipendenti** senza specificare quali siano le loro classi concrete* » (GoF)
- Detto anche:
 - kit

Un esempio

- Vogliamo creare un labirinto:

```
public class MazeGame {  
    ...  
    public Maze newMaze() {  
        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.setDoor(MazeGame.East);  
        r2.setDoor(MazeGame.West);  
        return maze;  
    }  
}
```

*Creiamo un
labirinto con
due stanze e
una porta...*

Un esempio

- E se...
 - volessimo creare uno speciale labirinto “incantato”, con le stanze e le porte magiche?
 - oppure un labirinto con le bombe?
 - oppure con le trappole
 - oppure...

Un esempio, esteso

- Certo, potremmo fare così...

```
public Maze newMaze(int mazeType) {  
    switch(mazeType) {  
        case ENCHANTED:  
            Maze maze = new EnchantedMaze();  
            Room r1 = new EnchantedRoom();  
            Room r2 = new EnchantedRoom();  
            Door door = new DoorNeedingSpell(r1, r2);  
            break;  
        case BOMBED:  
            Maze maze = new BombedMaze();  
            Room r1 = new RoomWithBomb();  
            Room r2 = new RoomWithBomb();  
            Door door = new BombedDoor(r1, r2);  
            break;  
        case ...
```

- Ma cosa succede quando abbiamo tanti tipi diversi di labirinti?



Un esempio, esteso

- Problema: questo metodo è *inflexibile*
 - il metodo `newMaze()` deve conoscere tutti i tipi possibili di labirinti, e i nomi delle relative classi
 - idealmente, i compiti del nostro `newMaze()` dovrebbero essere solo:
 - 1) creare un labirinto, senza occuparsi del tipo
 - 2) creare due stanze e una porta
 - 3) aggiungere le stanze e la porta al labirinto
 - ... indipendentemente dal tipo particolare di labirinto creato!

Le *factory*

- Soluzione: spostare la *logica di creazione* da qualche altra parte
 - ovvero, **delegare** la creazione dei vari oggetti ad una classe apposita
 - una classe di questo tipo prende il nome di ***Factory***
 - gli oggetti creati da una *factory* vengono chiamati “prodotti”

Factory astratta

- Creiamo una *factory* per ogni tipo di labirinto
 - EnchantedMazeFactory
 - BombedMazeFactory
 - ...
- Tutte queste *factory* sono sottoclassi di un'unica **factory astratta**
- Il metodo newMaze() deve conoscere solo l'interfaccia della *factory* astratta

Ritornando al labirinto...

- La *factory* astratta:

```
public abstract class MazeFactory {  
    public Maze createMaze() { return new Maze(); }  
    public Room createRoom() { return new Room(); }  
    public Door createDoor(r1, r2) { return new Door(r1, r2); }  
    ... }
```

In questo caso i metodi hanno un'implementazione, ma in generale potrebbero anche essere astratti...

- e le *factory* concrete:

```
public class EnchantedMazeFactory extends MazeFactory {  
    public Maze createMaze() { return new EnchantedMaze(); }  
    public Room createRoom() { return new EnchantedRoom(); }  
    public Door createDoor(r1, r2) {  
        return new DoorNeedingSpell(r1, r2);  
    }  
    ... }
```

```
public class BombedMazeFactory extends MazeFactory { ... }
```


Il metodo newMaze()

- Ora il metodo newMaze() diventa:

