Progettazione

- Creare un oggetto specificandone la classe esplicitamente
  - Orienta ad una particolare implementazione invece che ad una interfaccia
  - Può complicare i cambiamenti futuri
  - E' meglio creare oggetti indirettamente
  - Pattern: Abstract Factory, Factory Method, Prototype
- Le dipendenze da operazioni specifiche
  - Vincolano ad un modo di soddisfare una richiesta
  - Modificare a compile-time e run-time le richieste è più facile se si evita che le richieste siano inserite nel codice
  - Pattern: Chain of Responsibility, Command

Progettazione

- Dipendenze da piattaforme hardware e software
  - API esterne al sistema cambiano
  - Progettare il sistema limitando le dipendenze dalla piattaforma
  - Pattern: Abstract Factory, Bridge
- Dipendenze da rappresentazioni o implementazioni di oggetti
  - Client che conoscono come un oggetto è rappresentato, conservato, o implementato possono necessitare di essere cambiati quando l'oggetto cambia
  - Nascondere le informazioni ai client evita cambiamenti in cascata
  - Pattern: Abstract Factory, Bridge, Memento, Proxy

Progettazione

- Dipendenze da algoritmi
  - Gli algoritmi sono spesso estesi, ottimizzati e rimpiazzati durante lo sviluppo ed il riuso
  - Gli algoritmi che sono soggetti a cambiamenti devono essere isolati
  - Pattern: Builder, Iterator, Strategy, Template Method, Visitor
- Stretto accoppiamento
  - Porta a sistemi monolitici
  - Classi strettamente accoppiate sono difficili da riusare singolarmente
  - Pattern: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer

Progettazione

- Estendere funzionalità tramite sottoclassi: white box reuse
  - Richiede profonda comprensione della superclasse
  - Override di una operazione può richiedere override di un'altra
  - Può condurre a un grande numero di classi (per semplici estensioni)
  - Quindi: meglio limitare il numero di livelli in una gerarchia e quando possibile usare la delegazione anziché ereditare: black box reuse
  - Pattern: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy
- Impossibilità di modificare classi
  - Sorgenti non disponibili
  - La modifica può richiedere cambiamenti a tante altre classi
  - Pattern: Adapter, Decorator, Visitor

Organizzazione Design Pattern

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Rimanda la creazione di un oggetto ad un'altra classe

Rimanda la creazione di un oggetto ad un altro oggetto

Descrive modi per assemblare oggetti

Descrive algoritmi per il controllo del flusso

Design Pattern di Creazione

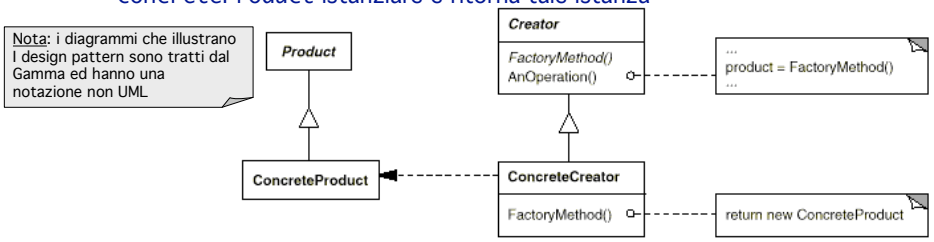
- Permettono di astrarre il processo di creazione oggetti
  - Rendono un sistema indipendente da come i suoi oggetti sono creati, composti, e rappresentati
- Sono importanti se i sistemi evolvono per dipendere più su composizioni di oggetti che su ereditarietà tra classi
  - L'enfasi va dal codificare un set ***fissato*** di comportamenti verso un più piccolo set 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
- Motivazioni
  - 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) incapsula la conoscenza su quale classe creare

Factory Method

- Soluzione
  - *Product* è l'interfaccia comune degli oggetti che il `FactoryMethod()` crea
  - `ConcreteProduct` è una implementazione dell'interfaccia *Product*
  - *Creator* è l'interfaccia che dichiara il `FactoryMethod()`
    - Tale metodo ritorna un oggetto di tipo *Product*
  - `ConcreteCreator` implementa il `FactoryMethod()` scegliendo quale `ConcreteProduct` istanziare e ritorna tale istanza



Factory Method

```
interface Shape {  //Product
  void draw();
  void fill();
}
interface ShapeCreator {  //Creator
  public Shape getShape(); //Factory Method
}
//ConcreteCreator
class CreatorA implements ShapeCreator {
  public Shape getShape() { //override
    //solo qui indico la classe da istanziare
    return new Circle();
  }
}
class Circle implements Shape { //ConcretProd
  public void draw() {
    System.out.println("A circle ( )");
  }
  public void fill() {
    System.out.println("Filled circle (o)");
  }
}
```

```
class Square implements Shape { //ConcretProd
  public void draw() {
    System.out.println("A Square [ ]");
  }
  public void fill() {
    System.out.println("Filled Square [X]");
  }
}
public class ShapeCreatorTest {
  public static void main(String args[]) {
    //istanzio il Concrete Creator
    ShapeCreator sc = new CreatorA();
    //ottengo una istanza di una
    //sottoclasse di Shape, non
    //decido qui quale sottoclasse
    Shape s = sc.getShape();

    s.draw();
    s.fill();
  }
}
```


