FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Analýza objektově orientovaných návrhových vzorů
Student:	Jan Šafář
Vedoucí:	Ing. Jiří Hunka
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Cílem práce je nastudovat a popsat nejvýznamnější objektově orientované návrhové vzory používané v oboru softwarového inženýrství. Výstupem bude teoretický přehled významných vzorů, včetně praktických ukázek jejich implementace – praktická část bude podrobně zdokumentována a veřejně dostupná, aby mohla sloužit jako výukový materiál pro začínající vývojáře.

Postupujte v těchto krocích:
1. Řádně nastudujte problematiku návrhových vzorů, včetně jejího rozdělení do základních kategorií (creational, structural, behavioral, concurrent, other).
2. Z každé kategorie zvolte několik konkrétních vzorů, které následně zanalyzuje (princip, výhody/nevýhody, případy užití, atd..).
3. Navrhněte a popište vhodný prototyp softwaru, na kterém bude demonstrovat použití jednotlivých návrhových vzorů, včetně jejich výhod i nevýhod, a to z hlediska implementace, rozšiřitelnosti a testovatelnosti.
4. Shrňte získané informace a zajistěte vhodnou přístupnost výsledků této práce pro začínající vývojáře jakožto studijního materiálu.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 19. září 2022 v Praze.

Bakalářská práce

ANALÝZA OBJEKTOVĚ ORIENTOVANÝCH NÁVRHOVÝCH VZORŮ

Jan Šafář

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jiří Hunka
2. ledna 2023

Obsah

Seznam obrázků

Seznam výpisů kódu

Seznam tabulek

Chtěl bych poděkovat především Ing. Jiří Hunkovi, za trpělivost při vedení mé bakalářské práce a cenné rady, které v průběhu zpracování poskytl. Dále bych rád poděkoval svým blízkým přátelům a rodině za jejich podporu během studia a kolegům za propůjčení čerstvé sady oči v závěrečných fázích tohoto projektu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 2. ledna 2023 .....

Abstrakt

Svět IT zažívá v několika posledních letech rozmach nepředstavitelných rozměrů. Pro ohromnou část civilizace se staly počítače neodmyslitelnou součástí lidský životů, ať už kvůli infrastruktuře, práci či osobním zájmům. Díky tomuto rozvoji se vývojářské pozice stávají velmi lukrativním a požadovaným zaměstnáním. Jeden z aspektů, který odděluje dobré vývojáře od špatných, je porozumění abstraktnějším myšlenkám a principům. Návrhové vzory, jejichž průzkumem se zabývá tato práce, jsou přesně odvětvím, jehož podstata uniká spoustě programátorů. Vyjma teoretických rozborů jednotlivých vzorů je v rámci praktické části vytvořen veřejně dostupný materiál, který slouží jakožto katalog praktický ukázek. Hlavní pointou za tímto katalogem je vytvoření alternativního výukového materiálu, který odráží reálné situace a má „vše pod jednou střechou“ — tj. veškeré příklady jsou postaveny nad jednotným kontextem. Tím je prototyp softwaru, jehož návrh je mimo jiné také součástí této práce.

Klíčová slova návrhový vzor, GoF, návrh aplikace, objektově orientovaný design, softwarová architektura, objektově orientované programování, prototyp softwaru

Abstract

The world of IT has been experiencing an explosion of unimaginable proportions over the last few years. For a huge part of our civilization, computers have become an integral part of human lives, whether it be because of infrastructure, work or personal interests. Due to this expansion, developer professions have become highly lucrative and sought-after. One aspect that separates good developers from bad ones is their ability to grasp more abstract concepts and principles. Design patterns, which are the main subject of this thesis, are precisely one of the more conceptual topics whose essence eludes many programmers. In addition to theoretical examinations of individual patterns, a publicly available material has been created in the practical portion, which serves as a repository of simulated real-world examples. The main idea behind this catalog is to develop an alternative educational medium, that reflects plausible situations while encapsulating everything „under one roof” — i.e. all examples are based on the same exact context of a certain software prototype. The prototype, amongst other things, is also covered within this thesis.

