Il pattern Builder

a cura di **Angelo Furfaro**
da “Design Patterns”, Gamma et al.
“Effective Java”, J. Bloch

Dipartimento di
Ingegneria Informatica, Elettronica, Modellistica e Sistemistica
Università della Calabria, 87036 Rende(CS) - Italy
Email: a.furfaro@unical.it
Web: http://angelo.furfaro.dimes.unical.it

Builder

Classificazione

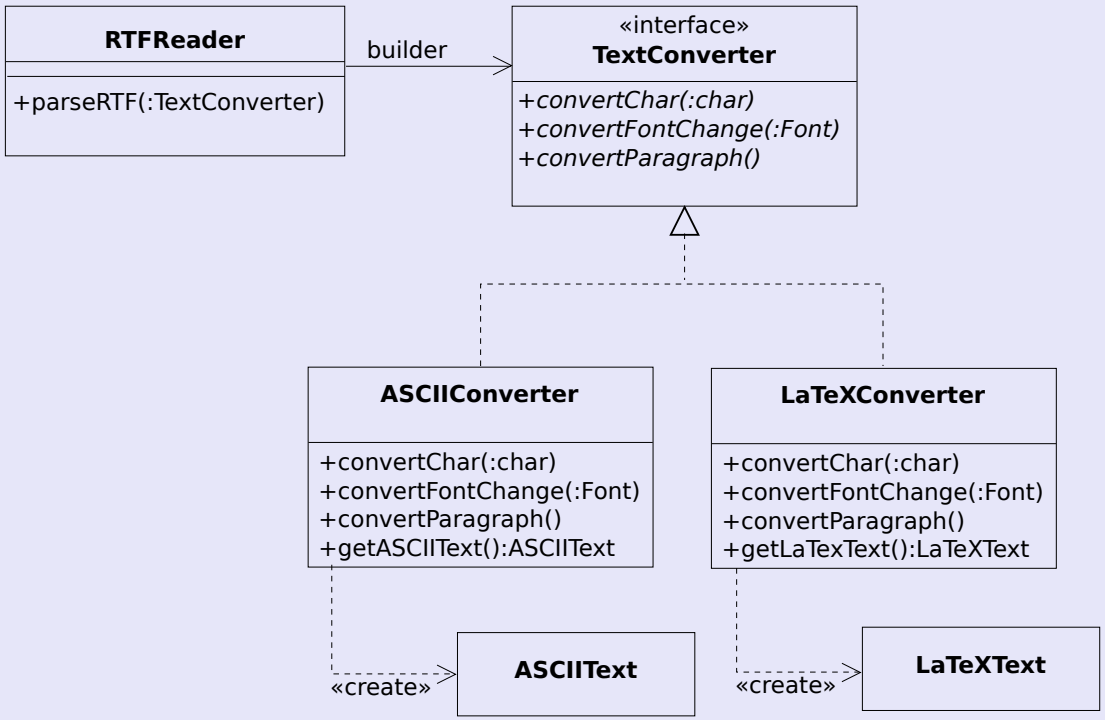
- Scopo: creazionale
- Raggio d'azione: basato su oggetti