Factory Method

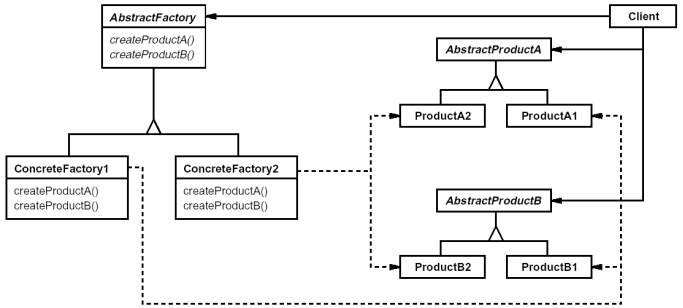
- Varianti
  - Il `FactoryMethod()` ha un parametro in ingresso per far scegliere al client la classe da creare
  - Il `FactoryMethod()` usa `Class.forName()` e `newInstance()` per togliere dal codice la dipendenza da una specifica classe
- Conseguenze
  - Il codice delle classi dell'applicazione conosce solo l'interfaccia *`Product`* e può lavorare con qualsiasi `ConcreteProduct`
  - E' necessario creare una sottoclasse di *`Creator`* solo per creare un `ConcreteProduct` (proliferazione di classi)

Abstract Factory

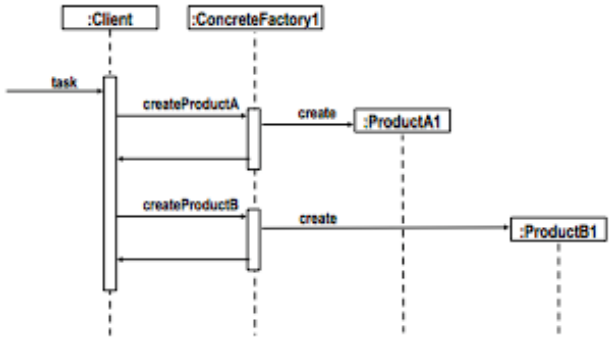
- Intento
  - Fornire una interfaccia per creare famiglie di oggetti che hanno qualche relazione senza specificare le loro classi concrete
- Problema
  - Il sistema complessivo dovrebbe essere indipendente dalle classi usate, così da essere configurabile con una di varie famiglie di classi. Le classi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

Abstract Factory

- Soluzione
  - *`AbtractFactory`* è l'interfaccia per la creazione di famiglie di oggetti specifici
  - `ConcreteFactory` implementa operazioni per creare famiglie di oggetti specifici
  - *`AbstractProduct`* è l'interfaccia per una famiglia di oggetti
  - `Product` definisce un oggetto, creato da un `ConcreteFactory` e 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 intercambiabili facilmente
  - Non è facile supportare nuove classi `Product` poiché bisogna aggiungere un metodo su *`AbstractFactory`* e su ogni `ConcreteFactory`

Abstract Factory

```
interface Icon { // AbstractProductA
  void draw();
  void fill();
}
interface Text { // AbstractProductB
  public void tell();
  public void shout();
}
interface Creator { // AbstractFactory
  public Icon getIcon(); // create method
  public Text getText();
}
// ConcreteFactory
class Creator1 implements Creator {
  public Icon getIcon() {
    return new Circle();
  }
  public Text getText() {
    return new Japanese();
  }
}
```

```
class Creator2 implements Creator {//ConcretFact
  public Icon getIcon() {
    return new Box();
  }
  public Text getText() {
    return new English();
  }
}
class Circle implements Icon { // ProductA1
  public void draw() {
    System.out.print("( ) ");
  }
  public void fill() {
    System.out.print("(o) ");
  }
}
class Box implements Icon { // ProductA2
  public void draw() {
    System.out.print("[ ] ");
  }
  public void fill() {
    System.out.print("[X] ");
  }
}
```


Abstract Factory

```
class Japanese implements Text { // ProductB1
  public void tell() {
    System.out.println("( Youkoso. Konnichiwa! Hajimemashite )");
  }
  public void shout() {
    System.out.println("( Shizukanishite kudasai )");
  }
}
class English implements Text { // ProductB2
  public void tell() {
    System.out.println("::::: Welcome. Nice to meet you :::::");
  }
  public void shout() {
    System.out.println("::::: Be quiet please! :::::");
  }
}
public class AbsFactorTest {
  public static void main(String args[]) {
    Creator c = new Creator1(); // istanzio un Creator
    Icon ic = c.getIcon();
    Text t = c.getText();
    ic.draw();
    t.tell();
  }
}
```