Abstract Factory

**Scopo e cosa è** È un design pattern che fornisce un interfaccia per creare oggetti connessi o dipendenti tra di loro, in modo che non ci sia necessità da parte del client di specificare i nomi delle classi concrete all’interno del proprio codice. In questo modo si permette che un sistema sia indipendente dall'implementazione degli oggetti concreti e che il client, attraverso l'interfaccia, utilizzi diverse famiglie di prodotti.

**Applicabilità** Questo pattern è utile quando

* si vuole un sistema indipendente da come gli oggetti vengono creati, composti e rappresentati
* si vuole permettere la configurazione del sistema come scelta tra diverse famiglie di prodotti
* si vuole che i prodotti che sono organizzati in famiglie siano vincolati ad essere utilizzati con prodotti della stessa famiglia
* si vuole fornire una libreria di classi mostrando solo le interfacce e nascondendo le implementazioni.

**Conseguenze**

1. *Isola le classi concrete*. Secondo il principio dell'incapsulamento dell'implementazione, la factory viene utilizzata solamente attraverso la sua interfaccia, per cui nei client non c'è traccia del codice per istanziare gli oggetti. In questo modo il sistema è indipendente dal tipo della classe che effettivamente implementa l'interfaccia del tipo di prodotto. I client usano i prodotti concreti attraverso la loro interfaccia comune ***AbstractProduct***, in modo che anche il codice successivo all'istanziazione sia indipendente dal nome della classe che effettivamente implementa il prodotto concreto.
2. *Consente di cambiare in modo semplice la famiglia di prodotti utilizzata*. La factory viene istanziata una sola volta nel codice, quindi è sufficiente cambiare il tipo di factory istanziato in quel punto del sorgente per utilizzare un diverso tipo di prodotti. La coerenza col resto del codice è assicurata dall'utilizzo delle interfacce astratte e non delle classi concrete secondo il principio di *programmazione verso l'interfaccia e non verso l'implementazione*
3. *Promuove la coerenza nell'utilizzo dei prodotti*. Se i prodotti di una famiglia sono stati esplicitamente progettati per lavorare insieme, l'interfaccia ***AbstractFactory*** permette di rispettare questo vincolo.
4. *Difficile aggiungere supporto per nuove tipologie di prodotti*. Dato che ***AbstractFactory*** definisce tutte le varie tipologie di prodotti che è possibile istanziare, aggiungere una famiglia significa modificare l'interfaccia della factory. La modifica si ripercuote a cascata nelle factory concrete e in tutte le sottoclassi, rendendo laboriosa l'operazione.

**Implementazione**

* *Istanziare una sola factory*. Creando ogni ConcreteFactory come [Singleton](https://it.wikipedia.org/wiki/Singleton) ci si assicura che esista una sola istanza della classe a *run-time*, accessibile pubblicamente.
* *Istanziare i prodotti*. Poiché AbstractFactory definisce solo l'interfaccia, la creazione dei prodotti è responsabilità delle classi ConcreteFactory. Si può utilizzare un [Factory method](https://it.wikipedia.org/wiki/Factory_method" \o "Factory method) per ogni prodotto, metodi che saranno sovrascritti dalle factory concrete. Lo svantaggio di questa tecnica è l'obbligo di dover implementare una factory diversa anche se i tipi di prodotto sono molto simili tra loro. Per risolvere questo problema si può usare il [Prototype pattern](https://it.wikipedia.org/wiki/Prototype_pattern" \o "Prototype pattern).

**public** **abstract** **class** **GUIFactory** {

**public** **static** GUIFactory getFactory(int value) {

**if** (value == 1) {

**return** **new** WinFactory();

} **else** {

**return** **new** OSXFactory();

}

}

**public** **abstract** Button createButton();

}

**class** **WinFactory** **extends** GUIFactory {

**public** Button createButton() {

**return** **new** WinButton();

}

}

**class** **OSXFactory** **extends** GUIFactory {

**public** Button createButton() {

**return** **new** OSXButton();

}

}

**public** **abstract** **class** **Button** {

**public** **abstract** void paint();

}

**class** **WinButton** **extends** Button {

**public** void paint() {

System.out.println("Sono un WinButton!");

}

}

**class** **OSXButton** **extends** Button {

**public** void paint() {

System.out.println("Sono un OSXButton!");

}

}

**public** **class** **Application** {

**public** **static** void main(String[] args) {

GUIFactory factory = GUIFactory.getFactory(1);

Button button = factory.createButton();

button.paint();

}

}