



DESIGN PATTERN


# CREAZIONALI INGEGNERIA DEL SOFTWARE

Università degli Studi di Perugia  
Dipartimento di Matematica e Informatica  
Corso di Laurea in Informatica



# DESIGN PATTERN CREAZIONALI



Relazioni tra		Campo di applicazione		
		Creational (5)	Structural (7)	Behavioral (11)
	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
Architetturali				
Model view controller				

# INTRODUZIONE



- Scopo dei *design pattern* creazionali
  - Rendere un **sistema indipendente** dall'**implementazione** concreta delle sue componenti
    - Si nascondono i tipi concreti delle classi realmente utilizzate
    - Si nascondono i dettagli sulla composizione e creazione
    - **Riduzione accoppiamento** e flessibilità
- Ampio uso dell'**astrazione** / interfacce

# SINGLETON



## ○ Scopo

- Assicurare l'**esistenza** di un'**unica istanza** di una classe
  - ... ed avere un punto di accesso **globale** a questa

## ○ Motivazione

- Alcune entità NON DEVONO avere più di un'istanza
  - Non è possibile utilizzare una variabile globale (C++)
- La classe deve tenere traccia della sua unica istanza

# SINGLETON



## ○ Applicabilità

- Deve esistere **una e una sola istanza** di una classe in tutta l'applicazione
  - L'istanza deve essere accessibile dai *client* in modo noto
- L'istanza deve essere **estendibile** con ereditarietà
  - I *client* non devono modificare il proprio codice

# SINGLETON



## ○ Struttura

Definisce *getInstance* che permette l'accesso all'unica istanza. È responsabile della creazione dell'unica istanza

Singleton	
<u>- singleton : Singleton</u>	
- Singleton()	
<u>+ getInstance() : Singleton</u>	

```
/*
 * Implementazione Java
 * "naive"
 */
public class Singleton {
    private static
        Singleton instance;

    private Singleton() {

        /* Corpo vuoto */
    }

    public static Singleton
    getInstance() {

        if (instance == nul) {
            instance =
                new Singleton();
        }
        return instance;
    }
}
```

# SINGLETON



## ○ Conseguenze

- **Controllo** completo di come e quando i *client* **accedono** all'interfaccia
- Evita il proliferare di variabili globali (C++)
- Permette la ridefinizione delle operazioni definite nel Singleton
- Può permettere un **numero massimo** e preciso di **istanze** attive
- Più flessibile delle operazioni di classe
  - Utilizzo del **polimorfismo**

# SINGLETON



## ○ Esempio

### Esempio

Un applicativo deve istanziare un oggetto che gestisce una stampante (*printer spooler*). Questo oggetto deve essere unico perché si dispone di una sola risorsa di stampa.



# SINGLETON



- Esempio
  - *Printer Spooler*

PrinterSpooler	
- <u>instance : PrinterSpooler</u>	
- PrinterSpooler()	
+ <u>getInstance() : PrinterSpooler</u>	•
+ print(file : String) : void	

```
static PrinterSpooler getInstance() {  
    if (instance == null) {  
        instance = new PrinterSpooler();  
    }  
    return instance;  
}
```

L'appoggio non è *thread safe* in Java: nessuna sincronizzazione sulla creazione di `instance`

# SINGLETON



## ○ Implementazione

- Assicurare un'unica istanza attiva (*lazy initialization*)
  - Si rende il/i **costruttore**/i **privato**/i (non accessibili)
  - Si rende disponibile un'operazione di classe di "recupero"
- Non è **possibile** utilizzare **variabili globali** (C++)
  - Non garantisce l'unicità dell'istanza
  - L'istanza è generata durante l'inizializzazione "statica"
  - Tutti i *singleton* sarebbero costruiti sempre, anche se mai utilizzati

# SINGLETON



## ○ Implementazione

- **Ereditare** da una classe *Singleton*
  - **È difficile** installare l'unica istanza nel membro `instance`
  - La soluzione migliore è utilizzare un registro di *singleton*

```
public class Singleton {
    private static Singleton instance;
    private static Map<String, Singleton> registry;

    private Singleton() { /* Corpo vuoto */ }

    public static register(String name, Singleton)
    { /* ... */}

    protected static Singleton lookup(String name)
    { /* ... */}

    public static Singleton
    getInstance(String singletonName) {

        /* Utilizza lookup per recuperare l'istanza */
    }
}
```

```
public class MySingleton
    extends Singleton {

    static {

        Singleton.register("MySingleton",
            new MySingleton())
    }

    /* ... */
}
```

# SINGLETON



Soffrono di attacco per  
*Reflection* sul costruttore

## ○ Implementazione

- Java ⇨ Costruttore privato (omesso per spazio)

```
public class PrinterSpooler {  
    private static final PrinterSpooler INSTANCE = new PrinterSpooler();  
    public static PrinterSpooler getInstance() {  
        return INSTANCE;  
    }  
}
```

*no lazy, thread safe, no subclassing, no serializable*

```
public class PrinterSpooler {  
    private static volatile PrinterSpooler INSTANCE;  
    public static PrinterSpooler getInstance() {  
        if (INSTANCE == null) {  
            synchronize (PrinterSpooler.class) {  
                if (INSTANCE == null { INSTANCE = new PrinterSpooler(); }  
            }  
        }  
        return INSTANCE;  
    }  
}
```

*lazy, thread safe, subclassing possibile, no serializable*

# SINGLETON



## ○ Implementazione

### ● Java

- Versione accettata da JDK  $\geq 1.5$

```
public enum PrinterSpooler {  
    INSTANCE;  
  
    public void print(String file) { /* ... */ }  
}
```

- Item 3 di *Effective Java* *no lazy, thread safe, no subclassing, serializable*
  - Usa costrutti nativi del linguaggio
  - Non soffre di alcun attacco
  - Conciso, leggibile e manutenibile

# SINGLETON



- Implementazione
  - Scala

Cons: meno controllo sull'inizializzazione

```
object PrinterSpooler extends ... {  
  def print(file: String) {  
    // do something  
  }  
}
```