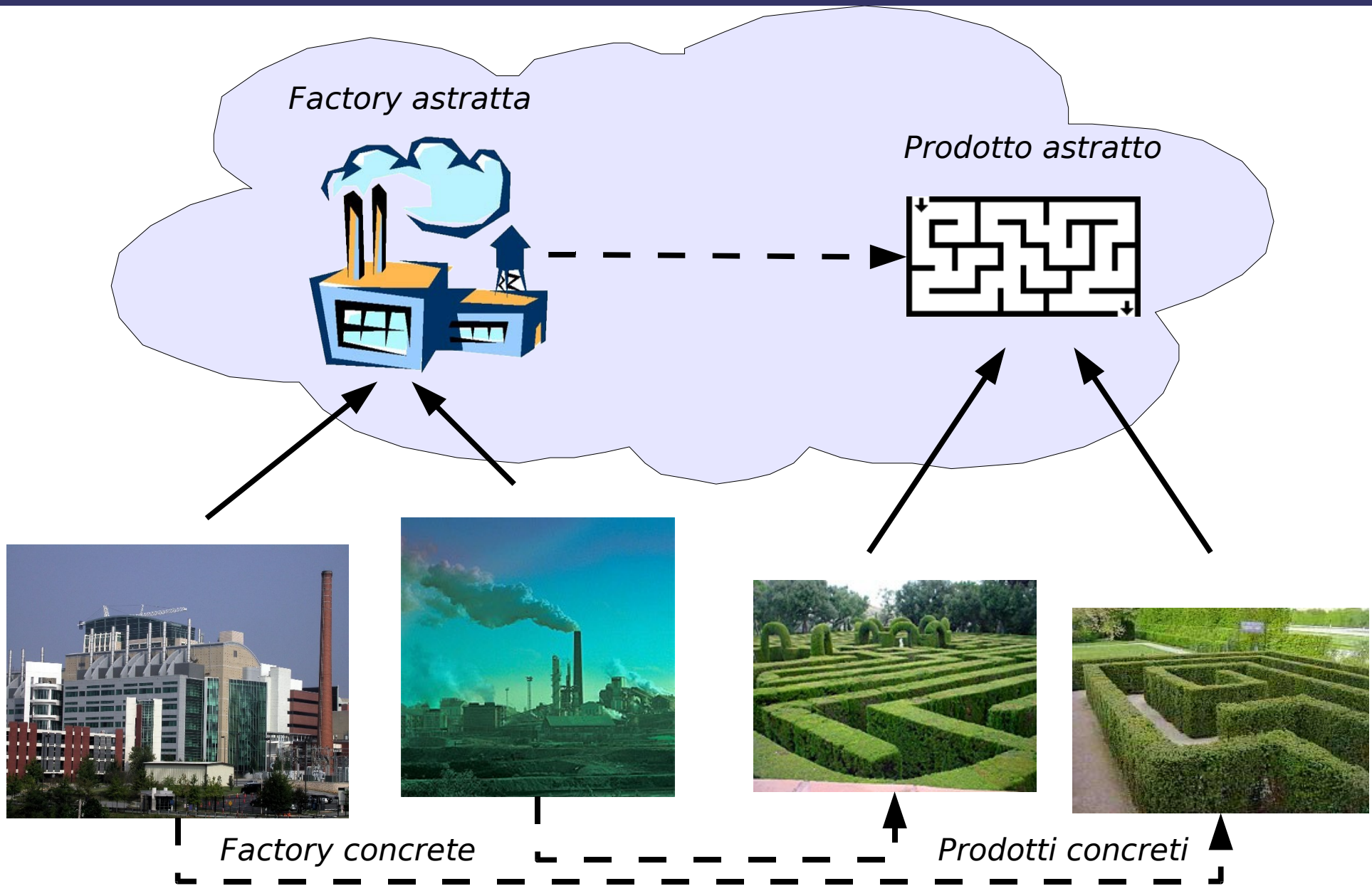
```
public Maze newMaze(MazeFactory factory) {  
    Maze maze = factory.createMaze();  
    Room r1 = factory.createRoom();  
    Room r2 = factory.createRoom();  
    Door door = factory.createDoor(r1, r2);  
    ...  
    return maze;  
}
```



