Factory Method

Rif.: Gamma, Helm, Johnson, Vlissides, Design Patterns, Addison Wesley
Corso di Ingegneria del Software, CdL triennale in Informatica
Università degli studi di Catania
Proff. Emiliano Tramontana, Andrea Calvagna
Aprile 2022

1

Design pattern creazionali

- Singleton, **Factory Method**, Abstract Factory, Builder, Prototype
- Permettono di astrarre il processo di creazione oggetti: rendono un sistema indipendente da come i suoi oggetti sono creati, composti, e rappresentati
- Il pattern Factory Method consente di non specificare subito la classe di un oggetto che so di dover usare

2

Factory Method

- **Intento**:
  - In un algoritmo, so **quando** deve essere creato un oggetto di servizio, ma non so o non voglio stabilire (e/o voglio poter cambiare) di **quale** classe istanziarlo.
    - Legato a doppio filo al concetto di *interfaccia**: so cosa devo fare e che devo farlo adesso ma non so ancora cosa usare per farlo

3

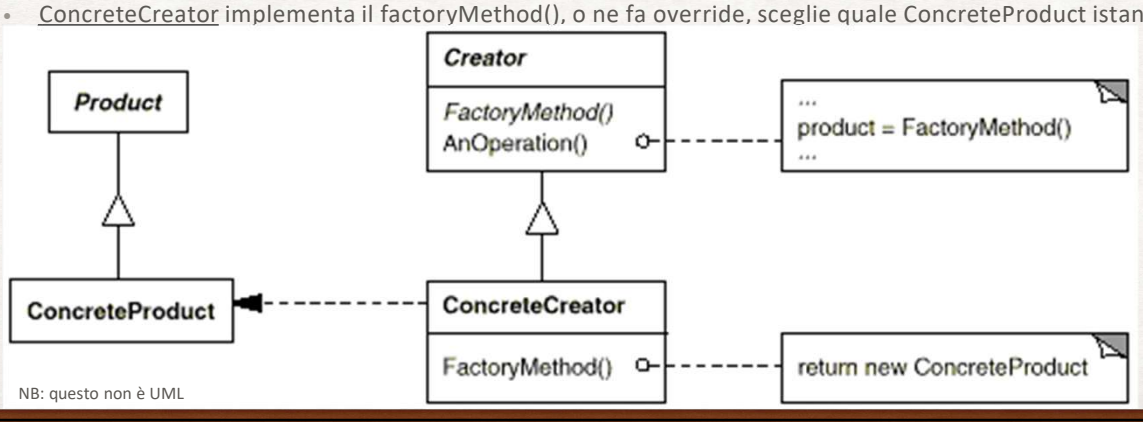
Interfaccia vs Classe:

- Una interfaccia definisce una **qualità parziale**, un possible modo di interagire con un oggetto, non il suo profilo completo (la classe)
- Per definizione **condivisa** tra più classi scorrelate tra loro.
- Una classe puo' avere molte interface
- Una classe (in Java) definisce in modo completo un **tipo (ADT)**: identifica una categoria unica di oggetti (e la sua implementazione)
- Una classe non condivide mai implem. con altre classi, a meno che:
  - Sono nella stessa gerarchia
  - Le hai progettate male: codice duplicato
  - Stai programmando in C++: ereditarietà multipla