- In Java il *pattern* Singleton è uno dei più utilizzati
  - Mancanza a livello di linguaggio
  - *Error prone*
- Scala risolve introducendo il tipo *object*
  - Implementazione del *pattern* Singleton nel linguaggio
  - *Thread-safe*
  - Gli *Object* sono inizializzati su richiesta (*laziness*)

# SINGLETON



## ○ Implementazione

- Javascript: si utilizza il *module pattern*

*Private  
namespace*

```
var mySingleton = (function () {  
  var instance;  
  function init() {  
    // Private methods and variables  
    function privateMethod() { console.log( "I am private" ); };  
    var privateRandomNumber = Math.random();  
  }  
  
  return {  
    // Public methods and variables  
    publicMethod: function () {  
      console.log( "The public can see me!" );  
    },  
    getRandomNumber: function() { return privateRandomNumber; }  
  };  
};  
  
return {  
  getInstance: function () {  
    if ( !instance ) { instance = init(); }  
    return instance;  
  }  
}; })();
```

*Public  
functions / variables*

# SINGLETON



- Casi d'uso del pattern:
  - Accesso ad interfacce hardware: es. printer spooler (visto prima)
  - **Logger**: nel caso di applicazioni dove l'utilità di logging deve produrre un unico file di log per i messaggi ricevuti dagli utenti
  - **Configuration file**: per creare una singola istanza del file di configurazione a cui si può accedere da chiamate multiple concorrentemente.
  - **Cache**: usare la cache come un oggetto singleton che può per avere un riferimento globale



# BUILDER



## ○ Scopo

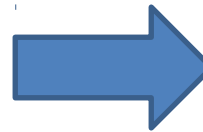
- Separa la **costruzione** di un oggetto complesso dalla sua **rappresentazione**

## ○ Motivazione

- Necessità di **riutilizzare** un medesimo algoritmo di costruzione per più oggetti di tipo **differente**
- Processo di costruzione *step-by-step*

Il cassiere deve:

- Inserire il panino
- Inserire le patate
- Inserire la frutta
- ...



## ○ Applicabilità

- La procedura di **creazione** di un oggetto complesso deve essere **indipendente** dalle **parti** che compongono l'oggetto
- Il processo di **costruzione** deve permettere **diverse rappresentazioni** per l'oggetto da costruire

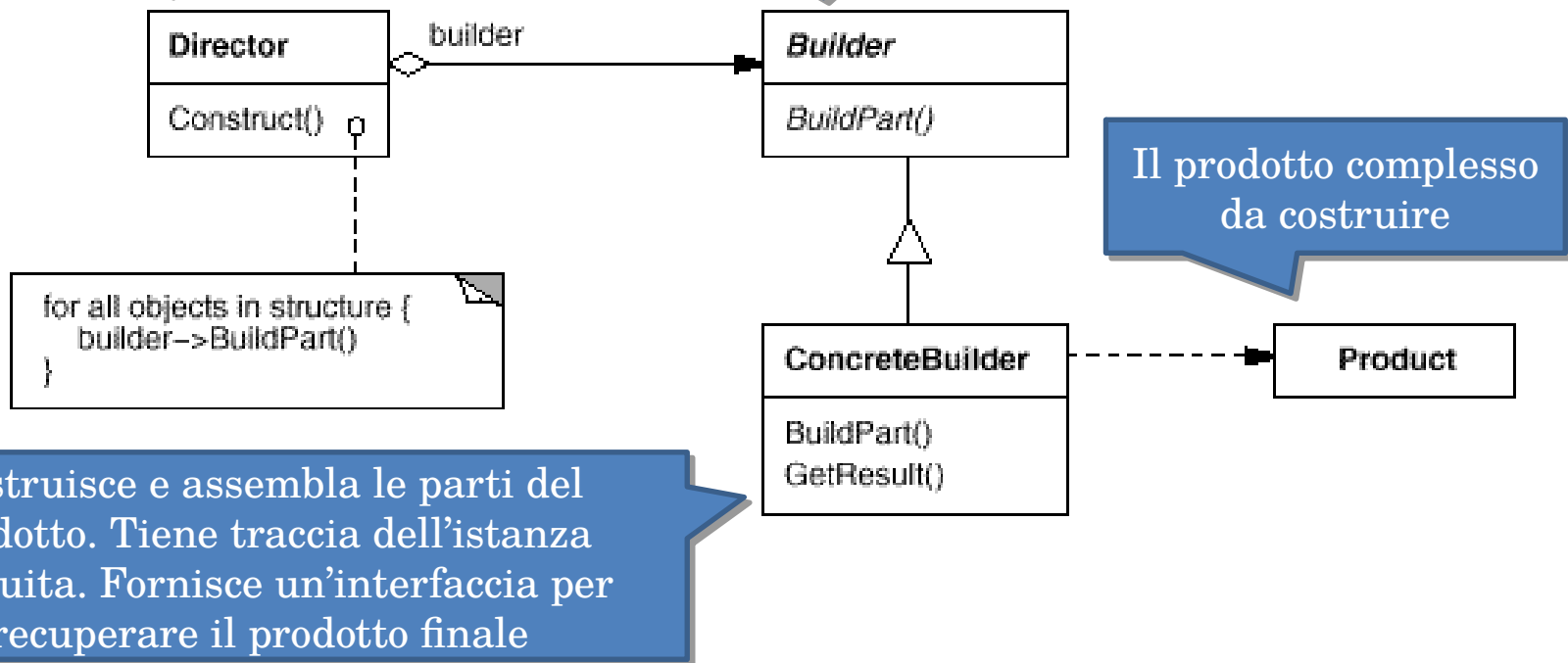
# BUILDER



## ◦ Struttura

Costruisce un oggetto utilizzando il *builder* (procedura)

