Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative

SUPSI

Approfondimento Design Patterns: Builder Pattern

Studente/i
De Santi Massimo

Relatore
-

Correlatore
-

Committente
Giancarlo Corti

Corso di laurea
Ingegneria Informatica

Modulo
Ingegneria e sviluppo del software 2

Anno
2019/2020

Data
19.06.2020

Indice generale

Abstract

Il seguente documento è un approfondimento sul Design Pattern "Builder".

L'obbiettivo è quello di evidenziare benefici e svantaggi dell'adozione di questo pattern. Oltre alla versione originale proposta da E.Gamma, R.Helm, R.Johnson, J.Vlissides [1] nel libro *"Design Patterns: Elements of Reusable Object Oriented Software"* [Gam94], viene analizzata anche una soluzione più semplice proposta da Joshua Bloch nel libro *"Effective Java, 3rd Edition"* [Bloch17].

Viene inoltre fornito un caso pratico, presentando dapprima un problema riscontrato durante l'implementazione di un progetto universitario, e spiegando successivamente come il pattern sia stato applicato per mitigare questo problema. Considerando che il progetto è stato implementato in Java, anche gli esempi proposti nel documento faranno riferimento a questo linguaggio di programmazione.

Come descritto nelle conclusioni, il pattern proposto da Bloch è di facile comprensione e di immediata implementazione. A mio avviso però, si discosta dall'obiettivo originale dei Creational patterns, ovvero: *"Creational patterns ensure that your system is written in terms of interfaces, not implementations"* [Gam94].

Viene quindi proposta una possibile modifica al pattern di Bloch.

Infine viene sottolineata la responsabilità che ogni Software Developer ha nelle decisioni di design, e come la formazione e l'esperienza nel campo sia fondamentale per poter prendere decisioni ponderate.

[1] Gamma, Helm, Johnson e Vlissides vengono spesso citati col nomignolo Gang of Four (GoF)

Introduzione

Questo paper è organizzato nel seguente modo. Il Capitolo 1 *("Motivazione e contesto")* descrive il contesto e il motivo che mi hanno spinto a proporre questo lavoro. Nello specifico viene presentata una breve descrizione del progetto di laboratorio del modulo di "Ingegneria del Software 2". Il Capitolo 2 *("Problema")* introduce un problema che ho dovuto affrontare durante lo sviluppo del progetto. Il Capitolo 3 *("Review del Design Pattern scelto")* approfondisce il Pattern Builder attraverso una selezione della letteratura disponibile. Il Capitolo 4 *("Implementazione e difesa")* propone una possibile soluzione al problema presentato nel Capitolo 2, sfruttando il pattern scelto. Il Capitolo 5 *("Conclusioni")* conclude l'analisi riassumendo i vantaggi e svantaggi della soluzione proposta.

1. Motivazione e contesto

Il modulo di "Ingegneria e Sviluppo del Software 2" fa parte del percorso formativo del Bachelor in Ingegneria Informatica alla SUPSI (Scuola Universitaria Professionale della Svizzera Italiana). Come parziale requisito per la certificazione di questo modulo è richiesta la redazione e consegna di un lavoro scritto individuale nella forma di un documento scientifico.

Questo documento sfrutta il progetto di gruppo sviluppato in laboratorio per mostrare un'applicazione pratica di un design pattern, ed evidenziare i problemi che il pattern scelto risolve. Qui di seguito viene proposta una breve descrizione del progetto.

Descrizione Progetto

Il software che ho realizzato, in collaborazione con due miei compagni, implementa il gioco di carte "Blackjack" (detto anche ventuno[2]) attraverso un'applicazione desktop. Allo stato attuale il software permette di eseguire una partita in modalità giocatore singolo contro computer (Dealer).

L'obiettivo del gioco è quello di ottenere un valore della mano superiore a quella del mazziere (Dealer). Il valore della mano viene calcolato sommando i singoli valori delle carte. Inizialmente il giocatore e il mazziere ricevono due carte. Dopo la distribuzione delle carte, ognuno può chiedere altre carte fino a quando reputa di avere una buona mano oppure supera il valore massimo consentito (Bust). Nel secondo caso, l'avversario vince automaticamente la mano.

Per rendere più interessante il gioco, il giocatore ha a disposizione un valore di fiches pari a 1000. Il giocatore può giocare, round dopo round, fino a quando non consuma tutte le fiches. Ad ogni momento il giocatore può abbandonare la partita ed eventualmente ricominciarne un'altra.

Qui di seguito (Figura 1) uno screenshot dell'applicazione Blackjack.