4

Factory Method

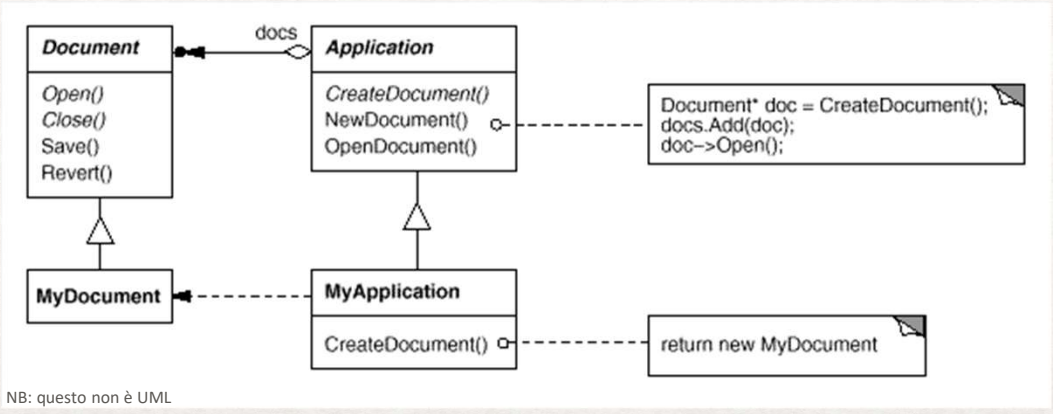
- **Soluzione:**
  - Product è l'interfaccia comune degli oggetti creati da factoryMethod()
  - ConcreteProduct è un'implementazione di Product
  - Creator dichiara il factoryMethod(), quest'ultimo ritorna un oggetto di tipo Product. Usa l'oggetto Product nel suo codice
  - ConcreteCreator implementa il factoryMethod(), o ne fa override, sceglie quale ConcreteProduct istanziare



5

Factory Method

- **Intento**:
  - Incapsulo la creazione dentro un **metodo** (dà l'*oggetto come valore di ritorno*): saranno le sottoclassi a decidere la classe da istanziare
  - Il mio algoritmo (astratto) può essere scritto conoscendo (e/o usando) solo l'**interfaccia** dell'oggetto di servizio: *primo principio*



6

```java
public interface Product {
    void request();
}

class CProduct implements Product {
    public void request() {
        System.out.println("CProduct");
    }
}

public class Client { //client application
    public static void main(String[] args) {
        Creator myApp = new CCreator(); // incapsulation!
        myApp.AnOperation(); //use the product service indirectly
         ...

        Product myProduct = myapp.getProduct(); // incapsulation!
        myProduct.request();      // use the product service directly
    }
}
```

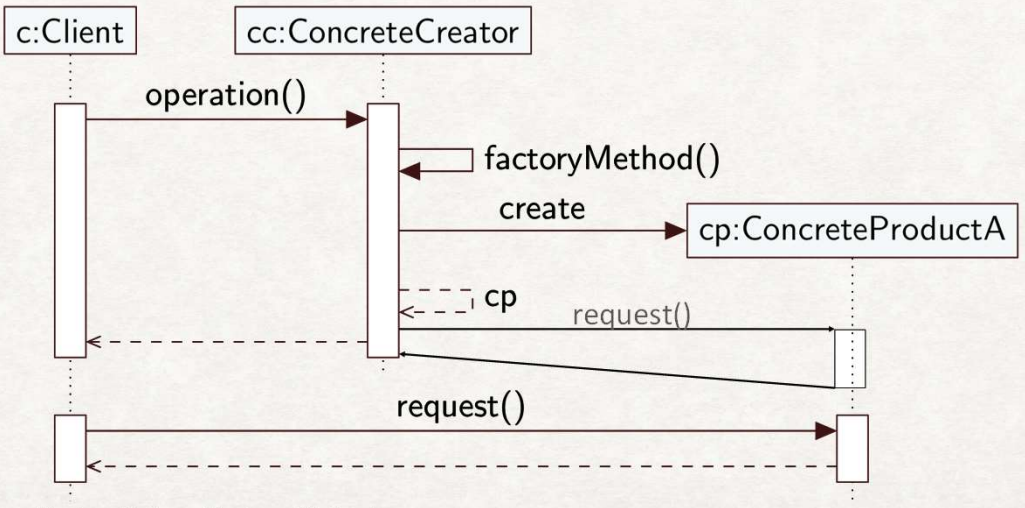
```java
abstract class  Creator {
    abstract public Product getProduct();

    public void AnOperation(){
          Product p = this.getProduct();
          p.request();
    }

}

public class CCreator extends Creator {
    @Override
    public Product getProduct() {
      return new CProduct();
    }
}
```