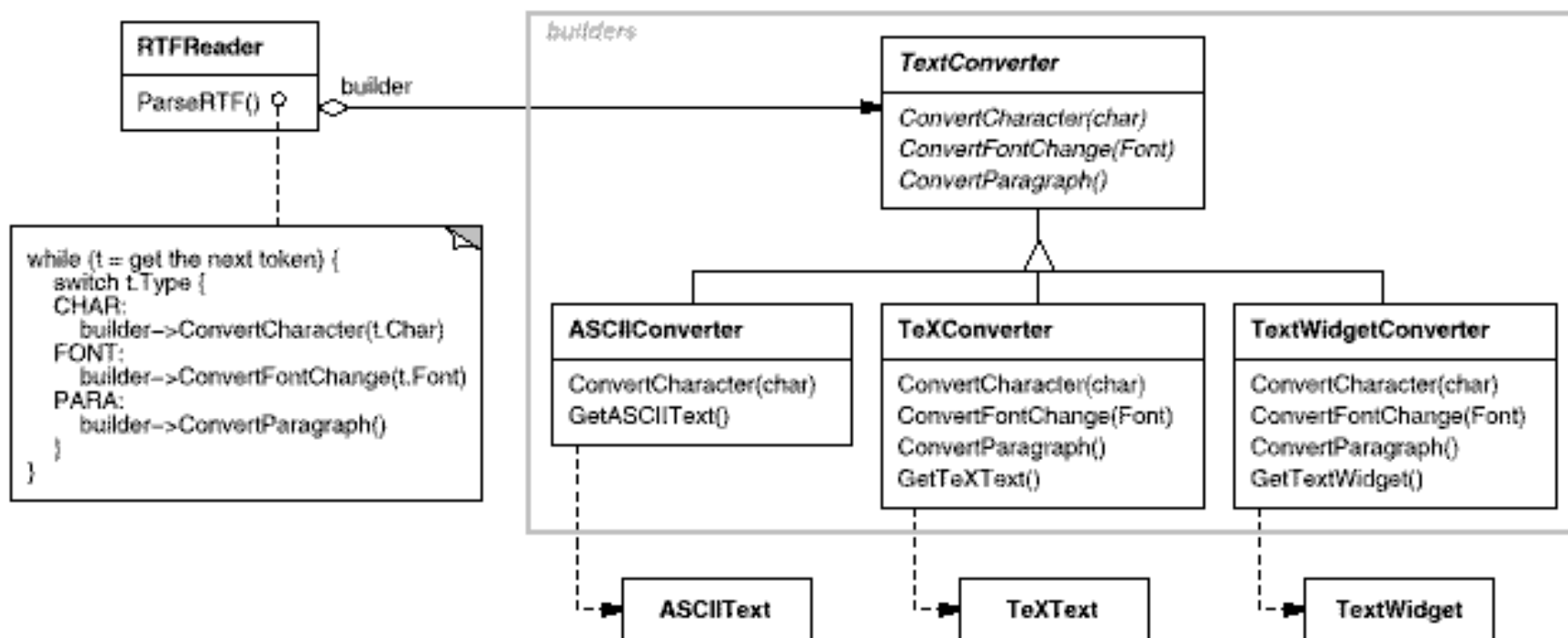
Specifica l'interfaccia da utilizzare per assemblare il prodotto



