UNIVERSITÀ PARTHENOPE
DiST
Dipartimento di Scienze e Tecnologie

Programmazione 3
e
Laboratorio di Programmazione 3

Creational Patterns

Angelo Ciaramella

Creational Patterns

- Creational Patterns
  - I pattern creazionali nascondono i costruttori delle classi e mettono dei metodi al loro posto creando un'interfaccia
    - In questo modo si possono utilizzare oggetti senza sapere come sono implementati

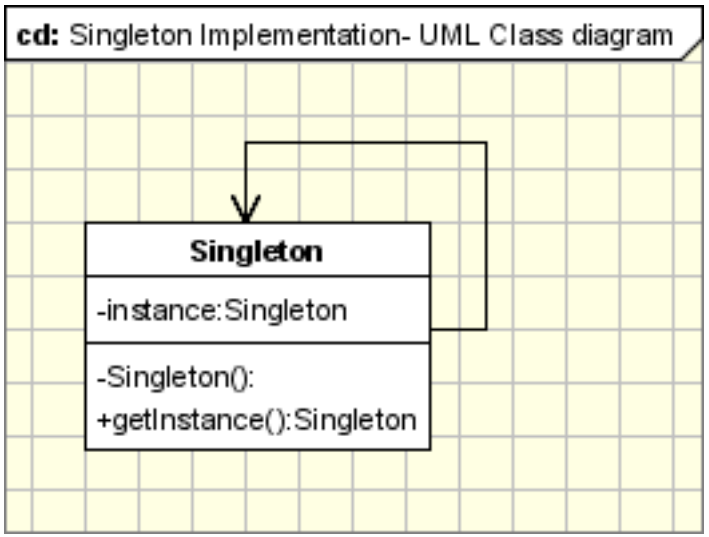
- Design Pattern
  - Singleton ("singoletto")
  - Factory Method ("metodo fabbrica")
  - Abstract factory ("fabbrica astratta")
  - Factory Pattern
  - Builder ("costruttore")
  - Prototype ("prototipo")

Singleton

- Scopo
  - *Assicurare che una classe abbia una sola istanza e fornire un punto globale di accesso ad essa*
- Motivazione
  - Per alcune classi è importante avere una sola istanza
    - e.g., un singolo spooler per diverse stampanti
  - La classe assicura che non possono essere create altre istanze e prevede un modo per accedere all'istanza
- Applicabilità
  - Il pattern Singleton è usato quando
    - deve esistere solo un'istanza della classe e deve essere accessibile da un punto noto
    - l'unica istanza deve essere estesa e i client devono essere capaci di usare un'istanza estesa senza modificare il loro codice

Singleton

- Struttura



Struttura del pattern `Singleton`

Lazy initialization

```
public class SingletonExample {
private static SingletonExample instance;
private SingletonExample () {
}
public static SingletonExample getInstance() {
if (instance == null) {
instance = new SingletonExample();
}
return instance;
}

...
public void doSomething()
{
                ...
}
}
```

Esempio di implementazione del pattern `Singleton`

Implementazione

- `getInstance()`
  - essendo interno alla classe può utilizzare il metodo costruttore (anche se privato) per istanziare la classe stessa
  - verrà creato *un oggetto solo la prima volta* che verrà chiamato questo metodo ed assegnato all'attributo statico `instance`
  - dalla seconda chiamata in poi questo *metodo restituirà sempre la stessa istanza*

- Per ottenere l'unica istanza della classe `SingletonExample` le altre classi dovranno usare la sintassi

```
SingletonExample unicaIstanza = SingletonExample.getInstance();
```

- Punto globale di accesso

```
SingletonExample.getInstance().doSomething();
```

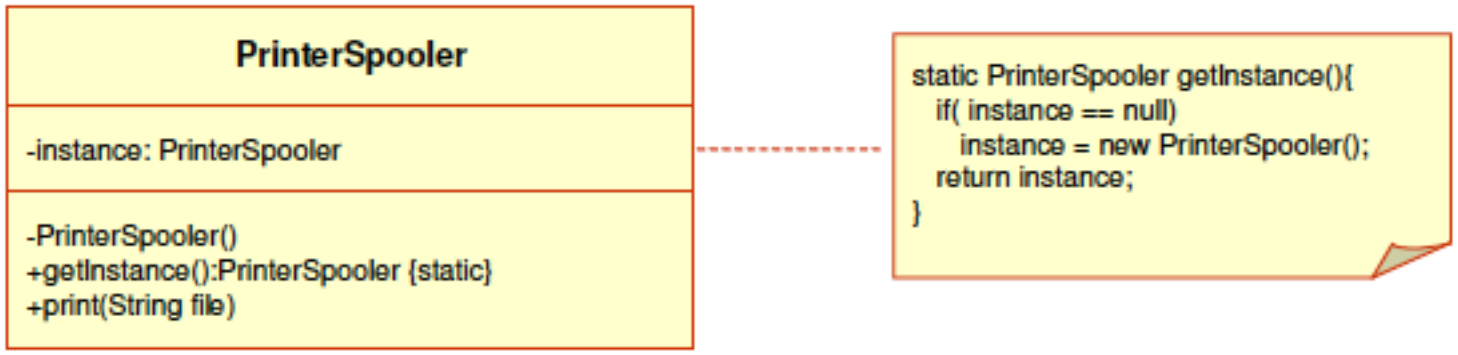

Eager initialization

```
public class EagerInitializedSingleton {
private static final EagerInitializedSingleton instance =
new EagerInitializedSingleton();
//private constructor to avoid client applications to use
constructor

private EagerInitializedSingleton(){}

public static EagerInitializedSingleton getInstance(){
return instance;
}
}
```

Esempio di implementazione del pattern `Singleton`

Esempio

- Un applicativo deve istanziare un oggetto che gestisce una stampante
  - oggetto unico (una sola istanza di esso) altrimenti potrebbero risultare dei problemi nella gestione della risorsa



Schema del modello `Singleton`

Implementazione

```
public class PrinterSpooler {
private static PrinterSpooler instance;
private PrinterSpooler() {
}
public static PrinterSpooler getInstance() {
if ( instance==null) {
instance = new PrinterSpooler();
}
return instance;
}
public void print (String msg) {
System.out.println( msg );
}
}
```

Esempio di implementazione del pattern `Singleton`

Implementazione

```
public class PrinterSpooler {
private static PrinterSpooler instance;
private PrinterSpooler() {
}
public static synchronized PrinterSpooler getInstance() {
if ( instance==null) {
instance = new PrinterSpooler();
}
return instance;
}
public void print (String msg) {
System.out.println( msg );
}
}
```

Esempio di implementazione del pattern `Singleton` per una esecuzione multithread

Esempi di applicazione

- Logger classes
  - Prevedere un punto di accesso di login generale per tutte le applicazioni senza creare oggetti ad ogni login
- Configuration classes
  - Configurazione dei parametri per un'applicazione
- Accedere alle risorse condivise
  - Applicazioni che per esempio usano porti seriali. In un ambiente multithreading può essere usato per gestire le operazioni sulla porta seriale
- Factory
  - Spesso il pattern Singleton è associato con i pattern Abstract Factory e Factory Method

Esercizio

- Scrivere una classe `Singleton` che permette di visualizzare "`Hello World`".

