Design patterns - Command, Strategy

- Responsabil: Theodor Stoican
- Data ultimei modificări: 12.12.2017

Obiective

Scopul acestui laborator este familiarizarea cu folosirea unor pattern-uri des întâlnite în design-ul atât al aplicațiilor, cât și al API-urilor - *Command* și *Strategy*.

Introducere

Design pattern-urile reprezintă soluții generale și reutilizabile ale unei probleme comune în design-ul software. Un design pattern este o descriere a soluției sau un template ce poate fi aplicat pentru rezolvarea problemei, nu o bucata de cod ce poate fi aplicata direct. În general pattern-urile orientate pe obiect arată relațiile și interacțiunile dintre clase sau obiecte, fără a specifica însă forma finală a claselor sau a obiectelor implicate.

Se consideră că există aproximativ 2000 de design patterns [2], iar principalul mod de a le clasifica este următorul:

- **“Gang of Four” patterns**
- Concurrency patterns
- Architectural patterns - sunt folosite la un nivel mai inalt decat design patterns, stabilesc nivele și componente ale sistemelor/aplicațiilor, interacțiuni între acestea (e.g. Model View Controller şi derivatele sale). Acestea descriu structura întregului sistem, iar multe framework-uri vin cu ele deja încoporate, sau faciliteaza aplicarea lor (e.g. Java Spring). În cadrul laboratoarelor nu ne vom lega de acestea.

O carte de referință pentru design patterns este “Design Patterns: Elements of Reusable Object-Oriented Software” [1], denumită și “Gang of Four” (GoF). Aceasta definește 23 de design patterns, foarte cunoscute și utilizate în prezent. Aplicațiile pot încorpora mai multe pattern-uri pentru a reprezenta legături dintre diverse componente (clase, module).
În afară de GoF, și alți autori au adus în discuție pattern-uri orientate în special pentru aplicațiile enterprise și cele distribuite.

Pattern-urile GoF sunt clasificate în felul următor:

- **Creational Patterns** - definesc mecanisme de creare a obiectelor
  - Singleton, Factory etc.
- **Structural Patterns** - definesc relații între entități
  - Decorator, Adapter, Facade, Composite, Proxy etc.
- **Behavioural Patterns** - definesc comunicarea între entități
  - Visitor, Observer, Command, Mediator, Strategy etc.

Design pattern-urile nu trebuie privite drept niște rețete care pot fi aplicate direct pentru a rezolva o problemă din design-ul aplicației, pentru că de multe ori pot complica inutil arhitectura. Trebuie întâi înțeles dacă este cazul să fie aplicat un anumit pattern, si de-abia apoi adaptat pentru situația respectivă. Este foarte probabil chiar să folosiți un pattern (sau o abordare foarte similară acestuia) fără să vă dați seama sau să îl numiți explicit. Ce e important de reținut după studierea acestor pattern-uri este un mod de a aborda o problemă de design.

În laboratorul Visitor Pattern au fost introduse design pattern-urile și aplicabilitatea Visitor-ului. Acesta este un pattern comportamental, și după cum ați observat oferă avantaje în anumite situații, în timp ce pentru altele nu este potrivit. Pattern-urile comportamentale modelează interacțiunile dintre clasele și componentele unei aplicații, fiind folosite în cazurile în care vrem sa facem un design mai clar și ușor de adaptat și extins. /*În afară de acest tip de pattern-uri, mai se folosesc și cele *structural* și *creational*, prezentate în clasificarea următoare:

- *Creational Patterns* - mecanisme de creare a obiectelor
  - **Singleton**, **Factory** etc
- *Structural Patterns* - definesc relații între entități
  - Decorator, Adapter, Facade, Composite, Proxy etc.
- *Behavioural Patterns* - definesc comunicarea între entități
  - Visitor, **Observer**, **Command**, Mediator, Strategy etc.