# BUILDER



## ○ Esempio



# BUILDER

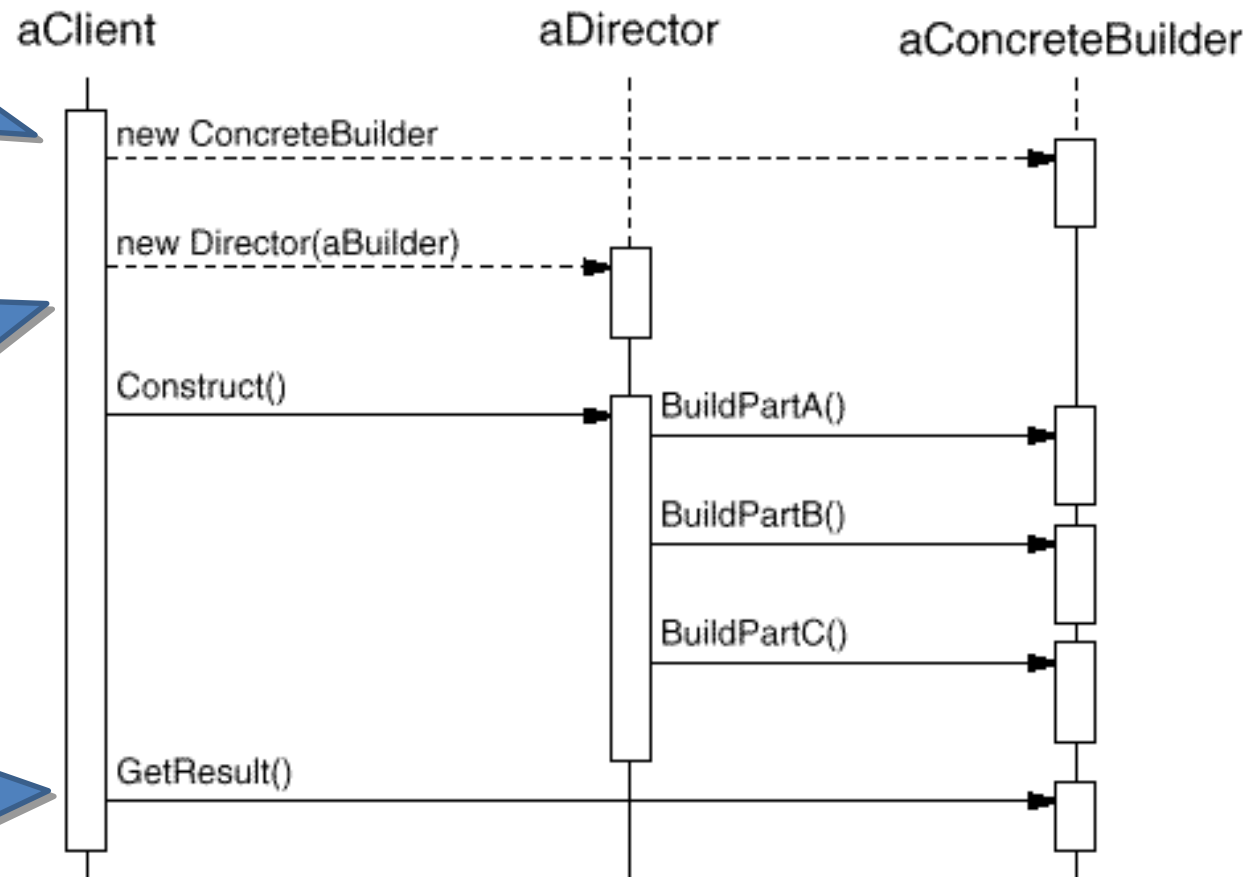


## ○ Struttura

Il *client* conosce il tipo *builder* concreto

Il *director* notifica il *builder* delle parti che devono essere costruite

Il *client* recupera dal *builder* concreto il prodotto finito



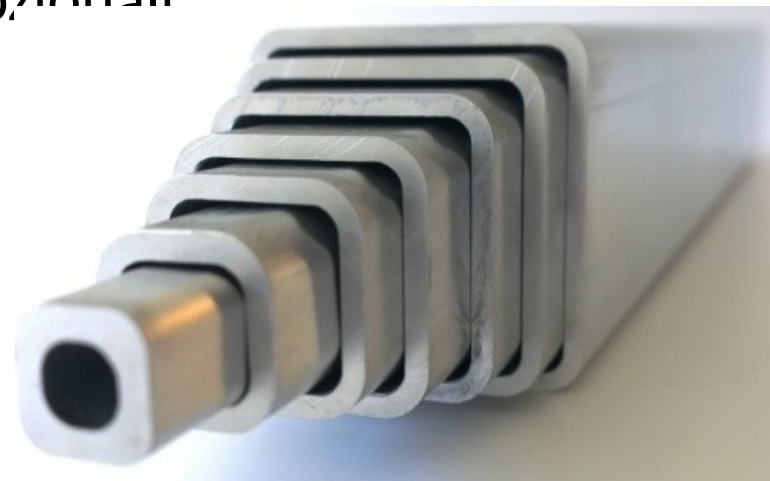
- Conseguenze
  - Facilita le **modifiche** alla rappresentazione interna di un prodotto
    - È sufficiente costruire un nuovo *builder*: NO *telescoping*!
  - **Isola** il codice dedicato alla costruzione di un prodotto dalla sua rappresentazione
    - Il *client* non conosce le componenti interne di un prodotto
    - **Encapsulation**
    - L'orchestrazione dei processi di costruzione è unica
  - Consente un **controllo migliore** del processo di costruzione
    - Costruzione *step-by-step*
    - Accentramento logica di validazione

## ○ Esempio di telescoping:

```
public class User {  
    private final String firstName;    //required  
    private final String lastName;    //required  
    private final int age;            //optional  
    private final String phone;       //optional  
    private final String address;     //optional  
  
    public User(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    public User(String firstName, String lastName, int age) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.age = age;  
    }  
  
    public User(String firstName, String lastName, int age, String phone) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        /// ...  
    }  
}
```

## ○ Telescoping:

- utilizzo di costruttori in overloading
- per ogni change request - informazioni non previste dalla analisi iniziale del progetto - è stato aggiunto un nuovo costruttore incrementando il costruttore più “esteso” con dei parametri opzionali
- codice poco leggibile!!!





# BUILDER



## ◦ Soluzione:

```
public class User{
    private final String firstName; // required
    private final String lastName; // required
    private final int age; // optional
    private final String phone; // optional
    private final String address; // optional

    private User(UserBuilder builder) {
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.age = builder.age;
        this.phone = builder.phone;
        this.address = builder.address;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }
}
```

**Attributi privati e final:** sono inizializzabili solo dal costruttore

**Costruttore privato:** la classe non può essere istanziata direttamente

# BUILDER



## ◦ Soluzione:

```
public static class UserBuilder {  
    private final String firstName;    //required  
    private final String lastName;    //required  
    private int age;                  //optional  
    private String phone;              //optional  
    private String address;            //optional  
  
    public UserBuilder(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    public UserBuilder age(int age) {  
        this.age = age;  
        return this;  
    }  
  
    // ...  
  
    public User build() {  
        return new User(this);  
    }  
}
```

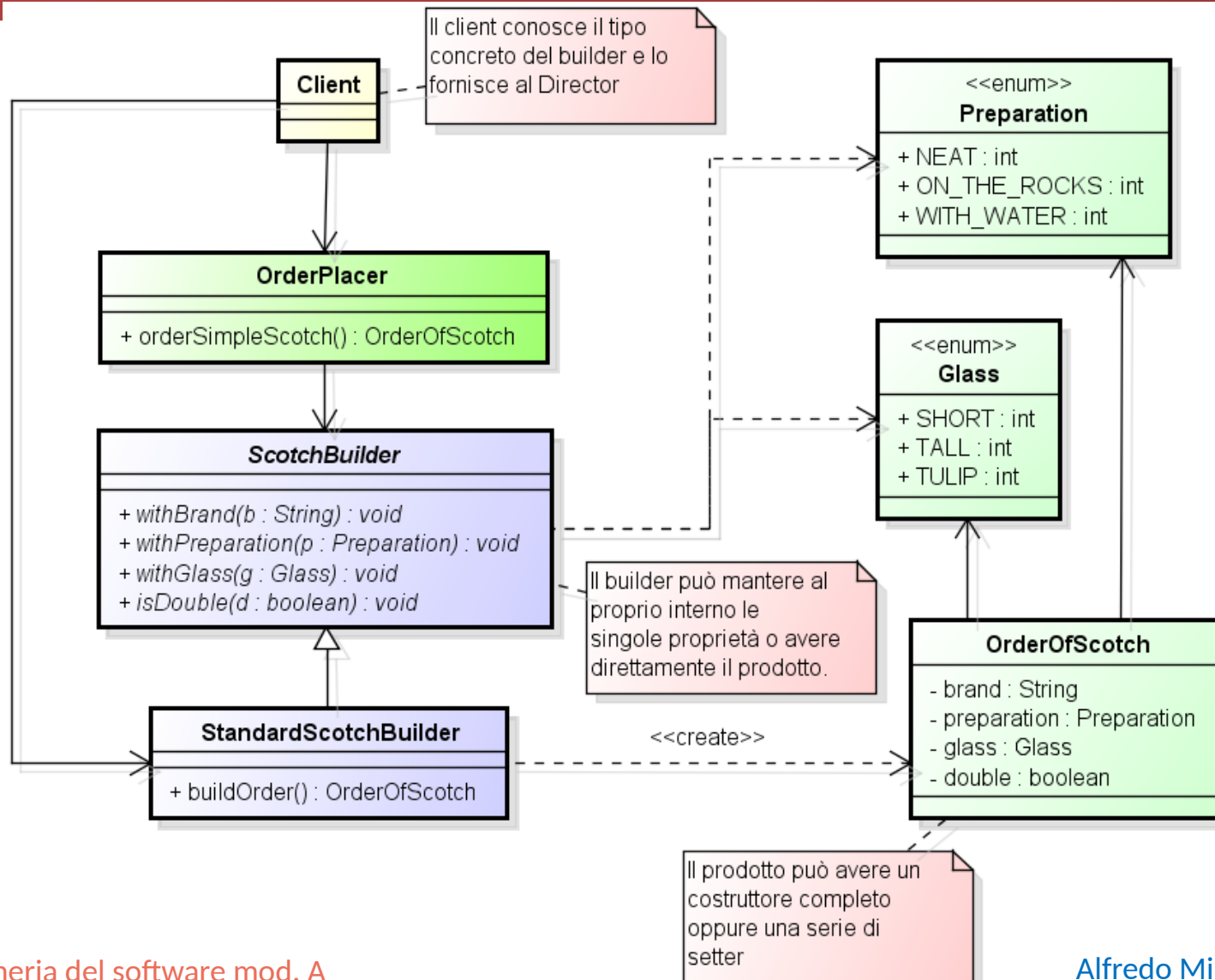
Si può introdurre la  
validazione della  
classe base