[2] Il gioco è nato in Francia nel XVII secolo, con il nome di Vingt-et-un (ossia "ventuno"). https://it.wikipedia.org/wiki/Blackjack

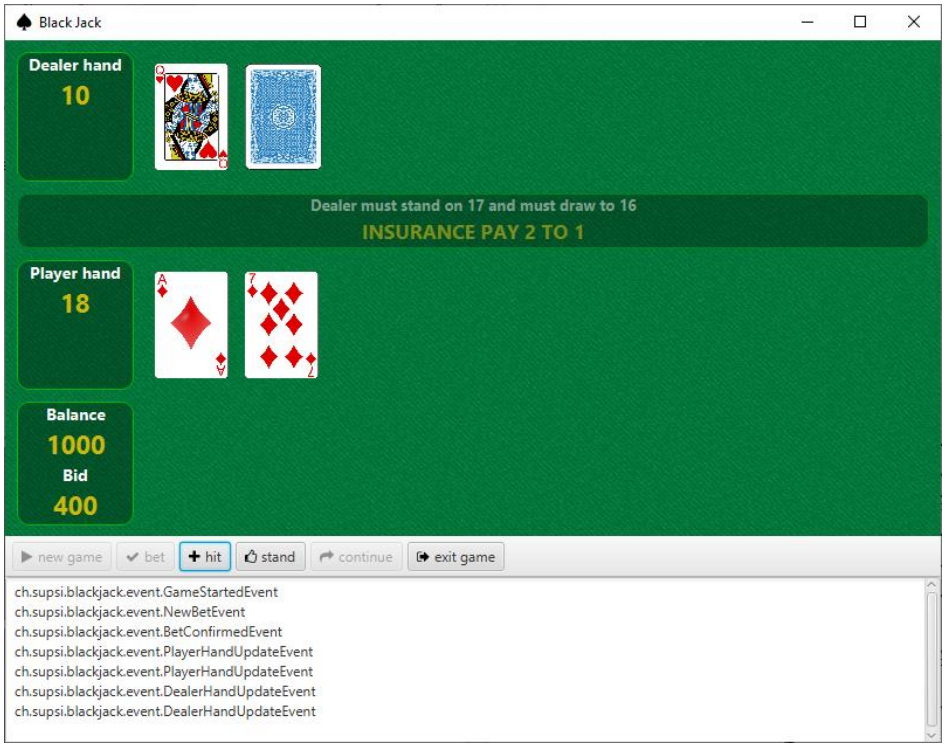

Figura 1 – Screenshot Blackjack (Fase di distribuzione carte)

Architettura

Il software è stato sviluppato in Java 11. A supporto dell'implementazione dell'interfaccia grafica abbiamo usato il framework JavaFX. Due moduli Maven separano la parte di logica (Backend) dalla parte di presentazione (Frontend).

L'architettura dell'applicazione segue il pattern MVC (Model/View/Controller). Lo stato della partita è mantenuto nel Modello (Backend). Nel Frontend invece, viste multiple (*ContentAreaView*, *LogView*, *Menu*View) implementano l'interfaccia *PropertyChangeListener* per poter essere notificate dei cambiamenti di stato del modello. Ogni view ha abbinato una classe controller (*ContentAreaController*, *LogController*, *MenuController*) per gestire l'interazione con l'utente.

2. Problema

Le regole più comuni del gioco del Blackjack prevedono l'uso di uno o più mazzi di carte francesi. Il mazzo francese è costituito da quattro semi (Cuori, Quadri, Fiore, Picche) e 13 valori per ogni seme (Asso, 2, 3, 4, 5, 6, 7, 8, 9, 10, Fante, Donna, Re) per un totale di 52 carte. Inoltre, il mazzo francese prevede anche due Jolly, non usati nel gioco del Black Jack.

L'insieme dei mazzi viene rappresentato nel nostro modello logico dalla classe *DecksContainer*. Questa classe mantiene la lista delle carte disponibili e delle carte usate, e fornisce i metodi per distribuire le carte. Durante la sua creazione, la classe *DeckContainer* sfrutta la classe *Deck* per la costruzione delle *Card*.
Il seguente schema (Figura 2) modella la creazione dell'insieme di mazzi.