- Implementare la classe di Test, `TestSingleton`

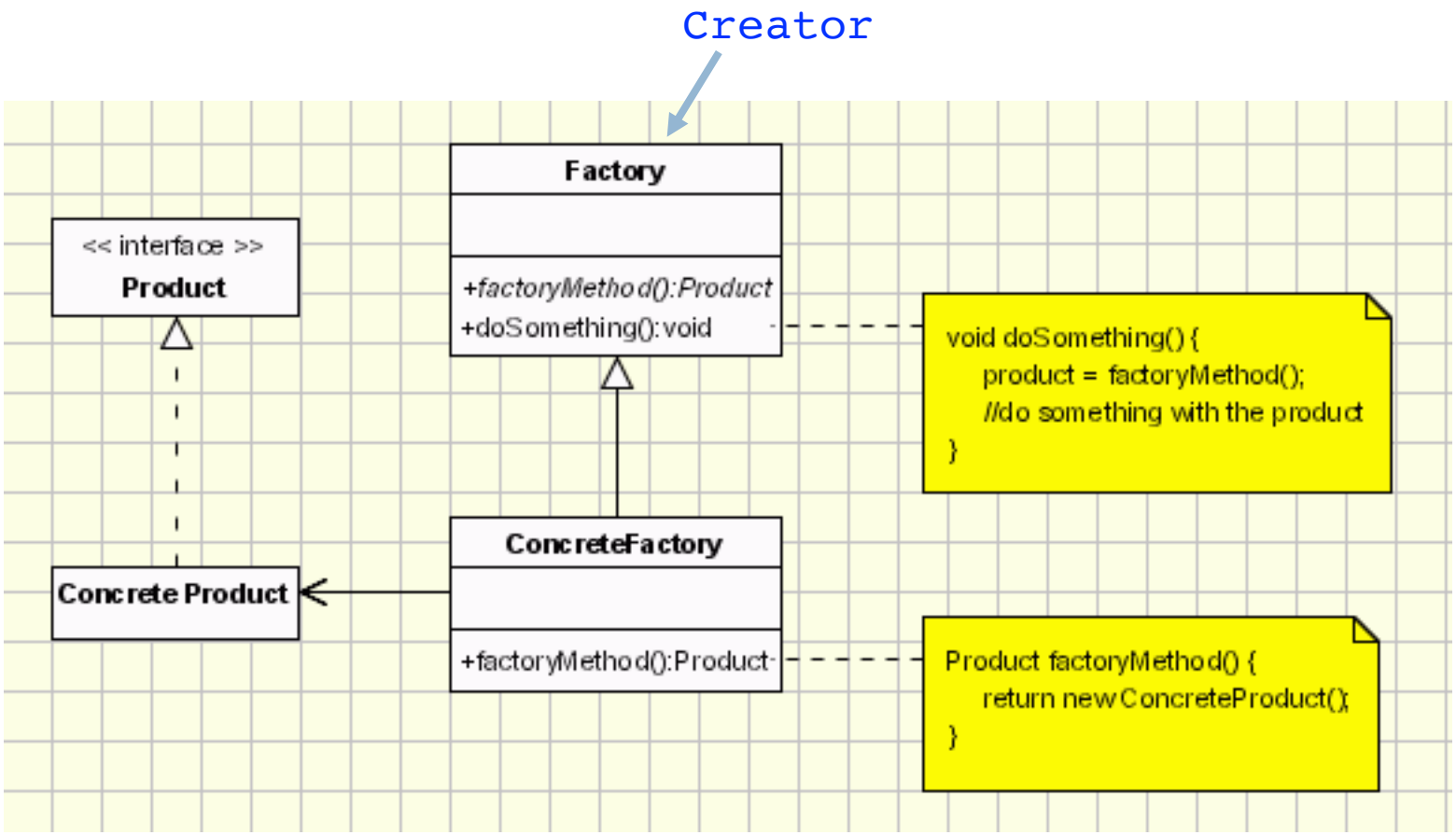
Factory Method

- Scopo
  - *Definire un'interfaccia per creare un oggetto ma lasciare la scelta del suo tipo alla sottoclasse essendo la creazione differita a run-time*
- *Anche conosciuto come*
  - Virtual Constructor
- Motivazione
  - I Framework usano classi astratte per definire e mantenere le relazioni tra oggetti
    - e.g., framework per applicazioni che presenta diversi documenti all'utente
    - e.g., in hotel
      - Stanza (factory)
      - Chiamata telefonica (factory)
      - …

Factory Method

- Applicabilità
  - una classe non può anticipare la classe di oggetti che deve creare
  - una classe vuole che le sottoclassi specificano gli oggetti da creare
  - le classi delegano le responsabilità ad una delle diverse sottoclassi "helper"

Factory Method - Struttura



Struttura del pattern `Factory Method`

Implementazione

```
public interface Product { ... }

public abstract class Creator
{
        public void anOperation()
        {
                Product product = factoryMethod();
        }

        protected abstract Product factoryMethod();
}

public class ConcreteProduct implements Product { ... }
```

Esempio di implementazione del pattern `Factory Method`

Implementazione

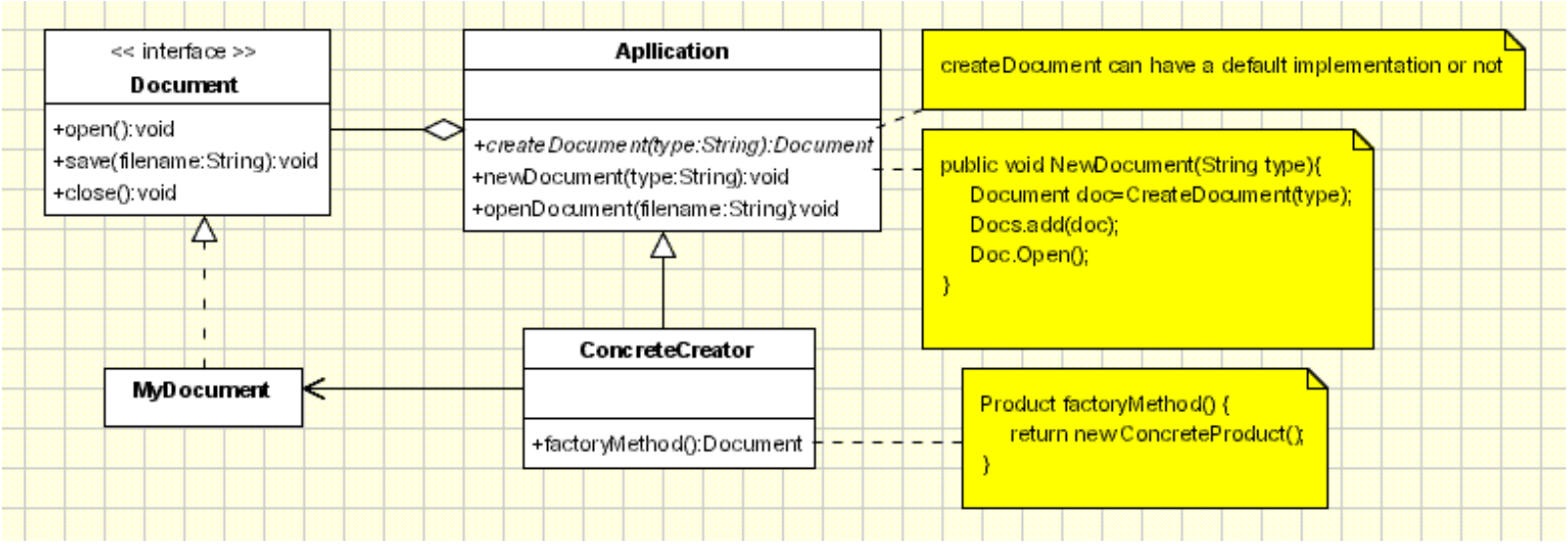
```
public class ConcreteCreator extends Creator
{
        protected Product factoryMethod()
        {
                return new ConcreteProduct();
        }
}

public class Client
{
        public static void main( String arg[] )
        {
                Creator creator = new ConcreteCreator();
                creator.anOperation();
        }
}
```

Esempio di implementazione del pattern `Factory Method`

Esempi di applicazione

- Documenti
  - Framework per applicazioni di desktop
    - apertura, creazione e salvataggio di documenti
  - Le classi principali sono `Application` e `Document`
  - La classe `Application` ha il compito *di gestire i documenti come chiesto dall'utente*

Esempi di applicazione



Gestione dei documenti tramite il pattern `Factory Method`

Esempi di applicazione

```
public Document CreateDocument(String type){
        if (type.isEqual("html"))
                return new HtmlDocument();
        if (type.isEqual("proprietary"))
                return new MyDocument();
        if (type.isEqual("pdf"))
                return new PdfDocument ();
}
```

```
public void NewDocument(String type){
        Document doc=CreateDocument(type);
        Docs.add(doc);
        Doc.open();
}
```

Esempio di implementazione del pattern `Factory Method`

Considerazioni

- E' un pattern molto usato
  - per la separazione tra applicazioni e famiglie di classi
  - cambiamento minimo nel codice dell'applicazione
  - oggetti customizzati possono facilmente rimpiazzare gli oggetti originali
  - contro – può essere usato *solo su una una famiglia di oggetti*