## ○ Esempio

Si vuole ordinare un bicchiere di *scotch*. È necessario fornire al barman alcune informazioni: la marca del whiskey, come deve essere preparato (liscio, *on the rocks*, allungato) e se lo si desidera doppio. È inoltre possibile scegliere il tipo di bicchiere (piccolo, lungo, a tulipano).

[se fossimo veramente intenditori, anche la marca e la temperatura dell'acqua sarebbero fondamentali...]

# BUILDER



## ○ Implementazione

- Il *builder* defisce un'**interfaccia** per ogni parte che il *director* può richiedere di costruire
  - Abbastanza **generale** per la costruzione di prodotti differenti
  - *Appending process*
- Nessuna classe astratta comune per i prodotti
  - **Differiscono** notevolmente fra loro
  - Se simili, valutare l'utilizzo di un *Abstract Factory Pattern*
- Fornire metodi **vuoti** come *default*
  - I *builder* concreti ridefiniscono solo i metodi necessari

## ○ Implementazione

- Java: il *builder* diventa classe interna del prodotto

```
public class OrderOfScotch {  
    private String brand;  
    private Preparation preparation;    // Si puo' dichiarare enum interna  
    // ...  
    private OrderOfScotch() { }    // Costruttore privato per il prodotto  
    private OrderOfScotch(ScotchBuilder builder) {  
        this.brand = builder._brand;  
        // ...  
    }  
    public static class Builder {  
        private String _brand;    // Inserisco '_' per diversificare  
        private Preparation _preparation;  
        // ...  
        public ScotchBuilder withBrand(String brand) {  
            this._brand = brand;  
            return this;    // Appending behaviour  
        }  
        // CONTINUA...
```

# BUILDER



## ○ Implementazione

### ● Java

```
// CONTINUA...
public OrderOfScotch build() { return new OrderOfScotch(this); }
} // public static class Builder

public static void main(String[] args) {
    // Il client non conosce il processo di costruzione e le classi
    // prodotto e builder sono correttamente accoppiate tra loro.
    // Non e' possibile costruire un prodotto se non con il builder.
    OrderOfScotch oos = new OrderOfScotch.Builder()
        .withBrand("Brand 1")
        .withPreparation(NEAT)
        // ...
        .build();
}
} // public class OrderOfScotch
```

## ○ Implementazione

- Javascript

```
var Builder = function() {  
  var a = "defaultA";    // Valori default  
  var b = "defaultB";  
  return {  
    withA : function(anotherA) {  
      a = anotherA;  
      return this;    // Append behaviour  
    },  
    withB : function(anotherB) {  
      b = anotherB;  
      return this;  
    },  
    build : function() {  
      return "A is: " + a + ", B is: " + b;  
    }  
  };  
};  
  
var first = builder.withA().withB("a different value for B").build();
```

Approccio classico,  
con l'utilizzo degli  
*scope* per la  
creazione degli  
oggetti



## ○ Implementazione

- Javascript – **JQuery**
  - Costruzione *step-by-step* del DOM

```
// I metodi appendTo, attr, text permettono di costruire un oggetto  
// all'interno del DOM in modo step-by-step
```

```
$( '<div class="foo">bar</div>' );
```

```
$( '<p id="test">foo <em>bar</em></p>' ).appendTo( "body" );
```

```
var newParagraph = $( "<p />" ).text( "Hello world" );
```

```
$( "<input />" )  
    .attr({ "type": "text", "id": "sample" })  
    .appendTo( "#container" );
```

## ○ Implementazione

### ● Scala

- In Scala è possibile utilizzare i principi dei linguaggi funzionali
  - Immutabilità del *builder*
- *Type-safe builder*
  - Assicura staticamente che sono state fornite tutte le informazioni necessarie alla costruzione
- Si utilizzano *feature* avanzate del linguaggio
  - *Type constraints*
    - <http://blog.rafaelferreira.net/2008/07/type-safe-builder-pattern-in-scala.html>
    - <http://www.tikalk.com/java/type-safe-builder-scala-using-type-constraints/>

# ABSTRACT FACTORY



- Scopo
  - Fornisce un'interfaccia per creare famiglie di prodotti senza specificare classi concrete
- Motivazione
  - Applicazione configurabile con diverse famiglie di componenti
    - Toolkit grafico
  - Si definisce una classe astratta *factory* che definisce le interfacce di creazione.
    - I *client* non costruiscono direttamente i “prodotti”
  - Si definiscono le interfacce degli oggetti da creare
  - Le classi che concretizzano *factory* vengono costruite una volta sola