Keywords design pattern, GoF, application design, object-oriented design, software architecture, object-oriented programming, software prototype

Seznam zkratek

AI	Umělá inteligence (Artificial Intelligence)
API	Aplikační programovací rozhraní (Application Programming Interface)
BE	Serverová část aplikace (BackEnd)
CI/CD	Kontinuální integrace a kontinuální nasazení (Continuous Integration/Continuous Deployment)
CRC	Třída-Zodpovědnost-Spolupráce (Class-Responsibility-Collaboration)
DB	Databáze
DWH	Datový sklad (Data WareHouse)
FE	Klientská část aplikace (FrontEnd)
GRASP	Softwarové principy přidělování odpovědnosti (General Responsibility Assignment Software Principles)
GUI	Grafické uživatelské rozhraní (Graphical User Interface)
HW	Hardware
I/O	Vstup/Výstup (Input/Output)
IT	Informační technologie
JSON	JavaScriptový objektový zápis (JavaScript Object Notation)
OO	Objektově orientovaný
OOP	Objektově orientované programování
OS	Operační systém
SI	Softwarové inženýrství
SRP	Princip jedné odpovědnosti (Single Responsibility Principle)
SW	Software
TDD	Vývoj řízený testy (Test-Driven Development)
UI	Uživatelské rozhraní (User Interface)
XML	Rozšiřitelný značkovací jazyk (eXtensible Markup Language)

Úvod

V dnešní době s ohromně se rozrůstajícím IT sektorem je stále větší a větší poptávka po technicky zdatných lidech a vývojářích. Víceméně každá sebemenší společnost má své vlastní IT oddělení, jehož úkolem je udržovat a případně vyvíjet interní SW, což může v extrémnějších případech čítat až několik desítek či stovek produktů. Perfektním příkladem je segment bankovnictví, kde udržování celého ekosystému v rámci jedné banky zabere spolupráci několika různých týmů vývojářů, s tím že každý má na starosti svou vlastní sbírku aplikací. Takováto úroveň robustnosti vyžaduje, aby dané aplikace byly co nejoptimálněji navržené od počátku své existence a každá titěrná změna nebo nová funkčnost nevytvářela zbytečný *technický dluh*[1].

Většina nynějšího SW je vyvíjena pomocí OOP, zejména díky jeho schopnosti namapování se na objekty reálného světa a následné úspoře práce dosažené pomocí jejich znovupoužitelnosti. Návrh objektově orientovaného SW je ovšem velice těžký, a návrh lehce udržitelného objektově orientovaného SW je ještě mnohem těžší. Je potřeba nalézt relevantní objekty, zvolit vhodnou granularitu, určit mezi nimi vhodné vztahy a vyřešit desítky dalších obdobných problémů. Výsledný SW by zároveň měl být šitý na míru konkrétnímu účelu, ale zároveň natolik obecný, aby dovoloval adresovat budoucí problémy a požadavky. Jak je vidět nejedná se o žádný triviální úkol ani pro sebevíce zkušené vývojáře, nemluvě o nezkušených začátečnících, kteří se s ničím obdobným nesetkali.

Zde přichází na scénu návrhové vzory, jakožto jeden ze základních stavebních kamenů objektově orientovaného SW, se kterým by měl být obeznámen každý vývojář. Díky aktivnímu růstu IT sektoru v posledních letech, je poměrně nepravděpodobné střetnout se s problémem, se kterým se doposud nikdo nikdy nesetkal. Načež vyvstává otázka — proč se zabývat řešením něčeho, co již bylo vyřešeno, a snažit se takzvaně znovu vynalézt kolo? Nesmysl… Zmíněné návrhové vzory vznikly jako odpověď na nejtypičtější problémy vyskytující se v rámci OOP. Hlavní snahou individuálních vzorů je poskytnout efektivní řešení konkrétního problému, společně s ustanovením jeho výhod a potenciálních kompromisů.