Scopo

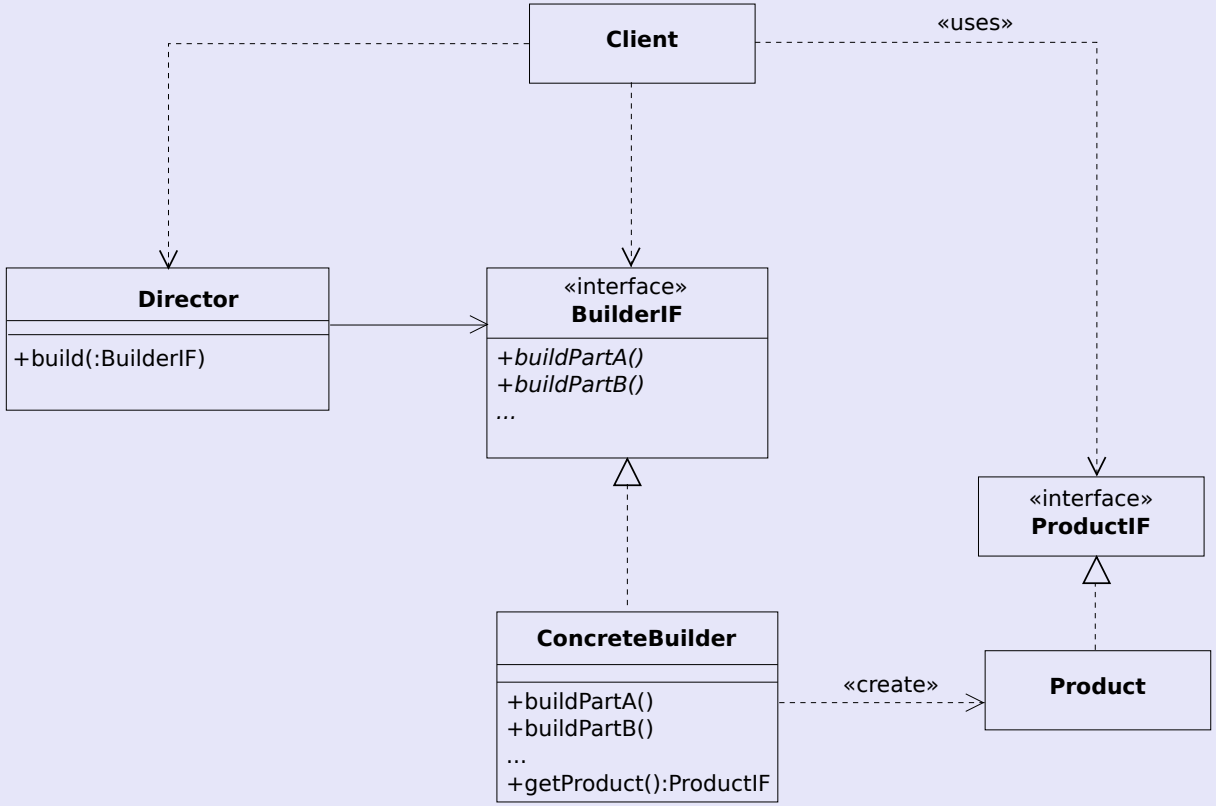
- Separare la costruzione di un oggetto complesso dalla sua rappresentazione, in modo che lo stesso processo di costruzione possa essere utilizzato per creare rappresentazioni diverse.

Motivazione

- Un'applicazione capace di leggere documenti in formato RTF può supportare la conversione in altri formati.
- Ad esempio, conversione in testo ASCII, in formato LaTeX o altro.
- Una soluzione consiste nel configurare la classe `RTFReader` con un oggetto conforme all'interfaccia `TextConverter` in grado di gestire la conversione in un altro formato.
- Il documento nel formato di uscita viene costruito man mano che gli elementi del documento RTF sono analizzati.



