SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

NÁVRHOVÉ VZORY: BEHAVIORAL PATTERNS (MEDIATOR, MEMENTO, OBSERVER, STATE)

SEMINÁRNA PRÁCA

2022

Tomáš Popík
Katarina Popíková
Ľuboš Sremaňák
Matej Peluha

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

NÁVRHOVÉ VZORY: BEHAVIORAL PATTERNS (MEDIATOR, MEMENTO, OBSERVER, STATE)

SEMINÁRNA PRÁCA

Študijný program:	Aplikovaná informatika
Predmet:	I-ASOS – Architektúra softvérových systémov
Prednášajúci:	RNDr. Igor Kossaczký, CSc.
Cvičiaci:	Ing. Stanislav Marochok

Bratislava 2022

Tomáš Popík
Katarina Popíková
Ľuboš Sremaňák
Matej Peluha

Obsah

Úvod **1**

1 Webová aplikácia **2**

1.1 Inštalácia a spustenie 2

1.2 Používateľská príručka 2

2 Mediator pattern **4**

2.1 Problém 4

2.2 Riešenie 4

2.3 Využitie vzoru 4

2.4 Výhody a nevýhody 6

2.5 Implementácia 6

3 Observer pattern **8**

3.1 Problém 8

3.2 Riešenie 8

3.3 Využitie vzoru 8

3.4 Výhody a nevýhody 9

3.5 Implementácia 9

4 State pattern **11**

4.1 Využitie vzoru 11

4.2 Problém 11

4.3 Riešenie 12

4.4 Výhody a nevýhody 13

4.5 Implementácia 13

5 Memento pattern **15**

5.1 Návrh 15

5.1.1 Problém 15

5.1.2 Riešenie 16

5.2 Implementácia 17

5.3 Využitie vzoru 19

5.4 Výhody a nevýhody 19

Záver **21**

Zdroje **22**

Prílohy **I**

A UML class diagram našej aplikácie **II**

Zoznam obrázkov a tabuliek

Obrázok 1 Screenshot našej webovej aplikácie 3
Obrázok 2 UML class diagram štruktúry použitia Mediator patternu 5
Obrázok 3 UML class diagram Mediator patternu v našej aplikácií 7
Obrázok 4 UML diagram použitia Observer patternu 8
Obrázok 5 UML class diagram Observer patternu v našej aplikácií 10
Obrázok 6 UML diagram použitia State v Context [3] 12
Obrázok 7 Implementácia vzoru State v našej aplikácii 14
Obrázok 8 Pred vykonaním operácie sa uloží stav editora do histórie [4] . . 15
Obrázok 9 UML class diagram štruktúry použitia Memento patternu [4] . . 17
Obrázok 10 UML class diagram Memento patternu v našej aplikácii [4] . . . 18
Obrázok 11 Výpis histórie v konzole aplikácie 19
Obrázok 12 2. UML class diagram štruktúry použitia Memento patternu pri jazykoch s podporou vnorených tried [4] 19

Zoznam skratiek

DOM Document Object Model
GUI Graphical user interface
UML Unified Modeling Language

Zoznam algoritmov

Úvod

Medzi základné návrhové vzory patrí podskupina Behavioral patterns, v preklade vzory správania. Behavioral design patterns nám slúžia na komunikáciu medzi objektami. Tieto návrhové vzory zvyšujú efektivitu a flexibilitu v kóde, čím zamedzujú rast robustnosti kódu. Tým zamedzujeme budúcim chybám v programe a zlepšujeme rozšíriteľnosť kódu.

Z tejto skupiny návrhových vzorov si v našej práci predstavíme:

- Mediator pattern
- Observer pattern
- State pattern
- Memento pattern

Tieto návrhové vzory vznikli na zlepšenie komunikácie medzi objektami, oproti pôvodným jednoduchým riešeniam v kóde. Tie často narazili na rôzne problémy s flexibilitou a efektivitou kódu pri väčších projektoch.

V jednotlivých kapitolách si každý z týchto návrhových vzorov predstavíme, ukážeme si jeho využitie, rozoberieme jeho návrh, implementujeme do funkčného programu a zhrnieme jeho výhody s nevýhodami.

Pre účely nášho projektu sme vytvorili funkčnú webovú aplikáciu ako príklad použitia Behavioral design patterns. Táto aplikácia je dostupná na našom github repozitáry v ktorom je priložený aj tento dokument. Viac si o jej funkcionalite povieme v prvej kapitole a implementáciu konkrétnych návrhových vzoroch v ďalších prislúchajúcich kapitolách.

1 Webová aplikácia

Pre účely ukážky Behavioral design patterns sme vytvorili funkčnú webovú aplikáciu, pri ktorej tieto návrhové vzory používame. V tejto kapitole popíšeme funkcionalitu aplikácie a návod na to, ako ju používať. Implementáciu jednotlivých návrhových vzorov si vysvetlíme v ďalších kapitolách.

1.1 Inštalácia a spustenie

Na spustenie našej aplikácie je potrebné urobiť niekoľko krokov. Z nášho github repozitára stiahneme zdrojový kód lokálne do počítača na tomto linku.

Najdôležitejšia prerekvizita je stiahnutie a nainštalovanie Node. To môžeme urobiť cez inštalátor alebo zdrojový kód tu alebo cez package manager s príkazmi tu. Vďaka inštalácii Node vieme používať package manager nazývaný npm.

Na naše návrhové vzory potrebujeme používať plnohodnotne triedy, kvôli čomu sme si zvolili používanie Typescript namiesto Javascript. Kvôli tomu potrebujeme nainštalovať Typescript v projekte, čo zabezpečujeme cez náš *package.json*, ktorý je súčasťou programu. Na inštalovanie všetkých dependencies vrátane Typescriptu je potrebné spustiť cez konzolu v priečinku so súborom *package.json* príkaz `npm install`.

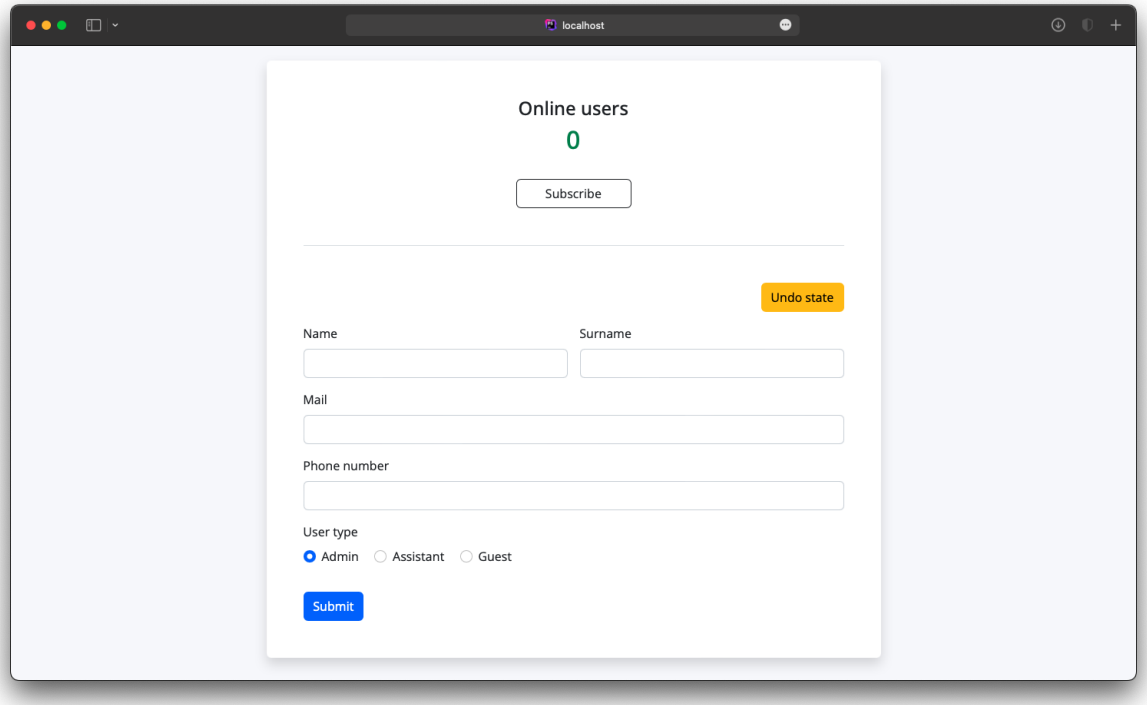
Pre spustenie aplikácie je potrebné najprv spustiť Typescript kód pomocou príkazu cez konzolu `npx tsc`. Následne keď otvoríte HTML súbor, zobrazí sa vám naša webová aplikácia v zvolenom prehliadači na localhost-e, teda lokálne u vás v počítači.