Command Pattern

Design pattern-ul *Command* încapsulează un apel cu tot cu parametri într-o clasă cu interfață generică. Acesta este *Behavioral* pentru ca 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 întârziată
- 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

Functionare si necesitate

In esenta, Command pattern (asa cum v-ati obisnuit si lucrand cu celelate Pattern-uri pe larg cunoscute) presupune incapsularea unei informatii referitoare la actiuni/comenzi folosind un wrapper pentru a “tine minte aceasta informatie” si pentru a o folosi ulterior. Astfel, un astfel de wrapper va detine informatii referitoare la tipul actiunii respective (in general un asemenea wrapper va expunde o metoda execute(), care va descrie comportamentul pentru actiunea respectiva).

Mai mult inca, cand vorbim de Command Pattern, in terminologia OOP o sa intalniti deseori si notiunea de Invoker. Invoker-ul este un middleware ca functionalitate care realizeaza managementul comenzilor. Practic, un Client, care vrea sa faca anumite actiune, va instantia clase care implementeaza o interfata Command. Ar fi incomod ca, in cazul in care aceste instantieri de comenzi provin din mai multe locuri, acest management de comenzi sa se face local, in fiecare parte (din ratiuni de economie, nu vrem sa duplicam cod). Invoker-ul apare ca o necesitate de a centraliza acest proces si de a realiza intern management-ul comenzilor (le tine intr-o lista, tine cont de eventuale dependinte intre ele, totul in functie de context).

Si nu in cele din urma, un client (generic spus, un loc de unde se lanseaza comenzi) instantiaza comenzile si le paseaza Invoker-ului. Din acest motiv Invoker-ul este un middleware intre client si receiver, fiindca acesta va apela execute pe fiecare Command, in functie de logica sa interna.

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*).

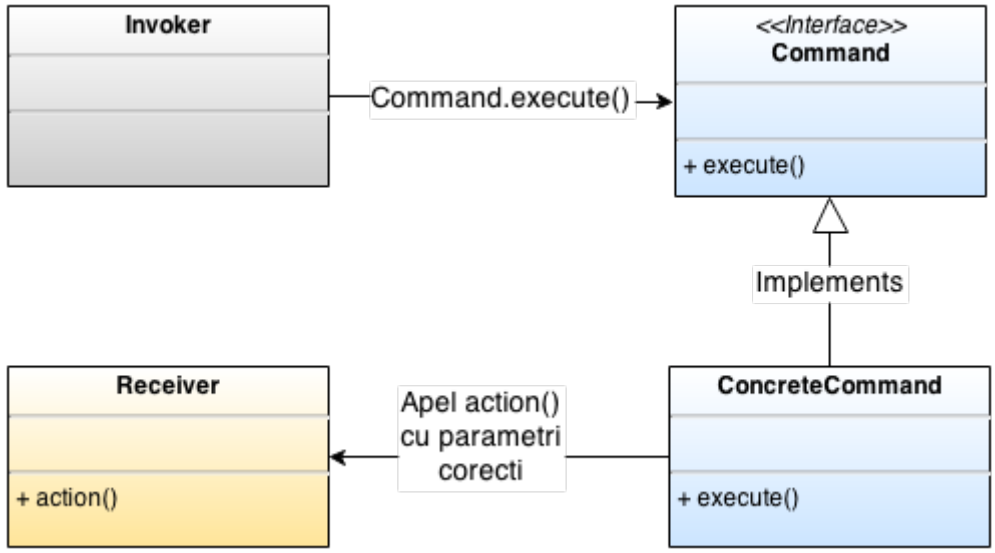

Fig. 1: Diagrama de stări pentru CommandPattern

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 tuturor 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 diagrama din figure 1, 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.

Implementare

Diagrama de secvență din figure 2 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 *CropCommand* 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.