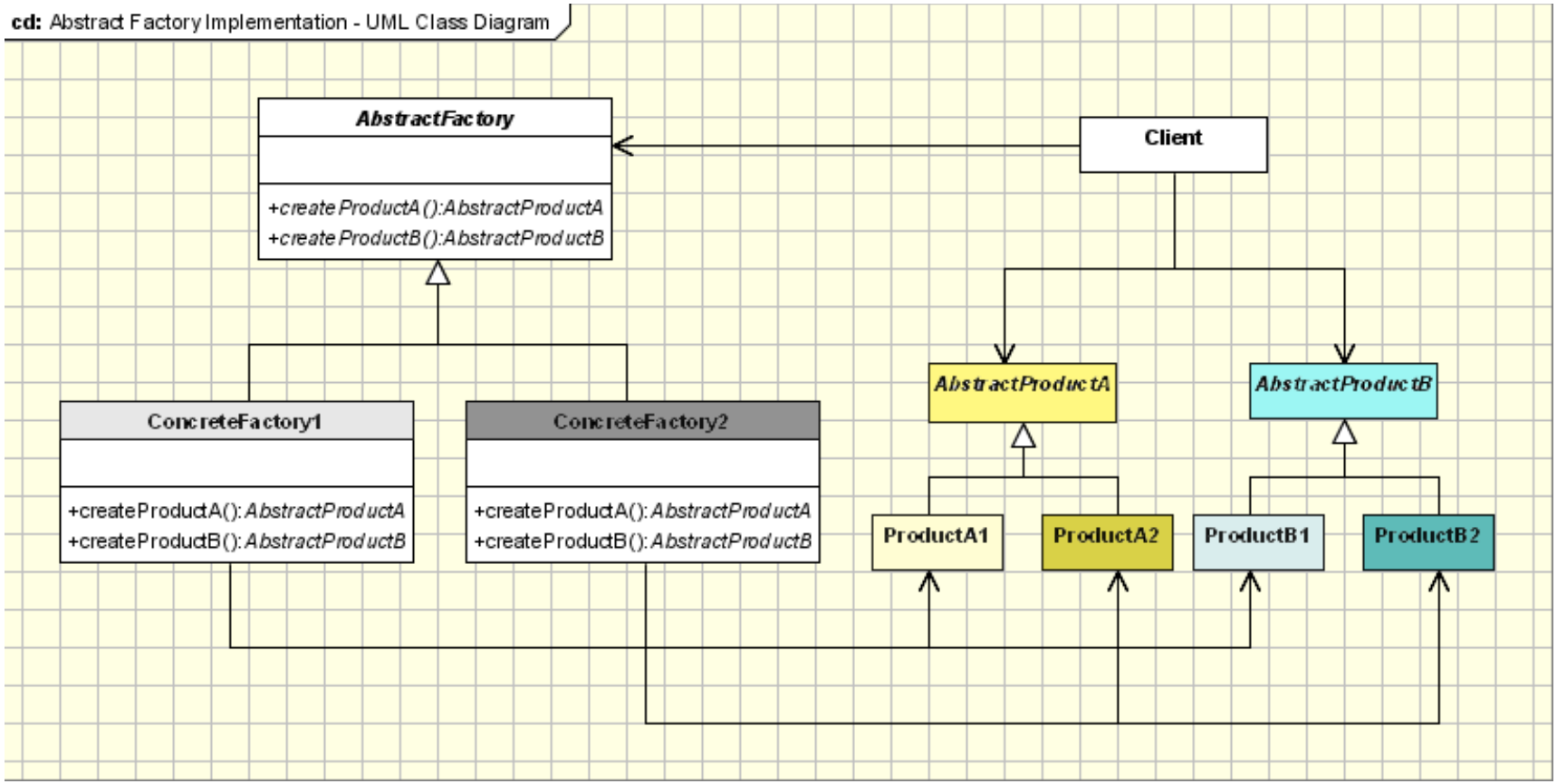
Abstract Factory

- Scopo
  - *Disporre di un'interfaccia per creare una famiglia di oggetti connessi o dipendenti senza specificare le loro classi concrete*
- *Anche conosciuto come*
  - *Kit*
- Motivazione
  - Modularizzazione
    - Aggiungere codice a classi esistenti in modo da incapsulare informazioni più generali
    - e.g., gestore telefonico, ogni numero è identificato dall'area e dal paese. Aggiungere numeri di altri paesi potrebbe essere complicato

Abstract Factory

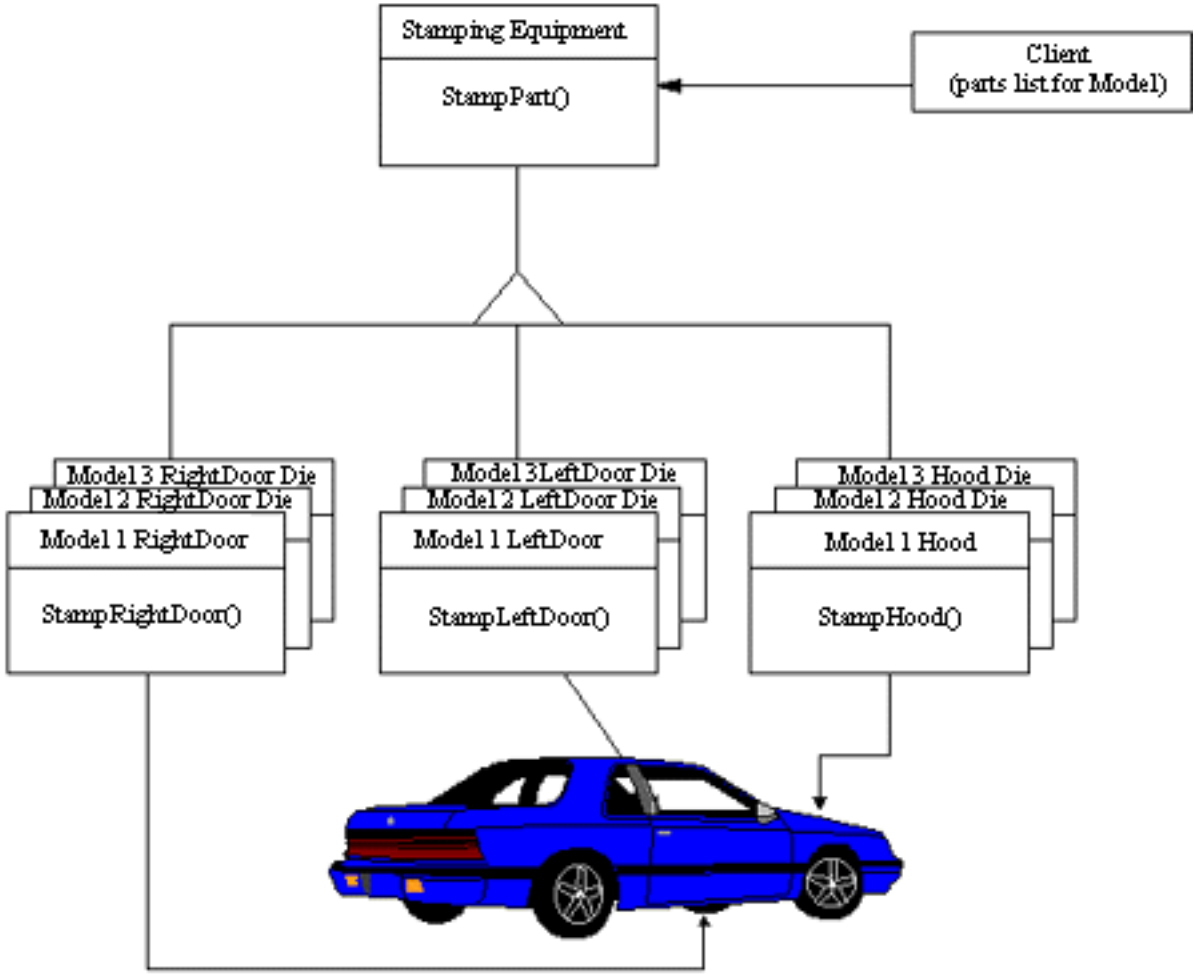
- Applicabilità
  - un sistema indipendente dalla creazione, composizione e rappresentazione dei suoi prodotti
  - un sistema configurato con molte famiglie di prodotti
  - creare una libreria di prodotti e vogliamo conoscere solo le loro interfacce e non l'implementazione

Abstract Factory - Struttura



Struttura del pattern `Abstract Factory`

Abstract Factory



Esemplificazione del pattern `Abstract Factory`

Implementazione

```
abstract class AbstractProductA{
        public abstract void operationA1();
        public abstract void operationA2();
}

class ProductA1 extends AbstractProductA{
        ProductA1(String arg){
                System.out.println("Hello " +arg);
        } // Implement the code here
        public void operationA1() { };
        public void operationA2() { };
}

class ProductA2 extends AbstractProductA{
        ProductA2(String arg){
                System.out.println("Hello " +arg);
        } // Implement the code here
        public void operationA1() { };
        public void operationA2() { };
}
```


Esempio di implementazione del pattern `Abstract Factory`

Implementazione

```
abstract class AbstractProductB{
        //public abstract void operationB1();
        //public abstract void operationB2();
}

class ProductB1 extends AbstractProductB{
        ProductB1(String arg){
                System.out.println("Hello " +arg);
        } // Implement the code here
}

class ProductB2 extends AbstractProductB{
        ProductB2(String arg){
                System.out.println("Hello " +arg);
        } // Implement the code here
}
```

Esempio di implementazione del pattern `Abstract Factory`

Implementazione

```
abstract class AbstractFactory{
        abstract AbstractProductA createProductA();
        abstract AbstractProductB createProductB();
}
class ConcreteFactory1 extends AbstractFactory{
        AbstractProductA createProductA(){
                return new ProductA1("ProductA1");
        }
        AbstractProductB createProductB(){
                return new ProductB1("ProductB1");
        }
}

class ConcreteFactory2 extends AbstractFactory{
        AbstractProductA createProductA(){
                return new ProductA2("ProductA2");
        }
        AbstractProductB createProductB(){
                return new ProductB2("ProductB2");
        }
}
```