1.2 Používateľská príručka

Používanie našej aplikácie je veľmi jednoduché. Ide o jednoduchý formulárový dokument (viď Obr.1). Základná funkcionalita tkvie v tom, že viete vypĺňať polia formuláru ako sú krstné meno, priezvisko, mail, adresa, telefónne číslo atď.

Tento formulár vieme odoslať cez tlačidlo *send* pod textovými vstupmi formulára. Odoslanie sa nedeje reálne a dostanete len informáciu o tom kam sa súbor odoslal a aké informácie sme poslali.

Na spodku dotazníka viete prepínať medzi jednotlivými rolami: *admin*, *assistant* a *guest*. Pri zmene role sa taktiež premaže formulár. K dispozícii máme aj tlačidlo *Undo state*, ktoré vráti predošlú rolu aj s vyplneným formulárom v takom stave, akom bol pred zmenou role.



Obr. 1: Screenshot našej webovej aplikácie

2 Mediator pattern

Mediator je návrhový vzor, vďaka ktorému sa vieme vyhnúť chaotickým závislostiam medzi objektami odstránením ich priamej komunikácie. Teda komponenty komunikujú nepriamo, prostredníctvom Mediator objektu, ktorý uľahčuje úpravy, rozšírenia a opätovné použitia jednotlivých komponentov [1].

2.1 Problém

Predstavme si formulár, ktorý obsahuje textové polia a tlačidlá. Chceme dosiahnuť aby na stlačenie tlačidla „Mám dieťa“ sa nám zobrazilo textové pole na zadávanie mena dieťaťa. Iné tlačidlo by zas validovalo zadané vstupy. Ak by sme túto logiku mali implementovanú rovno v kóde komponentu (textové pole, tlačidlo), bolo by veľmi náročné si tieto komponenty prepoužívať v iných formulároch. Napríklad bolo by potrebné vykonať zmeny v komponente tlačidla, ak to tlačidlo si prepoužijeme v inom formulári a jeho úloha je zobrazovať textové pole pre zadávanie mena dieťaťa [1].

2.2 Riešenie

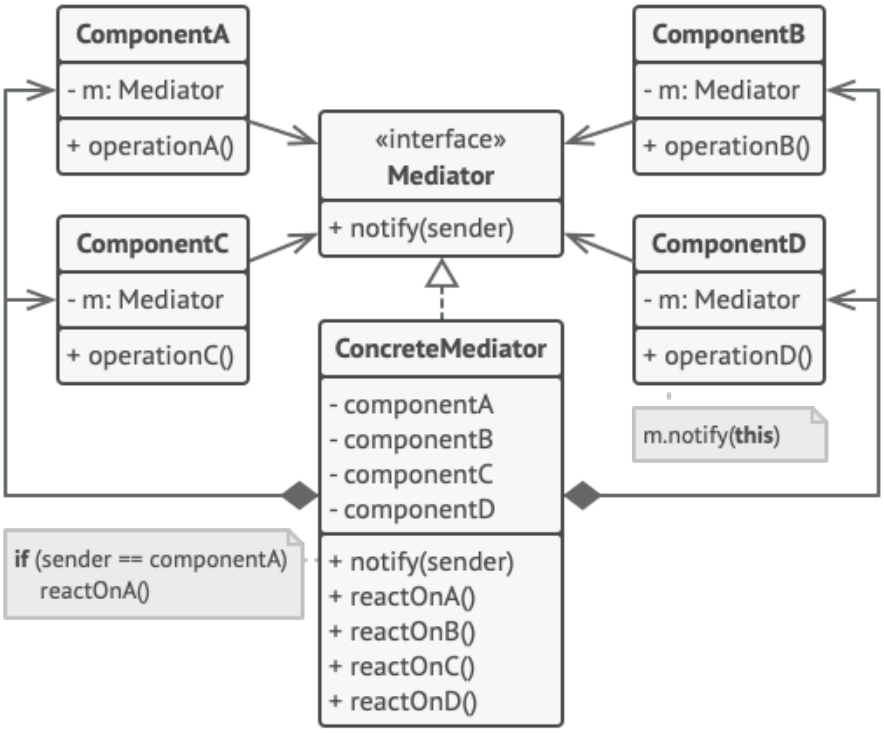
Vyššie spomenutý problém rieši Mediator pattern tak, že komponenty musia komunikovať nepriamo cez Mediator objekt. Teda komponenty sú závislé na jednom Mediator objekte a nie na viacero iných komponentov. Napríklad ak budeme chcieť na stlačenie tlačidla zobraziť textové pole pre zadávanie mena dieťaťa alebo zvalidovať celý formulár upozorníme Mediator, že kto ako komponent som, a čo chcem vykonať. Zadaný mediator už bude pripravený na operáciu, ktorá sa má pri tejto situácií vykonať. Tlačidlo a príslušné textové polia sa zbavili viacerých závislosti a sú závislé len na Mediator objekte.

2.3 Využitie vzoru

Mediator pattern nie je veľmi populárny na použitie, pretože je lepšie siahnuť po Observer patterne 3. Hlavným cieľom Mediator patternu je eliminovať vzájomné závislosti medzi komponentami. Tým pádom sa tieto komponenty stanú závislými od jedného objektu Mediatora. Cieľom Observer patternu je vytvoriť dynamické jednosmerné spojenia medzi objektami, kde niektoré objekty sú podradené iným. Existuje aj populárna implementácia Mediator patternu, ktorý sa spolieha na Observer. Objekt Mediator patternu má úlohu publishera a komponenty fungujú ako subscriberi, ktorý vykonajú prihlasovanie a odhlasovanie od udalosti Mediatora. Pri tejto implementácií sa Mediator veľmi podobá na Observer.

