



DESIGN PATTERN CREAZIONALI

INGEGNERIA DEL SOFTWARE

Università degli Studi di Padova
Dipartimento di Matematica

Corso di Laurea in Informatica, A.A. 2016 – 2017

rcardin@math.unipd.it



DESIGN PATTERN CREAZIONALI



		Campo di applicazione		
		Creational (5)	Structural (7)	Behavioral (11)
Relazioni tra	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		

Ingegneria del software mod. A

Riccardo Cardin

2

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

Ingegneria del software mod. A

Riccardo Cardin

3

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

Ingegneria del software mod. A

Riccardo Cardin

4

SINGLETON



o Applicabilità

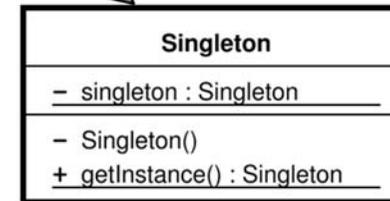
- Deve esistere una e una sola istanza di una classe in tutta l'applicazione
 - o L'istanza deve essere accessibile dai *client* in modo noto
- L'istanza deve essere estendibile con ereditarietà
 - o I *client* non devono modificare il proprio codice

SINGLETON



o Struttura

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



```
/*
 * 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



o 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
 - o Utilizzo del polimorfismo

SINGLETON



o Esempio

Esempio

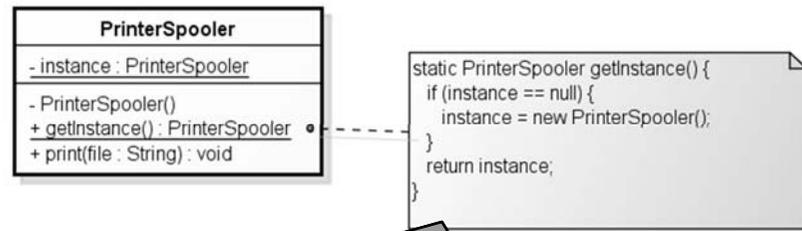
Un applicativo deve istanziare un oggetto che gestisce una stampante (*printer spooler*). Questo oggetto deve essere unico perché si dispone di una sola risorsa di stampa.

SINGLETON



o Esempio

- *Printer Spooler*



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

SINGLETON



o Implementazione

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

SINGLETON



o Implementazione

- Ereditare da una classe *Singleton*
 - o È difficile installare l'unica istanza nel membro *instance*
 - o 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 void register(String name, Singleton s) {
        /* ... */
    }

    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



o Implementazione

Soffrono di attacco per *Reflection* sul costruttore

- 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



o Implementazione

- Java
 - o Versione accettata da JDK ≥ 1.5

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

no lazy, thread safe, no subclassing, serializable

- o Item 3 di *Effective Java* di Joshua Bloch
 - o Usa costrutti nativi del linguaggio
 - o Non soffre di alcun attacco
 - o Conciso, leggibile e manutenibile

SINGLETON



o 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
 - o Mancanza a livello di linguaggio
 - o *Error prone*
- Scala risolve introducendo il tipo *object*
 - o Implementazione del *pattern* Singleton nel linguaggio
 - o *Thread-safe*
 - o Gli *Object* sono inizializzati su richiesta (*laziness*)

SINGLETON



o Implementazione

- Javascript: si utilizza il *module pattern*