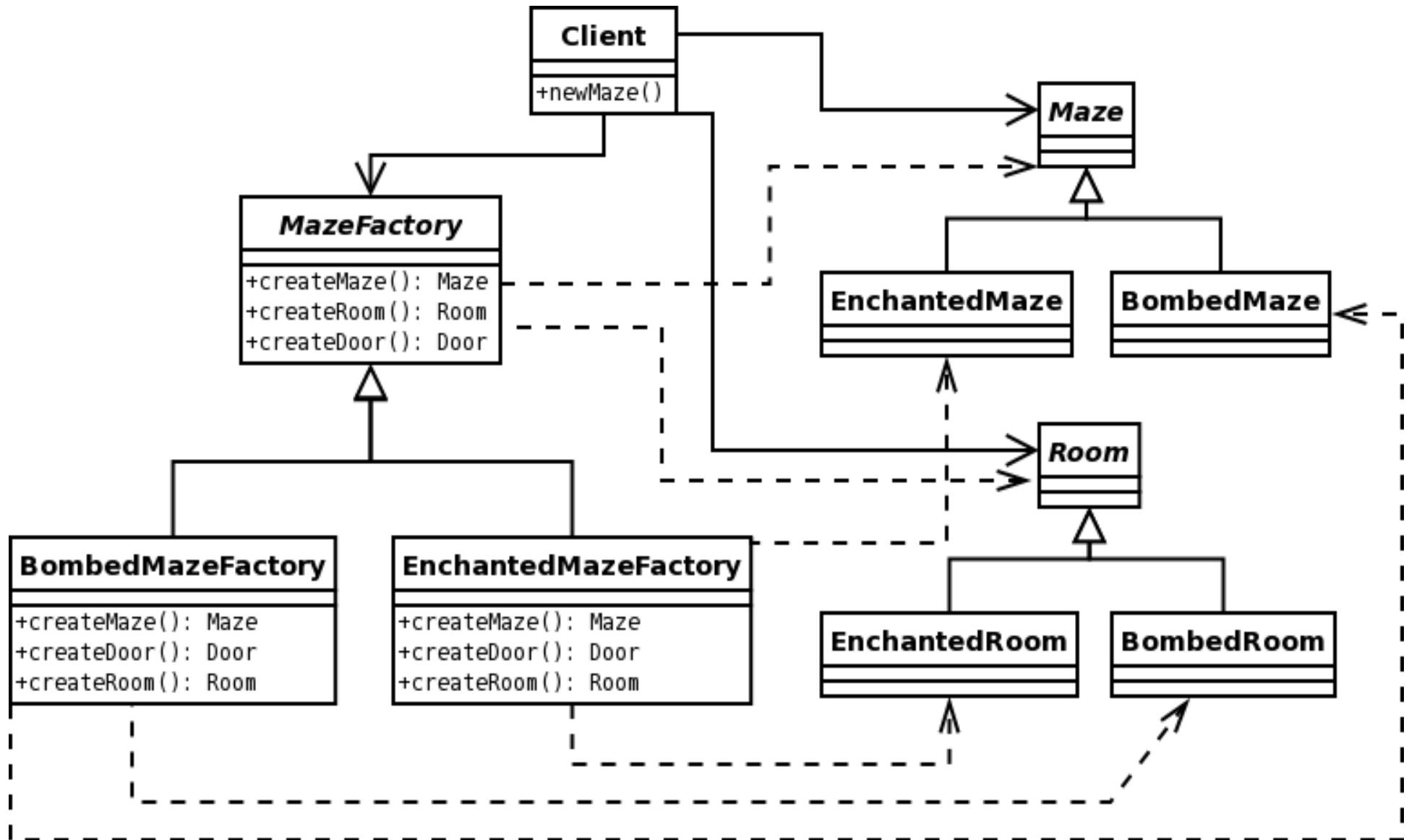
Il metodo newMaze()

- La *factory* concreta da usare viene scelta in un altro punto del programma
 - in un unico punto! Quindi è facile da cambiare
 - per esempio la *factory* concreta può essere scelta durante l'inizializzazione del programma
- newMaze() **non sa più niente** circa i diversi tipi di “prodotti” (labirinti, porte, stanze, ...)
 - ovvero, non conosce i “prodotti” concreti, ma solo quelli **astratti**
 - quindi non deve conoscere le *factory* concrete, ma solo quella **astratta**