Avšak aj pri jeho nízkej popularite je najčastejšie používaný na uľahčenie komuni-

kácie medzi GUI komponentami aplikácie. Mediator pattern môže byť aj synonymom Controllera z MVC patternu [1]. Na obrázku 2 vidíme UML class diagram, ktorý popisuje štruktúru použitia Mediator patternu. Triedy *ComponentA*, *ComponentB*, *ComponentC* a *ComponentD* si môžeme predstaviť ako GUI komponenty, ktoré reagujú na nejaké udalosti ako kliknutie, zmenu a veľa ďalších. Všetky tieto komponenty sú asociované s Mediatorom, takže v objekte komponentu máme prístup ku Mediatorovi. To nám umožnuje upozorniť Mediatora, že nastala zmena a je potrebné vykonať príslušné operácie či už v aplikácií alebo na iných komponentoch. Všimnime si ale aj, že Mediator je v kompozícií s komponentami, čo znamená, že objekty komponentov žijú len v Mediatorovi. Teda ak zanikne Mediator, zaniknú aj všetky komponenty, ktoré v ňom žili. Vďaka tejto kompozícií je Mediator schopný vykonávať zmeny na jednotlivých komponentoch. Ak by sme potrebovali vykonávať iné zmeny, napríklad v nejakom inom formulári, zmeníme len objekt Mediátora a do objektov komponentov nie je potrebné zasahovať. Týmto Mediator plní svoj hlavný účel. Interface Mediator, ktorý taktiež vidíme na UML diagrame (viď Obr. 2) slúži na to, aby bolo zabezpečené, že každá Mediator trieda bude mať implementovanú *notify()* metódu. Táto metóda bude volaná komponentami a bude rozhodovať pri akej udalosti sa má vykonať daná zmena alebo operácia.



Obr. 2: UML class diagram štruktúry použitia Mediator patternu

2.4 Výhody a nevýhody

[1] Mediator pattern nám prináša veľa výhod:

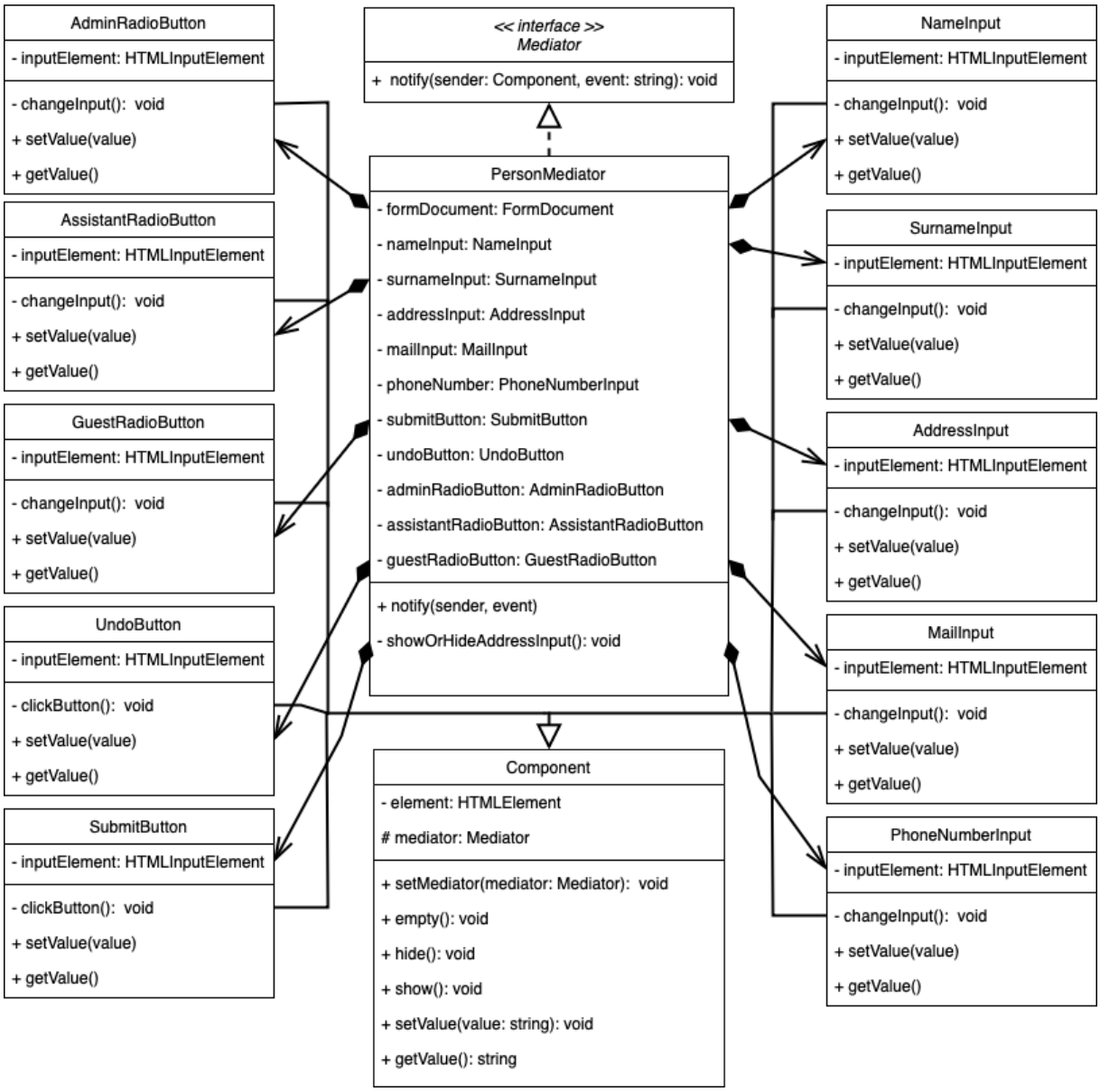
- Single responsibility principle - Komunikácia medzi komponentami sa extrahuje na jedno miesto čo uľahčuje udržiavanie kódu a lepšiu rozšíriteľnosť.
- Open/Closed principle - Vieme jednoducho vytvoriť a používať nový Mediator objekt bez toho, aby sme museli niečo meniť v komponentoch.
- Obmedzíme priame prepojenia medzi komponentami.
- Jednoduchšie prepoužitie komponentov.

Mediator pattern má jednu veľkú nevýhodu a to, že sa vie veľmi rýchlo stať God objectom. God object je anti-pattern, ktorý v sebe obsahuje veľa rozličných typov tried a má veľa nekategorizovateľných metód. S touto nevýhodou sme sa stretli aj pri našej aplikácií.

2.5 Implementácia

V našej aplikácií je Mediator objekt hlavný stavebný prvok. Uchováva všetky komponenty a taktiež aj FormDocument objekt, v ktorom sa nachádza State aplikácie a list Memento objektov na vytvorenie snapshotov. Komponenty sú reprezentované prichytením DOM elementu ku premennej inputElement. Každý komponent dedí od abstraktnej triedy Component, ktorá hovorí že musia obsahovať objekt typu Mediator. Taktiež potomkovia triedy Component musia obsahovať metódu *setMediator(mediator: Mediator)* na nastavenie Mediator objektu a funkcie *empty()*, *hide()*, *show()*, *getValue()*, *setValue(value: string)* pre prácu s elementom.

Mediator interface nám zaručí aby každý Mediator objekt, ktorý ho implementuje, mal funkciu *notify(sender: Component, event: string)*. Táto funkcia je jadro Mediator objektu, pretože implementuje operácie, ktoré sa majú vykonať pri jednotlivých udalostiach komponentov. Pri vytvorení PersonMediator objektu sa do konštruktora nastavia vytvorené komponenty, ktoré už majú nalinkované príslušné DOM. V konštruktore sa týmto komponentom nastaví prostredníctvom funkcie *setMediator(mediator: Mediator)* Mediator objekt, v ktorého konštruktore sa nachádzame. Následne vedia už naše komponenty pri udalostiach ako zmena alebo kliknutie volať *notify* funkciu priradeného Mediator objektu, ktorý už rozhodne, aká operácia sa má vykonať. Táto implementácia je znázornená na UML diagrame nižšie (viď Obr. 5). Tento UML diagram kvôli obmedzujúcej veľkosti nezobrazuje všetky asociácie aplikácie, ale iba tie, ktoré sú dôležité pre Mediator pattern.