Esempio di implementazione del pattern `Abstract Factory`

Implementazione

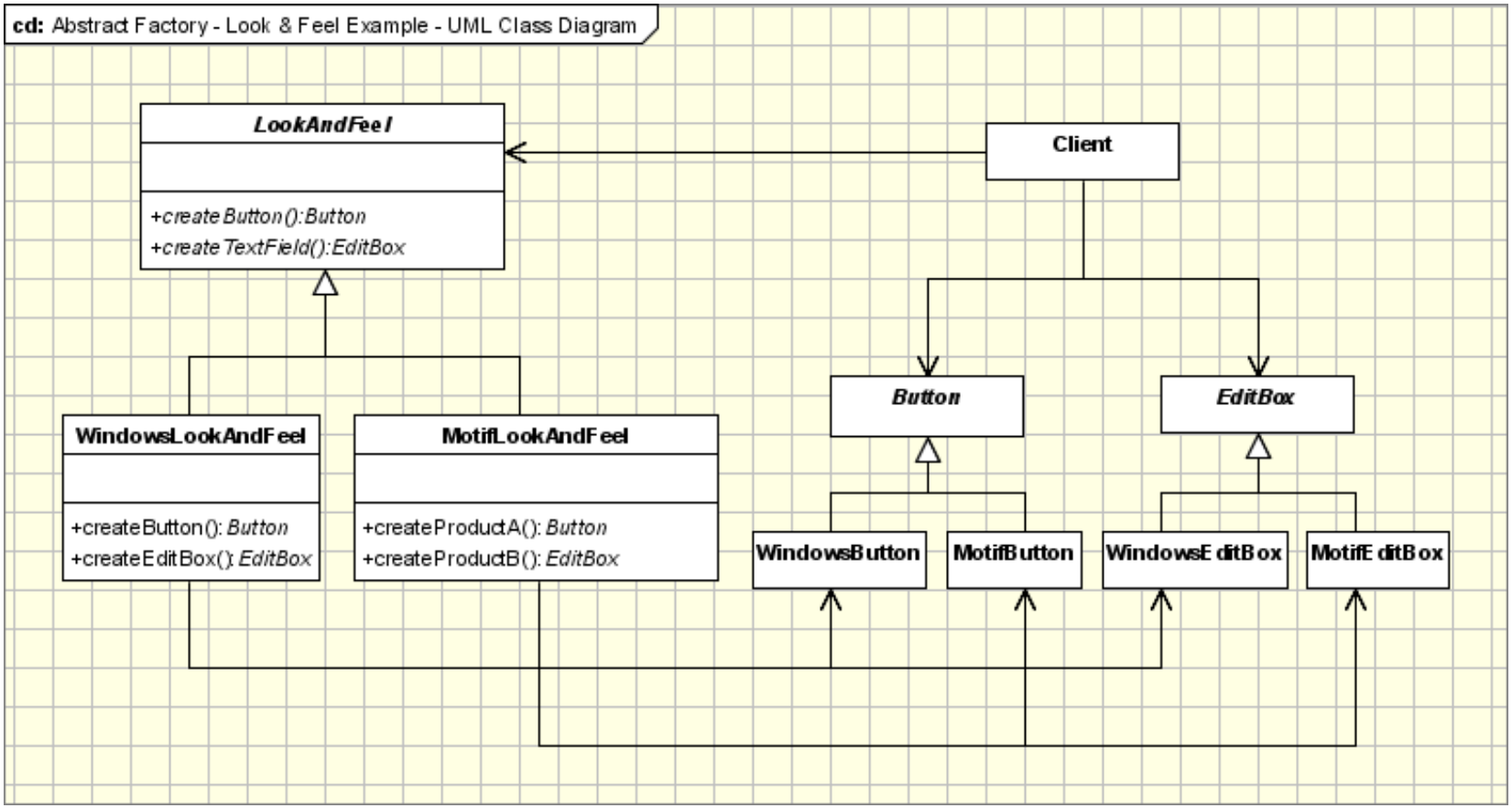
```
//Factory creator - an indirect way of instantiating the
factories
class FactoryMaker{
        private static AbstractFactory pf=null;
        static AbstractFactory getFactory(String choice){
                if(choice.equals("a")){
                        pf=new ConcreteFactory1();
                }else if(choice.equals("b")){
                                pf=new ConcreteFactory2();
                        } return pf;
        }
}
// Client
public class Client{
        public static void main(String args[]){
        AbstractFactory pf=FactoryMaker.getFactory("a");
                AbstractProductA product=pf.createProductA();
                //more function calls on product
        }
}
```

Esempio di implementazione del pattern Abstract Factory

Esempi di applicazione

- Look & Feel (interfaccia grafica)
  - Una GUI che supporta diversi aspetti grafici
    - Motif e Window
  - Ogni stile definsice alcuni controlli
    - Button e Edit Boxes

Esempi di applicazione



Gestione della GUI tramite il pattern `Abstract Factory`

Implementare la soluzione in Java

Considerazioni

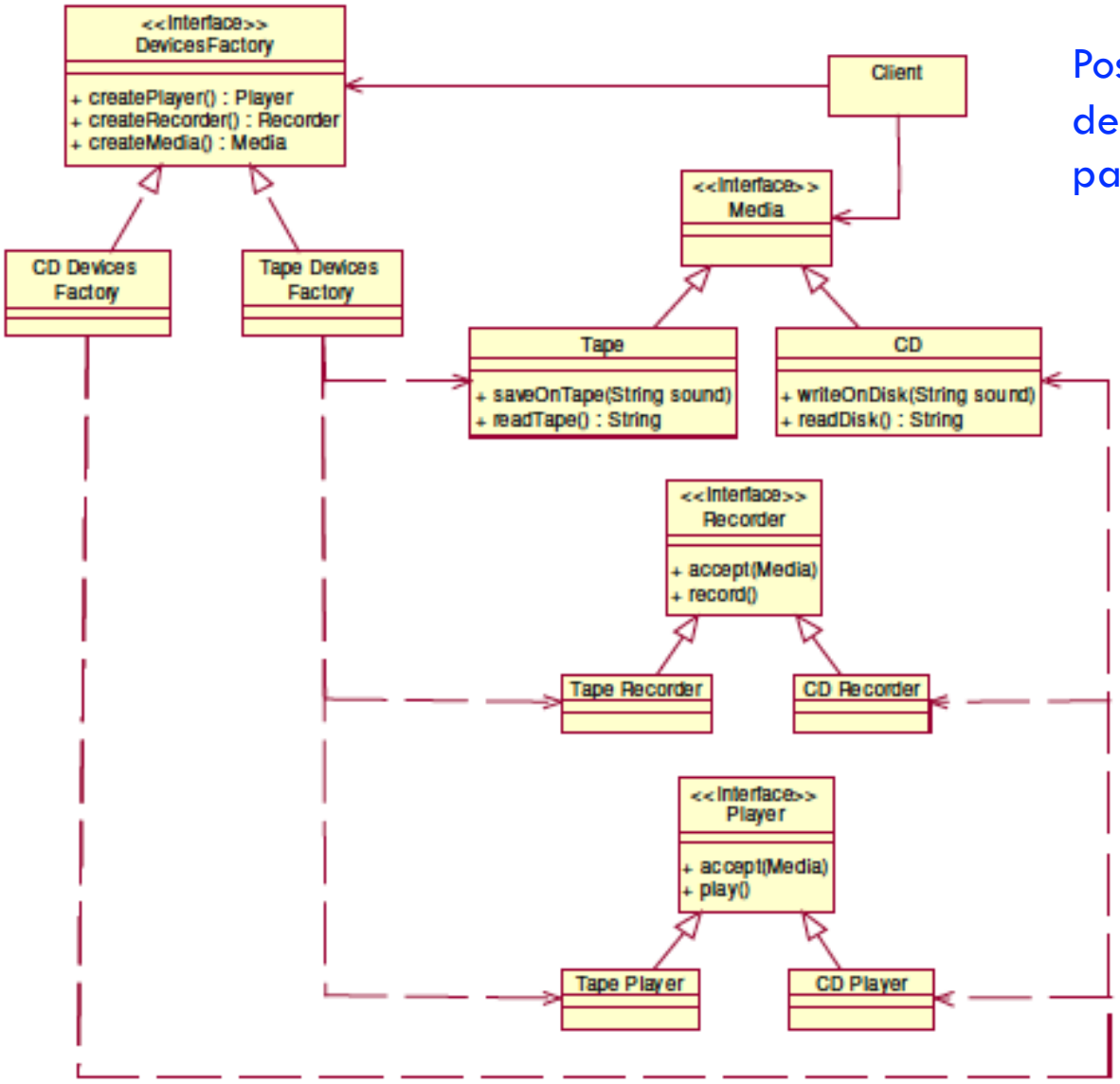
- Un'applicazione generalmente ha bisogno di una sola istanza di `ConcreteFactory`
  - Potrebbe essere utile implementarla come `Singleton`
- Per semplificare e incrementare le performance può essere usato il pattern `Prototype`
- Esempi in JDK
  - `javax.xml.parsers.DocumentBuilderFactory#newInstance()`
  - `javax.xml.transform.TransformerFactory#newInstance()`
  - `javax.xml.xpath.XPathFactory#newInstance()`