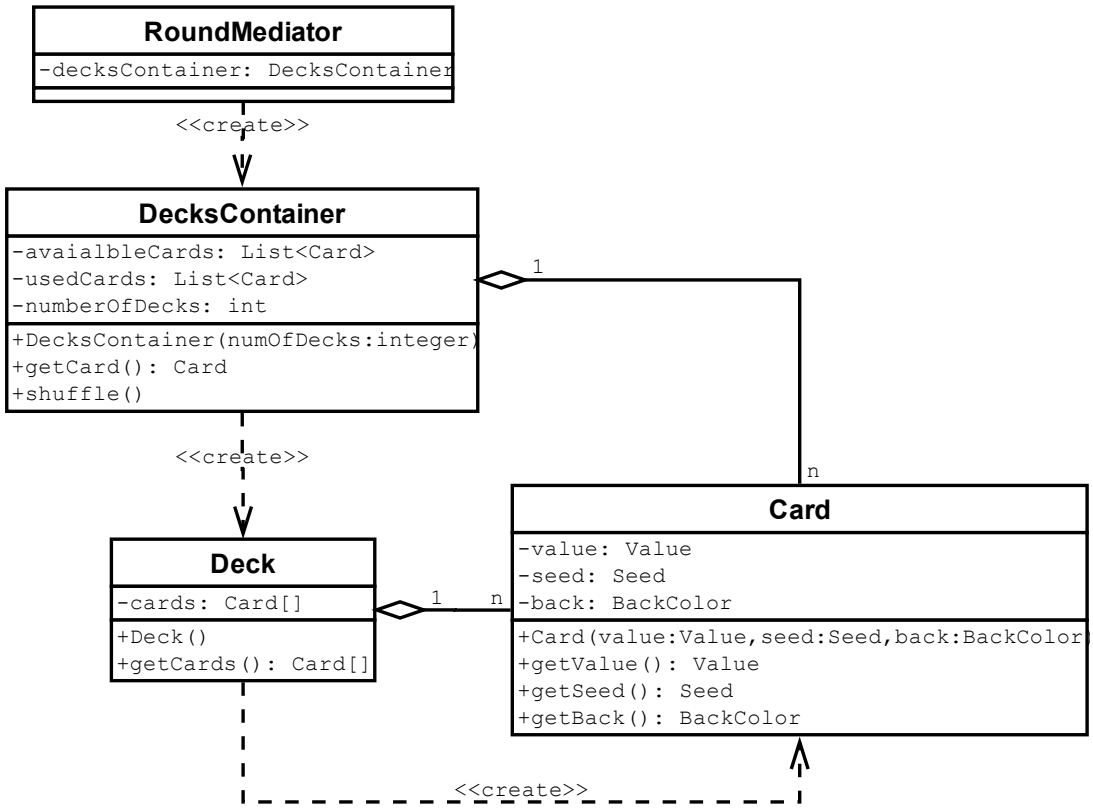

Figura 2 – Composizione insieme di mazzi di carte.

La costruzione del *DecksContainer* avviene attraverso l'invocazione diretta del costruttore dentro la classe *RoundMediator*. Il *DecksContainer* a sua volta crea un insieme di *Deck* che in cascata crea una lista di *Card*. Ogni classe espone uno o più costruttori pubblici.

In generale la presenza di più costruttori pubblici in overload serve per rendere più flessibile la creazione degli oggetti. Alcuni parametri del costruttore possono essere obbligatori mentre altri possono essere opzionali (per esempio definendo dei valori di default). Esistono diverse tecniche per non duplicare lo stesso codice nei vari costruttori. Qui di seguito ne presento alcune.

Telescoping Constructor

Una di queste tecniche è chiamata *telescoping constructor pattern* [Bloch17]. La classe prevede un costruttore con tutti i parametri obbligatori, un secondo costruttore che include un parametro opzionale, un terzo costruttore che include due parametri opzionali e così via.

Il codice seguente mostra come i vari costruttori si richiamino fino ad invocare il costruttore che prevede tutti i parametri.

```
public class Card {
    // ... many other fields could be there
    private final Value value; // mandatory
    private final Seed seed;   // optional
    private final Color back;  // optional

    public Card(Value value) {
        this(value, Seed.H);
```

```
    }

    public Card(Value value, Seed seed) {
        this(value, seed, Color.RED);
    }

    public Card(Value value, Seed seed, Color back) {
        this.value = value;
        this.seed = seed;
        this.back  = back;
    }

    // ...Getters
}

// Client code
Card card1 = new Card(Value.ACE);
Card card2 = new Card(Value.ACE, Seed.H);
Card card3 = new Card(Value.ACE, Seed.H, Color.RED);
```

Purtroppo, quando il numero dei parametri del costruttore inizia a crescere, la manutenzione e leggibilità del codice ne risente. Inoltre, se due o più parametri del costruttore sono dello stesso tipo, la probabilità che l'utilizzatore della classe faccia un errore (es. inversione di due parametri) diventa più alta.

