Laboratorul 10: Design patterns - Command și Builder

Video introductiv: link [https://www.youtube.com/watch?v=F2HMN4mvVYY]

Obiective

Scopul acestui laborator este familiarizarea cu folosirea design pattern-ului comportamental *Command* și a design pattern-ului creațional *Builder*.

Introducere

În laboratoarele precedent am prezentat pattern-uri ce vă ajută în realizarea unei arhitecturi mai decuplate, modulare și extensibile a aplicațiilor:

- pattern-uri creaționale ce decuplează logica creării obiectelor: Singleton, Factory
- pattern-uri comportamentale ce decuplează comunicarea dintre componente: Visitor, Observer, Strategy

În acest laborator vom prezenta *Command* și *Builder*, un pattern comportamental care decuplează obiectele care execută anumite acțiuni de obiectele care le invocă și un pattern creațional care oferă o soluție mai flexibilă pentru a crea obiecte complexe și este adesea însoțită de un model de interfață fluentă.

Command

Design pattern-ul *Command* incapsulează un apel cu tot cu parametri într-o clasă cu interfată generică. Acesta este *Behavioral* pentru că modifică interacțiunea dintre componente, mai exact felul în care se efectuează apelurile.

Acest pattern este recomandat în următoarele cazuri:

- pentru a ușura crearea de structuri de delegare, de callback, de apelare intarziată
- pentru a reține lista de comenzi efectuate asupra obiectelor
- accounting
- liste de Undo, Rollback pentru tranzacții-suport pentru operații reversibile (*undoable operations*)

Exemple de utilizare:

- sisteme de logging, accounting pentru tranzacții
- sisteme de undo (ex. editare imagini)
- mecanism ordonat pentru delegare, apel întârziat, callback

Funcționare și necesitate

În esentă, Command pattern (așa cum v-ați obișnuit și lucrând cu celelate Pattern-uri pe larg cunoscute) presupune încapsularea unei informații referitoare la acțiuni/comenzi folosind un wrapper pentru a "ține minte această informație" și pentru a o folosi ulterior. Astfel, un astfel de wrapper va deține informații referitoare la tipul acțiunii respective (în general un asemenea wrapper va expunde o metodă execute(), care va descrie comportamentul pentru acțiunea respectivă).

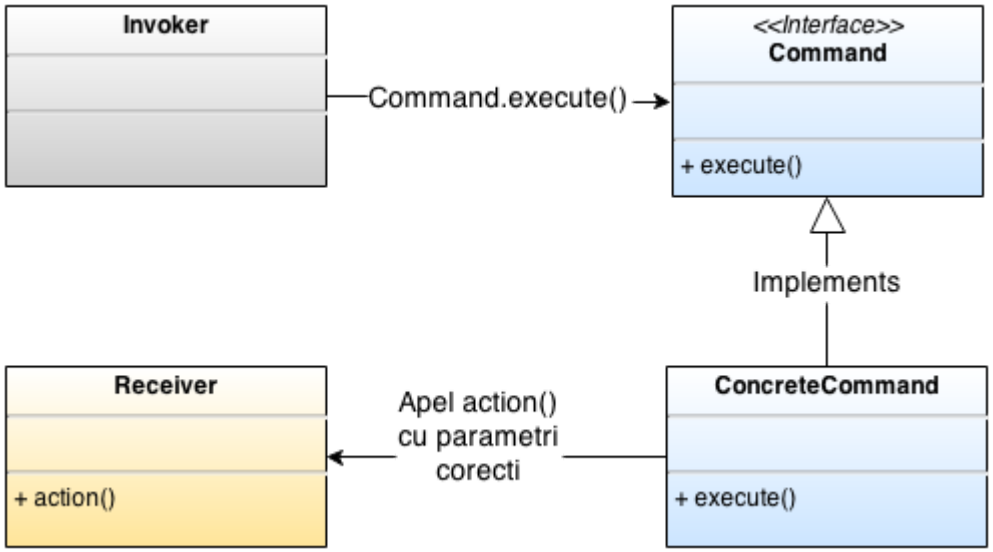
Mai mult încă, când vorbim de Command Pattern, în terminologia OOP o să întâlniți deseori și noțiunea de *Invoker*. Invoker-ul este un middleware ca funcționalitate care realizează managementul comenzilor. Practic, un *Client*, care vrea să facă anumite acțiune, va instanția clase care implementează o interfață *Command*. Ar fi incomod ca, în cazul în care aceste instanțieri de comenzi provin din mai multe locuri, acest management de comenzi să se facă local, în fiecare parte (din rațiuni de economie, nu vrem să duplicăm cod). Invoker-ul apare ca o necesitate de a centraliza acest proces și de a realiza intern management-ul comenzilor (le ține într-o listă, ține cont de eventuale dependențe între ele, totul în funcție de context).