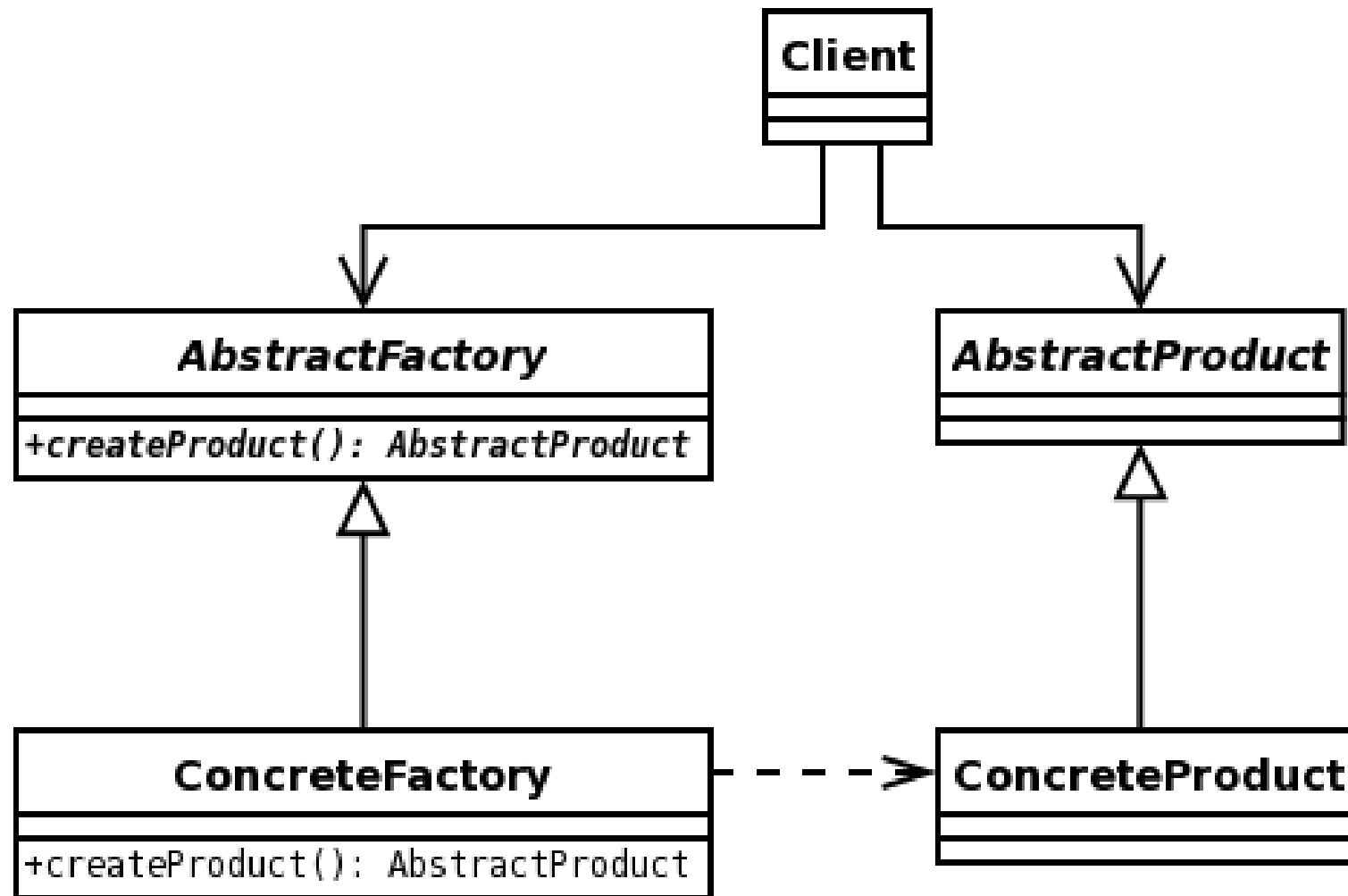
In immagini...



In UML...



Lo schema UML generale



I partecipanti

- Le *factory*
 - **AbstractFactory**: dichiara un'interfaccia per la *creazione* di oggetti prodotto astratti
 - **ConcreteFactory**: implementa le operazioni di *creazione* degli oggetti prodotto concreti
- I prodotti
 - **AbstractProduct**: dichiara un'interfaccia per una tipologia di oggetti prodotto
 - **ConcreteProduct**: definisce un oggetto prodotto che verrà creato dalla corrispondente *factory* concreta
- Il Client: utilizza soltanto le interfacce dichiarate dalle classi AbstractFactory e AbstractProduct

Applicabilità

- Quando applicare il pattern *Abstract Factory*?
 - 1) quando vogliamo un sistema **indipendente** dalle modalità di creazione, composizione, rappresentazione dei suoi “prodotti”
 - aggiungiamo altri 100 tipi di labirinti: dobbiamo cambiare qualcosa in `newMaze()` ?



Platform
Independence!

Manufacturer
Independence!

Applicabilità

- Quando applicare il pattern *Abstract Factory*?
 - 2) quando vogliamo avere la possibilità di scegliere tra più **famiglie** (“kit”) di prodotti
 - 3) quando vogliamo essere sicuri di usare solo i prodotti della famiglia scelta
 - non vogliamo mescolare prodotti appartenenti a famiglie diverse!
 - 4) quando abbiamo una libreria di classi di cui vogliamo rivelare solo le interfacce, non le implementazioni

Un altro esempio

- Vogliamo scrivere un'applicazione grafica in grado di funzionare su desktop diversi
 - Windows
 - Mac OSX
 - GTK
 - ...
- Le operazioni per creare gli elementi grafici (*widget*) cambiano a seconda del desktop