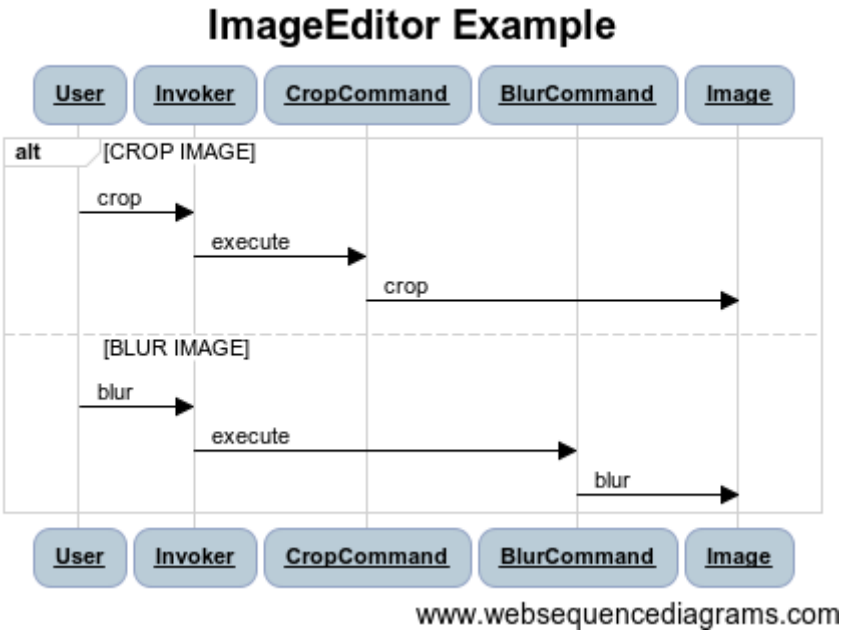

Fig. 2: Diagrama de secvență pentru comenzile de prelucrare a imaginilor

Pe wikipedia puteți analiza exemplul PressSwitch. Flow-ul pentru acesta este ilustrat în figure 3

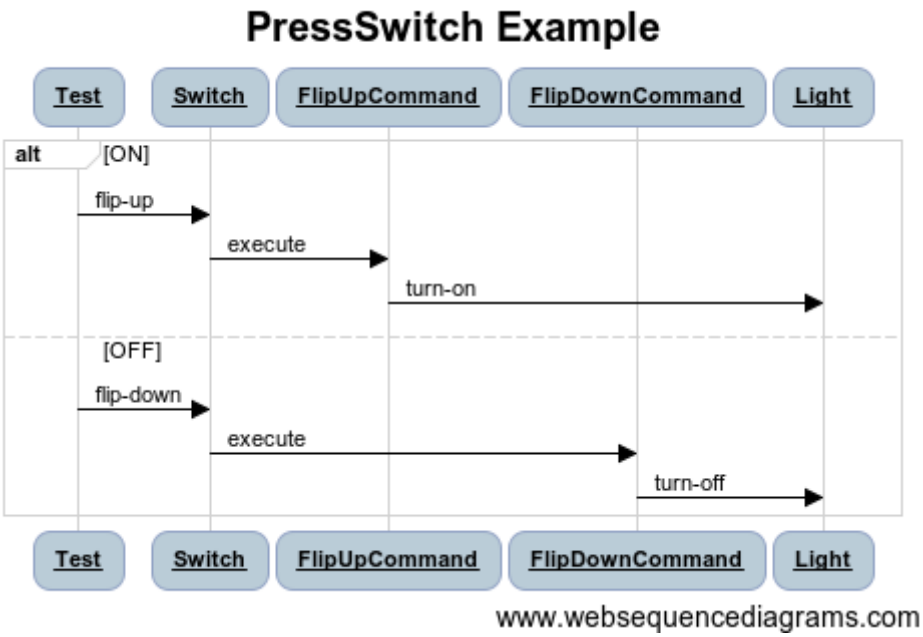

Fig. 3: Diagrama de secvență pentru comenzile de aprindere/stingere a switch-ului

Strategy Pattern

Design pattern-ul *Strategy* încapsulează algoritmii în clase ce oferă o anumită interfață de folosire, și pot fi selecționați la runtime. Ca și Command, acest pattern este *behavioral* pentru ca permite decuplarea unor clase ce oferă un anumit comportament și folosirea lor independentă în funcție de situația de la runtime.