# ABSTRACT FACTORY



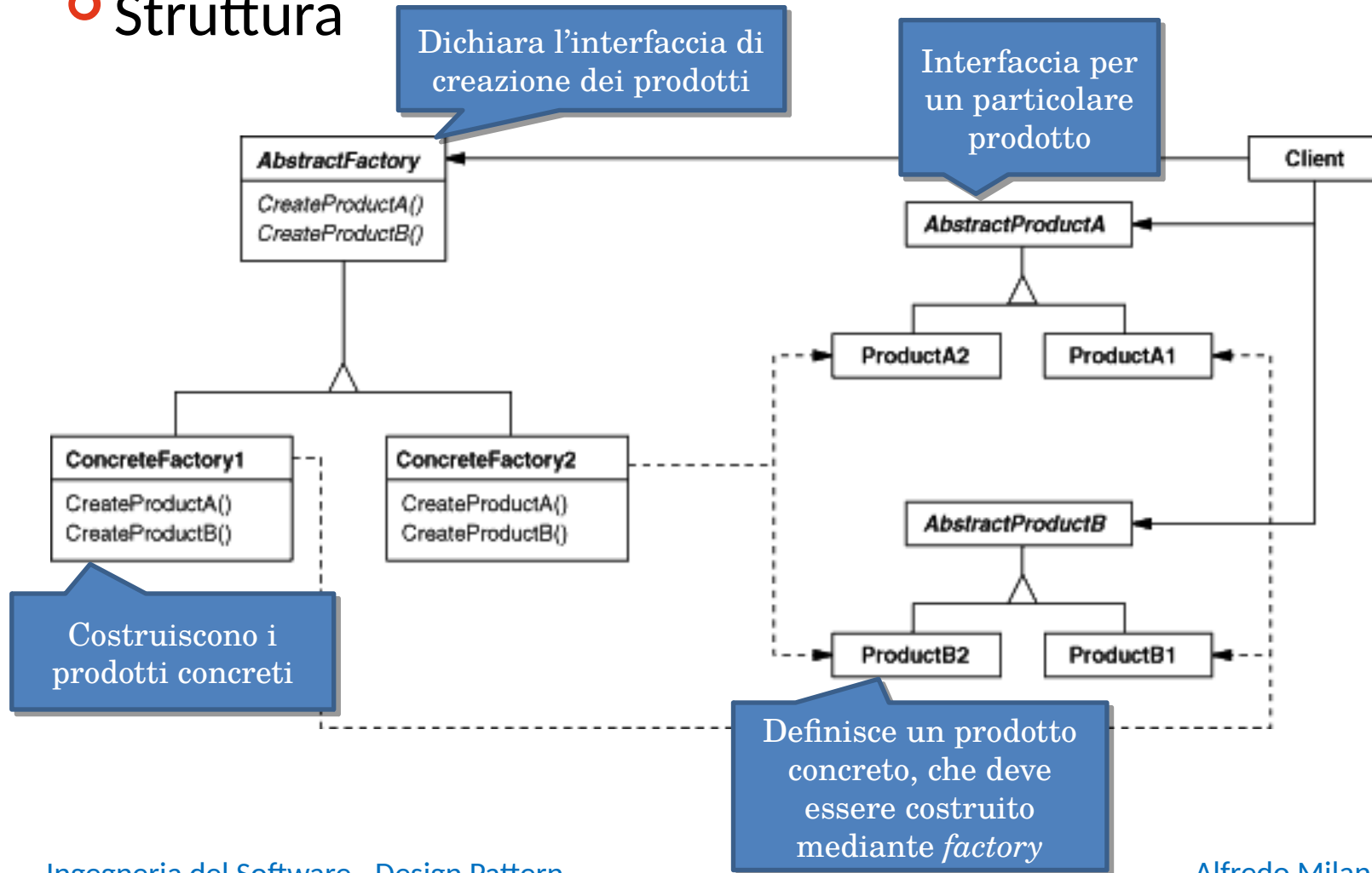
## ○ Applicabilità

- Un sistema **indipendente** da come i **componenti** sono creati, composti e rappresentati
- Un sistema **configurabile** con più famiglie prodotti
- Le **componenti** di una famiglia DEVONO essere **utilizzate insieme**
- Si vuole fornire una libreria di classi prodotto, senza rivelarne l'implementazione.

# ABSTRACT FACTORY



## ○ Struttura



# ABSTRACT FACTORY



- Conseguenze
  - Isolamento dei tipi concreti
    - I *client* manipolano unicamente interfacce, i nomi dei prodotti sono nascosti
  - Semplicità maggiore nell'*utilizzo* di una diversa famiglia di prodotti
    - La *factory* concreta appare solo una volta nel programma
  - Promuove la *consistenza* fra i prodotti
  - Difficoltà nel supportare nuovi prodotti
    - Modificare l'interfaccia della *factory* astratta costringe il cambiamento di tutte le sotto classi.

# ABSTRACT FACTORY



## ○ Esempio

### Esempio

Si vuole realizzare un negozio di vendita di sistemi Hi-Fi, dove si eseguono dimostrazioni dell'utilizzo dei prodotti.

Esistono due famiglie di prodotti, basate su tecnologie diverse:

- supporto di tipo nastro (tape)
- supporto di tipo digitale (CD).