```
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;  
    }  
}; }());
```

Private namespace

Public functions/variables

BUILDER



o Scopo

- Separa la costruzione di un oggetto complesso dalla sua rappresentazione

o Motivazione

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

Il cassiere deve:

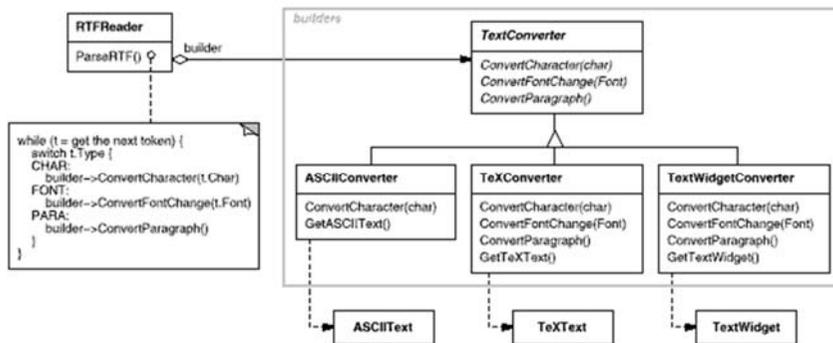
1. Inserire il panino
2. Inserire le patate
3. Inserire la frutta
4. ...



BUILDER



o Esempio



BUILDER



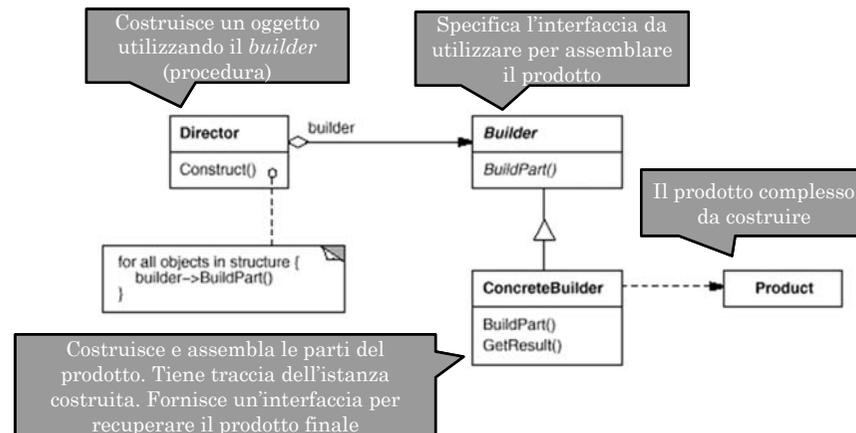
o Applicabilità

- 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



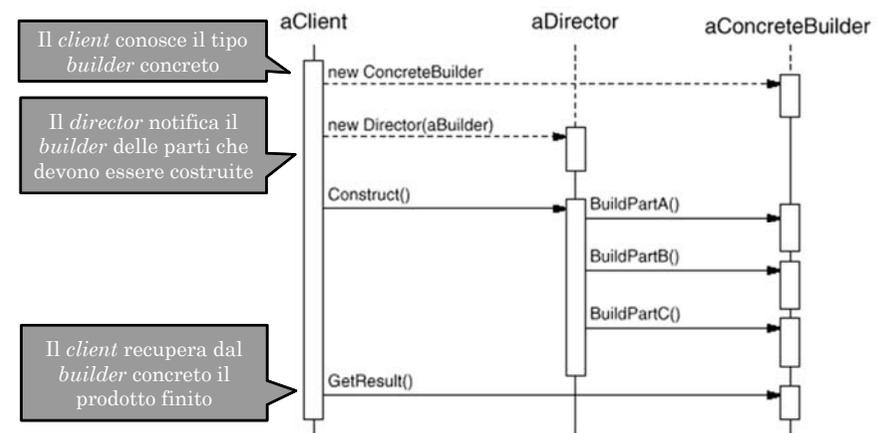
o Struttura



BUILDER



o Struttura



BUILDER



o 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

BUILDER

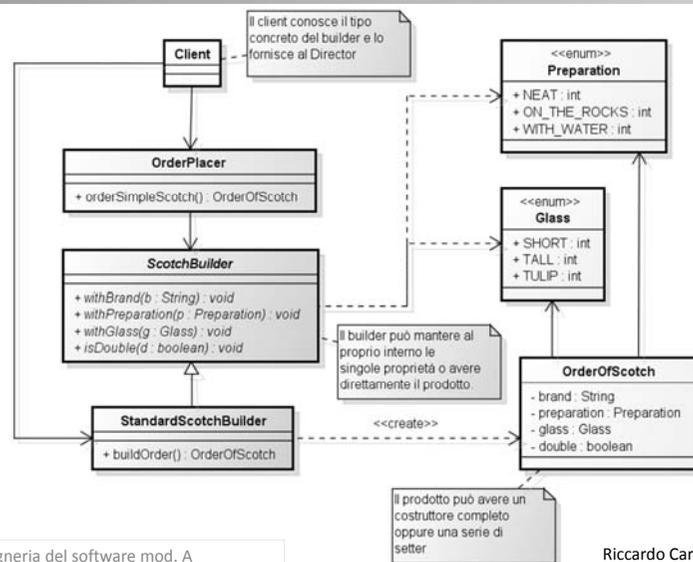


o Esempio

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



BUILDER



o 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

BUILDER



o 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(Builder builder) {
        this.brand = builder._brand;
        // ...
    }
    public static class Builder {
        private String _brand; // Inserisco '_' per diversificare
        private Preparation _preparation;
        // ...
        public ScotchBuilder setBrand(String brand) {
            this._brand = brand;
            return this; // Appending behaviour
        }
        // CONTINUA...
    }
}
```