Un client (generic spus, un loc de unde se lansează comenzi) instanțiază comenzile și le pasează Invoker-ului. Din acest motiv Invoker-ul este un middleware între client și receiver, fiindcă acesta va apela execute pe fiecare Command, în funcție de logica să internă.

Recomandare: La Referinte aveti un link catre un post pe StackOverflow, pentru a intelege mai bine de ce aveti nevoie de Pattern-ul Command si de ce nu lansati comenzi pur si simplu.

Structura

Ideea principală este de a crea un obiect de tip *Command* care va reține parametrii pentru comandă. Comandantul reține o referință la comandă și nu la componenta comandată. Comanda propriu-zisă este anunțată obiectului *Command* (de către comandant) prin execuția unei metode specificate asupra lui. Obiectul *Command* este apoi responsabil de trimiterea (*dispatch*) comenzii către obiectele care o îndeplinesc (*comandați*).



Tipuri de componente (**roluri**):

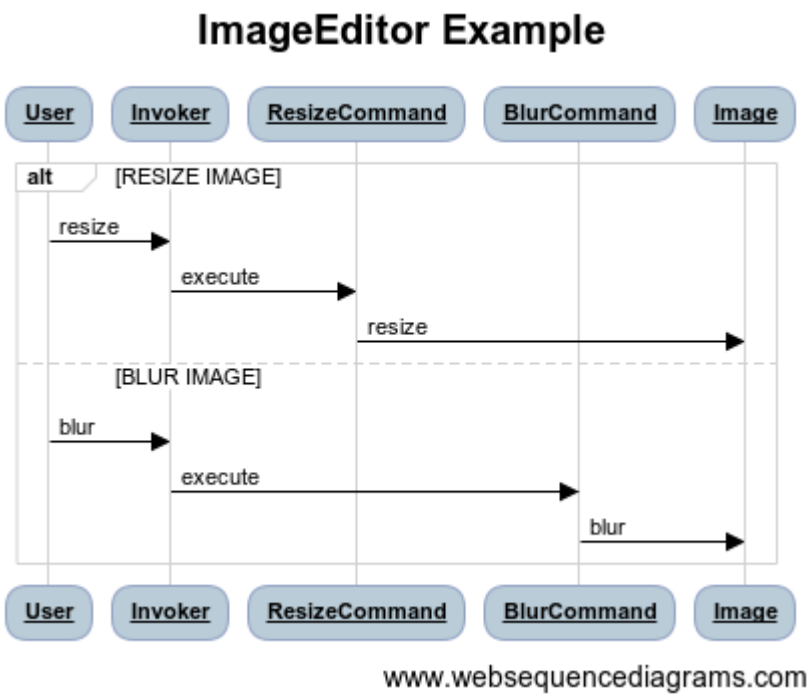
- **Invoker** - comandantul
  - apelează acțiuni pe comenzi (invocă metode oferite de obiectele de tip *Command*)
  - poate menține, dacă e cazul, o *listă a tutoror comenzilor aplicate* pe obiectul (obiectele) comandate. Este necesară reținerea acestei liste de comenzi atunci când implementăm un comportament de undo/redo al comenzilor.
  - primește clase *Command* pe care să le invoce
- **Receiver** - comandatul
  - este clasa asupra căreia se face apelul
  - conține implementarea efectivă a ceea ce se dorește executat
- **Command** - obiectele pentru reprezentarea comenzilor implementează această interfață/o extind dacă este clasă abstractă
  - *concrete command* - ne referim la implementări/subclasele acesteia
  - de obicei conțin metode cu nume sugestiv pentru executarea acțiunii comenzii (e.g. `execute()`). Implementările acestora conțin apelul către clasa *Receiver*.
  - în cazul implementării unor acțiuni *undoable* adăugăm metode pentru `undo` și/sau `redo`.
  - țin referințe către comandați (receivers) pentru a aplica/invoca acțiunea ce reprezintă acea comandă