Esercizio

- Prova di sistemi Hi-Fi
  - famiglia con supporto il nastro (tape)
  - famiglia con supporto il compact disc
  - In entrambi casi un masterizzatore (recorder) e un riproduttore (player)

- I prodotti offrono agli utenti una stessa interfaccia
  - un cliente potrebbe essere in grado di eseguire lo stesso processo di prova su prodotti di entrambe famiglie di prodotti
  - e.g., registrazione e player

Esercizio



Possibile diagramma delle classi usando il pattern `Abstract Factory`

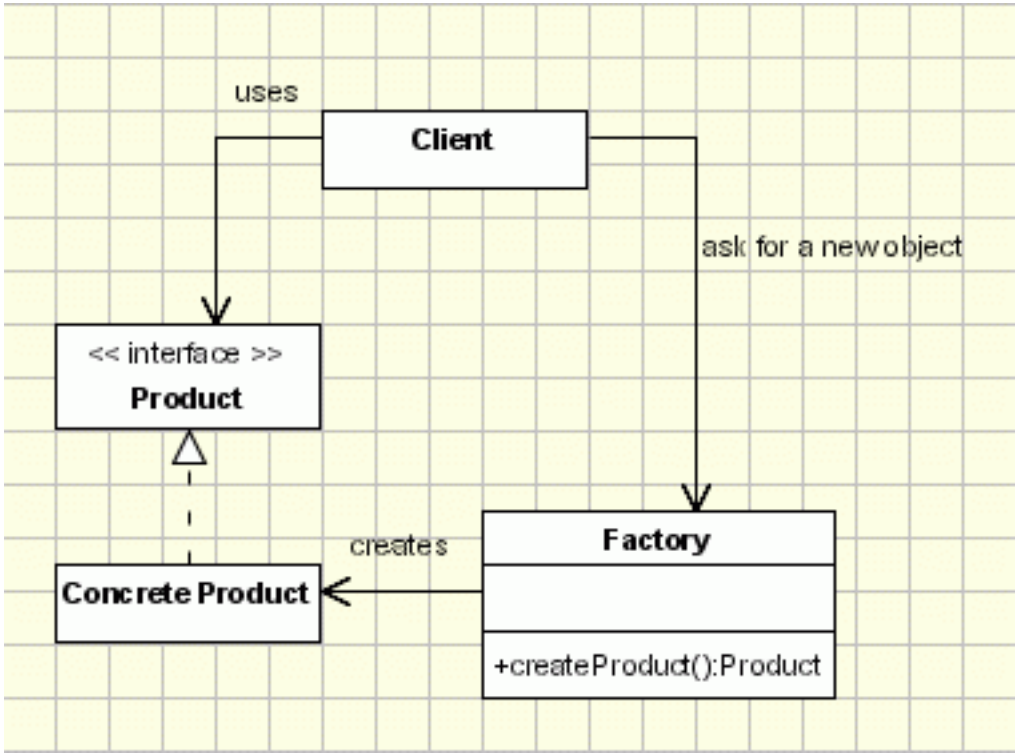
Implementare la soluzione in Java

Factory Pattern

- Scopo
  - *Creare oggetti senza esporre la logica di instanziazione al client. Creazione di oggetti attraverso un'interfaccia comune.*

- Motivazione
  - E' probabilmente il più usato pattern nei moderni linguaggi di programmazione
    - JDK, Spring, Struts lo usano
  - Ha differenti varianti e deriva dal Factory Method e Abstract Factory

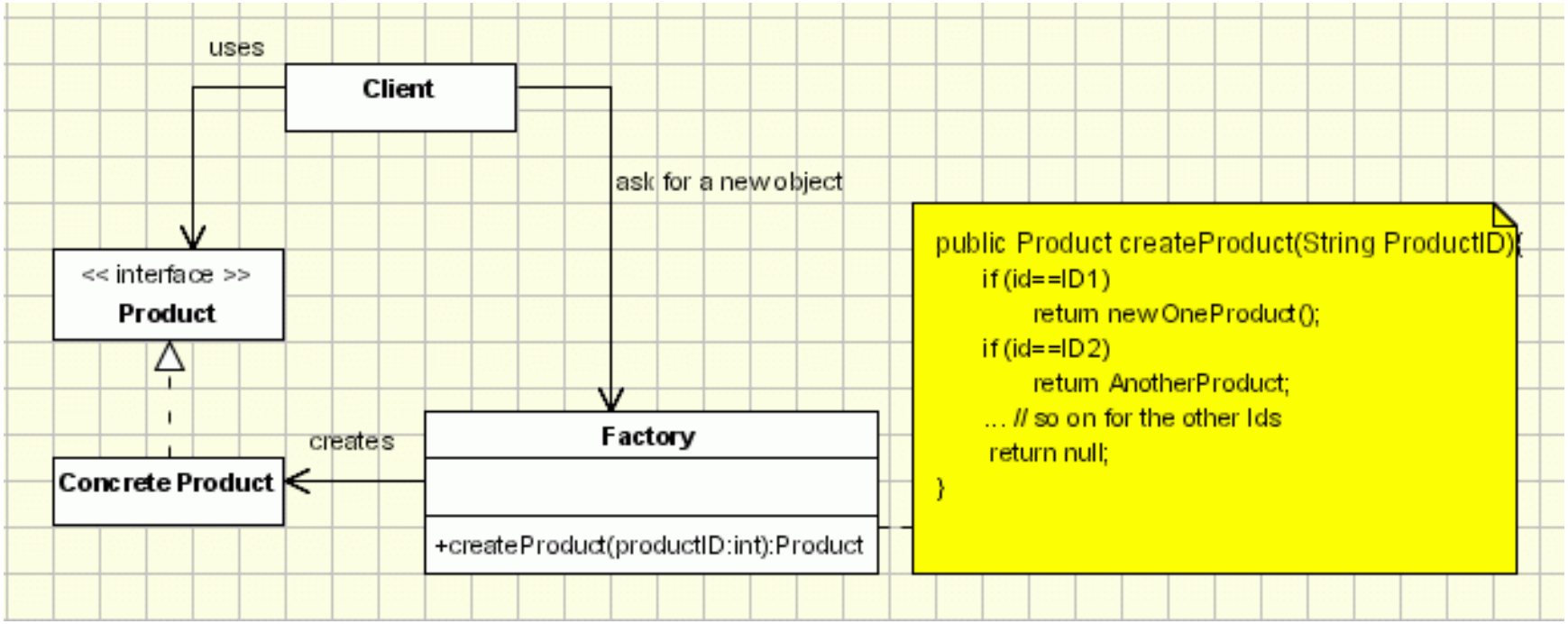
Factory Pattern - Struttura



Struttura del pattern `Factory Pattern`

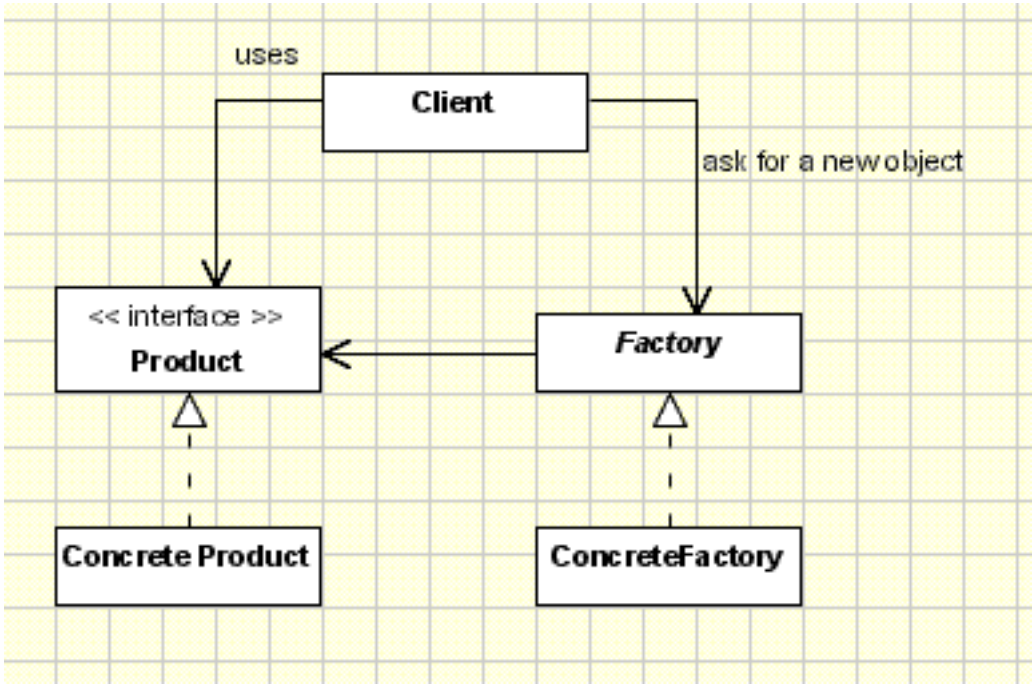
Implementare la struttura in Java

Factory Pattern – Imp. Specifiche



Procedural Solution (Parameterized Factory) – Può violare l'OCP aggiungendo un nuovo prodotto