Obr. 3: UML class diagram Mediator patternu v našej aplikácií

Tak ako bolo aj spomínané pri nevýhodách Mediator patternu, tak aj v našej aplikácií sa začal PersonMediator pomaly podobať na anti-pattern God object. Začal sa veľmi nafukovať kódom a funkcia *notify* sa začala prepĺňať podmienkami, ktoré reagujú na udalosti komponentov.

Pre plné pochopenie a taktiež zobrazenie všetkých funkcionalít aplikácie si odporúčam pozrieť zdrojový kód na tomto linku.

3 Observer pattern

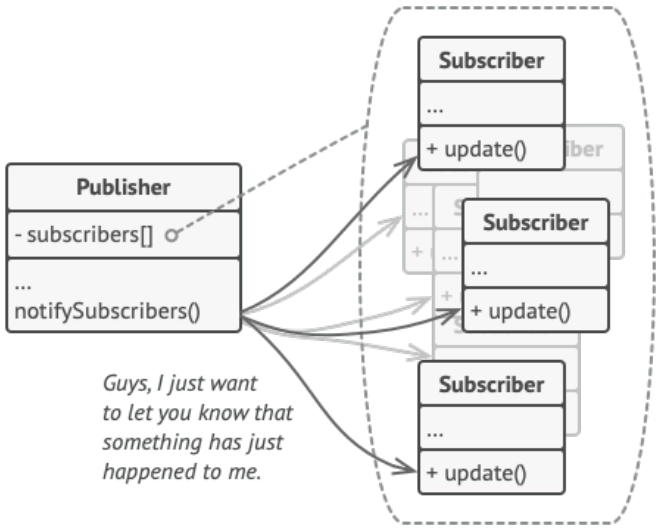
Observer je návrhový vzor, definuje mechanizmus, ktorý oboznámi všetky sledujúce objekty, že nastala zmena danej udalosti.[2]

3.1 Problém

Predstavme si, že máme dva objekty Obchod a Zákazník, pričom Zákazník si chce zakúpiť Iphone 14. Za predpokladu, že nevie v ktorom obchode je dostupný, potrebuje obísť všetky obchody a skúšať šťastie, čo je neefektívne. Ďalší problém nastáva, keď Obchod oznámi všetkým Zákazníkom, že si môžu dôjsť po Iphone 14, v tomto prípade Zákazník, ktorý chce Android, zbytočne dostáva informáciu, ktorá ho nezaujíma.

3.2 Riešenie

Spomínané problémy, vieme vyriešiť dvoma entitami Subject/Publisher a Subscribers. Subject je väčšinou objekt, ktorý drží nejaký zaujímavý stav a keďže rozosiela tento stav Subsrciberom, je taktiež volaný Publisher, v našom probléme to bude Obchod. Subscribers sú objekty ktoré čakajú na zmenu stavu Publishera.



Obr. 4: UML diagram použitia Observer patternu

3.3 Využitie vzoru

Typickým príkladom sú udalosti/akcie (frontendové aj backendové) a reakcie na nich, napríklad ak používateľ klikne na tlačidlo a musí sa niečo vykonať. Observers sa hodia aj v prípade, že je potrebné aby nejaký objekt sledoval iný objekt na určitý čas, napríklad, ak chceme vykonávať volanie API počas toho ako používateľ píše do textoveho poľa, ale

chceme tieto volania optimalizovať a vykonávať s oneskorením.

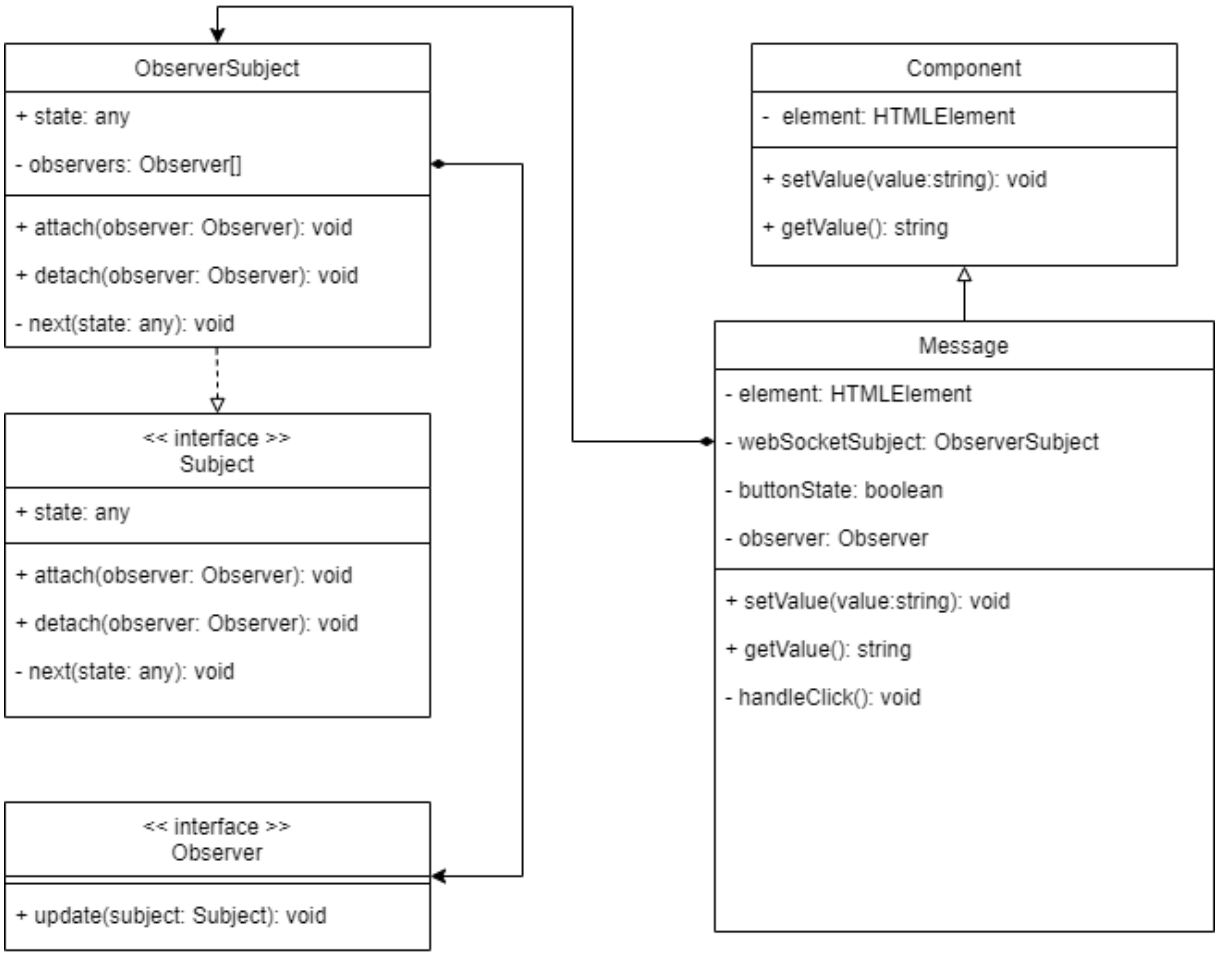
3.4 Výhody a nevýhody

- Nezávislosť kódu subscribera na kóde Publishera, vieme vypnuť/zapnuť počúvanie Publishera, bez zmeny jeho kódu
- Vzťahy medzi objektami, je možné stanoviť za behu

Nevýhodou je náhodné poradie, v ktorom Publisher oznamuje zmenu udalosti.