JavaBeans

Un'alternativa al telescoping è l'uso di *JavaBeans pattern* [Bloch17], in cui il client crea l'oggetto tramite un costruttore vuoto e successivamente invoca una serie di *setter methods* per impostare i valori dei campi.

```
// JavaBean
public class Card {
    // … many other fields could be there
    private final Value value; // mandatory
    private final Seed seed;   // optional
    private final Color back;  // optional

    // Parameterless Constructor
    public Card() {}

    // Setters
    public void setValue(Value val)  { value = val; }
    public void setSeed(Seed val)    { seed = val; }
    public void setColor(Color val)  { color = val; }

    // ... Getters
}

// ... Client Code
Card card = new Card();
card.setValue(Value.ACE);
card.setSeed(Seed.H);
card.setColor(Color.RED);
```

Nonostante risolva alcuni dei problemi riscontrati con il telescoping, questa tecnica ha un grosso difetto. Siccome la costruzione è divisa in più chiamate, un *JavaBean* può trovarsi in uno stato inconsistente. Inoltre, il pattern JavaBean preclude la possibilità di definire l'oggetto *Immutable*.

Complex Objects

In altri casi, un oggetto è una composizione di altri sotto oggetti. In questo caso l'algoritmo di creazione dell'oggetto potrebbe non essere banale.

Nel nostro caso l'algoritmo di creazione del mazzo esegue un doppio loop su tutti i possibili valori e segni delle carte. Inoltre, per rendere più piacevole la rappresentazione grafica, un valore random assegna il colore (rosso, blu) al retro della carta.

```
// Deck ctor
public Deck(){
    List<Card> cardList = new ArrayList<>();
    String back = Math.random() < 0.5 ? "red" : "blu";

    for(Seed seed : EnumSet.allOf(Seed.class)) {
        for(Value value : EnumSet.allOf(Value.class)) {
            cardList.add(new Card(seed, value, back));
        }
    }
    cards = new Card[cardList.size()];
    cardList.toArray(cards);
}

// DecksContainer ctor
public DecksContainer(int numberOfDecks)
        throws InvalidDeckContainerSizeException {
    if (numberOfDecks > MIN_NUMBER_OF_DECKS &&
        numberOfDecks < MAX_NUMBER_OF_DECKS){
        this.numberOfDecks=numberOfDecks;
        for (int i=0; i<numberOfDecks; i++){
            Deck deck = new Deck();
            cards.addAll(Arrays.asList(deck.getCards()));
        }
    } else {
        throw new InvalidDeckContainerSizeException()
    }
}


// ... Client Code
DecksContainer decksContainer = new DecksContainer(numberOfDecks);
```

Questa implementazione, anche se funzionante, presenta una serie di problemi:

1. Bassa riusabilità: impossibilità di variare la composizione del mazzo (*Deck*), per esempio usando un set limitato di valori o segni.
2. Complessità: il costruttore della classe *Card* richiede diversi parametri, ma non tutti indispensabili al fine del gioco. Alcuni parametri sono puramente estetici. Inoltre, il costruttore del *DecksContainer* si occupa sia della costruzione dei Deck che della validazione del numero dei mazzi.
3. Testabilità: la casualità del colore assegnato al mazzo rende poco testabile la classe Deck, visto che il colore è impredicibile.

3. Review del Design Pattern scelto

Il pattern scelto per risolvere il problema presentato nel Capitolo 2 è il pattern "*Builder*", classificato da Gamma et al. come *Object Creational Pattern* [Gam94]. Gli altri creational pattern sono: Factory Method, Abstract Factory, Prototype e Singleton.

L'intento del Builder pattern è:
"Separate the construction of a complex object from its representation so that the same construction process can create different representations." [Gam94].

Per chiarire l'intento, la GoF discute il problema di un RTF (Rich Text Format) reader e dei possibili formati di conversione (Figura 3). L'obiettivo è di poter aggiungere nuovi formati di conversione mantenendo invariato il reader.