V neposlední řadě stojí za zmínku, že ačkoliv návrhové vzory poskytují velmi solidní základ, nejsou žádným magickým nástrojem, který by vyřešil veškeré nesnáze. Pečlivé promyšlení všech okolností stále zůstává neopominutelným aspektem. Nestačí pouze slepě implementovat jeden vzor bez širšího pohledu na daný SW jakožto na celek. Stejně tak výraz „čím více vzorů, tím lepší software" rozhodně není univerzálně pravdivý. Nejdůležitější součástí vývoje stále zůstává rozum a zkušenosti samotného vývojáře.

[1] metaforický koncept, který popisuje upřednostnění rychlosti vývoje na úkor kvality

Cíl práce

Hlavním cílem této práce je podrobně rozebrat problematiku SI, kterou řeší objektově orientované návrhové vzory, a blíže seznámit čtenáře s konkrétními příklady z této oblasti.

V textu budou zpracovány jednotlivé vzory, které mohou být v praxi využity pro překonání širokého spektra překážek. Základem je rozbor několika nejvýznamnějších exemplářů ze standardních kategorií. Tyto vzorky budou pro úplnost rozšířeny i o atypičtější modely. Mezi předvybrané kategorie patří vzory týkající se tvorby objektů, vzory týkající se struktury programu, vzory týkající se chování, vzory týkající se souběžného řešení úloh a případně „nekategorizovatelné vzory".

Každý probraný vzor bude podrobně popsán a vysvětlen. Zanalyzuje se problém, který řeší, princip jeho fungování, jeho struktura a jeho výhody i nevýhody. Ve vhodných případech budou zanalyzovány i vzájemné vztahy s jinými vzory. Dále bude pomocí metodik SI navržen prototyp SW, který bude využit pro demonstrace jednotlivých návrhových vzorů. Tyto ukázkové příklady budou co nejlépe odrážet reálné situace a potenciální problémy. Pokud bude potřeba, budou v rámci ukázek zmíněny jak kladné, tak i záporné dopady spojené s využitím daného vzoru, a to ať už z hlediska implementace, rozšiřitelnosti či testování.

Závěrem a výstupem této práce bude veřejně dostupný katalog s přehledem probraných návrhových vzorů. Součástí tohoto katalogu budou demonstrační implementace, které budou v rámci navrženého prototypu simulovat jisté situace a problémy, k jejichž možným řešením lze využít individuálních vzorů.

Kapitola 1
Rešerše

Tato kapitola se zabývá základními informacemi potřebnými pro plnohodnotné porozumění zbytku práce. Do tohoto obsahu spadá zejména obecný popis OO návrhových vzorů a prozkoumáním problémů, které řeší. Na úvod je stručně vysvětleno samotné OOP a jeho základní principy. Následuje uvedení několika typických překážek, se kterými se lze v kontextu zmíněné programovacího paradigmatu setkat. Na závěr je čtenář seznámen s pár obecnostmi týkajícími se hlavní náplně celého textu, kterou jsou návrhové vzory.

1.1 Objektově orientované programování

V oboru informatiky existuje několik desítek různých přístupů k programování. Od nejznámějších, mezi které patří třeba *procedurální* či *funkcionální*, přes *logické*, *meta*, až po méně známé, kterým je kupříkladu *aspektově orientované*[1]. Následující pojmy se týkají nejrozšířenějšího typu — OOP.