3.5 Implementácia

V našej aplikácii, sme využili observer na reprezentáciu socketových eventov. Po pripojení na socketový server, sa vytvorí Publisher, ktorý na prijatú správu reprezentujúcu počet aktívnych používateľov zavolá metódu next(), ktorá oboznámi všetkých Subscriberov o tom, že nastala zmena udalosti. Component Message vytvorí Subscribera, ktorý po zmene vyvolanej Publisherom, vypíše obsah správy zo socketov. Taktiež obsahuje tlačidlo, ktoré pridáva/odstraňuje daného Subscribera zo zoznamu príjemcov informácie od Publishera, konkrétne volá attach() a detach(), čím docielime zapnutie a vypnutie aktualizovania správy reprezentujúcej počet aktívnych používateľov.



Obr. 5: UML class diagram Observer patternu v našej aplikácií

4 State pattern

State, v preklade stav, je návrhový vzor, ktorý umožňuje meniť správanie objektu na základe jeho vnútorného stavu. Existuje viacero možností ako využiť vzor state [3].

4.1 Využitie vzoru

Dá sa použiť aj v algoritmickom programovaní na backende pre špecifické správanie objektu, aby sme zovšeobecnili fungovanie určitých funkcií, podľa údajov v objekte, teda jeho stave.

Najznámejšie použitie vzoru state je však na webových frontendových frameworkoch. Tie sú postavené, na výstavbe takzvaných komponentov. Každý komponent reprezentuje funkčnú časť webovej aplikácie, napríklad tabuľka, menu alebo formuláre. Moderné frameworky majú implementovaný state ako základný princíp programovania ale trochu v inom význame ako používame my.

Vo frameworkoch je State reprezentovaný len ako objekt s uloženými údajmi. Vždy keď sa tento objekt zmení, celý komponent sa znovu vyrenderuje s novými údajmi z objektu. Framework automaticky pri zmene volá funkcionalitu render, ktorá vytvorí komponent podľa údajov v stave. Náš príklad, ktorý sa dá používať v rôznych algoritmoch, využíva tiež tieto princípy, len jednoduchšie na pochopenie [3].

4.2 Problém

V programovaní sa často využíva takzvaný state machine, ktorý je prechodca state pattern. State machine je vo svojej podstate konštrukcia if-ov alebo switch, ktoré na základe stavu objektu, vykonáva inú funkcionalitu.

Predstavme si, že pečieme. Pokiaľ náš stav je *pizza*, zapne sa trúba na 150°C a pokiaľ náš stav je *lasagne*, zapne sa trúba na 200°C. V state machine toto určujeme pomocou použitia if-ov a switchu.

Tento prístup je jednoduchý a vhodný pre malý počet stavov. Pokiaľ ale existuje veľký počet stavov, veľmi sa nám rozširujú podmienky alebo switch. Tým sa stáva kód veľmi robustný a nečitateľný, čím vzniká problém s rozširovaním funkcionality aj rizikovosťou vzniku chýb. Čím väčší program je, tým nevhodnejší je state machine na použitie.

if $state \geq pizza$ **then**
  $bakeWithLowTemperature()$
else if $state \geq lasagne$ **then**
  $bakeWithHighTemperature()$
end if

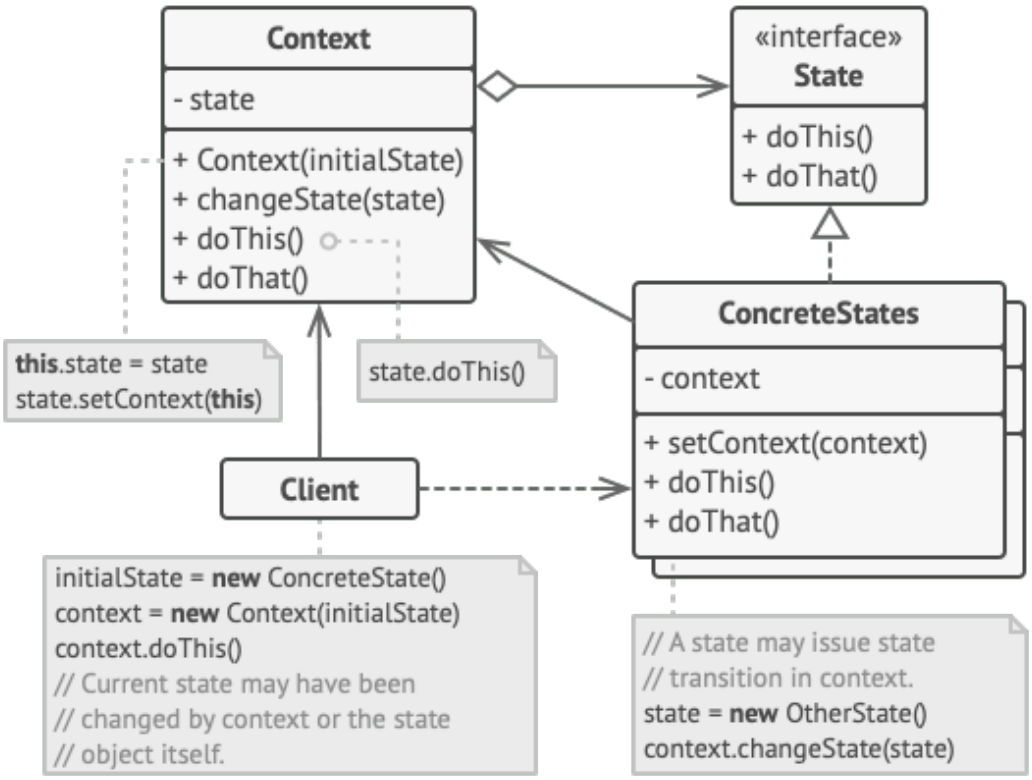
4.3 Riešenie

Pri väčších programoch s viacerými možnosťami stavov, je state machine nevhodný. Ako riešenie vznikol návrhový vzor *state*. State je druh objektu ktorý je implementovaný medzi konkrétne druhy stavov. Drží definíciu tej istej funkcie naprieč konkrétnymi stavmi ale každý jeden konkrétny stav, ktorý implementuje z nadradeného State, má inú implementáciu tejto funkcie. [3]

State by sa mal nachádzať v takzvanom kontexte, v ktorom sa menia konkrétne stavy implementujúce tento State. V našom programe existujú takzvane AdminState, AssistantState a GuestState. Ich funkcionalitu a implementáciu preberieme neskôr.

Vďaka tomu návrhovému vzoru, vieme v kontexte meniť konkrétny State (stav) podľa potreby a volať tú istú definíciu funkcie, ktorá bude podľa zvoleného State robiť vždy niečo iné.

Na nižšie zvolenom UML diagrame vidíme implementáciu State [3]. Naše konkrétne stavy ktoré implementujú State sú DraftState, PublishedState, ...atď. Documnet funguje ako Context, ktorý drží State a cez ktorý vyvolávame zmenu stavu ako aj volanie definovaných funkcií v State.



