Návrhové vzory: Behavioral patterns- (Chain Of Responsibility, Command, Interpreter, Iterator)

Jakub Pôčka, Michal Fusatý, Adam Ružička, Michal Grznár, Samuel Hudák

Obsah

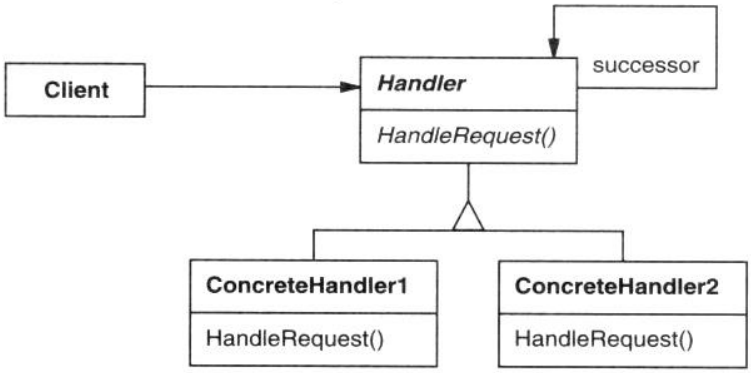
Chain of Responsibility

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

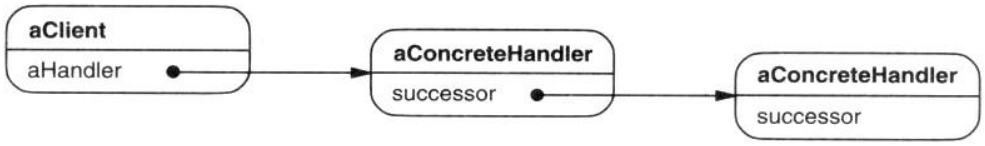
V mnohých aplikáciách komponenty reagujú na udalosti, ktoré vznikajú v iných komponentoch. Vtedy je vždy potrebné vytvoriť mechanizmus na odovzdanie notifikácie komponentom, ktoré na udalosť chcú reagovať. Jednou možnosťou je priama notifikácia: ak komponent vie, ktoré všetky komponenty má pri tej-ktorej udalosti notifikovať, môže napríklad zavolať ich dohodnutú metódu.

Často to tak nie je. V čase návrhu komponentu, kde udalosť vzniká, nie je známe, v ktorých komponentoch sa bude spracovávať. Môže sa to aj dynamicky meniť, v závislosti od toho, kde a ako sa príslušný komponent používa. Napríklad, predstavme si robotického manžela a jeho plánovací kalendár. V čase začiatku naplánovanej udalosti kalendár vygeneruje notifikáciu. Podľa toho, o akú udalosť ide, má naštartovať zodpovedajúci modul - napr. modul vysávania, modul umývania riadu, modul vešania prádla a pod. pričom kalendár vôbec nemá informáciu o tom, aké rôzne moduly existujú a aké udalosti v ňom môžu byť zaradené. V inom príklade si predstavme dialógové okno, v ktorom je viacero ovládacích prvkov rozložených do hierarchickej štruktúry - panely, podpanely, skupiny rádiobuttonov a pod. Pri kliknutí myši potrebujeme, aby na túto udalosť zareagoval ten správny komponent, na ktorý sa kliklo. Podobne, ak je časť dialógového okna dočasne zakrytá a po odkrytí sa má znovu aktuálne nakresliť - netreba kresliť celé okno, ale len tie podčasti, ktoré majú prienik s oblasťou, ktorá sa má prekresliť. V takýchto situáciách potrebujeme nejaký univerzálny mechanizmus na doručovanie notifikácií o udalostiach.