Struttura

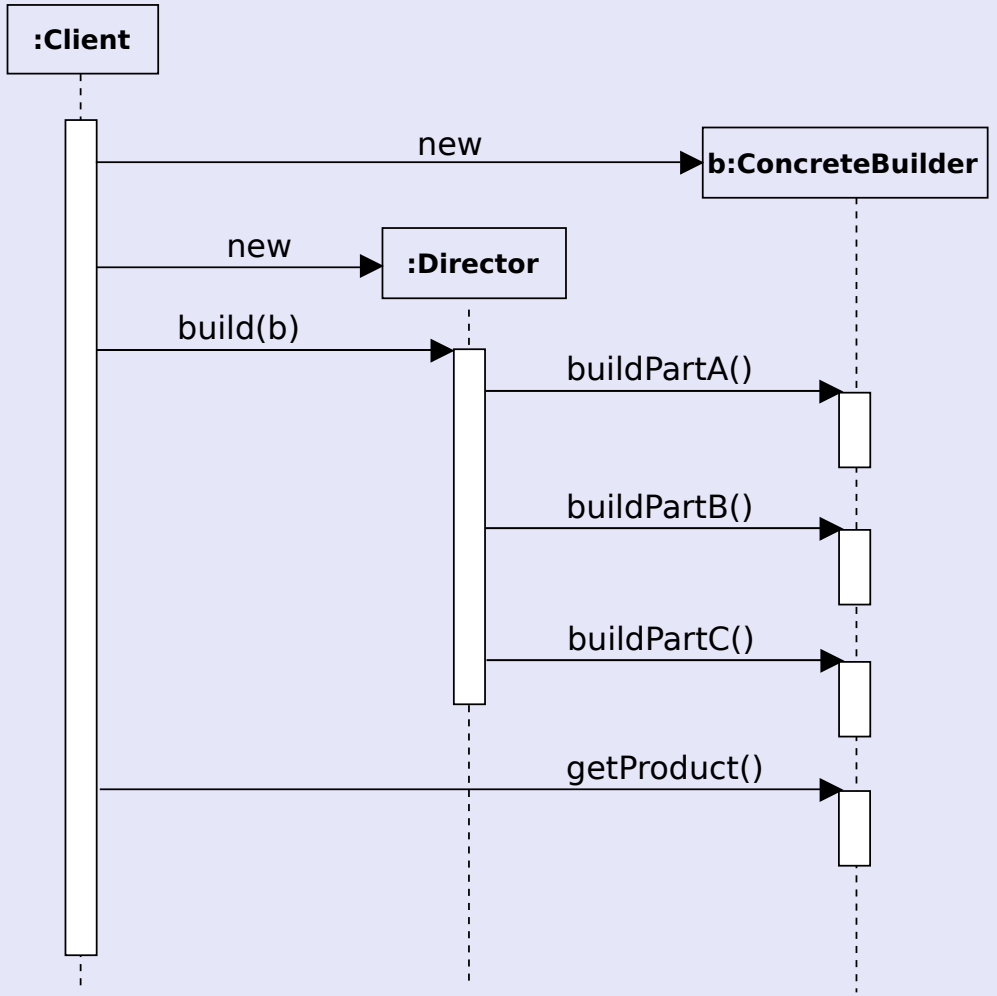


Partecipanti

- **BuilderIF**: specifica l'interfaccia astratta che crea le parti dell'oggetto Product.
- **ConcreteBuilder**: costruisce e assembla le parti del prodotto implementando l'interfaccia Builder; definisce e tiene traccia della rappresentazione che crea.
- **Director**: costruisce un oggetto utilizzando l'interfaccia Builder.
- **Product**: rappresenta l'oggetto complesso e include le classi che definiscono le parti che lo compongono, includendo le interfacce per assemblare le parti nel risultato finale.

Interazioni

- Il client crea l'oggetto `Director` e lo configura tramite l'oggetto `Builder` desiderato.
- Il `Director` informa il `Builder` ogni volta che una parte di `Product` deve essere costruita.
- Il `Builder` riceve e gestisce le richieste dal `Director` e aggiunge le parti al `Product`.
- Il client ottiene dal `Builder` il `Product` creato.



Conseguenze

- *Consente di variare la rappresentazione interna di un prodotto.*
  L'interfaccia del `Builder` consente di nascondere la rappresentazione, la struttura interna dell'oggetto `Product` ed il processo di costruzione. Per modificare la rappresentazione del `Product` basta definire un nuovo tipo di `Builder`.
- *Isola il codice per la costruzione e la rappresentazione.*
  Migliora la modularizzazione incapsulando il modo di costruire e rappresentare un oggetto complesso. Per i client non è necessario conoscere le classi che definiscono la struttura interna del prodotto poiché esse non appaiono nell'interfaccia del `Builder`.
- *Consente un migliore controllo del processo di costruzione.*
  A differenza di altri pattern creazionali che costruiscono il prodotto nella sua interezza, in questo caso il prodotto viene costruito passo dopo passo sotto il controllo del `Director`.

Utilizzare un builder al posto di un costruttore con molti parametri (Effective Java Item 2)

- I metodi factory statici ed i costruttori condividono una limitazione: non scalano bene quando ci sono molti parametri.
- Si consideri una classe che rappresenti le etichette delle informazioni nutrizionali di un prodotto alimentare.
- Ci sono alcuni campi che devono essere necessariamente specificati quali: dimensione di ciascuna porzione, numero di porzioni per contenitore e calorie per porzione.
- Molti altri campi sono opzionali : grassi totali, grassi saturi, colesterolo, sodio e così via.
- Molti prodotti specificano valori diversi da zero solo per alcuni dei campi opzionali.
- Che tipo di costruttore o metodo factory occorre usare ?

Costruttori telescopici

Di solito, la soluzione adottata consiste nel fornire più costruttori.

- Un costruttore solo con i parametri richiesti.
- Un costruttore con un parametro opzionale, uno con due e così via.
- Fino ad un costruttore con tutti i parametri.

```
public class NutritionFacts {
  private final int servingSize;   // (mL) required
  private final int servings;      // (per container) required
  private final int calories;      // optional
  private final int fat;           // (g) optional
  private final int sodium;        // (mg) optional
  private final int carbohydrate; // (g) optional
  public NutritionFacts(int servingSize, int servings) { this(servingSize, servings, 0); }
  public NutritionFacts(int servingSize, int servings, int calories) { this(servingSize, servings, calories, 0); }
  public NutritionFacts(int servingSize, int servings, int calories, int fat) {
    this(servingSize, servings, calories, fat, 0);
  }
  public NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium) {
    this(servingSize, servings, calories, fat, sodium, 0);
  }
  public NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium, int carbohydrate) {
    this.servingSize = servingSize; this.servings = servings; this.calories = calories;
    this.fat = fat; this.sodium = sodium; this.carbohydrate = carbohydrate;
  }
}
```