Obr. 6: UML diagram použitia State v Context [3]

4.4 Výhody a nevýhody

[3] State pattern má mnohé výhody oproti state machine napríklad:

- Single responsibility principle - Princíp, pri ktorom je kód pekne organizovaný do tried podľa stavov komponentov či celého systému, vďaka čomu vieme volaním jednej funkcie, vykonať vždy inú funkcionalitu.
- Open/Closed principle - Princíp, pri ktorom vieme voľne pridávať stavy bez menenia Context-u alebo už existujúcich State-ov
- Zjednodušuje kód Context-u eliminovaním zložitých prebytočných podmienok v state machine

State pattern má minimum nevýhod ale vieme spomenúť napríklad:

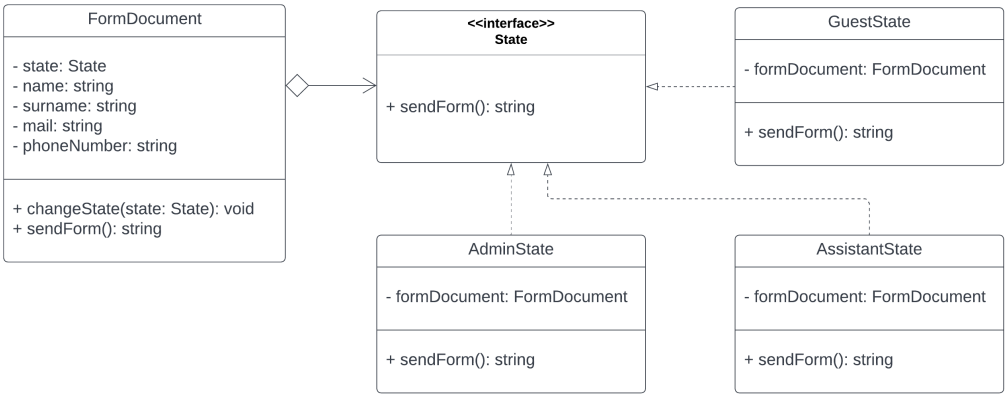
- Aplikovanie tohto návrhového vzoru môže byť zbytočné pri malom počte stavov v jednoduchých aplikáciach, kde postačuje state machine.

4.5 Implementácia

V našej aplikácii sme našli využitie pre návrhový vzor State pri prepínaní užívateľských režimov. Viete si zvoliť jeden z 3 užívateľských režimov: admin, assistant a guest. Vždy pri prepnutí režimu sa formulár premaže. Po stlačení tlačidla na odoslanie formulára, sa formulár odošle inam podľa zvoleného režimu. Namiesto použitia state machine a teda viacerých podmienok, jednoducho implementujeme tieto tri režimy ako State objekty AdminState, AssistantState a GuestState. Všetky zdieľajú rovnaká funkciu sendForm(), v každom stave ale robí niečo iné.

Formulár sa reálne nikam neodošle ale vypíše sa v aplikácii iný text podľa toho v akom režime sme. Text nás informuje o tom kam sa posiela formulár. Zatiaľ čo admin ho ukladá priamo v databáze, assistant ho posiela adminovi a guest ho posiela na automatizovanú kontrolu.

Ako náš Context, ktorý drží state a dodatočné informácie vhodné pre používanie State, sme zvolili abstraktnú reprezentáciu formulára cez triedu FormDocument. Tu ukladáme aj hodnoty z poľa. Toto bude mať pre nás zmysel neskôr pri návrhovom vzore Memento. Náš State nám tu vyjadruje, v akom užívateľskom režime sme a teda čo sa udeje po zavolaní funkcie sendForm(). Bližšiu štruktúru si viete pozrieť v tomto UML diagrame, ktorý pracuje s využitím State v našom príklade.



Obr. 7: Implementácia vzoru State v našej aplikácii

5 Memento pattern

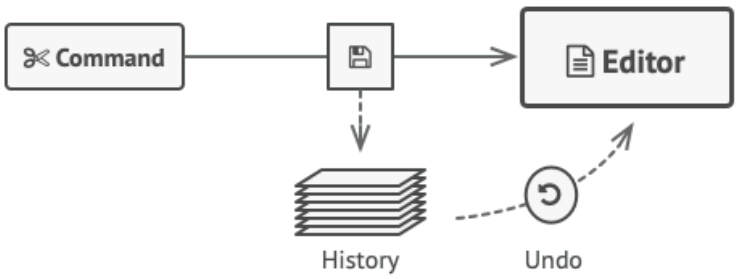
Memento rieši na ukladanie histórie. Po určitej akcii uloží stav (state) objektu. V prípade potreby obnoví predchádzajúci stav. Zároveň nerieši, akým spôsobom sa tento stav zmenil [4].

5.1 Návrh

Ako bolo spomenuté už aj pri inom patterne v tejto dokumentácii, tak aj tu si môžeme predstaviť vytváranie aplikácie textového editora. Bude v nej využívané editovanie textu, jeho formátovanie, vkladanie obrázkov a niekoľko ďalších akcií, ktorými je možné meniť stav editora.

Po vykonaní viacerých operácií budeme chcieť tieto akcie vrátiť. Keďže memento nerieši, akým spôsobom sa stav zmenil, tak pred každou úpravou sa zaznamená stav editora a uloží sa do úložiska. Ak sa vyžiada o vrátenie zmeny, nevráti sa operácia, ale natiahne sa posledný uložený stav. Takže keď sa stav editora reprezentuje objekt, tak sa celý jeho stav uloží pri každej zmene (pokiaľ si to tak definujeme) a po vyžiadaní sa obnoví stav objektu editora na posledne uložený v histórii.

Ukladanie stavu editora sa nazýva „snapshot“, pretože sa vytvorí akoby snímka stavu objektu, ktorá sa neskôr priamo bez komplikovaného zásahu nahradí za aktuálne zmenený stav objektu.



Obr. 8: Pred vykonaním operácie sa uloží stav editora do histórie [4]

5.1.1 Problém

Otázkou však je, ako takýto snapshot vytvoriť. Veľmi jednoduché riešenie by sa mohlo zdať, že pred vykonaním nejakej akcie si iný objekt skopíruje všetky prvky objektu editora, ktoré následne uloží do úložiska. Takéto zmýšľanie by bolo dobré, pokiaľ by všetky prvky editora boli verejné, teda mali stav `public`. Avšak pri programoch nechceme, aby bol stav editora takto voľne dostupný, aby ho nemohli rôzne objekty meniť.

Aj keď by to bolo nebezpečné riešenie a implementovali by sme ho, tak by nastal iný

problém. Pokiaľ by sa v postupnom vyvíjaní aplikácie rozhodlo určité vlastnosti objektu zrušiť, alebo naopak, o ďalšie rozšíriť, nestačilo by upraviť len triedu editora. Zmena by bola potrebná všade, kde sa k objektu editora pristupuje.

Ďalší problém vzniká, keď chceme ukladať históriu snapshot-ov. Keďže s najväčšou pravdepodobnosťou sa ukladá viacero hodnôt (text, pozícia kurzora, štýl...), tak sa pre uloženie snapshotu vytvorí nový objekt a tieto novovytvorené objekty sa budú ukladať do poľa. Aby sme mohli k hodnotám pristúpiť pri vyžiadaní obnovy, teda z iného objektu, museli by mať `public` stav. A znova vzniká problém, že aj objekty, ktoré by nemali mať prístup k snapshotom, tak vedeli by sa dostať k celej histórii. Takže buď to nebude bezpečné riešenie, alebo sa k hodnotám nebude dať dostať.

Tieto problémy sú spôsobené kvôli „broken encapsulaion“. Niektoré objekty sa snažia robiť viac, ako by sa predpokladalo. Aby získali údaje, zasahujú do priestoru iných objektov, ktorý by mal byť `private`, avšak pre fungovanie sa ich stav musel nastaviť na `public`. [4].

5.1.2 Riešenie

Memento pattern šikovne obíde problém „broken encapsulaion“, aj keď Snapshot stavu objektu sa uloží do oddeleného objektu, pričom niektoré jeho prvky majú `private` stav. Tieto uložené Snapshot-y sa budú využívať aj na obnovenie.