Factory Pattern – Imp. Specifiche



Factory Pattern con astrazioni

Builder

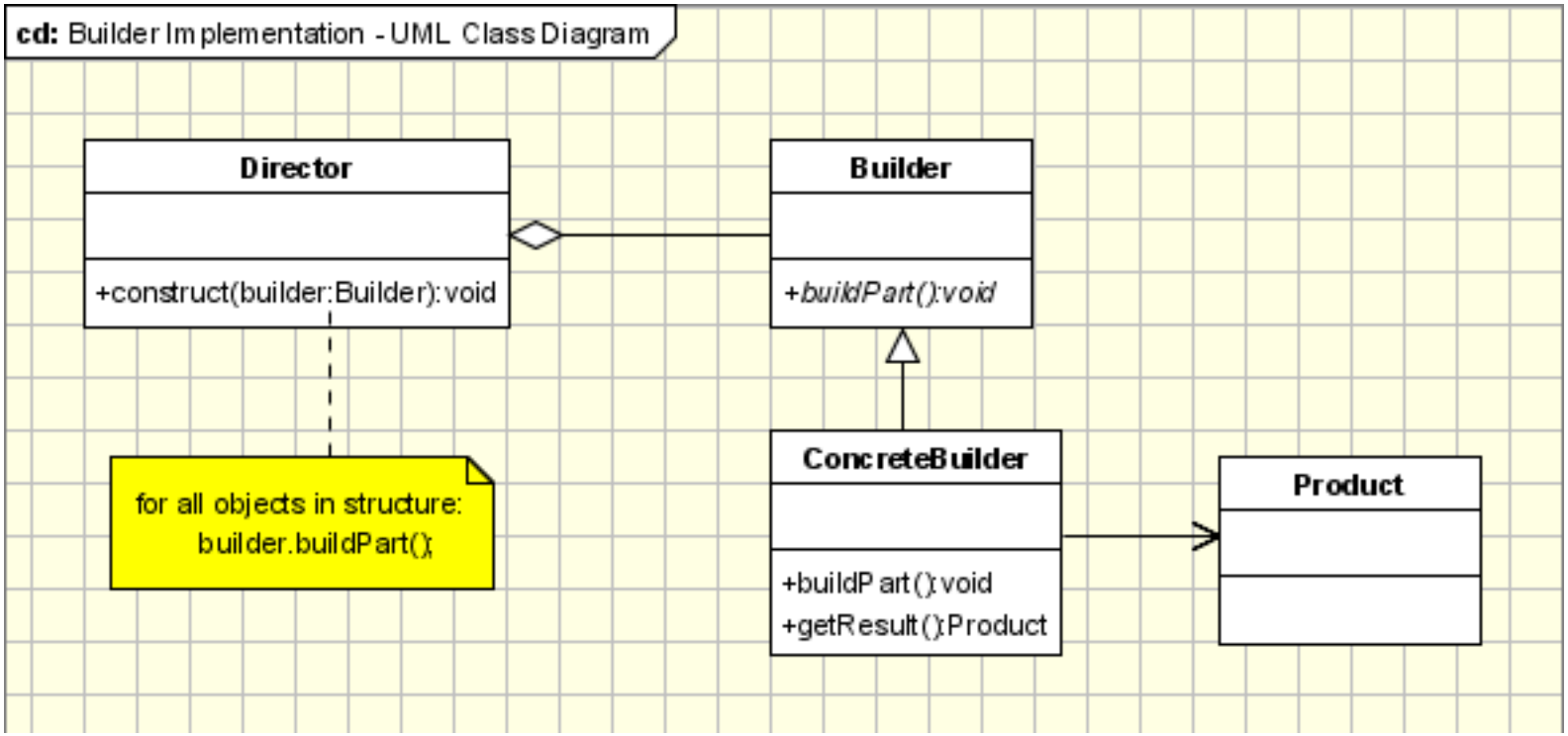
- Scopo
  - *Separare la costruzione di un oggetto complesso dalla sua rappresentazione in modo tale che lo stesso processo di costruzione può creare differenti rappresentazioni*

- Motivazione
  - Un'applicazione potrebbe avere bisogno di un meccanismo per la costruzione di oggetti complessi che è indipendente da quelli che compongono l'oggetto
  - Definisce un'istanza per creare un oggetto ma dando la possibilità alle sottoclassi di decidere quali classi istanziare
  - Riferimento ai nuovi oggetti creati attraverso un'interfaccia comune

Builder

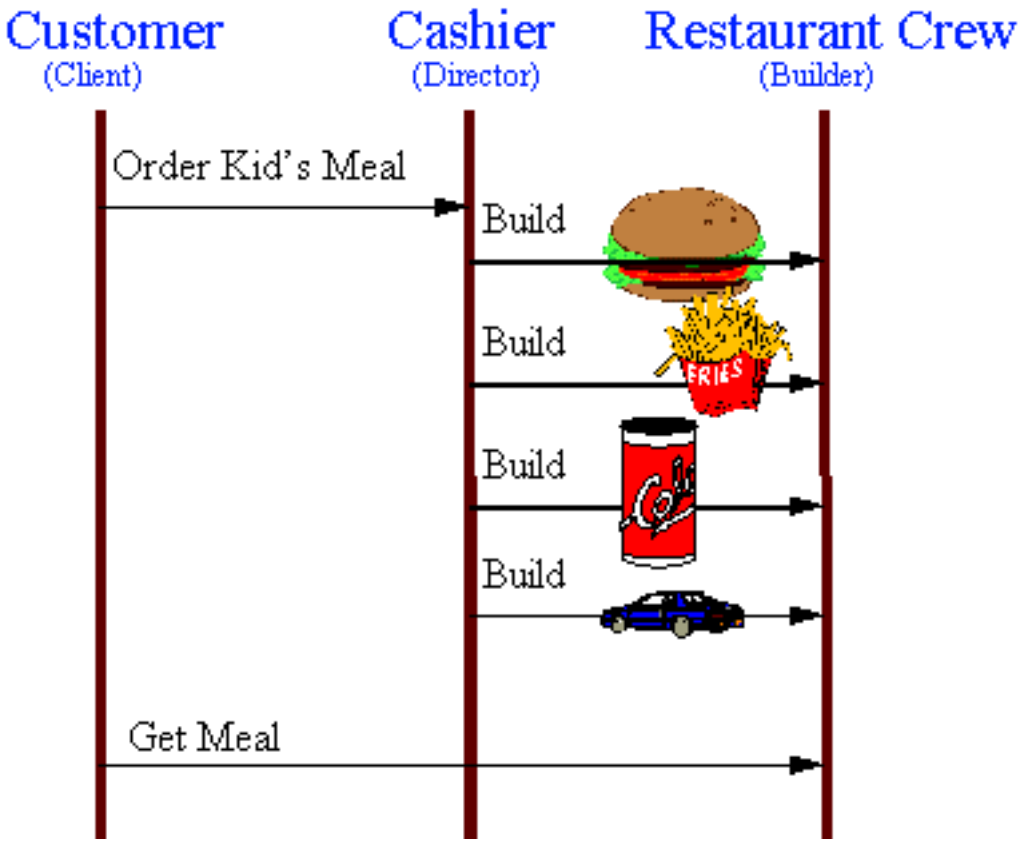
- Applicabilità
  - Un algoritmo per creare un oggetto complesso deve rendere indipendenti le parti per costruire l'oggetto e per il loro assemblaggio
  - Il processo di costruzione deve permettere differenti rappresentazioni per gli oggetti costruiti