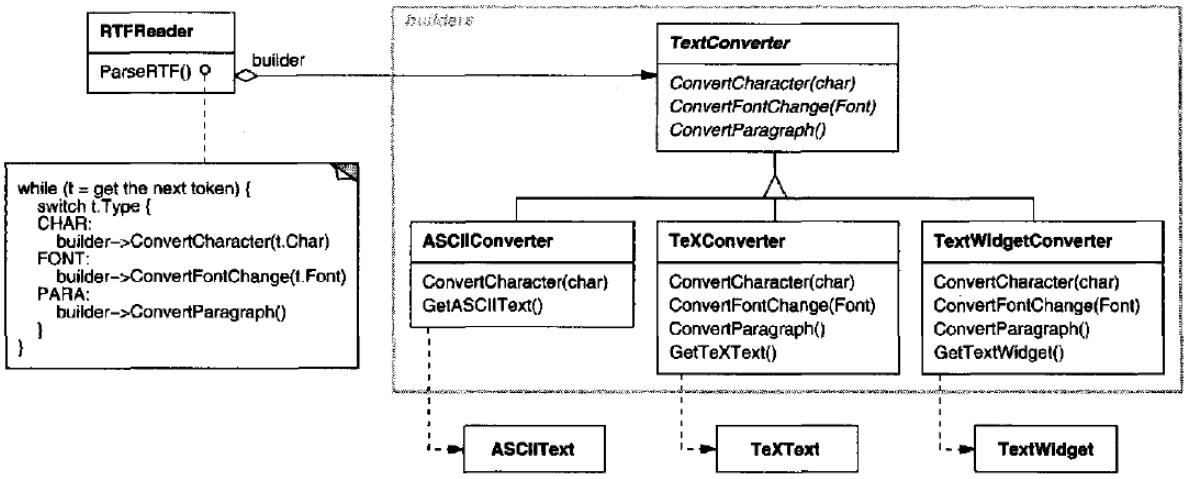

Figura 3 - RTF Reader [Gam94]

All'interno del pattern si possono isolare i seguenti attori:

- Director *(RTFReader)*
  - Costruisce un oggetto attraverso il Builder
- Builder *(TextConverter)*
  - Specifica l'interfaccia per creare parti del prodotto finale
- ConcreteBuilder *(ASCIIConverter, TeXConverter, TextWidgetConverter)*
  - Costruisce e assembla parti del prodotto implementando l'interfaccia Builder
  - Definisce e tiene traccia della rappresentazione che crea
  - Fornisce un'interfaccia per ottenere il prodotto specifico
- Product *(ASCIIText, TeXText e TextWidget)*
  - Rappresenta l'oggetto complesso da costruire.
  - Include le classi delle parti che definiscono il prodotto finale

Il seguente class diagram (Figura 4) evidenzia la relazione tra i vari attori all'interno del pattern.

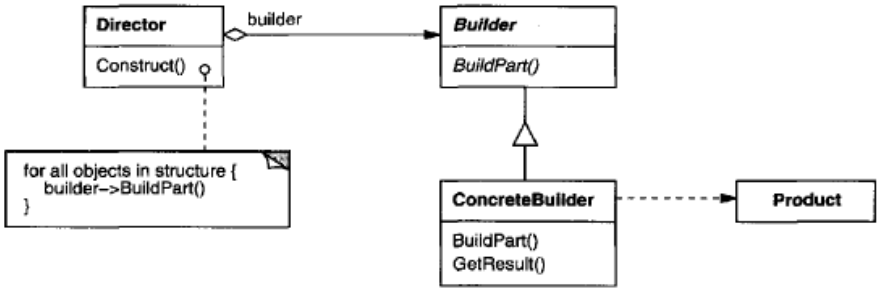

Figura 4 - Struttura del pattern [Gam94]

Il seguente sequence diagram (Figura 5) evidenzia l'interazione tra i vari attori all'interno del pattern.

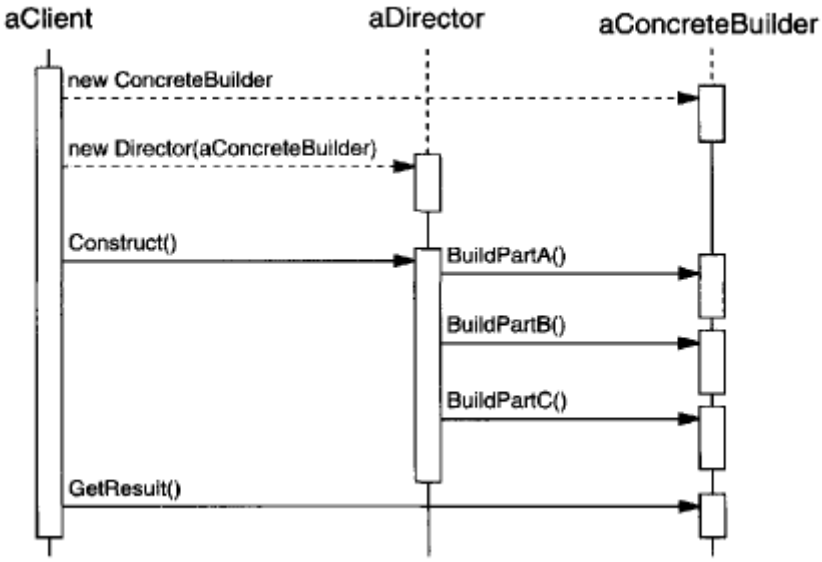

Figura 5 - Collaborazione nel pattern [Gam94]

Nell'esempio precedente, il pattern Builder separa l'algoritmo di interpretazione del testo dalla creazione e rappresentazione del formato di output. Questo permette il riuso dell'algoritmo di parsing della classe *RTFReader* per creare nuovi formati di rappresentazione (estendendo TextConverter).