Memento pattern necháva vytvorenie snapshotu pre uchovanie svojho stavu na samotnom objekte. Ak si chce ďalší objekt kopírovať stav editora „zvonku“, snaphost vytvorí samotná trieda editora, ktorá má k nemu prístup. Preto má označenie „Originator“. Zároveň môže svoj stav obnoviť na základe uloženého snapshoot-u.

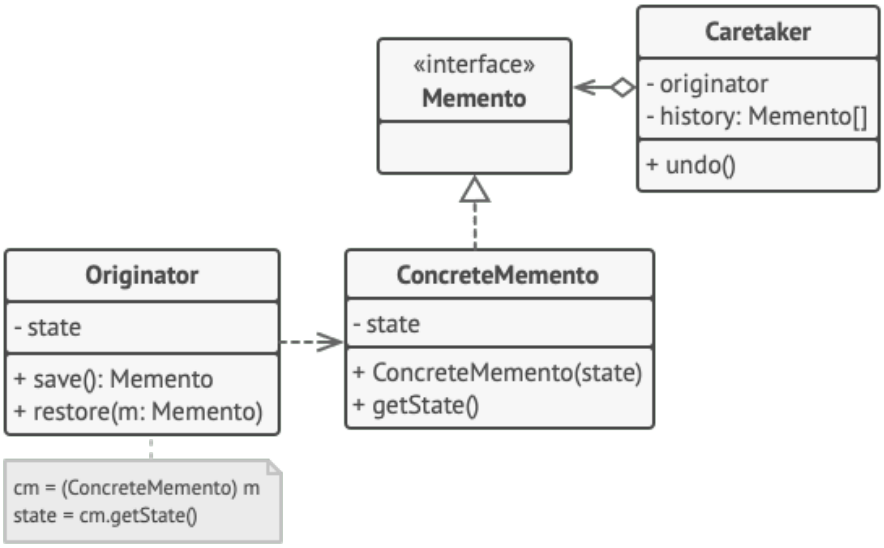
Tento snapshot sa uloží do objektu „memento“. Obsah mementa je prístupný iba pre objekt, ktorý ho vytvoril a zvyčajne je nemenný. Môžu sa do mementa doplniť metadáta snapshotu (názov, čas vytvorenia), ktoré budú prístupné aj pre ďalšie objekty.

O to, kedy a prečo sa má uložiť snapshot, sa stará „Caretaker“. Ten požiada objekt Originator, aby vytvoril snapshot a vytvorené Memento uloží do poľa, ktoré sa bude správať ako stack. Takže memento sa vždy ukladá na koniec poľa a po vyžiadaní o obnovu sa bude odkonca z poľa vyberať. To nám zabezpečí presúvanie sa po histórií postupne po najposlednejších predchádzajúcich stavoch. Vytiahnuté memento z poľa sa následne pošle objektu Originator, aby obnovilo svoj stav.

Pre obmedzenie prístupu k prvkom Mementa sa vytvára konvencia. Caretaker pracuje s triedou Memento iba cez interface Memento, v ktorom sú deklarované iba metódy, ku ktorým môžu mať prístup aj iné objekty ako Originator. To sú napríklad spomínané metadáta ako čas vytvorenia, či názov mementa. Originator môže pristúpiť k metódam

mementa priamo, keďže ho vytvoril, vďaka čomu sa dostane ku uloženému stavu.

Keďže priamo s mementom komunikuje iba Originator, to nám zabezpečí, že uložené údaje ostanú v bezpečí, pretože ostatné objekty k nemu nemajú prístup a nemôžu čítať uložený stav.



Obr. 9: UML class diagram štruktúry použitia Memento patternu [4]

5.2 Implementácia

V nami vytvorenej aplikácii si pomocou patternu Memento ukladáme stav formuláru. Keďže stav sa ukladá pred nejakou akciou, bolo na nás, ktorú si vyberieme. Taktiež sme si ich mohli zvoliť viac. Vhodné akcie pre ukážku by mohli byť po každom vyplnení elementu. Po každom vstupe z klávesnice, či kliknutí myši, ktorá vykoná zmenu. My sme sa pre jednoduchosť rozhodli ukladať stav pri každej zmene užívateľských režimov. Následne sa vieme vrátiť pomocou tlačidla „Undo state“. Vrátenie stavu nám spôsobí, že sa vo formulári zobrazia hodnoty, ktoré boli predtým, ako nastala zmena užívateľského režimu.

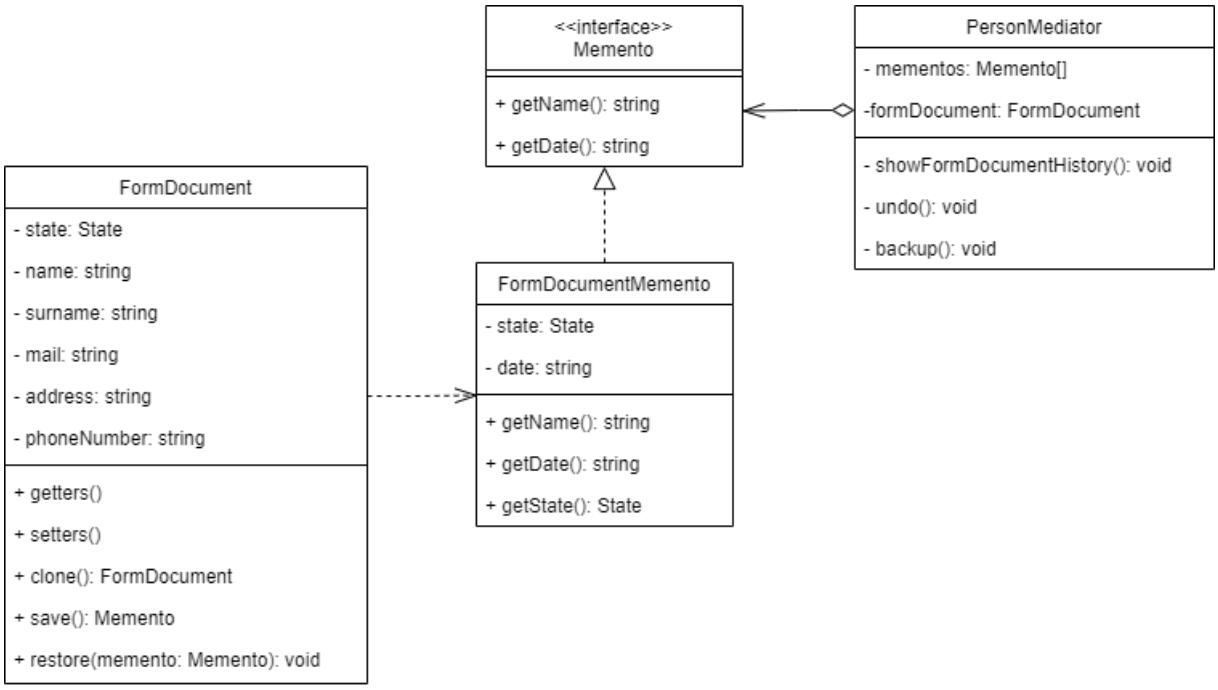
V našom programe to znamená, že si ukladáme stav abstraktnej triedy `FormDocument`. V podkapitole 4.5, ktorá patrí State patternu, bolo spomínané, že state drží abstraktná reprezentácia formulára cez triedu `FormDocument`. Keďže sme tieto patterny využili súčasne, reálne nám stačí uložiť atribút `state`, ktorý obsahuje referenciu na objekt triedy `FormDocument`. Vďaka tomu vieme získať ostatné hodnoty pre náš formulár. Rozšírnejšie vysvetlenie by bolo zbytočne komplikované. Preto, v prípade záujmu, je odporúčané pozrieť si kód dostupný k tejto dokumentácii. Pre jednoduchosť si môžeme predstaviť to, že si ukladáme všetky hodnoty patriace objektu `FormDocument`.