Builder - Struttura



Struttura del pattern `Builder`

Builder - Esempio

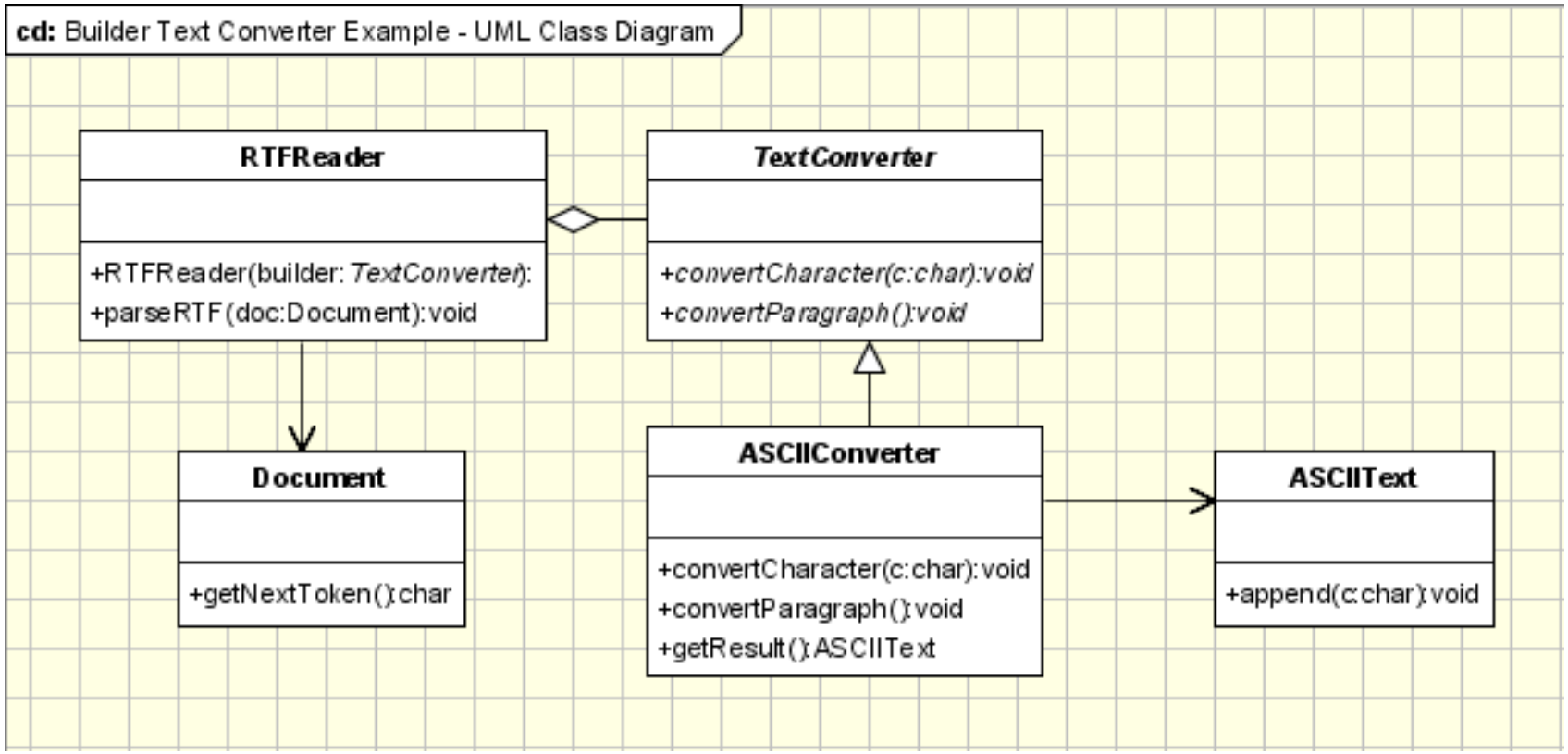


Esempio di utilizzo del pattern `Builder`

Esempi di applicazione

- Applicazione
  - Applicazione che converte un documento da format RTF ad ASCII

Esempi di applicazione



Esempio di applicazione per la conversione del testo medinate il pattern `Builder`

Implementazione

```
//Abstract Builder
class abstract class TextConverter{
        abstract void convertCharacter(char  c);
        abstract void convertParagraph();
}

// Product
class ASCIIText{
        public void append(char  c){ //Implement the code here
}
}

//This class abstracts the document object
class Document{
        static int value;
        char token;
        public char getNextToken(){
                //Get the next token
                return token;
        }
}
```


Implementazione

```
//Concrete Builder
class ASCIIConverter extends TextConverter{
        ASCIIText asciiTextObj;//resulting product

        /*converts a character to target representation and
appends to the resulting object*/
        void convertCharacter(char c){
                char asciiChar = new Character(c).charValue();
                        //gets the ascii character
                asciiTextObj.append(asciiChar);
        }
        void convertParagraph(){}
        ASCIIText getResult(){
                return asciiTextObj;
        }
}
```

Esempio di implementazione del pattern `Builder`

Implementazione

```
//Director
class RTFReader{
        private static final char EOF='0';
//Delimitor for End of File
        final char CHAR='c';
        final char PARA='p';
        char t;
        TextConverter builder;
        RTFReader(TextConverter obj){
                builder=obj;
        }
        void parseRTF(Document doc){
                while ((t=doc.getNextToken())!= EOF){
                        switch (t){
                        case CHAR: builder.convertCharacter(t);
                        case PARA: builder.convertParagraph();
                        }
                }
        }
}
```


Esempio di implementazione del pattern `Builder`

Implementazione

```
//Client
public class Client{
        void createASCIIText(Document doc){
        ASCIIConverter asciiBuilder = new ASCIIConverter();
        RTFReader rtfReader = new RTFReader(asciiBuilder);
                rtfReader.parseRTF(doc);
        ASCIIText asciiText = asciiBuilder.getResult();
        }
        public static void main(String args[]){
                Client client=new Client();
                Document doc=new Document();
                client.createASCIIText(doc);

        system.out.println("This is an example of Builder
Pattern");
        }
}
```

Esempio di implementazione del pattern `Builder`

Ulteriori esempi

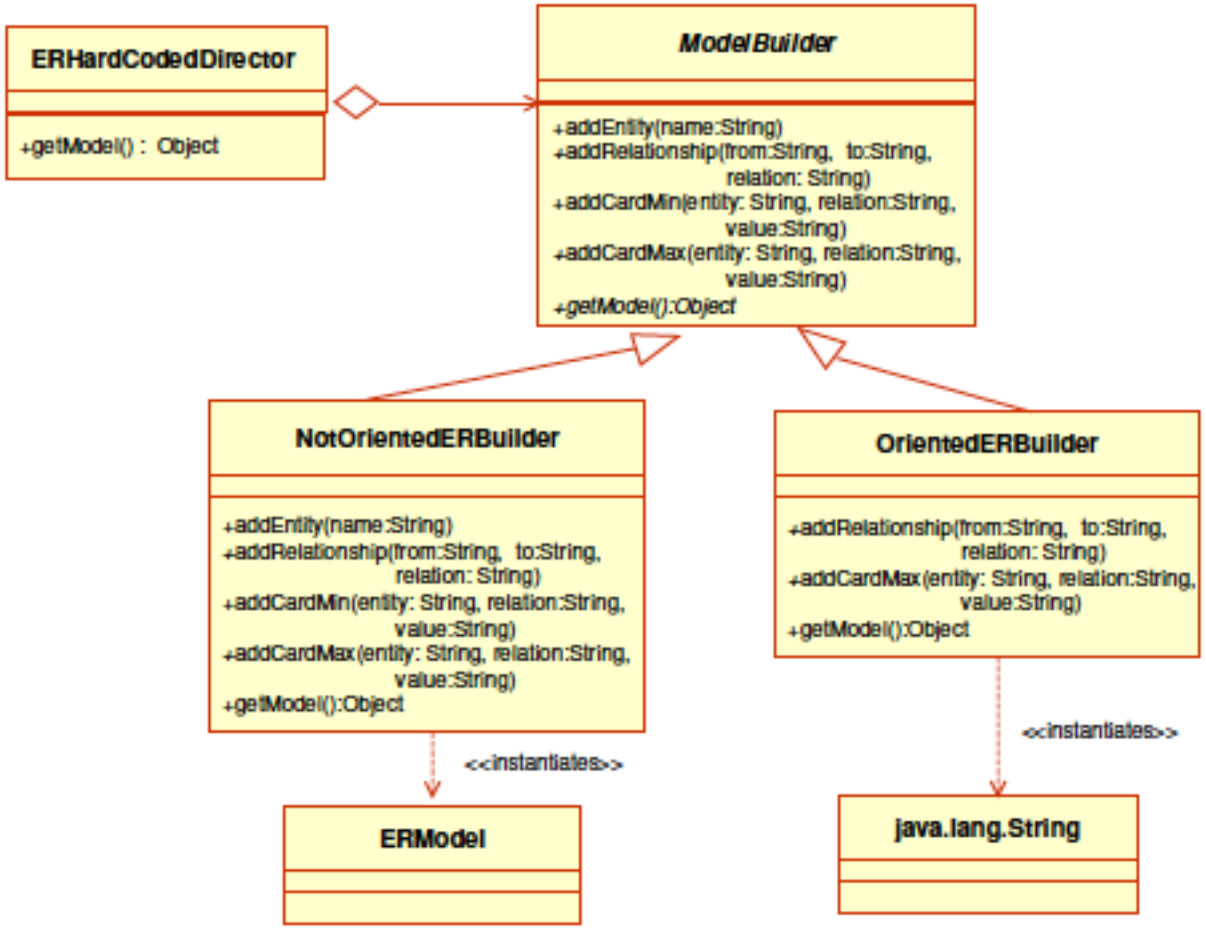
- Casa automobilistica
  - può costruire auto, biciclette, motociclette e scooter
  - `Builder` diventa `VehicleBuilder`

- Applicazione per gli esami studenti
  - lista e informazioni di esami
  - differenti utenti di login (`admin` e `user`)
  - `Builder` fornisce un'interfaccia che fornisce informazioni in base all'utente

Considerazioni

- Esempi in JDK
  - `java.lang.StringBuilder#append()` (unsynchronized)
  - `java.lang.StringBuffer#append()` (synchronized)