Sottoclassando Creator fornisco versioni alternative di AnOperation/request

7

Factory Method

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

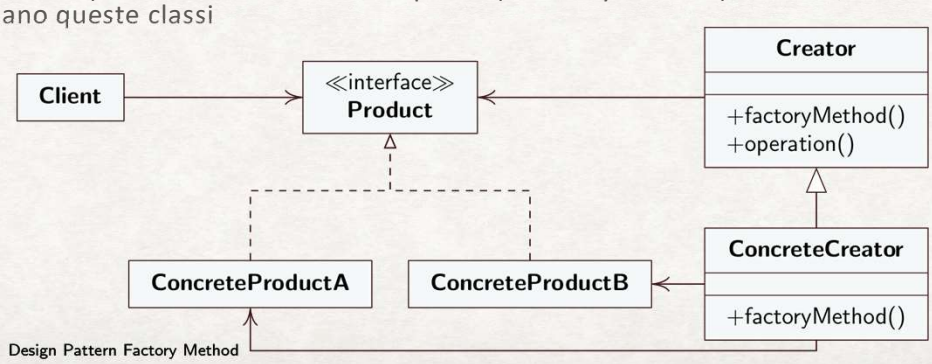


Design Pattern Factory Method

8

Factory Method

- **Applicabilità**:
  - Una classe non è in grado di sapere in anticipo la classe dell'oggetto che deve creare
  - Una classe vuole delegare alle sottoclassi future **la scelta di quale** oggetto creare e utilizzare
  - Una classe (creator/client) delega responsabilità (metodi) a una (o più) classe di supporto (product) e vuole localizzare in un punto (il factory method) la scelta/conoscenza di quali siano queste classi



9

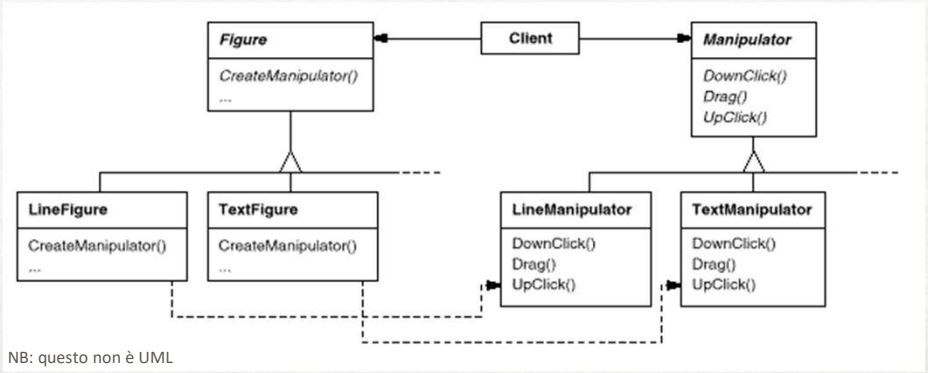
Conseguenze

- Il codice delle classi del**l'applicazione conosce solo l'interfaccia** Product e può lavorare con qualsiasi ConcreteProduct. I ConcreteProduct sono facilmente intercambiabili
- Fornisce hooks per le classi derivate. Creare oggetti dentro una classe con un factory method è molto **più flessibile che creare un oggetto esplicitamente**. Un Factory Method dà alle classi derivate un hook per fornire una versione estesa di un dato oggetto.
- Se si implementa una sottoclasse di Creator per ciascun ConcreteProduct da istanziare si ha una **proliferazione di classi** (soprattutto se servono solo per implementare il fm)

10

Gerarchie di classi parallele

- Il factory method non è chiamato per forza solamente dai Creators.
- Altri clients possono trovare i factory methods utili, specialmente nei casi di gerarchie di classi parallele.
- Le Gerarchie di classi parallele si hanno quando una classe delega alcune delle sue responsabilità a una classe separata.