Návrhový vzor Chain of Responsibility rieši takéto prípady. Všetky komponenty, ktoré majú záujem spracovávať udalosti z nejakého zdroja, sú zoradené v reťazi "handlerov". Zdroj, ktorý notifikáciu o udalosti generuje, má len referenciu na prvý handler v reťazi. Ten udalosť buď spracuje, alebo ju pošle ďalšiemu handleru v reťazi. Hovoríme, že notifikácia nemá explicitný (zreteľne uvedený) cieľ, ale má len implicitného adresáta (takého, čo vyplynie z aktuálne platnej reťaze handlerov). Handler sa môže rozhodnúť, či udalosť spracoval kompletne, alebo či na ňu síce nejak zareaguje, ale pošle ju aj ďalším handlerom v reťazi, alebo či sa ho netýka a iba ju pošle ďalej. V princípe sa môže stať, že na niektoré udalosti žiaden handler nezareaguje - je na návrhu a implementácii, aby boli všetky potrebné prípady pokryté korektne. Návrhový vzor nepredpisuje, či sa reťaz handlerov má vytvoriť pomocou novo-zadefinovaných referencií (ako ukazuje obrázok nižšie), alebo či sa využije už nejaká existujúca hierarchická štruktúra handlerov - ako by to mohlo byť v prípade hierarchických grafických ovládacích prvkov v dialógovom okne.



A typical object structure might look like this:



Štruktúra návrhového vzoru Chain of Responsibility

Implementácia

Táto implementácia predpokladá, že jednotlivé články majú odkazy na nasledujúce články a spracovanie je teda v rukách každého článku.

```
interface Handler<T> {
    void handle(T request);
}
public abstract class AbstractHandler<T> implements Handler<T> {
    private Handler<T> next;

    public void setNext(Handler<T> next) {
        this.next = next;
    }

    protected void handleByNext(T request) {
        if (this.next != null) {
            // předává řízení dalšímu článku v řetězci
            this.next.handle(request);
        }
    }
}
public class LoginHandler extends AbstractHandler<Request> {
    @Override
    public void handle(Request request);
        if (containsLoginHeaders(request)) {
            if (tryLogin(request)) {
                request.setAuthorized(true);
```

```
            }
        }

        handleByNext(request);
    }
}
public class AuthorizingHandler extends AbstractHandler<Request> {
    @Override
    public void handle(Request request);
        if (request.isAuthorized()) {
            handleByNext(request);
        } else {
            request.setResponseCode(403);
            request.setResponseBody("You are not authorized to do this!");
        }
    }
}
public class PerformingHandler extends AbstractHandler<Request> {
    @Override
    public void handle(Request request);
        try {
            performAction(request);
            request.setResponseCode(200);
            request.setResponseBody("Action was performed!");
        } catch (Exception e) {
            request.setResponseCode(500);
            request.setResponseBody("Action failed: " + e.toString());
        }
    }
}
```