Esercizio



Implementare la struttura in Java

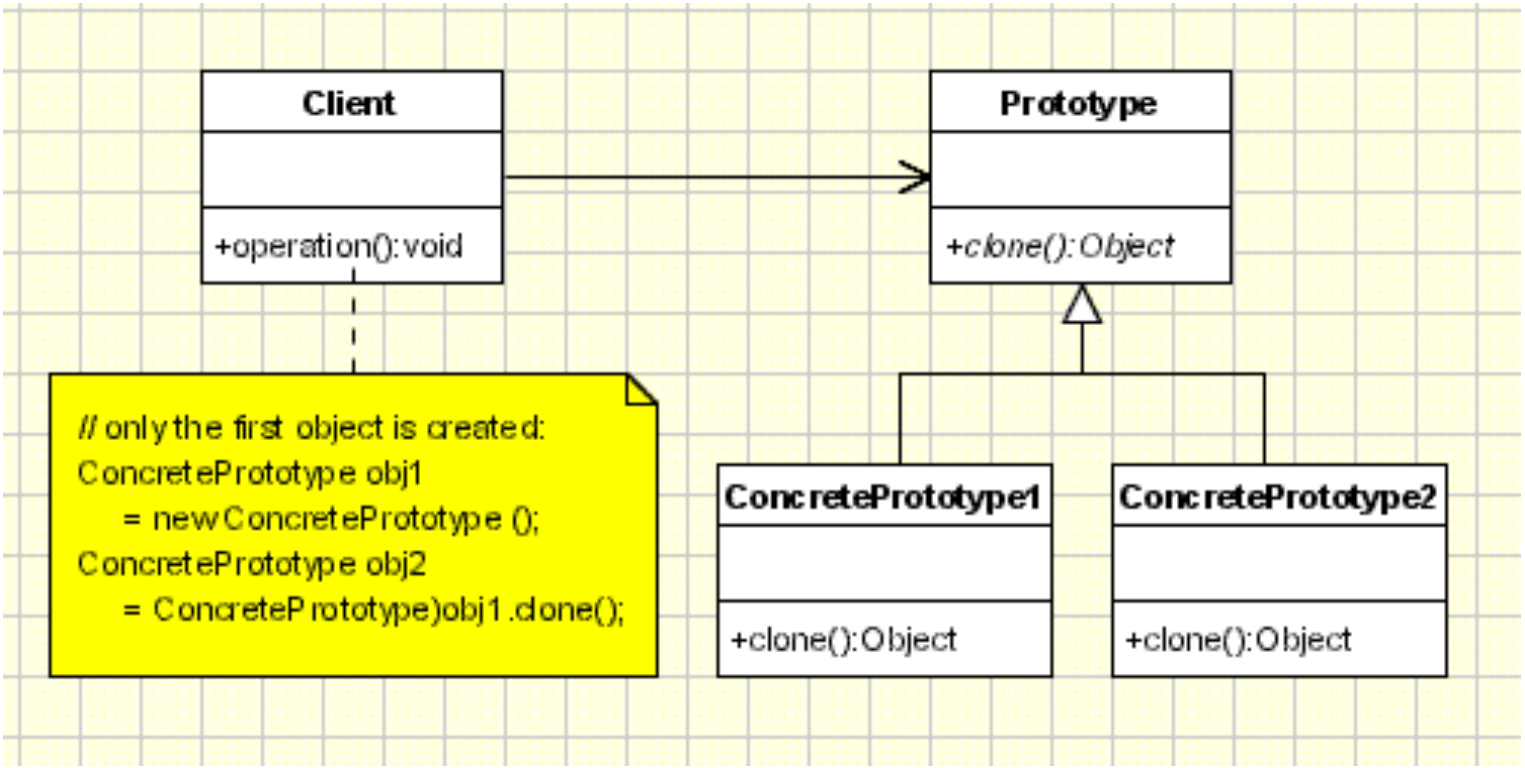
Prototype

- Scopo
  - *Specifica il tipo di oggetti da creare usando un'istanza prototipale e creando nuovi oggetti copiando questi oggetti*
- Motivazione
  - Permette ad un oggetto di creare oggetti senza conoscere la loro classe o i dettagli per crearli

Prototype

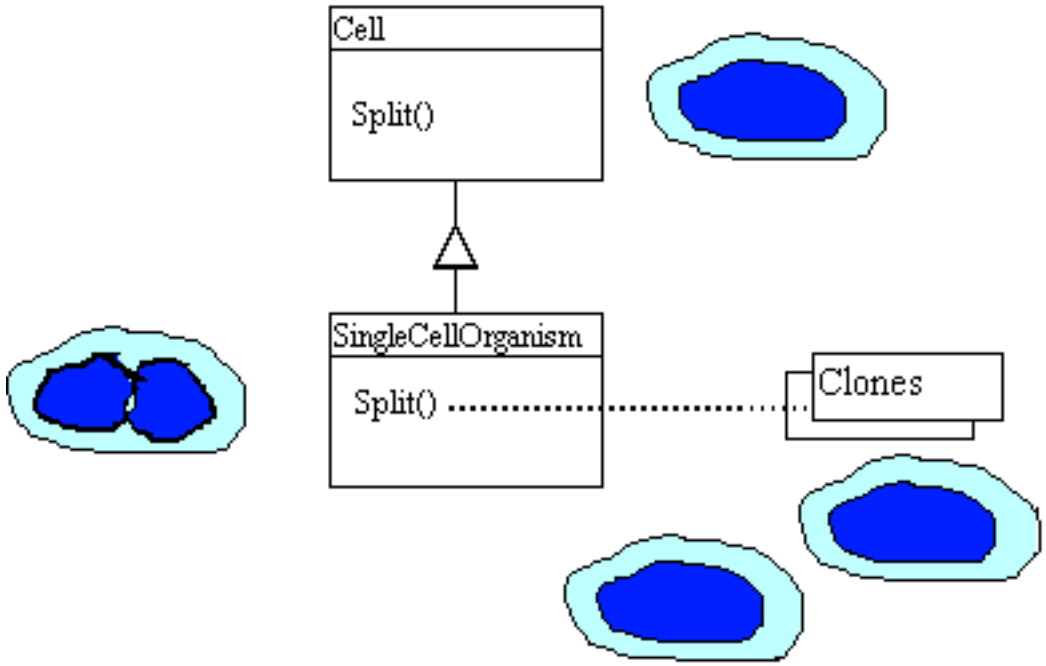
- Applicabilità
  - sistema indipendente da come i suoi prodotti sono creati, composti e rappresentati
  - le classi da istanziare sono specificate a run-time
  - per evitare di scrivere una gerarchia di classi
  - è più conveniente copiare un'istanza esistente che crearne una nuova

Prototype - Struttura



Struttura del pattern `Prototype`

Prototype - Esempio



Esempio di utilizzo del pattern `Prototype`

Implementazione

```
public interface Prototype {
                    public abstract Object clone ( );
                }

public class ConcretePrototype implements Prototype {
                    public Object clone() {
                            return super.clone();
                    }
                }

public class Client {

        public static void main( String arg[] )
                    {
ConcretePrototype obj1= new ConcretePrototype ();
ConcretePrototype obj2 = (ConcretePrototype)obj1.clone();
                    }

                }
```


Esempio di utilizzo

```
/** Prototype Class **/
 public class Cookie implements Clonable {

    public Object clone()
    {
        try{
            Cookie copy = (Cookie)super.clone();
//In an actual implementation of this pattern you might now
change references to
//the expensive to produce parts from the copies that are
held inside the prototype.

            return copy;
        }
        catch(CloneNotSupportedException e)
        {
           e.printStackTrace();
           return null;
        }
    }
 }
```


Esempio di implementazione del pattern `Prototype`

Clonable

- `Clonable`
  - interfaccia che non contiene metodi
  - è una interfaccia marker implementando la quale si esplicita il fatto che un oggetto può essere clonato

- `Clone`
  - L'eccezione `CloneNotSupportedException` viene lanciata dal metodo `clone` di `Object` se l'oggetto su cui è invocato il metodo non implementa questa interfaccia

Esempio di utilizzo

```
/** Concrete Prototypes to clone **/
 public class CoconutCookie extends Cookie { }

 /** Client Class**/
 public class CookieMachine
 {
   private Cookie cookie;
//could have been a private Cloneable cookie;
     public CookieMachine(Cookie cookie) {
         this.cookie = cookie;
     }
     public Cookie makeCookie() {
       return (Cookie)cookie.clone();
     }
     public Object clone() { }

     public static void main(String args[]){
         Cookie tempCookie =  null;
         Cookie prot = new CoconutCookie();
         CookieMachine cm = new CookieMachine(prot);
         for(int i=0; i<100; i++)
             tempCookie = cm.makeCookie();
     }
 }
```


Ulteriori esempi

- Game
  - un labirinto con diversi oggetti visuali
  - per generare diverse mappe del labirinto
    - Muri, porte, passaggi, stanze, …
  - diversi prototipi per i componenti

- Vendite
  - analisi di dati da un database
  - per ogni analisi sugli stessi dati possiamo clonare le informazioni estratte dal database