În Java, se pot folosi atât interfețe cât și clase abstracte, pentru Command, depinzând de situație (e.g. clasă abstractă dacă știm sigur ca obiectele de tip Command nu mai au nevoie să extindă și alte clase).

În prima diagramă de mai jos, comandantul este clasa *Invoker* care conține o referință la o instanță (command) a clasei (Command). *Invoker* va apela metoda abstractă `execute()` pentru a cere îndeplinirea comenzii. *ConcreteCommand* reprezintă o implementare a interfeței *Command*, iar în metoda `execute()` va apela metoda din *Receiver* corespunzătoare acelei acțiuni/comenzi.

Exemplu

Prima diagramă de secvență prezintă apelurile în cadrul unei aplicație de editare a imaginilor, ce este structurată folosind pattern-ul Command. În cadrul acesteia, Receiver-ul este *Image*, iar comenzile *BlurCommand* și *ResizeCommand* modifică starea acesteia. Structurând aplicația în felul acesta, este foarte ușor de implementat un mecanism de undo/redo, fiind suficient să menținem în Invoker o listă cu obiectele de tip *Command* aplicate imaginii.



Pornind de la această diagramă, putem realiza o implementare a pattern-ului Command. Vom construi clasa *Image*, care va juca rolul Receiver-ului. Acesteia îi vom asocia un câmp *blurStrength*, care ne va oferi informații despre intensitatea filtrului de blur, și încă două câmpuri *length* și *width* care ne vor spune ce dimensiune are imaginea. Valorile acestor câmpuri vor fi alterate în urma aplicării comezilor de *blur* și *resize*.

```
public class Image {
    private int blurStrength;
    private int length;
    private int width;

    public Image(int length, int width) {
        this.length = length;
        this.width = width;
    }

    public int getBlurStrength() {
        return blurStrength;
    }

    public void setBlurStrength(int blurStrength) {
        this.blurStrength = blurStrength;
    }

    public int getLength() {
        return length;
    }
```

```
    public void setLength(int length) {
        this.length = length;
    }

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }
}
```

Command va fi o interfață, căreia pe langă metoda de *execute()* îi vom asocia și o metodă de *undo()*.

```
interface Command {
    void execute();

    void undo();
}
```

BlurCommand și *ResizeCommand* vor implementa interfața *Command*. La apelul *execute()*, *BlurCommand* va modifica câmpul *blurStrength* din clasa *Image*, iar *ResizeCommand* va modifica dimensiunea, lungimea și înălțimea imaginii. Întrucât ne dorim să implementăm un mecanism de undo, este nevoie să reținem valoare anterioară.

```
// Concrete command

public class BlurCommand implements Command {
    private final Image image;
    private int previousBlurStrength;
    private int nextBlurStrength;

    public BlurCommand(Image image, int blurStrength) {
        this.image = image;
        this.nextBlurStrength = blurStrength;
    }

    @Override
    public void execute() {
        previousBlurStrength = image.getBlurStrength();
        image.setBlurStrength(nextBlurStrength);
    }

    @Override
    public void undo() {
        nextBlurStrength = previousBlurStrength;
        previousBlurStrength = image.getBlurStrength();
        image.setBlurStrength(nextBlurStrength);
    }
}

public class ResizeCommand implements Command {
```