Siccome il builder si occupa dell'algoritmo di costruzione, prima ancora di restituire l'oggetto, esso ha la possibilità di verificarne la validità e consistenza. In questo modo il codice del costruttore dell'oggetto *Product* (*ASCIIText, TeXText e TextWidget*) può essere sgravato dalla logica di validazione. La classe *Product* diventa più semplice e focalizzata sulla logica di funzionamento.

Un altro vantaggio offerto da questo pattern è la possibilità di costruire un oggetto a step. Per esempio, se volessimo implementare un *Parser* di espressioni matematiche, avremmo la necessità di costruire un oggetto complesso rappresentato da un albero di espressioni. Questo albero verrà costruito passo dopo passo durante il parsing del testo e alla fine, dopo un eventuale controllo di consistenza, reso disponibile. Nel caso di costruzione di oggetti complessi/gerarchici (es. alberi), può essere utile appoggiarsi anche al pattern *Composite* per definire la struttura interna degli oggetti.

Variante di Joshua Bloch

Nel libro *"Effective Java, 3rd Edition",* Joshua Bloch propone un'implementazione più semplice del pattern Builder. Nella variante di Bloch, l'obiettivo del pattern è ridimensionato ma non per questo meno interessante.

Anche in questo caso, il client non usa più direttamente il costruttore dell'oggetto desiderato (*Product*), ma si appoggia a una classe separata (*Builder*) che espone una serie di metodi *setter-like* per ogni parametro opzionale del *Product*. Per rendere il codice ancora più semplice, la classe *Builder* viene definita come *inner static class* della classe *Product* e il costruttore della classe *Product* viene reso inaccessibile al cliente. Il seguente sequence diagram (Figura 6) mostra l'interazione del client col builder.

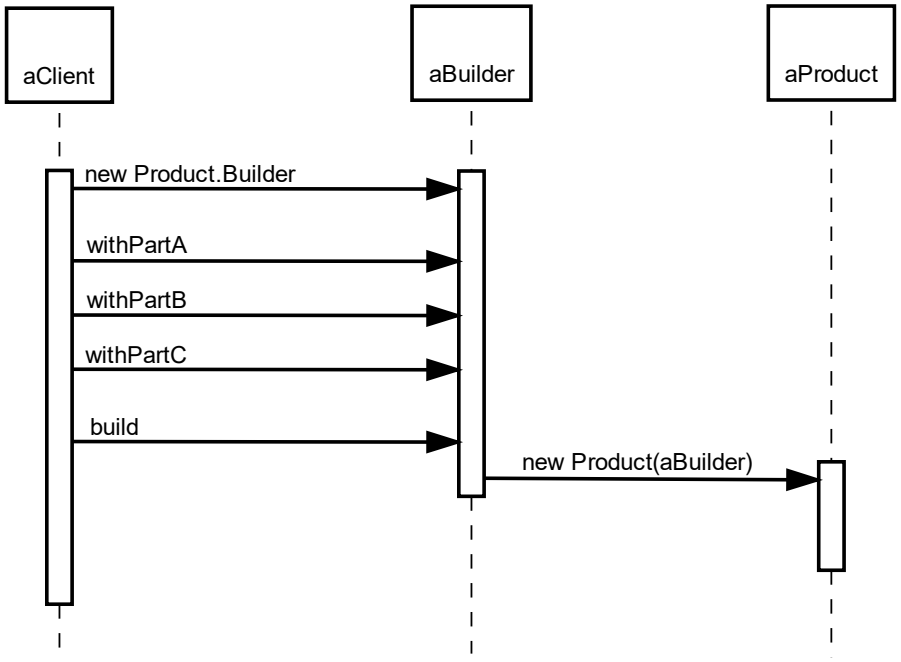

Figura 6 - Bloch Builder Pattern

Nel prossimo capitolo verrà illustrata un'applicazione del pattern di Bloch e di come il client interagisca col *Builder* per poter ottenere un'istanza del *Product*.

4. Implementazione e difesa

Per risolvere i problemi presentati nel Capitolo 2, abbiamo applicato il pattern Builder proposto da Bloch alle seguenti classi: *Card, Deck, DecksContainer.* Ognuna di queste classi ha ora un Builder, implementato come *static inner class,* della classe che va a produrre. Il seguente class diagram (Figura 7) evidenzia la nuova relazione tra le classi.