NB: questo non è UML

11

FM nella Java Library

- Collection: aggregato astratto (interfaccia) di oggetti *iterabili (enumerabili)*

```
public interface Collection<E> extends Iterable<E> {
        Iterator<E>  iterator()   //Returns an iterator over the elements in this collection.
        void         forEach(..) // invoca al suo interno il FM per ottenere l'iteratore
        …
}
```

- *Come iterare* dipende dagli oggetti : (get)**Iterator()**
  - *è un factory method che restituisce l'implementazione giusta per ogni tipo di aggregato da itearare*
  - *Collego il tipo di elemento al tipo di iteratore corrispondente*
  - *Nb anche l'iteratore è a sua volta un pattern (lo vedremo in seguito)*

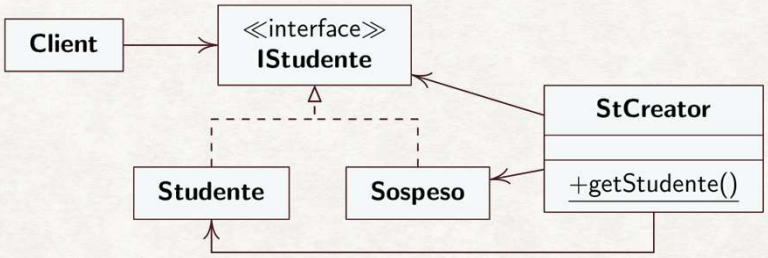
12

Pausa 10 min.

13

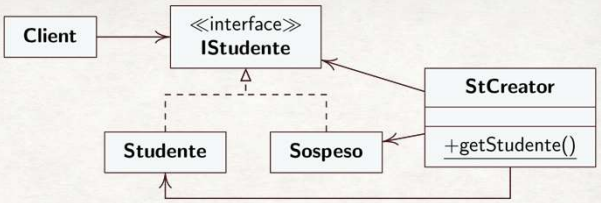
VARIANTI

- Il factoryMethod() come metodo static
  - Se devo solo di incapsulare una scelta di classe
  - Ho un solo il concreteCreator
  - Non devo istanziare il creator
  - Non estensibile



14

```
public interface IStudente {
  public void nuovoEsame(String m, int v);
  public float getMedia();
}

public class Studente implements IStudente {
  private List<Esame> esami = new ArrayList<>();
  public void nuovoEsame(String m, int v) {
    Esame e = new Esame(m, v);
    esami.add(e);
  }
  public float getMedia() {
    if (esami.isEmpty()) return 0;
    float sum = 0;
    for (Esame e : esami) sum += e.getVoto();
    return sum / esami.size();
  }
}

public class Sospeso implements IStudente {
  private float media;
  public Sospeso(float m) {
    media = m;
  }
  public void nuovoEsame(String m, int v) {
    System.out.println("Non e' possibile sostenere esami");
  }
  public float getMedia() {
    return media;
  }
```



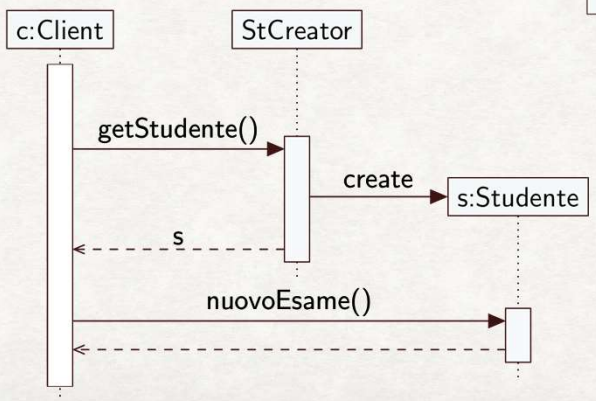

```
public class StCreator {
  private static boolean a = true;

  public static IStudente getStudente() {
    if (a) return new Studente();
    return new Sospeso(0);
  }
}

public class Client {

public void registra() {
   IStudente s = StCreator.getStudente();
   s.nuovoEsame("Maths", 8);
  }
}
```

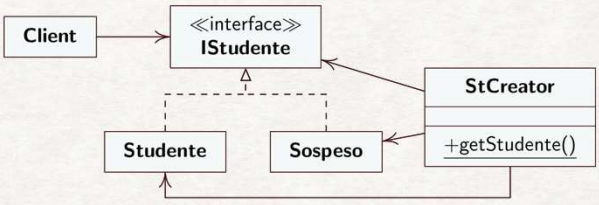
Implementando IStudente ottengo versioni alternative di registra()