Costruttori telescopici

- Quando occorre creare un'istanza si sceglie il costruttore con il minor numero di parametri che abbia almeno quelli che si vogliono specificare:
  ```
  NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);
  ```
- Di solito ci sono più parametri di quelli che si vogliono specificare ma si è costretti a fornirli ugualmente.
- La situazione peggiora al crescere del numero di parametri.
- I costruttori telescopici funzionano ma il codice cliente risulta difficile da scrivere e ancor più difficile da leggere.
- Chi leggendo si chiede il significato di ciascun parametro deve calcolare con cura la sua posizione.
- Lunghe sequenze di parametri dello stesso tipo può dare origine a bachi difficili da rilevare a tempo di esecuzione.

Soluzione JavaBeans

- Si introduce un costruttore senza parametri per creare un oggetto.
- I parametri richiesti si specificano invocando gli appositi metodi setter.

```
public class NutritionFacts {
// Parameters initialized to default values (if any)
  private int servingSize = -1; // Required; no default value
  private int servings = -1; // idem
  private int calories = 0;  private int fat = 0;
  private int sodium = 0; private int carbohydrate = 0;
  public NutritionFacts() { }

  public void setServingSize(int val){ servingSize = val; }
  public void setServings(int val){ servings = val; }
  public void setCalories(int val){ calories = val; }
  public void setFat(int val){ fat = val; }
  public void setSodium(int val){ sodium = val; }
  public void setCarbohydrate(int val){ carbohydrate = val; }
}
```

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
```

Soluzione basata su Builder

- Non ha nessuno degli svantaggi del costruttore telescopico.
- Tuttavia, poiché il processo di costruzione è diviso in varie chiamate l'oggetto creato potrebbe trovarsi in uno stato inconsistente.
- La classe non ha modo di verificare la consistenza dei parametri.
- Preclude la possibilità di rendere gli oggetti della classe immutabili.
- La soluzione migliore consiste nel ricorrere all'uso del pattern Builder.
- La classe `NutritionFacts` definisce un builder come inner class statica `Builder`.
- La classe `Builder` riproduce nei suoi campi quelli della classe `NutritionFacts`.
- Il costruttore di `Builder` riceve solo i parametri obbligatori.
- La classe `Builder` introduce un metodo per ciascun parametro opzionale.
- Questi metodi settano il parametro e restituiscono il builder stesso.
- Il metodo `build` crea l'oggetto fornendo atomicamente tutti i parametri per mezzo del builder.

```
public class NutritionFacts {
  private final int servingSize;
  private final int servings;
  private final int calories;
  private final int fat;
  private final int sodium;
  private final int carbohydrate;
  public static class Builder {
    // Required parameters
    private final int servingSize;
    private final int servings;
    // Optional
    private int calories = 0; private int fat = 0;
    private int carbohydrate = 0; private int sodium = 0;

    public Builder(int servingSize, int servings) {
     this.servingSize = servingSize; this.servings = servings;
    }
    public Builder calories(int val) { calories = val; return this;}
    public Builder fat(int val) { fat = val; return this; }
    public Builder carbohydrate(int val) { carbohydrate = val; return this; }
    public Builder sodium(int val) { sodium = val; return this; }
    public NutritionFacts build() { return new NutritionFacts(this); }
  }//Builder
  private NutritionFacts(Builder builder) {
    servingSize = builder.servingSize; servings = builder.servings;
    calories = builder.calories; fat = builder.fat;
    sodium = builder.sodium; carbohydrate = builder.carbohydrate;
  }
}
```

Soluzione basata su Builder

- Si noti che la classe `NutritionFacts` è immutabile.
- Ecco come appare il codice client:

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)
                    .calories(100).sodium(35).carbohydrate(27).build();
```

- Questo codice è molto più facile da leggere e da scrivere.
- Il builder può imporre degli invarianti sui parametri durante l'invocazione di `build()`
- Un altro modo per imporre degli invarianti è quello di raggruppare i parametri coinvolti nello stesso invariante in unico metodo setter del builder.
- Un singolo builder può essere utilizzato per creare più oggetti.
- I parametri del builder possono essere messi a punto tra le varie creazioni per variare gli oggetti creati.
- Il builder può riempire alcuni campi automaticamente come ad esempio un numero seriale che si incrementa automaticamente ad ogni creazione.