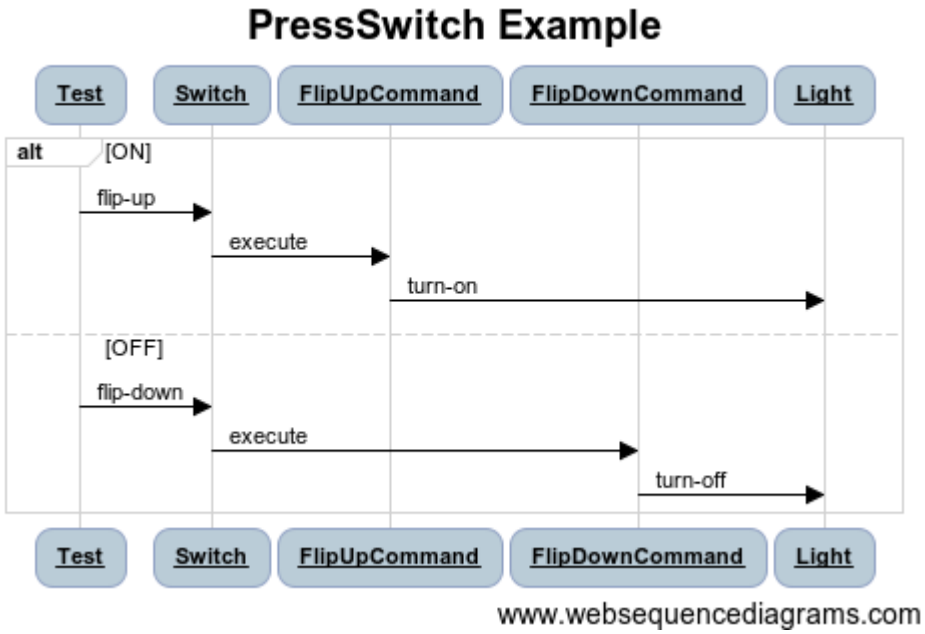
```
    private final Image image;
    private int previousWidth;
    private int previousLength;
    private int nextWidth;
    private int nextLength;

    public ResizeCommand(Image image, int width, int length) {
        this.image = image;
        nextWidth = width;
        nextLength = length;
    }

    @Override
    public void execute() {
        previousWidth = image.getWidth();
        image.setWidth(nextWidth);

        previousLength = image.getLength();
        image.setLength(nextLength);
    }

    @Override
    public void undo() {
        nextWidth = previousWidth;
        previousWidth = image.getWidth();
        image.setWidth(nextWidth);

        nextLength = previousLength;
        previousLength = image.getLength();
        image.setLength(nextLength);
    }
}
```

Invoker-ul este clasa *Editor*. Aceasta va avea două metode, edit si undo, care vor fi apelate de către user. În plus, vom păstra în această clasă și o listă a comenzilor aplicate pe imagine. Cu acestă listă vom putea implementăm un comportament de undo.
Metoda edit va primi ca parametru o referiță la o instanță command, apoi va fi inițiată o cerere de către *Editor* prin apelarea metodei *execute()*, cerând astfel execuția comenzii.

```
// Invoker

public class Editor {
    // LinkedList este folosit ca stivă în Java
    private LinkedList<Command> history = new LinkedList<>(); // păstrează comenzile aplicate pe imagine

    public void edit(Command command) {
        history.push(command);
        command.execute();
    }

    public void undo() {
        if (history.isEmpty()) return;

        Command command = history.pop();
        if (command != null) {
```

```
            command.undo();
        }
    }
}
```

Pe Wikipedia [http://en.wikipedia.org/wiki/Command_pattern] puteți analiza exemplul PressSwitch. Flow-ul pentru acesta este ilustrat în diagrama de mai jos



Builder

Design pattern-ul Builder este un design pattern creațional, cu alte cuvinte, este utilizat pentru a crea și configura obiecte. Un builder este utilizat în mod obișnuit pentru eliminarea supraincarcarilor de constructori multipli și oferă o soluție mai flexibilă la crearea obiectelor complexe.