Acest pattern este recomandat în cazul în care avem nevoie de un tip de algoritm (strategie) cu mai multe implementări posibile si dorim să alegem dinamic care algoritm îl folosim, fără a face sistemul prea strâns cuplat.

Exemple de utilizare:

- sisteme de tip Layout Managers din API-urile pentru UI
- selectarea in mod dinamic la runtime a unor algoritmi de sortare, compresie, criptare etc.

Structură:

- trebuie să definiți o **interfață comună** pentru strategiile pe care le implementați (fie ca o «interface» sau ca o clasa abstractă)
- implementați strategiile respectând interfața comună
- clasa care are nevoie să folosească strategiile **va ști doar despre interfața lor**, nu va fi legată de implementările concrete

Denumirile uzuale în exemplele acestui pattern sunt: *Strategy* (pt interfață sau clasa abstractă), *ConcreteStrategy* pentru implementare, *Context*, clasa care folosește/execută strategiile.

Recomandare: Urmariti link-ul de la referinte catre postul de pe Stack Overflow care descrie necesitatea pattern-ului Strategy. Pe langa motivul evident de incapsulare a prelucrarilor/algoritmilor (care reprezinta strategiile efective), se prefera o anumita abordare: la runtime se verifica mai multe conditii si se decide asupra strategiei. Concret, folosind mecanismul de polimorfism dinamic, se foloseste o anumita instanta a tipului de strategie (*ex. Strategy str = new CustomStrategy*), care se

paseaza in toate locurile unde este nevoie de Strategy. Practic, in acest fel, utilizatorii unei anumite strategii vor deveni agnostici in raport cu strategia utilizata, ea fiind instantiata intr-un loc anterior si putand fi gata utilizata. Ganditi-va la browserele care trebuie sa detecteze daca device-ul este PC, smartphone, tableta sau altceva si in functie de acest lucru sa randeze in mod diferit. Fiecare randare poate fi implementata ca o strategie, iar instantierea strategiei se va face intr-un punct, fiind mai apoi pasata in toate locurile unde ar trebui sa se tina cont de aceasta strategie.

Exerciții

Acest laborator și cel precedent au ca temă comună a exercițiilor realizarea unui joc controlat din consolă. Jocul constă dintr-o lume (aka hartă) în care se plimbă eroi de trei tipuri, colectează comori și se bat cu monștrii. În acestă săptămână terminam jocul inceput in laboratorul precedent folosind pattern-urile studiate (*Strategy*, *Command*).

Detalii joc:

- *Harta*
  - reprezentată printr-o matrice. Fiecare element din matrice reprezintă o zonă care poate fi liberă, poate conține obstacole sau poate conține o comoară (în laboratorul următor poate conține și monștrii).
  - este menținută în clasa `GameState`.
- *Eroii*
  - sunt reprezentați prin clase de tip `Hero` și sunt de trei tipuri: *Mage*, *Warrior*, *Priest*.
  - puteți adăuga oricâți eroi doriți pe hartă (cât vă permite memoria :))
  - într-o zonă pot fi mai mulți eroi
  - acțiunile pe care le pot face:
    - `move` - se mută într-o zonă învecinată
    - `attack` - ataca un monstru cand se afla pe aceeasi pozitie cu el
    - `collect` - eroul ia comoara găsită în zona în care se află
- Entry-point-ul în joc îl consitituie clasa `Main`.
- **(5p)** Folosiți design pattern-ul *Command* pentru a implementa functionalitatea de undo si redo la comanda `move`.
  - Momentan, aveti erori de compilare in clasele Main si GameState. Dupa ce veti implementa acest exercitiu, se vor rezolva, nu modificati in mod direct acolo.
  - Va trebui sa completati clasa `MoveCommand` care implementeaza interfata `Command`. Urmariti *TODO-urile* din aceasta clasa.
  - Hint: Pentru Undo, de exemplu, daca v-ati deplasat la dreapta, ar trebui sa va deplasati la stanga. Creati-va o metoda ajutatoare care trateaza astfel de cazuri.
  - Precum si clasa `CommandManager` care va tine evidenta comenzilor si ordinea lor.
  - Hint: Amintiti-va de la cursul de Structuri de Date cum se implementeaza operatiile Redo si Undo. Folositi doua stive.
- **(5p)** Folosiți design pattern-ul *Strategy* pentru a implementa logica de atac a unui monstru.
  - Pentru acest exercitiu va trebui sa implementati 2 strategii: **AttackStrategy** si **DefenseStrategy**. Ambele vor implementa Strategy si metodele aferente. Fiecare din aceste Strategy, va retine o referinta interna la un Hero. Abordarea este urmatoarea:
    - Exista 3 clase Hero, fiecare cu cate un DamageType aferent: Warrior - Blunt, Mage - Magic, Priest - Poison
    - Fiecare monstru are un weakness (slabiciune la atacurile de tip Blunt, Magic sau Poison)

        - **AttackStrategy**: In metoda attack(), veti itera prin inventory-ul eroului si veti verifica daca exista un obiect Treasure care are DamageType-ul identic cu cel al eroului. Daca da, atunci damage-ul pe care il veti da unui obiect Monster este **3 x damage-ul treasure-ului**. Daca nu aveti un astfel de Treasure, atunci cautati un Treasure cu un DamageType identic cu cel al mob-ului (mobul poate fi vulnerabil la Magic, de ex.). Daca gasiti un astfel de Treasure, damage-ul pe care il veti imprima mob-ului este **2 x damage-ul treasure-ului**. Daca nu, veti scade din HP-ul mob-ului rezultatul apelului metodei getBaseDamage() asupra Hero-ului.
        - **DefenseStrategy**: In metoda attack(), veti itera prin inventory-ul eroului si veti verifica, la fel, daca exista un obiect Treasure care are DamageType-ul identic cu cel al eroului. Daca da, atunci veti da un boost de HP egal cu treasure.getBoostHp() + getBaseHpBoost() eroului (getBaseHpBoost este implementata in clasa Hero). Daca nu, veti da doar un boost egal cu ce va intoarce getBaseHpBoost(). Damage-ul imprimat mobului va fi, de asemenea, egal cu ce va intoarce getBaseDamage(). Adaugati log-uri in clasa DefenseStrategy, pentru a va asigura ca i se mareste viata eroului.
        - Urmariti si TODO-urile din cele doua clase
    - Implementati metoda *attack* din clasa **Hero** astfel incat, daca eroul are mai mult de **50HP**, folositi strategia **AttackStrategy**. Altfel, folositi **DefenseStrategy**. Urmariti TODO-urile din cod.
- (**Bonus 2p**) Implementați coliziunile cu obstacolele de pe harta
    - Va trebui sa creati un nou obiect `Obstacle` precum si un `ObstacleObserver`
    - Cand eroul ajunge pe un obstacol se va printa un mesaj `Can't move there !` si se va apela automat undo pe ultima comanda de move pentru a reveni in pozitia anterioara coliziunii. Acest feature de wall collision va fi implementat in `ObstacleObserver`

Resurse

- Schelet
- Soluție
- PDF laborator

Referințe

- Descriere Command Pattern si exemplu pentru operatii undo/redo
- O resursa buna pentru a intelege cele mai des intalnite design patterns
- De ce avem nevoie de Command Pattern?
- De ce avem nevoie de Strategy Pattern?