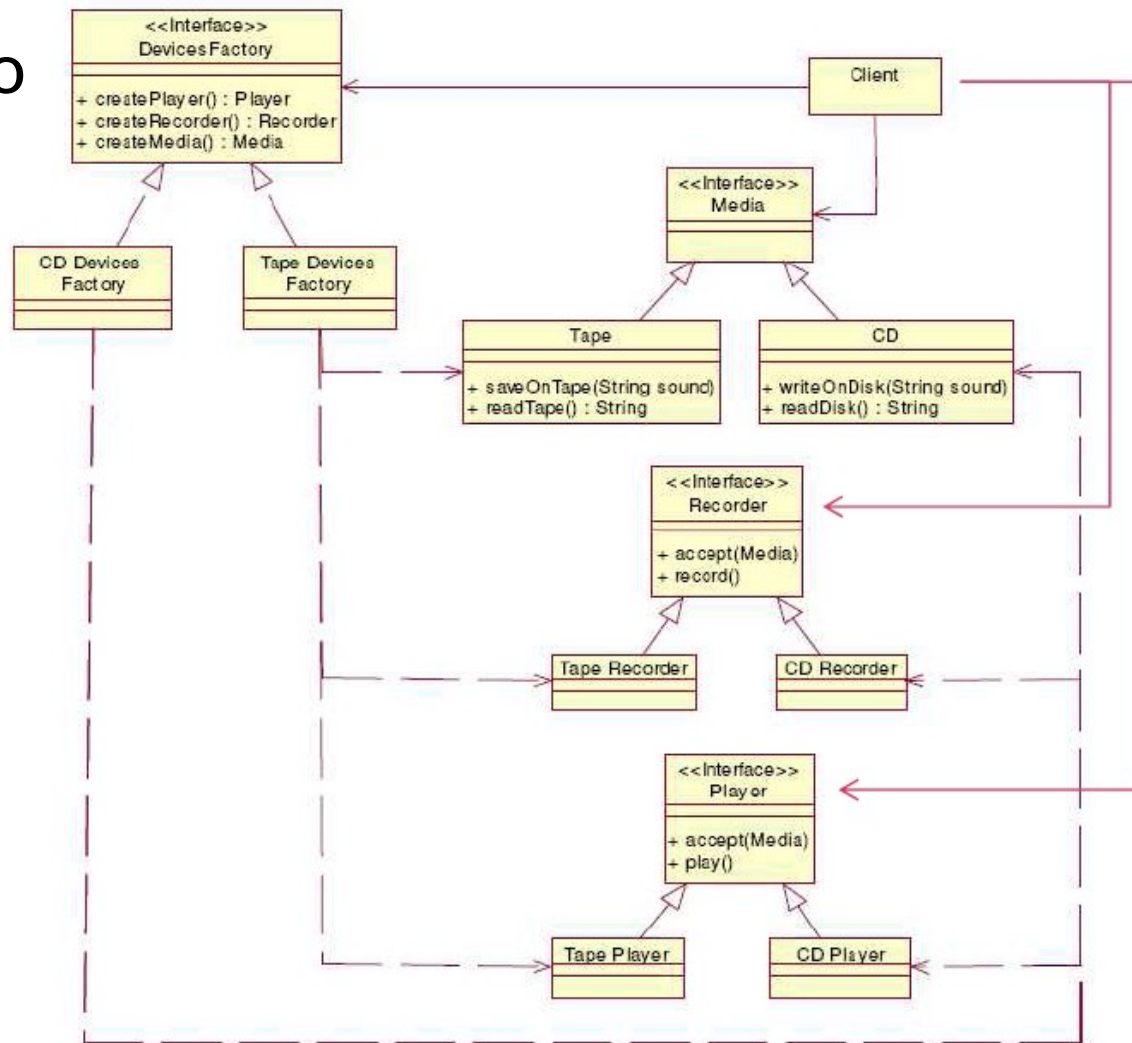
Ogni famiglia è composta da:

- supporto (tape o CD)
- masterizzatore (recorder)
- riproduttore (player).

# ABSTRACT FACTORY



## ○ Esempio





# ABSTRACT FACTORY



## ○ Esempio

### Esempio

Si vuole realizzare un negozio di vendita di device mobile, dove si eseguono dimostrazioni dell'utilizzo dei prodotti.

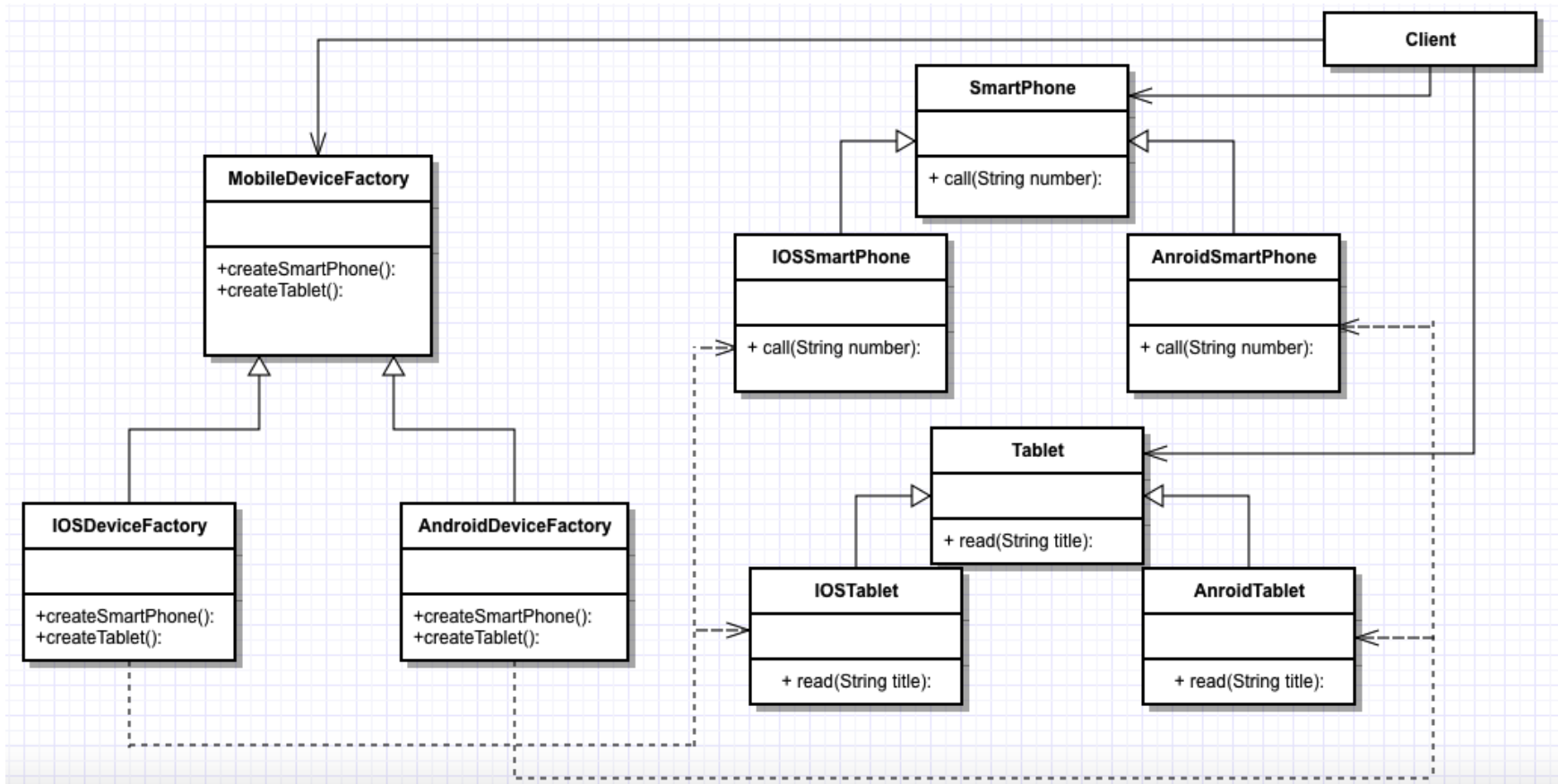
Esistono due famiglie di prodotti, basate su sistemi operativi diversi:

- sistemi IOS;
- sistemi android.

Ogni famiglia è composta da:

- smartphone.
- tablet.

# ABSTRACT FACTORY



# ABSTRACT FACTORY



## ○ Implementazione

```
public interface MobileDeviceFactory {  
    public SmartPhone createSmartPhone();  
    public Tablet createTablet();  
}
```

AbstracFactory

```
public class IOSDeviceFactory implements MobileDeviceFactory{  
    public SmartPhone createSmartPhone(){  
        return new IOSSmartPhone();  
    }  
  
    public Tablet createTablet(){  
        return new IOSTablet();  
    }  
}
```

Factory concreta per  
creare i prodotti  
della famiglia IOS

```
public class AndroidDeviceFactory implements MobileDeviceFactory{  
    public SmartPhone createSmartPhone(){  
        return new AndroidSmartPhone();  
    }  
  
    public Tablet createTablet(){  
        return new AndroidTablet();  
    }  
}
```

Factory concreta per  
creare i prodotti  
della famiglia  
Android

# ABSTRACT FACTORY



## ○ Implementazione

```
public interface SmartPhone {  
    public void call(String number);  
}  
  
public interface Tablet {  
    public void read(String titolo);  
}
```

Interfacce per la  
creazione dei due  
prodotti

```
public class IOSSmartPhone implements SmartPhone{  
    public void call(String number) {  
        System.out.println("IOS: Chiamata in corso "+number);  
    }  
}
```

```
public class AndroidTablet implements Tablet{  
    public void read(String titolo) {  
        System.out.println("ANDROID: Read: "+titolo);  
    }  
}
```

# ABSTRACT FACTORY



## ○ Implementazione

```
public class Test {  
  
    public static void main(String[] args){  
        MobileDeviceFactory iosMobileDevice = new IOSDeviceFactory();  
        SmartPhone iosSmartPhone = iosMobileDevice.createSmartPhone();  
        iosSmartPhone.call("075 585 5001");  
        Tablet iosTablet = iosMobileDevice.createTablet();  
        iosTablet.read("Manifesto Corso di Informatica");  
  
        MobileDeviceFactory androidMobileDevice = new AndroidDeviceFactory();  
        SmartPhone androidSmartPhone = androidMobileDevice.createSmartPhone();  
        androidSmartPhone.call("075 5851");  
        Tablet androidTablet = androidMobileDevice.createTablet();  
        androidTablet.read("Centralino Universita");  
    }  
}
```

Si creano i due prodotti  
della famiglia IOS

Si creano i due  
prodotti dlla  
famiglia Android

# ABSTRACT FACTORY



## ○ Esempio

- Scala: *companion object* (**Factory Method**)

```
trait Animal
private class Dog extends Animal
private class Cat extends Animal
```

```
object Animal {
  def apply(kind: String) =
    kind match {
      case "dog" => new Dog()
      case "cat" => new Cat()
    }
}
```

Equivale a  
Animal(...)

```
val animal = Animal("dog")
```

- *Apply* è tradotto in un simil - costruttore
- Si utilizza per la costruzione delle *factory concrete*

# ABSTRACT FACTORY



## ○ Esempio

### ● Javascript: varie tecniche di implementazione

```
var AbstractVehicleFactory = (function () {  
    // Storage for our vehicle types  
    var types = {};  
  
    return {  
        getVehicle: function ( type, customizations ) {  
            var Vehicle = types[type];  
            return (Vehicle ? new Vehicle(customizations) : null);  
        },  
        registerVehicle: function ( type, Vehicle ) {  
            var proto = Vehicle.prototype;  
            if ( proto.drive && proto.breakDown ) {  
                types[type] = Vehicle;  
            }  
            return AbstractVehicleFactory;  
        }  
    };  
})();
```

Registro solamente gli  
oggetti che soddisfano  
un contratto (*abstract  
product*)

# ABSTRACT FACTORY



## ○ Implementazione

- Solitamente si necessita di una sola istanza della *factory* (Singleton design pattern)
- Definizione di *factory* estendibili
  - Aggiungere un **parametro** ai metodi di **creazione** dei prodotti
    - Il parametro specifica il tipo di prodotto
    - Nei linguaggi tipizzati staticamente è possibile solo se tutti i prodotti condividono la stessa interfaccia
    - Può obbligare a *down cast* pericolosi ...



# RIFERIMENTI



- Design Patterns, Elements of Reusable Object Oriented Software, GoF, 1995, Addison-Wesley
- Design Patterns  
[http://sourcemaking.com/design\\_patterns](http://sourcemaking.com/design_patterns)
- Java DP  
<http://www.javacamp.org/designPattern/>
- Exploring the Decorator Pattern in Javascript  
<http://addyosmani.com/blog/decorator-pattern/>
- Design Patterns in Scala <http://pavelfatin.com/design-patterns-in-scala>
- Item 2: Consider a builder when faced with