15

Interazione

- Nel precedente esempio di codice, l'interfaccia IStudente svolge il ruolo Product, le classi Studente e Sospeso svolgono il ruolo ConcreteProduct, e la classe StCreator svolge il ruolo ConcreteCreator





16

FM vs Polimorfismo

- cambio sottoclasse per cambiare implem. Ma faccio anche molto di più
- Il pattern FM fa override, compone/delega e incapsula una creazione
- Delega: Il metodo factory collega il client/creator con classi, anche ignote, di altre gerarchie, cui delega responsabilità.

```
public class Client {
    BasicStud s = new Studente();
    public void registra(){
        s.nuovoEsame("Maths", 8);
    }
}
```

```
public class Sospeso extends BasicStud  {
        public void nuovoEsame(String m, int v) {
                System.out.println("Non possibile");
        }
}
```

```
public class Studente extends BasicStud{
 private List<Esame> esami = new ArrayList<>();
 public void nuovoEsame(String m, int v) {
  Esame e = new Esame(m, v);
  esami.add(e);
 }
```

17

VARIANTI

- Creator non astratto
  - per avere un factoryMethod di default
  - Creator e ConcreteCreator sono la stessa classe (estendibile)

```
public class Creator {
    public IStudente getStudente() {
        return new Studente();
    }
}

public class SoCreator extends Creator{
    public IStudente getStudente() {
        return new Sospeso();
    }
}
```

```
public class Client {
 public void registra( Creator sc) {
  if (sc==null) sc = new Creator();
  . . .
  IStudente s = sc.getStudente();
  s.nuovoEsame("Maths", 8);
 }
}
```

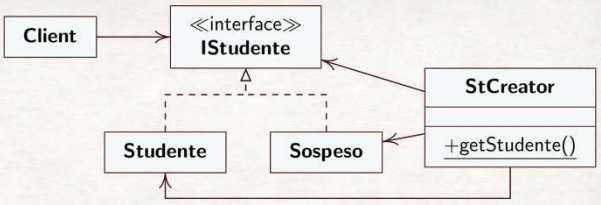
18

VARIANTI

- **factoryMethod() PARAMETRICO**
  - Istanzia una tra piu' classi possibili
  - Scelta dal client, direttamente o indirettamente
  - Varie versioni
    - **Creator non astratto, il client seleziona il prodotto**
    - Creator astratto, ConcreteCreator seleziona il prodotto
      - il client seleziona il ConcreteCreator da istanziare (già visto: caso base)

19

```
public interface IStudente {
  public void nuovoEsame(String m, int v);
  public float getMedia();
}

public class Studente implements IStudente {
  private List<Esame> esami = new ArrayList<>();
  public void nuovoEsame(String m, int v) {
    Esame e = new Esame(m, v);
    esami.add(e);
  }
  public float getMedia() {
    if (esami.isEmpty()) return 0;
    float sum = 0;
    for (Esame e : esami) sum += e.getVoto();
    return sum / esami.size();
  }
}

public class Sospeso implements IStudente {
  private float media;
  public Sospeso(float m) {
    media = m;
  }
  public void nuovoEsame(String m, int v) {
    System.out.println("Non e' possibile sostenere esami");
  }
  public float getMedia() {
    return media;
  }
```




```
public class StCreator {

  public static IStudente getStudente(boolean a) {
    if (a) return new Studente();
    return new Sospeso(0);
  }
}

public class Client {

public void registra() {
   IStudente s = StCreator.getStudente(true);
   s.nuovoEsame("Maths", 8);
  }
}
```

20

VARIANTI

- Creator **generico**,
  - Derivo classi ConcreteCreator collegate al prodotto giusto
  - il client seleziona indirettamente il prodotto giusto da usare
  - Non c'è dipendenza diretta del client col prodotto

21

```
public interface Product {
    void request();
}

class CProduct implements Product {
    public void request() {
        System.out.println("CProduct");
    }
}

abstract class Creator <P extends Product>{
    abstract public Product getProduct();

    public void doSomething(){
          P p = this.getProduct();
          p.request();
    }
}
```

```
public class Client { //client application
  public static void main(String[] args){
     Creator myApp = new ConcreteCreator();

     myApp.doSomething();
     Product p = myapp.getProduct()
     p.request();
  }
}
```