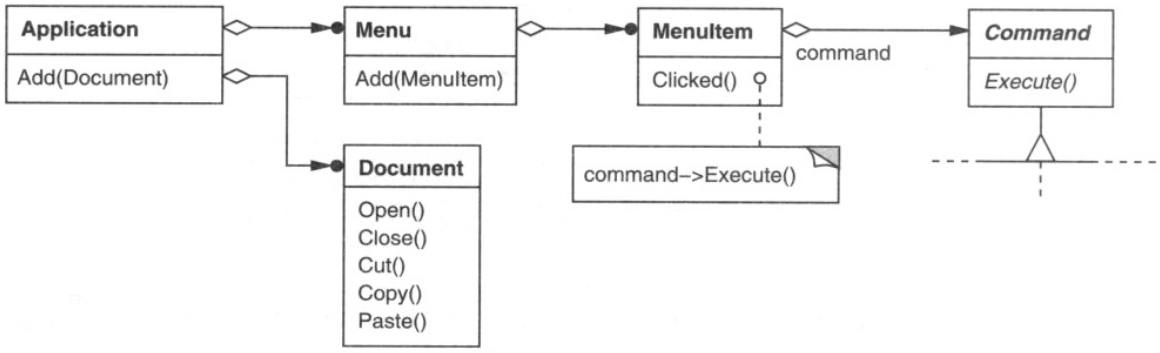
Command

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Ako sme uviedli v predchádzajúcom, princípy OOP vychádzajú z myšlienky dekompozície na základe dát, čiže informácií. Údaje, ktoré prirodzene patria spolu vytvárajú uzavreté štruktúry dát - objekty. V niektorých situáciách môže byť takouto informáciou (dátami) aj inštrukcia (príkaz). To je aj prípad návrhového vzoru Command, kde objekt triedy Command nesie informáciu o tom, aký príkaz sa má vykonať, resp. sa už vykonal. To nám umožňuje generovať požiadavky na vykonanie príkazov bez toho, aby sme vedeli o aké príkazy ide, alebo evidovať zoznam vykonaných príkazov počas nejakej postupnosti príkazov (session).

Predstavme si framework na vytváranie používateľského menu. Tento framework by mal aplikácii dovoliť pohodlne konfigurovať priradenie príkazov k jednotlivým položkám menu. Namiesto toho, aby sme to realizovali cez nejaké číselné kódy príkazov a potom v dlhej podmienke v štýle switch...case... rozlíšili podľa čísla o aký príkaz ide, môžeme frameworku priamo posunúť inštanciu nejakej podtriedy triedy Command, ktorá bude implementovať metódu execute(). Po zvolení zpodpovedajúcej položky menu framework zavolá metódu execute() nášho objektu a tá zabezpečí že sa vykoná požadovaný príkaz. Ak by sme do rozhrania pridali aj metódu undo(), postupnosť vykonávaných príkazov by framework mohol evidovať (história operácií) a po požiadavke používateľa

na vykonanie kroku späť (undo), prípadne viacnásobného návratu (undo-undo-undo...) by sa len volali metódy undo() na objektoch zodpovedajúcich príkazom vykonaným v predchádzajúcich krokoch. Podobne by sme mohli pridať redo(), aby sa dalo v histórii presúvať oboma smermi.



Príklad využitia návrhového vzoru Command vo frameworku na vytváranie používateľského menu

Skombinovaním vzoru Command so vzorom Composite je možné vytvárať makrá.

Implementácia

```
// The base command class defines the common interface for all
// concrete commands.
abstract class Command is
    protected field app: Application
    protected field editor: Editor
    protected field backup: text

    constructor Command(app: Application, editor: Editor) is
        this.app = app
        this.editor = editor

    // Make a backup of the editor's state.
    method saveBackup() is
        backup = editor.text

    // Restore the editor's state.
    method undo() is
        editor.text = backup

    // The execution method is declared abstract to force all
    // concrete commands to provide their own implementations.
    // The method must return true or false depending on whether
    // the command changes the editor's state.
    abstract method execute()


// The concrete commands go here.
class CopyCommand extends Command is
    // The copy command isn't saved to the history since it
    // doesn't change the editor's state.
    method execute() is
        app.clipboard = editor.getSelection()
        return false
```