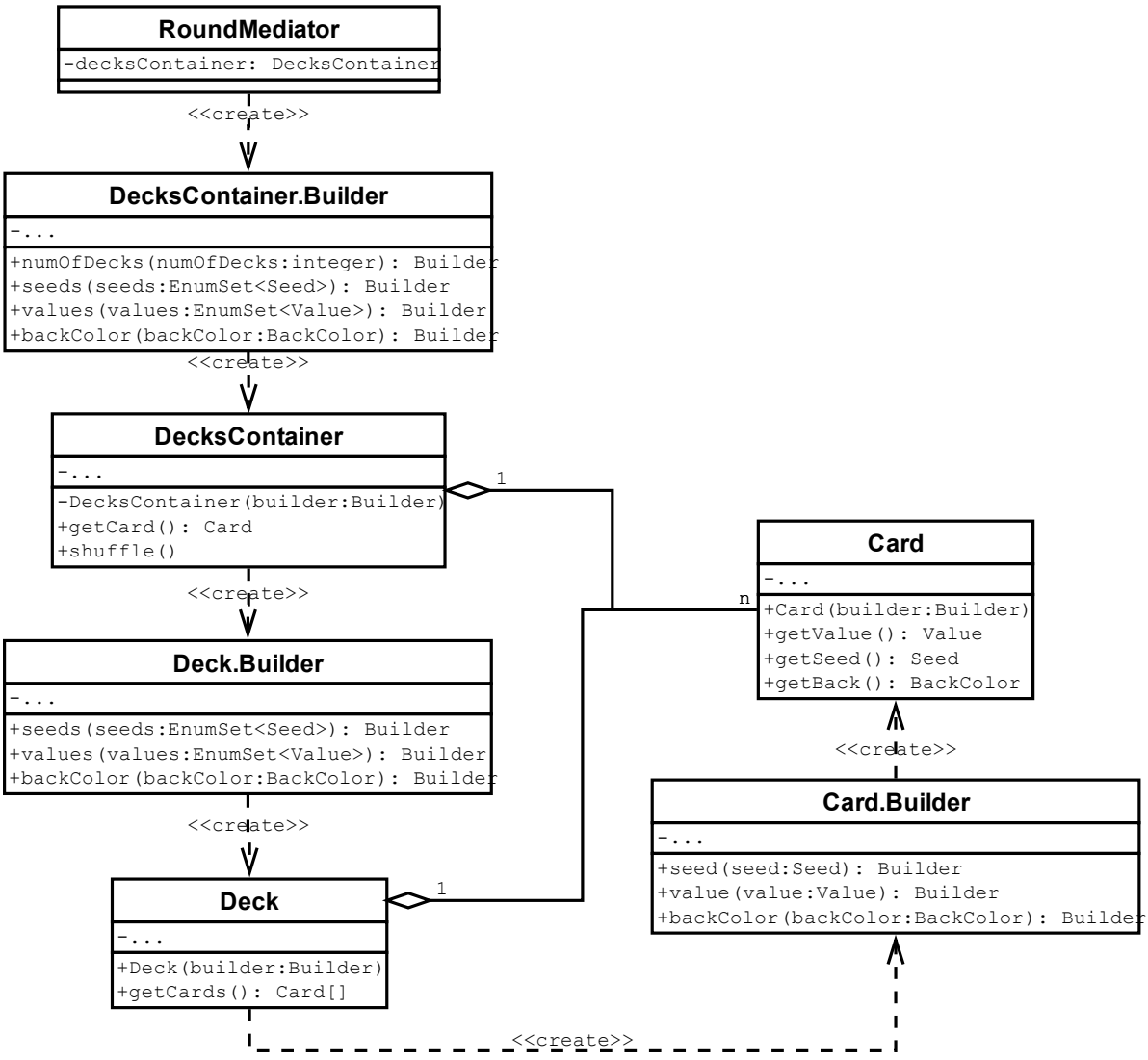

Figura 7 - Refactoring with Builder

Qui di seguito viene proposta l'implementazione della classe *Card* e un esempio di codice richiesto per istanziarla.

```
// Builder Pattern
public class Card {
    private final Value value; // mandatory
    private final Seed seed;   // optional
    private final BackColor back; // optional

    // Constructor is now accessible only from the Builder
    private Card(Builder builder) {
        value = builder.value;
        seed  = builder.seed;
        back  = builder.back;
    }

    // Builder as inner static class
    public static class Builder {
        // Required parameters
        private final Value value;

        // Optional parameters - initialized to default values
```

```
        private Seed seed = Seed.H;
        private BackColor back = BackColor.RED;

        // Constructor takes required parameters as input
        public Builder(Value value) {
            this.value = value;
        }

        // Optional parameters are set through builder methods
        public Builder seed(Seed val) {
            seed = val;
            return this;
        }

        // Optional parameters are set through builder methods
        public Builder back(BackColor val) {
            back = val;
            return this;
        }

        // build method finally returns the Card object
        public Card build() {
            return new Card(this);
        }
    }
}

// Client Code with fluent syntax
Card card = new Card.Builder(Value.ACE)
        .seed(Seed.H)
        .back(BackColor.RED)
        .build();
```

La nuova struttura, pur introducendo nuove classi, ha il vantaggio di rendere più flessibile la composizione del mazzo di carte. Questo refactoring agevola la riusabilità. Ora le classi *Card, Deck e DecksContainer* possono essere sfruttate anche in un altro tipo di gioco, utilizzando per esempio un subset di Valori o Segni.