Possibile pure Creator concreto generico e con la riflessione computazionale creo il prodotto indicato
Creator myApp = new Creator<CProduct>();

Vedi esempio nel seguito

```
public class ConcreteCreator extends Creator<CProduct> {
    @Override
    public CProduct getProduct() { return new CProduct();}
}
```

22

VARIANTI

- Creator con riflessione
  - Il Client specifica come parametro il prodotto
  - Non serve derivare sottoclassi di Creator
  - Il client dipende <per nome> dai concrete product

23

```
import java.lang.reflect.InvocationTargetException;
class  Creator {
 public Product getProduct(Class<?> cp){
   try {
     return (Product) cp.getDeclaredConstructor().newInstance();
   } catch (InstantiationException |
        OMISSIS … SecurityException e) {
       e.printStackTrace();
      return null;
   }
 }

  public void AnOperation(String cn)
             throws ClassNotFoundException{
      Class<?> c = Class.forName(cn);
      Product p = this.getProduct(c);
      p.request();
  }
  public void AnOperation(Class<?> cp){
     Product p = this.getProduct(cp);
     p.request();
  }
}
```

```
public class Client {
    public static void main(String[] args) {
        Product p;
        Creator myApp = new Creator();

        myApp.AnOperation(CProduct.class);

        try {
            myApp.AnOperation("CProduct");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        p = myApp.getProduct(CProduct.class);
        p.request();
    }
}
```

NB: Possibile anche parametrizzare direttamente il costruttore
new Creator(CProduct.class);
new Creator("Cproduct");
new Creator<CProduct>();

Passare nel costruttore istanze già create è una INIEZIONE di DIPENDENZA: new Creator(new Cproduct());
(ne parliamo tra poco)

24

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 fanno richieste, come visto prima per il Factory Method
  - I client dovranno dire 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 -> uso un Singleton

25

25

Esempio di Object Pool

```
import java.util.LinkedList;
// CreatorPool è un ConcreteCreator e implementa un Object Pool
public class CreatorPool extends ShapeCreator {
  private LinkedList<Shape> pool = new LinkedList<Shape>();
  // getShape() è un metodo factory che ritorna un oggetto prelevato dal pool
  public Shape getShape() {
        Shape s;
        if (pool.size() > 0) s = pool.remove();
        else s = new Circle();
        return s;
  }
  // releaseShape() inserisce un oggetto nel pool
  public void releaseShape(Shape s) {
        pool.add(s);
  }
}
```

26

26

Dependency Injection

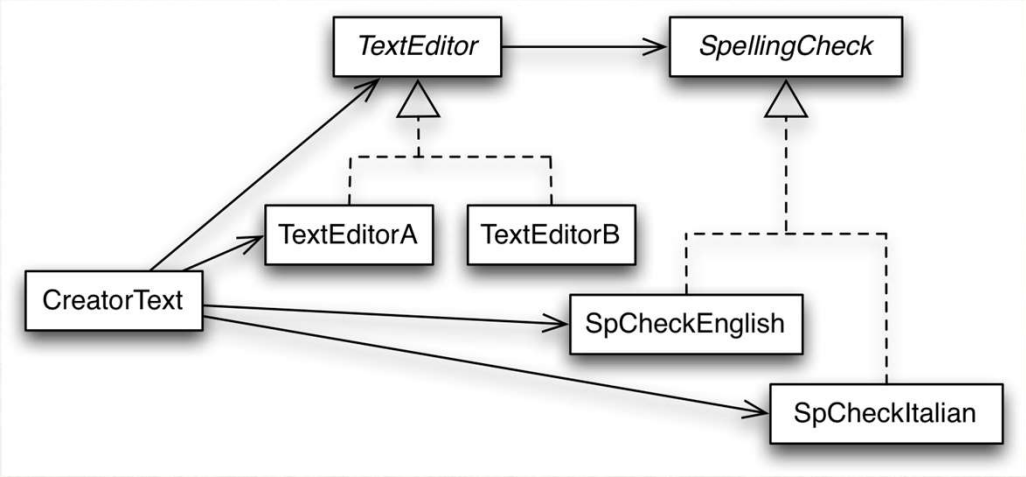
- Il design pattern Factory Method può essere usato per inserire le dipendenze necessarie ad altri oggetti (istanze di ConcreteProduct)
- Dependency injection
  - Una classe C usa un servizio S (ovvero C dipende da S)
  - Esistono tante implementazioni di S (ovvero S1, S2), la classe C non deve dipendere dalle implementazioni S1, S2
  - Al momento di creare l'istanza di C, indico all'istanza di C con quale implementazione di S deve operare
- Esempio di dependency
  - Una classe TextEditor usa un servizio SpellingCheck
  - Ci sono tante classi che implementano il servizio SpellingCheck, in base alla lingua usata: SpCheckEnglish, SpCheckItalian, etc.
  - TextEditor deve poter essere collegato ad una delle classi che implementano SpellingCheck