1.1.1 Základní pojmy

Objektově orientované programování je nejpopulárnějším z programovacích paradigmat dnešní doby a většina vývojářů je s ním seznámena s jakožto jistým profesním standardem. Staví na konceptu *tříd* a *objektů*, pomocí kterých jsou data společně s chováním zapouzdřena do jedné škatulky [1] (tomuto přístupu se říká *class-based* a je nejrozšířenějším, avšak existují i jiné, kupříkladu *prototype-based*[2]). Hlavní myšlenka za tímto přístupem je umožnění vytváření komplexních generických struktur s možností snadné znovupoužitelnosti napříč kódem. Pro účely této práce je zcela nezbytné se orientovat v následujících základních pojmech [2].

Třída je abstraktní šablona definující atributy a metody, pomocí které jsou vytvářeny konkrétní instance objektů.

Objekt je konkrétní instancí nějaké třídy vytvořený pomocí konkrétních dat, které jsou následně uloženy v jeho atributech.

Atribut je součást objektu, ve které jsou uložená data. Stav každého objektu v daný moment je definován právě těmito daty.

Metoda je funkce dostupná pouze objektu daného typu (resp. třídy) reprezentující jeho chování. Může například vracet informace o objektu nebo měnit jeho atributy.

[1] velice podobné OOP, avšak přináší několik rozšíření; snaží se zvýšit modularitu využitím tzv. *aspektů*
[2] nemá třídy jako takové, ale využívá již existujících objektů jakožto prototypů; model JavaScriptu

```
class Person
{
  public:

    Person(const std::string &name_): name(name_)
    {}

    void greet()
    {
      std::cout << "Hello, " << name << std::endl;
    }

  private:
    std::string name;
};

Person p = Person("John");
p.greet();
```

Výpis kódu 1.1 Úryvek C++ kódu demonstrující základní pojmy OOP: *Person* (třída), *p* (objekt), *name* (atribut) a *greet* (metoda)

1.1.2 Základní principy

V neposlední řadě OOP stojí na 4 fundamentálních pilířích. Jedná se o zásadní myšlenky, bez jejichž pochopení je nemožné navrhnout významově korektní OO software. Zde je nutné podotknout, že problematika těchto principů by vydala na knihu sama o sobě, a proto jsou následující definice poměrně zjednodušené. Pokud s nimi tedy čtenář není vůbec seznámen, je pro plné porozumění potřeba blíže prostudovat citovaný článek *The Four Pillars of Object-Oriented Programming* [3] nebo nějaké jiné alternativní zdroje.

Abstrakce je pojem, který zachycuje skrytí složitých implementačních detailů za něco s jednodušším rozhraním. Pointa této vlastnosti spočívá ve zobecnění pohledů, přes které koncový uživatelé interagují s danými subjekty.

Zapouzdření je technika, která umožňuje skrývání informací objektu. Tedy dovoluje navenek vystavit pouze vybrané atributy a metody, a zbytek zachovat viditelný pouze uvnitř objektů daného typu.

Dědičnost je způsob, jakým lze stanovit *is-a*[3] vztah mezi objekty. Říká nám, že potomci třídy (také nazývané podtřídy nebo odvozené třídy) podědí atributy a chování rodičovské třídy (také nazývané nadtřídy nebo super třídy).

Polymorfismus je velmi obecný pojem, nejlépe vystihnutelný výrazem „mnohotvárnost". Je velmi úzce svázán s předchozími a jeho hlavní síla spočívá v rozmanitém přetěžování metod a možnosti vzájemného zaměňování tříd potomků a rodičů.

Téměř všechny populární programovací jazyky podporují do určité míry OOP. Díky svým vlastnostem dovoluje vývojářům navrhovat bezpečný a znovupoužitelný SW, a tím předcházet zbytečnému opakování práce. Nicméně ačkoliv má OOP ohromné množství výhod, tak naše realita je stále nedokonalým chaotickým místem a ani takto mocný nástroj není natolik univerzální, aby posloužil k vyřešení libovolné úlohy [4].