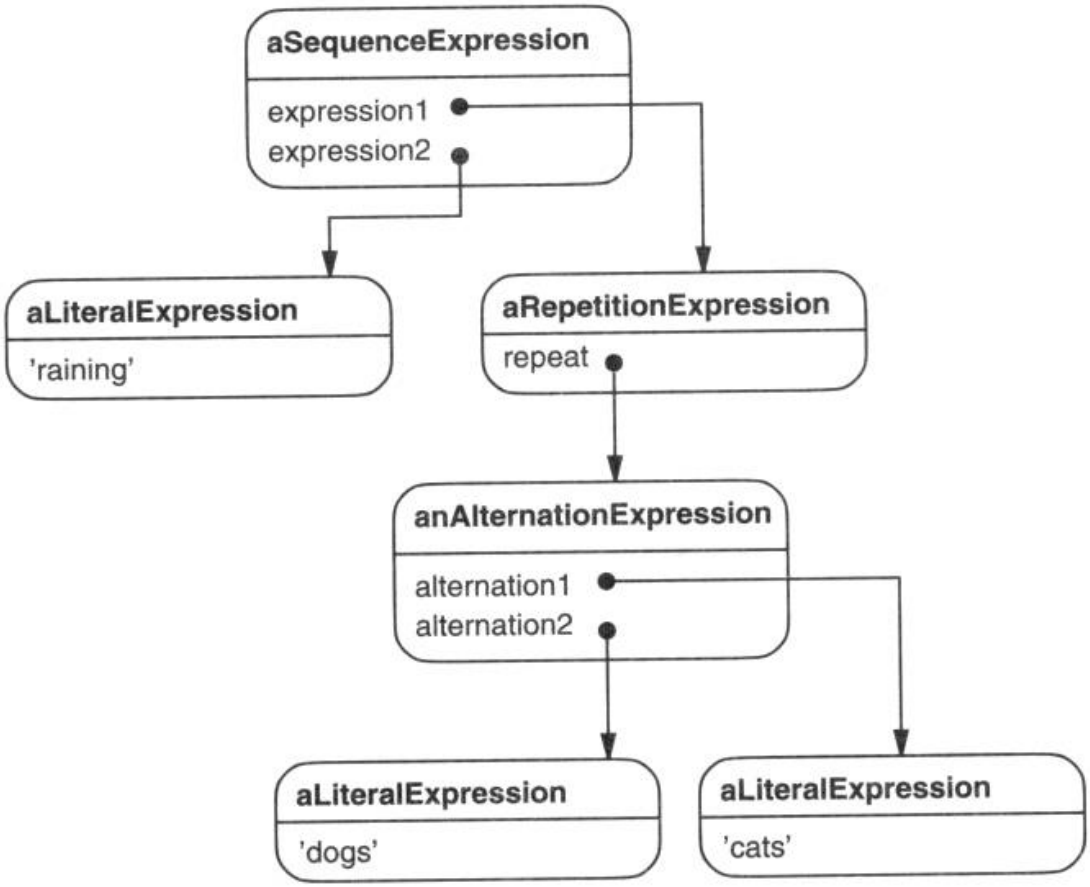
```
class CutCommand extends Command is
    // The cut command does change the editor's state, therefore
    // it must be saved to the history. And it'll be saved as
    // long as the method returns true.
    method execute() is
        saveBackup()
        app.clipboard = editor.getSelection()
        editor.deleteSelection()
        return true

class PasteCommand extends Command is
    method execute() is
        saveBackup()
        editor.replaceSelection(app.clipboard)
        return true

// The undo operation is also a command.
class UndoCommand extends Command is
    method execute() is
        app.undo()
        return false


// The global command history is just a stack.
class CommandHistory is
    private field history: array of Command

    // Last in...
    method push(c: Command) is
        // Push the command to the end of the history array.

    // ...first out
    method pop():Command is
        // Get the most recent command from the history.


// The editor class has actual text editing operations. It plays
// the role of a receiver: all commands end up delegating
// execution to the editor's methods.
class Editor is
    field text: string

    method getSelection() is
        // Return selected text.

    method deleteSelection() is
        // Delete selected text.

    method replaceSelection(text) is
        // Insert the clipboard's contents at the current
        // position.


// The application class sets up object relations. It acts as a
// sender: when something needs to be done, it creates a command
// object and executes it.
class Application is
    field clipboard: string
    field editors: array of Editors
    field activeEditor: Editor
    field history: CommandHistory
```

```
// The code which assigns commands to UI objects may look
// like this.
method createUI() is
    // ...
    copy = function() { executeCommand(
        new CopyCommand(this, activeEditor)) }
    copyButton.setCommand(copy)
    shortcuts.onKeyPress("Ctrl+C", copy)

    cut = function() { executeCommand(
        new CutCommand(this, activeEditor)) }
    cutButton.setCommand(cut)
    shortcuts.onKeyPress("Ctrl+X", cut)

    paste = function() { executeCommand(
        new PasteCommand(this, activeEditor)) }
    pasteButton.setCommand(paste)
    shortcuts.onKeyPress("Ctrl+V", paste)

    undo = function() { executeCommand(
        new UndoCommand(this, activeEditor)) }
    undoButton.setCommand(undo)
    shortcuts.onKeyPress("Ctrl+Z", undo)

// Execute a command and check whether it has to be added to
// the history.
method executeCommand(command) is
    if (command.execute)
        history.push(command)

// Take the most recent command from the history and run its
// undo method. Note that we don't know the class of that
// command. But we don't have to, since the command knows
// how to undo its own action.
method undo() is
    command = history.pop()
    if (command != null)
        command.undo()
```

Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Predstavme si nejaký jednoduchý špecializovaný jazyk - napr. na programovanie pohybov robotického ramena pripojeného k vesmírnej stanici. Návrhový vzor Interpreter rieši tento druh situácie, pričom účelom interpretra môže byť buď len sparsovanie programov zapísaných v jazyku a vybudovanie abstraktného syntaktického stromu (AST) k vstupnému programu, za účelom ďalšej analýzy a spracovania, prípadne vykonania, alebo môže byť úlohou interpretra rovno zadaný program aj vykonať. Pre danú gramatiku môžeme vybudovať hierarchiu tried zodpovedajúcich jednotlivým druhom pravidiel a pravidlám ako takým. V prípade zložitejších gramatík sa odporúča radšej použiť externé nástroje na parsovanie.

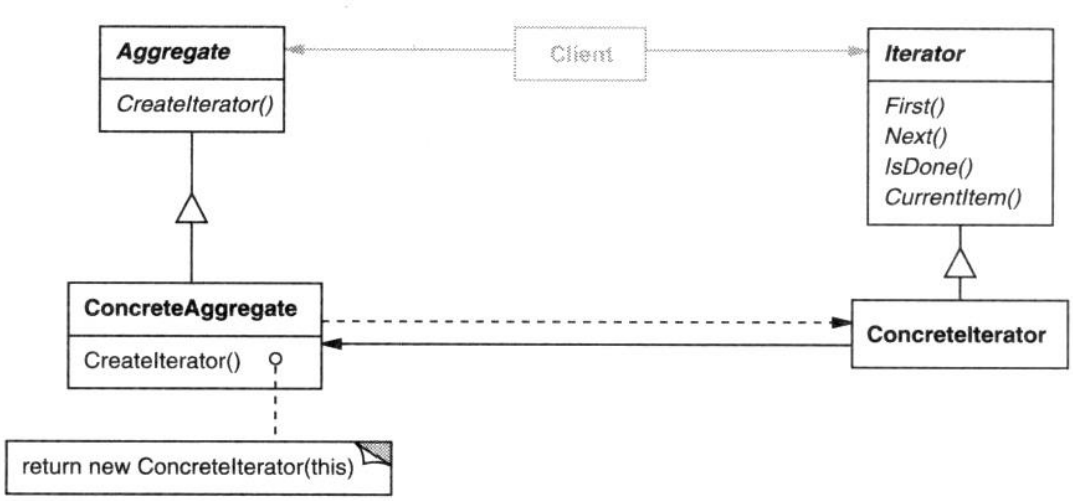


Príklad hierarchie tried pri použití návrhového vzoru Interpreter - objektová reprezentácia vyjadrujúca jedno pravidlo regulárneho výrazu "((cats|dogs) repeat) and raining"

Iterátor

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Iterator je jeden z najbežnejšie používaných návrhových vzorov, ktorý umožňuje postupne prechádzať cez všetky prvky združené v nejakom objekte, ktorý združuje množinu prvkov. Podstatné je, že interface iterátora je univerzálny, a použiteľný pre rozličné zbierky prvkov rôzneho typu. Dnešné moderné jazyky obsahujú zabudované syntaktické štruktúry na automatické vytváranie a používanie iterátorov, ale v tomto prípade máme na mysli používateľom definované agregované štruktúry, cez ktoré má zmysel potenciálne iterovať univerzálnym spôsobom nezávisle od vnúrornej reprezentácie objektu, ktorý interface iterátora spĺňa. Iterator predpisuje operácie posunutia sa na nasledujúci prvok, prezretia aktuálneho prvku, test na koniec procesu iterovania a prípadne reset na začiatočný prvok. Iterátor môže postupovať buď v náhodnom poradí, alebo podľa nejakého operátora usporiadania.