Problema

În Programarea Orientată pe Obiecte, cel mai adesea avem clase care dețin unele date pe care le setăm și le accesăm ulterior. Crearea instanțelor unor astfel de clase ar putea fi uneori cam laborioasă. Să luăm în considerare următoarea clasă de Pizza

Pizza.java

```
public class Pizza {
    private String pizzaSize;
    private int cheeseCount;
    private int pepperoniCount;
```

```
    private int hamCount;

    // constructor, getters, setters
}
```

O clasă foarte simplă la prima vedere însă lucrurile se complică pe masură ce vom crea acest obiect. Oricare pizza va avea o dimensiune, cu toate acestea, atunci când vine vorba de topping-uri, unele sau toate pot fi prezente sau deloc, prin urmare, unele dintre proprietățile clasei noastre sunt opționale, iar altele sunt obligatorii.

Supraîncărcarea constructorilor

Crearea unei instanțe noi *new Pizza("small", 1, 0, 0)* de fiecare dată când vreau să obțin pur și simplu un obiect pizza cu brânză și nimic altceva nu mi se pare o idee bună. Și aici vine prima soluție comună - supraîncărcarea constructorului.

Pizza.java

```
public class Pizza {
    private String pizzaSize; // mandatory
    private int cheeseCount; // optional
    private int pepperoniCount; // optional
    private int hamCount; // optional

    public Pizza(String pizzaSize) {
        this(pizzaSize, 0, 0, 0);
    }

    public Pizza(String pizzaSize, int cheeseCount) {
        this(pizzaSize, cheeseCount, 0, 0);
    }

    public Pizza(String pizzaSize, int cheeseCount, int pepperoniCount) {
        this(pizzaSize, cheeseCount, pepperoniCount, 0);
    }

    public Pizza(String pizzaSize, int cheeseCount, int pepperoniCount, int hamCount) {
        this.pizzaSize = pizzaSize;
        this.cheeseCount = cheeseCount;
        this.pepperoniCount = pepperoniCount;
        this.hamCount = hamCount;
    }

    // getters
}
```

Cu toate acestea, am rezolvat problema doar parțial. Nu putem, de exemplu, să creăm o pizza cu brânză și șuncă, dar fără pepperoni ca aceasta *new Pizza("small", 1, 1)*, deoarece al treilea argument al constructorului este pepperoni. Și aici vine a doua soluție comună - și mai multă supraîncărcare de constructori.

Pizza.java

```
public class Pizza {
    private String pizzaSize; // mandatory
```

```
    private String crust; // mandatory
    private int cheeseCount; // optional
    private int pepperoniCount; // optional
    private int hamCount; // optional
    private int mushroomsCount; // optional

    public Pizza(String pizzaSize, String crust) {
        this(pizzaSize, crust, 0, 0, 0, 0);
    }

    public Pizza(String pizzaSize, String crust, int cheeseCount) {
        this(pizzaSize, crust, cheeseCount, 0, 0, 0);
    }

    public Pizza(String pizzaSize, String crust, int cheeseCount, int pepperoniCount) {
        this(pizzaSize, crust, cheeseCount, pepperoniCount, 0, 0);
    }

    public Pizza(String pizzaSize, String crust, int cheeseCount, int pepperoniCount, int hamCount) {
        this(pizzaSize, crust, cheeseCount, pepperoniCount, hamCount, 0);
    }

    public Pizza(String pizzaSize, String crust, int cheeseCount, int pepperoniCount, int hamCount, int mushroomsCount) {
        this.pizzaSize = pizzaSize;
        this.crust = crust;
        this.cheeseCount = cheeseCount;
        this.pepperoniCount = pepperoniCount;
        this.hamCount = hamCount;
        this.mushroomsCount = mushroomsCount;
    }

    // getters
}
```

Gândiți-vă ce se va întampla dacă se schimbă ordinea parametrilor. Acest lucru minor va strica funcționalitatea completă a creative unei instanțe de Pizza.