Originator je v našom programe trieda `FormDocument`. Pre ukladanie snapshot-u sa

vytvára objekt typu `FormDocumentMemento` cez metódu `save()`. Pokiaľ bude vyžiadané vrátenie stavu formulára, tak sa v *Originator-e* vyvolá metóda `restore()`. Pre jej vyvolanie bude potrebné dodať parameter typu `Memento`.

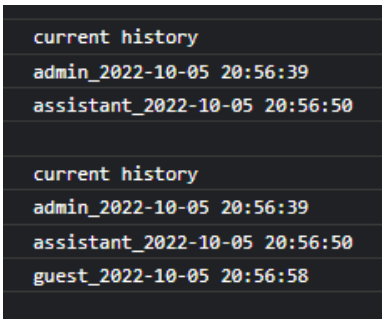
O to, kedy a prečo vytvoriť snapshot, rozhoduje trieda `PersonMediator`. Tým u nás získava označenie aj ako *Caretaker*. Ukladá históriu snapshotov v objektoch typu `Memento` do poľa `mementos`. Pred prepnutím užívateľského režimu použije metódu `backup()`, ktorá vyžiada Originator, aby jej vytvoril memento, ktoré si uloží. Teda sa zavolá metóda `save()`, patriaca triede `FormDocument`.

Pre vrátenie predchádzajúceho stavu formulára sa použije metóda `backup()`. Tá spôsobí to, že sa z poľa `mementos` vytiahne posledne uložené memento, ktoré sa pošle ako parameter metóde `restote()`.



Obr. 10: UML class diagram Memento patternu v našej aplikácii [4]

Caretaker môže komunikovať s mementom len obmedzene. Iba cez interface `Memento`. Ten mu povoľuje prístup iba ku metadátam (dátum a meno). Takúto komunikáciu sme využili pre vypísanie uloženej histórie stavov formulára pomocou metódy `showFormDocumentHistory()`. Výpis sa zrealizuje v konzole pri každej zmene v histórii.

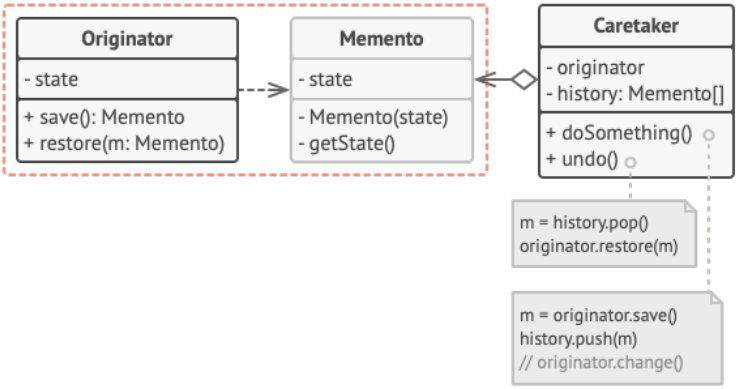


Obr. 11: Výpis histórie v konzole aplikácie

5.3 Využitie vzoru

Takýmto príkladom implementácie patternu Memento sme vhodne ilustrovali jeho štruktúru. Z akých tried pozostáva, aký vzťah majú medzi sebou a ako ich využívať.

Bohužiaľ pri dynamických jazykoch, akým je aj Typescript, nemôžeme zaručiť, že stav v memente zostane nedotknutý. Pri jazykoch, ako je napríklad Java, ktoré podporujú vnorené triedy, by sa trieda Memento vložila do Originator-a a nebol by potrebný interface pre komunikáciu s triedou Caretaker.



Obr. 12: 2. UML class diagram štruktúry použitia Memento patternu pri jazykoch s podporou vnorených tried [4]

5.4 Výhody a nevýhody

Výhodou patternu Memento je, že:

- rieši problém zapúzdrenia (Nedôjde k „broken encapsulaion“),
- Caretaker uchováva históriu stavu Originator-a, vďaka čomu je kód Originator-a jednoduchší.

Bohužiaľ takéto spracovávanie histórie prináša aj nevýhody:

- môže spotrebovať veľa pamäte RAM, pokiaľ sa uloží príliš veľa Snapshot-ov,
- väčšina dynamických programovacích jazykov (PHP, Python a JavaScript) nemôže zaručiť, že stav v memente zostane nedotknutý.

Záver

V našej seminárnej práci sme si ako tím naštudovali návrhové vzory, konkrétne behavioral patterns Mediator, Observer, Memento a State. Naštudovali sme si, aké problémy vznikajú pri komunikácií medzi objektami a ako naše patterny tieto problémy riešia. Pred implementáciou našej aplikácie sme uvažovali nad programovacím jazykom, v ktorom aplikáciu implementujeme. Keďže náš tím má blízko k webovým technológiám a radi sme chceli mať aj grafické rozhranie, vybrali sme si programovací jazyk Typescript.

Po naštudovaní teórie a štruktúry jednotlivých patternov sme si navrhli aplikáciu prostredníctvom UML class diagramov. Každý člen následne vytvoril UML class diagram patternu, ktorý mal na starosti. Tieto UML class diagramy sme pospájali do jedného veľkého (viď Príloha A), ktorý tvoril štruktúru našej aplikácie.

Implementovali sme aplikáciu, ktorá slúži na vyplnenie dotazníka s osobnými údajmi, kde je možné prepínať medzi typmi užívateľov a taktiež sledovať, koľko užívateľov práve vypĺňa dotazník. Na uloženie údajov z formulára a pri prepínaní užívateľov a odosielaní formulára sme využili State pattern, ktorý rozhoduje akým spôsobom sa formulár odošle. Mediator nám slúži na prepojenie všetkých komponentov formulára a reaguje na jednotlivé zmeny komponentov. Pri prepnutí užívateľa sa nám celý formulár vymaže a Memento nám zabezpečí uloženie stavu formulára, ktorý bol vyplnený pred prepnutím užívateľa. Tento uložený stav je taktiež možné z Mementa obnoviť. Zobrazovanie počtu práve pripojených užívateľov nám zabezpečuje Observer, ktorý je napojený na Websockety.

Teda naša práca vysvetľuje teóriu, využitie a štruktúru všetkých behaviorálnych vzorov, ktoré boli požadované v našom zadaní. Implementovaná webová aplikácia v Typescripte a k nej prislúchajúci UML class diagram (viď Príloha A) využíva a znázorňuje použitie týchto vzorov.

Zdroje

1. *Behavioral design patterns: Mediator pattern* [online]. [cit. 2022-10-03]. Dostupné z : `https://refactoring.guru/design-patterns/mediator`.
2. *Behavioral design patterns: Observer pattern* [online]. [cit. 2022-10-04]. Dostupné z : `https://refactoring.guru/design-patterns/observer`.
3. *Behavioral design patterns: State pattern* [online]. [cit. 2022-10-01]. Dostupné z : `https://refactoring.guru/design-patterns/state`.
4. *Behavioral design patterns: Memento pattern* [online]. [cit. 2022-10-03]. Dostupné z : `https://refactoring.guru/design-patterns/memento`.

Prílohy

A UML class diagram našej aplikácie II

A UML class diagram našej aplikácie