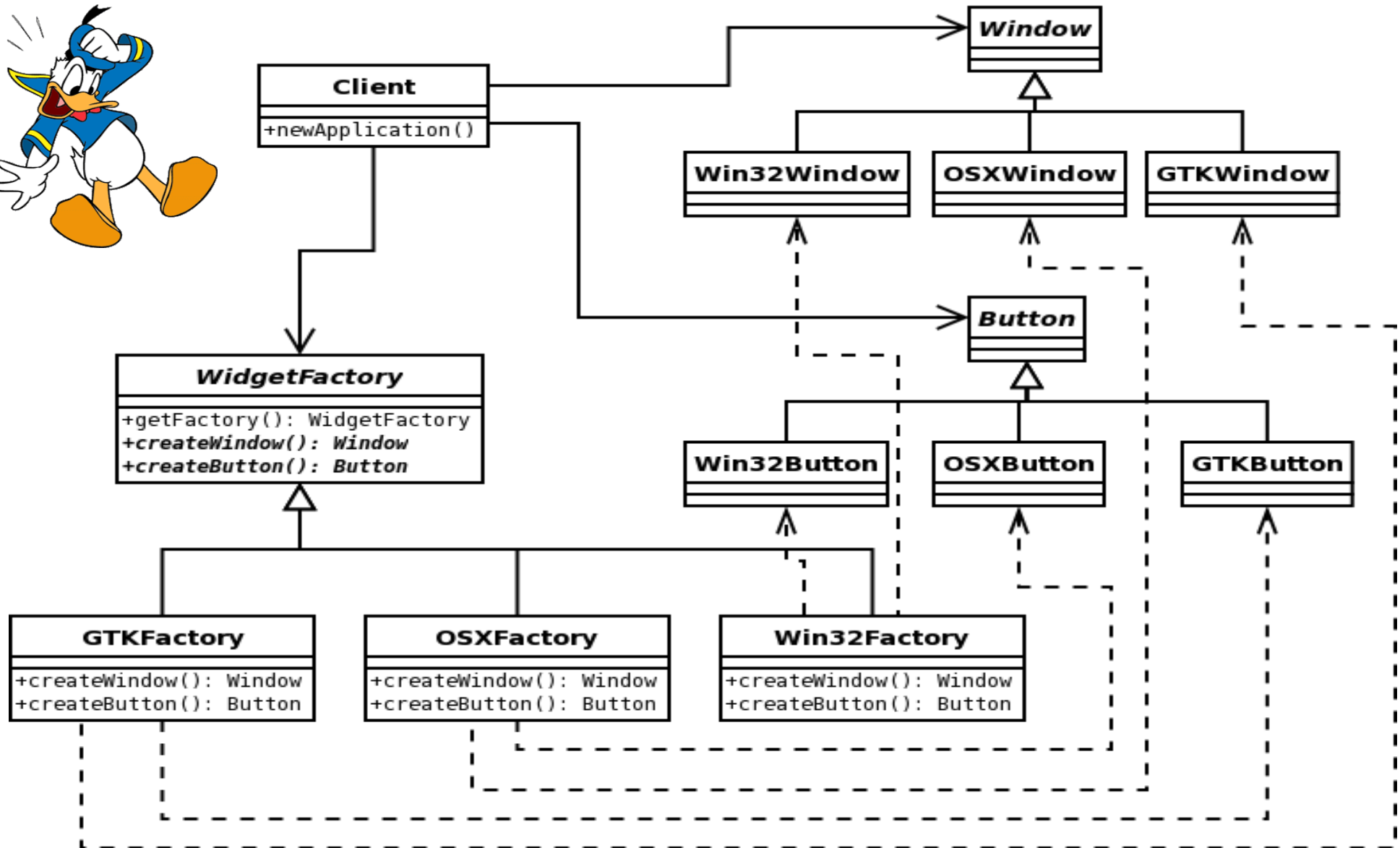
Un altro esempio

- Obiettivi:
 - rendere l'applicazione **indipendente** dal sistema desktop utilizzato
 - **non mescolare** insieme elementi appartenenti a differenti desktop
- L'applicazione vuole utilizzare vari *widget* (finestre, scrollbar, menu, bottoni)...
- ... ma **non vuole sapere** quale particolare famiglia di *widget* sta utilizzando!