În concluzie, modelul de constructori supraincarcati funcționează, dar este greu de menținut dacă se schimbă funcționalitatea și introducem noi parametri, numărul constructorilor va crește, de asemenea.

Folosirea de getters și setters

Pizza.java

```
Pizza pizza = new Pizza();

pizza.setPizzaSize("small");
pizza.setCrust("thin");
pizza.setMushroomsCount(1);
pizza.setCheeseCount(1);

// do something with pizza
```

Această soluție nu prezintă niciunul dintre dezavantajele modelului anterior. Este ușor să scalați clasa, mai ușor de instanțiat, mai ușor de citit și mai flexibil. Modelul are însă dezavantaje grave. Construcția clasei este împărțită în apeluri multiple, prin urmare instanța poate fi într-o stare parțial construită / invalidă.

Folosirea builder pattern

Pizza.java

```
public class Pizza {
    private String pizzaSize;
    private String crust;
    private int cheeseCount;
    private int pepperoniCount;
    private int hamCount;
    private int mushroomsCount;

    public static class Builder {
        private String pizzaSize; // mandatory
        private String crust; // mandatory
        private int cheeseCount = 0; // optional
        private int pepperoniCount = 0; // optional
        private int hamCount = 0; // optional
        private int mushroomsCount = 0; // optional

        public Builder(String pizzaSize, String crust) {
            this.pizzaSize = pizzaSize;
            this.crust = crust;
        }

        public Builder cheeseCount(int cheeseCount) {
            this.cheeseCount = cheeseCount;
            return this;
        }

        public Builder pepperoniCount(int pepperoniCount) {
            this.pepperoniCount = pepperoniCount;
            return this;
        }

        public Builder hamCount(int hamCount) {
            this.hamCount = hamCount;
            return this;
        }

        public Builder mushroomsCount(int mushroomsCount) {
            this.mushroomsCount = mushroomsCount;
            return this;
        }

        public Pizza build() {
            return new Pizza(this);
        }
    }
```

```
    private Pizza(Builder builder) {
        this.pizzaSize = builder.pizzaSize;
        this.crust = builder.crust;
        this.cheeseCount = builder.cheeseCount;
        this.pepperoniCount = builder.pepperoniCount;
        this.hamCount = builder.hamCount;
        this.mushroomsCount = builder.mushroomsCount;
    }

    // getters
}
```

Am făcut constructorul privat, astfel încât clasa noastră să nu poată fi instanțiată direct. În același timp am adăugat o clasă static Builder cu un constructor care are parametrii noștri obligatori pizzaSize și crust, metode de setare a parametrilor opționali și, în final, o metodă *build()* metoda care va returna o nouă instanță a clasei Pizza. Metodele setter returnează instanța de builder în sine, oferindu-ne astfel o interfață fluentă cu metoda de înlănțuire.

Pizza.java

```
Pizza pizza = new Pizza.Builder("large", "thin")
    .cheeseCount(1)
    .pepperoniCount(1)
    .build();
```

Este mult mai ușor să scrieți și, mai important, să citiți acest cod. La fel ca în cazul constructorului, putem verifica parametrii trecuți pentru orice încălcare, cel mai adesea în cadrul metodei *build()* sau a metodei setter, și putem arunca IllegalStateException dacă există încălcări înainte de a crea o instanță a clasei.

Modelul Builder are unele dezavantaje minore. În primul rând, trebuie să creați un obiect Builder înainte de a crea obiectul clasei în sine. Aceasta ar putea fi o problemă în unele cazuri critice de performanță iar clasa devine puțin mai mare când vine vorba de liniile de cod.