BUILDER



o 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()
        .setBrand("Brand 1")
        .setPreparation(NEAT)
        // ...
        .build();
} // public class OrderOfScotch
```

- Item 2 di *Effective Java* di Joshua Bloch

BUILDER



o 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

BUILDER



o Implementazione

- Javascript – JQuery

- o 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");
```

BUILDER



o Implementazione

- Scala
 - o In Scala è possibile utilizzare i principi dei linguaggi funzionali
 - o Immutabilità del *builder*
 - o *Type-safe builder*
 - o Assicura staticamente che sono state fornite tutte le informazioni necessarie alla costruzione
 - o Si utilizzano *feature* avanzate del linguaggio
 - o *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



o Scopo

- Fornisce un'interfaccia per creare famiglie di prodotti senza specificare classi concrete

o Motivazione

- Applicazione configurabile con diverse famiglie di componenti
 - o *Toolkit* grafico
- Si definisce una classe astratta *factory* che definisce le interfacce di creazione.
 - o 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



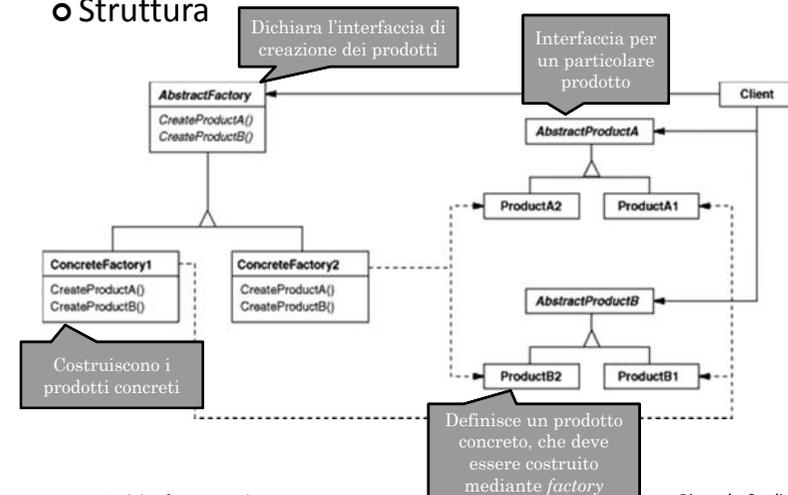
o Applicabilità

- 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



o Struttura



ABSTRACT FACTORY



o 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



o 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).

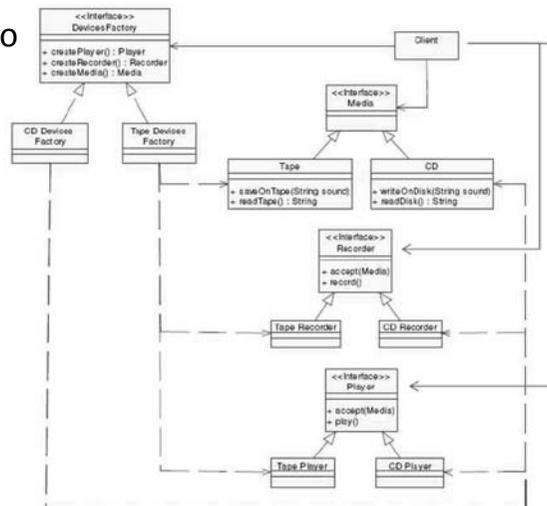
Ogni famiglia è composta da:

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

ABSTRACT FACTORY



o Esempio



ABSTRACT FACTORY



o 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()
    }
}

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

Equivale a
Animal(...)

- *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
- 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 many constructor parameters <http://www.informit.com/articles/article.aspx?p=1216151&seqNum=2>
- Item 3: Enforce the singleton property with a private constructor or an enum type <http://www.informit.com/articles/article.aspx?p=1216151&seqNum=3>
- Type-safe Builder Pattern in Scala <http://blog.rafaelferreira.net/2008/07/type-safe-builder-pattern-in-scala.html>