Un altro esempio

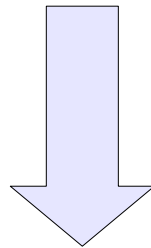
- Una piccola dimostrazione pratica:
 - `demoAbstractFactory.java`

Un altro esempio



Vantaggi di *Abstract Factory*

- **Isola** le classi concrete dei prodotti
 - il client manipola le istanze dei prodotti solo attraverso le loro interfacce astratte
 - la creazione di nuovi prodotti è responsabilità delle *factory*



Il client è **indipendente** dalle classi effettivamente utilizzate per l'implementazione dei prodotti

Vantaggi di *Abstract Factory*

- Consente di **cambiare** in modo semplice la famiglia di prodotti utilizzata
 - in tutta l'applicazione, la scelta della *factory* concreta compare in un punto solo
- Promuove la **coerenza** nell'utilizzo dei prodotti
 - i prodotti di una stessa famiglia sono progettati per essere utilizzati insieme
 - quando un'applicazione sceglie una *factory*, userà solo i prodotti di quella *factory*

Uno svantaggio...

- L'aggiunta del supporto per nuove tipologie di prodotti è difficile
 - il set di prodotti è determinato dall'interfaccia di *Abstract Factory*...
 - ... quindi, se vogliamo aggiungere un prodotto, dobbiamo modificare l'interfaccia di *Abstract factory* e, di conseguenza, tutte le *factory* concrete
 - una possibile soluzione: definire *factory* **estendibili**

Factory estendibili

- Nelle *factory*, invece di definire un metodo di creazione per ogni prodotto...
 - createMaze()
 - createDoor()
 - createRoom()
 - create...
- ... definiamo un **unico metodo** *make*
- *make* assume come input un parametro che indica il tipo di oggetto da creare
 - make(String type)

Factory estendibili

- Problema: che cosa ritorna *make*?
 - possiamo definire una unica interfaccia astratta
Product: `Product make(String type)`
 - ... e definiamo tutti i vari prodotti come sottoclassi di *Product*
 - in questo modo, però, il client può accedere ai prodotti solo attraverso l'interfaccia generica di *Product*
 - se ci sono operazioni specifiche per un prodotto, non sono accessibili attraverso *Product*
 - soluzione (poco sicura): *downcast*

Abstract Factory + Prototype

- Nell'esempio del labirinto, supponiamo di voler aggiungere dei nuovi tipi di labirinto
 - che magari si differenziano tra loro solo per alcuni dettagli
 - per esempio, un labirinto con le stanze incantate ma le porte normali...
- Per ogni nuova “famiglia” di labirinti dobbiamo definire una nuova *factory* concreta
 - anche se magari è simile ad altre *factory* già esistenti

Abstract Factory + Prototype

- ... oppure possiamo usare il pattern Prototype
 - definiamo **una sola** *factory* concreta
 - quando il client vuole usare la *factory*, la inizializza con i **prototipi** dei prodotti da creare
 - un prototipo è un'istanza di un prodotto
 - quando la *factory* deve creare un nuovo prodotto, lo **clona** dal prototipo

Abstract Factory + Prototype

- Per esempio:

```
public class ProtoMazeFactory extends MazeFactory {  
    private Maze protoMaze; private Room protoRoom; private Door protoDoor;  
    ...  
    public MazeFactoryPrototype(Maze, Room, Door) {  
        protoMaze = Maze; protoRoom = Room; protoDoor = Door;  
    }  
  
    public createMaze() {  
        return protoMaze.clone();  
    }  
    ...  
}
```

- Quando il client inizializza una nuova *factory*:

```
ProtoMazeFactory fancyFactory = new ProtoMazeFactory(  
    new simpleMaze(), new EnchantedRoom(), new BombedDoor());  
  
Maze maze = fancyFactory.createMaze();  
Door door = fancyFactory.createDoor();  
Room room = fancyFactory.createRoom();
```

Abstract Factory + Singleton

- Normalmente, ad una applicazione serve **una sola istanza** di una *factory* concreta
- Quindi è possibile combinare *Abstract factory* con *Singleton*
 - ogni *factory* concreta è un *singleton*
 - in questo modo ci assicuriamo che esista al più **una** istanza di ogni *factory* concreta

Conclusione

- I concetti chiave di *Abstract Factory*:
 - 1) delega della creazione:** la responsabilità della creazione di nuovi oggetti è incapsulata in classi apposite (le *factory*)
 - 2) famiglie di oggetti:** ci si assicura che i prodotti usati appartengano alla stessa famiglia (“kit”)
 - 3) indipendenza del client:** il client dipende dalle interfacce, non dalle implementazioni
- Riprendendo lo scopo:
 - « *fornire un'interfaccia per la **creazione di famiglie di oggetti correlati o dipendenti** senza specificare quali siano le loro classi concrete* » (GoF)