În ansamblu, modelul Builder este o tehnică foarte frecvent utilizată pentru crearea obiectelor și este o alegere bună de utilizat atunci când clasele au constructori cu parametri multipli (în special opționali) și este posibil să se schimbe în viitor. Codul devine mult mai ușor de scris și de citit în comparație cu constructorii supraincarcati, iar clasa este la fel de bună ca folosirea de getters și setters, dar este mult mai sigură.

Exerciții

Command

Implementați folosind patternul Command un editor de diagrame foarte simplificat. Scheletul de cod [https://github.com/oop-pub/oop-labs/tree/master/src/lab10] conține o parte din clase și câteva teste.

Componentele principale ale programului:

- *DiagramCanvas* - reprezintă o diagramă care conține obiecte de tip DiagramComponent
- *DrawCommand* - interfață pentru comenzile făcute asupra diagramei sau a componentelor acesteia

- *Invoker* - primește comenzile și le execută
- *Client* - entry-point-ul în program

Task 1 - Implementarea comenzilor (4p)

Implementați 5 tipuri de comenzi, pentru următoarele acțiuni:

- Draw rectangle - crează o DiagramComponent și o adaugă în DiagramCanvas
- Resize - modifică width și height al unei DiagramComponent pe baza unui procent dat
- Change color - modifică culoarea unei DiagramComponent
- Change text - modifică textul unei DiagramComponent
- Connect components - conectează o DiagramComponent la alta

Comenzile primesc în constructor referința către DiagramCanvas și alte argumente necesare lor. De exemplu, comanda pentru schimbarea culorii trebuie sa primească și culoarea nouă și indexul componentei.

Pentru acest task nu este nevoie să implementați și metoda *undo()*, doar *execute()*.

Comenzile implementează în afară de metodele interfeței și metoda toString() [https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Object.html#toString()] pentru a afișa comanda. Recomandăm folosirea IDE-ului pentru a o genera.

Task 2 - Testarea comenzilor (2p)

Scheletul conține în clasa `Test` metode pentru a testa comportamentul comenzilor. O parte sunt deja implementate, iar o parte trebuie implementate.

Task 3 - Undo/redo (2p)

Implementați în comenzi și în Invoker mecanismul de undo/redo al comenzilor. Recomandăm în Invoker sa folosiți două structuri de date, una care să mențină comenzile efectuate, iar una pentru comenzile făcute undo.

Task 4 - Test undo/redo (2p)

Scheletul conține în clasa Test o metodă pentru o verificare simplă a corectitudinii implementării undo și redo. Completați metoda *testComplexUndoRedo* în care să faceți multiple undo-uri și redo-uri pentru diverse tipuri de comenzi.

Builder

Task 5 - (0.75p)

Scrieți câmpuri în skeletul clasei House pentru câteva facilități obligatorii în construcția unei case, spre exemplu locația construcției, numărul de etaje, încălzire, camere dar și unele opționale pe care le poate selecta sau nu clientul, cum ar fi electrocasnice, piscină, panouri solare, securitate etc.

Completați constructorul privat, metodele de get și metoda toString.

Task 6 - (0.75p)

În clasa de builder, completați câmpurile, constructorul și metodele de adăugare a facilităților opționale.

Task 7 - (0.5p)

Finalizați metoda build și testați funcționalitatea într-o clasă Main creată de voi, acoperind cazuri în care se construiește o casa doar cu facilitați obligatorii și altele adăugând și pe cele opționale.

Resurse

- Exerciții din alți ani

Referințe

- Command design pattern [https://sourcemaking.com/design_patterns/command]
- De ce avem nevoie de Command Pattern? [https://stackoverflow.com/questions/32597736/why-should-i-use-the-command-design-pattern-while-i-can-easily-call-required-met]

poo-ca-cd/laboratoare/design-patterns2.txt · Last modified: 2021/12/13 01:00 by florin.mihalache