27

27

Diagramma UML

- Esempio di Dependency Injection



28

28

Esempio Dep. Injection

```
public class TextEditorA implements TextEditor { //TextEditorA è un ConcreteProduct
  private SpellingCheck speller;
  public TextEditorA(SpellingCheck sp) {  // inserisco la dipendenza fornendo al
        speller = sp;                     // costruttore l'istanza del servizio sp
  }
  public void put(String s) {
        if (speller.check(s)) ...
        else ...
  }
}
```

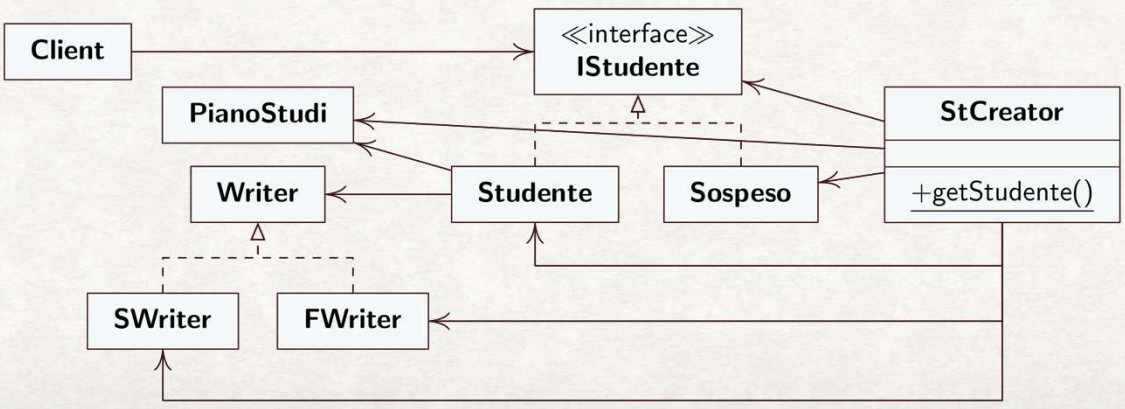
```
public class CreatorText {  // CreatorText è un
  ConcreteCreator
  public static TextEditor getEnglishEditor() {
        return new TextEditorA(new SpCheckEnglish());
  }
  public static TextEditor getItalianEditor() {
        return new TextEditorB(new SpCheckItalian());
  }
}
```

29

29

Altro Esempio

- Si abbiano Writer e PianoStudi che sono dipendenze per Studente
- Studente riceve nel suo costruttore le istanze di Writer e PianoStudi
- Studente conosce solo il tipo Writer non i suoi sottotipi



30

Principi di programmazione

- Fondamentali:
  - Programmare riferendosi ad una interfaccia, non ad una implementazione
  - Preferisci la composizione di oggetti rispetto all'ereditarietà di classe (delega!)
- Progetta per il cambiamento/riuso:
  - Mantieni alta la coesione e basso l'accoppiamento
  - Usa i tipi parametrici (generics)
  - usa i design patterns per aumentare la flessibilità del tuo codice

31