Inoltre, questa implementazione sfrutta la *fluent interface syntax*[3] che rende di facile lettura il codice client.

5. Conclusioni

Come per altri pattern presenti nel libro della GoF, il pattern Builder aggiunge un certo overhead nella struttura del sistema. Nuove classi vengono introdotte per la creazione degli oggetti, costringendo il client a scrivere qualche riga di codice in più. Questo overhead è però ripagato da una maggiore flessibilità nel riutilizzo dei componenti.

La variante di Bloch è interessante per la sua semplicità di implementazione e di utilizzo, visto che il numero di classi a supporto è decisamente minore rispetto al pattern originale. Rispetto all'uso diretto dei costruttori, l'introduzione di questo semplice pattern rende più leggibile e robusto il codice client.

[3] https://en.wikipedia.org/wiki/Fluent_interface

Per quanto interessante e di immediato utilizzo, bisogna far notare che esso perde di vista quello che in generale si prefiggono i Creational Patterns:
"Creational patterns ensure that your system is written in terms of interfaces, not implementations" [Gam94]

Per ovviare a questo problema, avremmo bisogno di introdurre un'interfaccia *Builder* e un'interfaccia *Product*. Implementazioni concrete di *Builder* potrebbero quindi costruire delle specializzazioni dell'interfaccia *Product,* garantendo il principio di design Object Oriented: *"Program to an interface, not to an implementation"* [Gam94]

Come possiamo vedere, è possibile trovare diverse soluzioni allo stesso problema. Ogni Software Developer deve saper bilanciare complessità e flessibilità in base alla situazione che deve affrontare.

Randy Stafford, nel capitolo 12 del libro "*97 Things Every Software Architect Should Know*", affronta questa tematica dal punto di vista dei Software Architect riprendendo un concetto già espresso nel 1991 da Eberhardt Rechtin.

Ogni Sofware Architect deve maturare un buon *"contextual sense":*
[A] better expression than 'common sense' is contextual sense — a knowledge of what is reasonable within a given context. [Rechtin90]

Infatti, non esiste una soluzione universale applicabile a tutti i problemi:
"There Is No One-Size-Fits-All Solution" [Monson09].

Questo senso va maturato nell'arco degli anni, formandosi, lavorando su diversi progetti e facendo tesoro delle esperienze passate. A mio avviso, possiamo estendere questo suggerimento anche al design dei singoli componenti. Va quindi valutato, progetto per progetto, se i vantaggi portati dall'introduzione del pattern valgono il costo di questa complessità.

Come suggerisce Stafford, la conoscenza piú importante dei pattern è sapere quando applicarli e quando no:
"The most important knowledge of software patterns is the knowledge of when to apply them and when not to apply them" [Monson09]

Riconoscimenti

Vorrei ringraziare i miei compagni Aleksandar Stojkovski e Attilio Baldini per l'esperienza di gruppo che abbiamo vissuto durante lo sviluppo di questo progetto.

Un sincero ringraziamento va anche a Giancarlo Corti per la disponibilità e la passione che mette nel suo lavoro di docente.

Bibliografia

[Gam94] Design Patterns: Elements of Reusable Object Oriented Software
Autori: E.Gamma, R.Helm,R.Johnson, J.Vlissides
Publisher: Addison-Wesley Professional Computing Series
Pubblicazione: ottobre 1994
ISBN: 0-201-63361-2

[Bloch17] Effective Java, 3rd Edition
Autore: Joshua Bloch
Publisher: Addison-Wesley Professional
Pubblicazione: dicembre 2017
ISBN: 9780134686097

[Monson09] 97 Things Every Software Architect Should Know
Autore: Richard Monson-Haefel
Publisher: O'Reilly Media, Inc.
Pubblicazione: febbraio 2009
ISBN: 9780596522698

[Rechtin90] Systems Architecting: Creating & Building Complex Systems
Autore: Eberhardt Rechtin
Publisher: Prentice Hall
Pubblicazione: dicembre 1990
ISBN: 0138803455