Štruktúra návrhového vzoru Iterator. V roli Aggregate je nejaký abstraktný interface pre kolekcie, v roli ConcreteAggregate je nejaká konkrétna kolekcia, napr. HashSet, ConcreteIterator je potom taký iterátor, ktorý dokáže prechádzať prvky toho konkrétneho HashSetu, pre ktorý bol vytvorený

Implementácia

```
// The collection interface must declare a factory method for
// producing iterators. You can declare several methods if there
// are different kinds of iteration available in your program.
interface SocialNetwork is
    method createFriendsIterator(profileId):ProfileIterator
    method createCoworkersIterator(profileId):ProfileIterator


// Each concrete collection is coupled to a set of concrete
// iterator classes it returns. But the client isn't, since the
// signature of these methods returns iterator interfaces.
class Facebook implements SocialNetwork is
    // ... The bulk of the collection's code should go here ...

    // Iterator creation code.
    method createFriendsIterator(profileId) is
        return new FacebookIterator(this, profileId, "friends")
    method createCoworkersIterator(profileId) is
        return new FacebookIterator(this, profileId, "coworkers")


// The common interface for all iterators.
interface ProfileIterator is
    method getNext():Profile
    method hasMore():bool


// The concrete iterator class.
class FacebookIterator implements ProfileIterator is
    // The iterator needs a reference to the collection that it
    // traverses.
    private field facebook: Facebook
```

```
    private field profileId, type: string

    // An iterator object traverses the collection independently
    // from other iterators. Therefore it has to store the
    // iteration state.
    private field currentPosition
    private field cache: array of Profile

    constructor FacebookIterator(facebook, profileId, type) is
        this.facebook = facebook
        this.profileId = profileId
        this.type = type

    private method lazyInit() is
        if (cache == null)
            cache = facebook.socialGraphRequest(profileId, type)

    // Each concrete iterator class has its own implementation
    // of the common iterator interface.
    method getNext() is
        if (hasMore())
            currentPosition++
            return cache[currentPosition]

    method hasMore() is
        lazyInit()
        return currentPosition < cache.length


// Here is another useful trick: you can pass an iterator to a
// client class instead of giving it access to a whole
// collection. This way, you don't expose the collection to the
// client.
//
// And there's another benefit: you can change the way the
// client works with the collection at runtime by passing it a
// different iterator. This is possible because the client code
// isn't coupled to concrete iterator classes.
class SocialSpammer is
    method send(iterator: ProfileIterator, message: string) is
        while (iterator.hasMore())
            profile = iterator.getNext()
            System.sendEmail(profile.getEmail(), message)


// The application class configures collections and iterators
// and then passes them to the client code.
class Application is
    field network: SocialNetwork
    field spammer: SocialSpammer

    method config() is
        if working with Facebook
            this.network = new Facebook()
        if working with LinkedIn
            this.network = new LinkedIn()
        this.spammer = new SocialSpammer()

    method sendSpamToFriends(profile) is
        iterator = network.createFriendsIterator(profile.getId())
        spammer.send(iterator, "Very important message")
```

```
method sendSpamToCoworkers(profile) is
    iterator = network.createCoworkersIterator(profile.getId())
    spammer.send(iterator, "Very important message")
```

Zdroje

refactoring.guru

voho.eu

wikipedia.org

dai.fmph.uniba.sk

oodesign.com

sourcemaking.com

javatpoint.com