[3] vztah reálného světa říkající, že *Typ_A* je skutečně podtypem *Typ_B*; např. Člověk *is-a* Savec

1.2 Překážky při OO návrhu

Návrh objektově orientovaného SW není vůbec jednoduchým procesem. Návrh „korektního" objektově orientovaného SW je pak ještě mnohem složitější. Pod pojmem „korektní" je zde myšlen takový SW, který neporušuje žádné zásadní OO konvence a principy. Hlavním cílem by vždy mělo být vytvoření SW, který bude po celý svůj životní cyklus jednoduché udržovat v provozu i přes eventuální budoucí změny [5]. Dosažení takového výsledného produktu je v praxi často velmi náročné a vyžaduje rozsáhlé zkušenosti a zběhlost v OO návrhu. Vývojáři zabývající se touto problematikou se dennodenně musí vypořádávat s desítkami poměrně záludných překážek. Konkrétní problémy jsou blíže rozebrány v kontextu jednotlivých návrhových vzorů — pro uvedení čtenáře do problematiky je zde uvedeno pár častých obecnějších příkladů [5].

1.2.1 Identifikace objektů s vhodnou granularitou

Dekompozice systému na jednotlivé objekty je jedním z nejobtížnějších úkolů OOP, ale zároveň jedním z nejdůležitějších. Jejich správná volba poskytuje stabilní základ, od kterého se bude odvíjet všechen další vývoj. Tento proces bere v úvahu nespočet okolností: zapouzdření, závislosti, znovupoužitelnost, výkon, zabezpečení a mnohé další. Nejhorší na tom je, že leckteré z nich se mnohdy navzájem ovlivňují rozporuplným způsobem.

Metodologie OO návrhu uvádí několik různých řešení, každé vhodné za jiných podmínek. Jedním způsobem může být vyčlenění podstatných jmen (objekt) společně s přídavnými jmény (atribut) a slovesy (metoda). Dalším kandidátem může být návrh založený na principech *GRASP*[4] a jejich striktním dodržením nebo využití *CRC karet*[5]. Každá z těchto metod poskytne seznam potenciálních objektů, který ale stále nebude finální a bude vyžadovat dodatečné úpravy v následujících fázích OO návrhu [6].

1.2.2 Specifikace rozhraní objektů

Vlastností každého objektu je nějaké předdefinované chování. To je blíže popsáno jeho metodami, z nichž každá je přesně určena svou signaturou, tj. názvem, přijímanými parametry a návratovou hodnotou. Sadě všech signatur definovaných nad objektem se poté říká rozhraní objektu [5].

Rozhraní objektu je velice důležitý pojem, jelikož jednoznačně stanovuje, co přesně objekt umí a co k tomu potřebuje. Takže v momentě, kdy je rozhraní zveřejněno nějakému konzumentovi (cílovému klientovi), efektivně je mu oznámena jakási garance poskytovaných služeb. Toto je naprosto zásadní myšlenka, protože pojem rozhraní neříká nic o vnitřní implementaci — může tedy bez problému existovat více objektů se stejným rozhraním, které se ale „pod kapotou" chovají zcela odlišně. OOP poté za pomoci principů popsaných v sekci 1.1.2 dovoluje zaměňovat objekty se stejným rozhraním.

Návrh rozhraní je tedy stěžejní částí, jelikož změn v implementaci lze poměrně lehce docílit, zatímco změny v rozhraní s sebou nesou vynucené změny všude, kde se daný objekt objevuje (kupříkladu i na straně klienta).

1.2.3 Flexibilní návrh

Další z ohromných výzev je návrh takového SW, který bude maximalizovat znovupoužitelnost již zhotovených částí a zároveň co nejvíce usnadňovat budoucí vývoj. Klíčem pro dosažení těchto vlastností je, zaměřit se již v počátečních fázích návrhu na důkladnou analýzu směru, jakým by se SW po vydání své první verze měl a případně i mohl nadále ubírat. Pečlivý průzkum možných budoucích změn a požadavků na rozšiřující funkčnosti je často velmi opomíjenou fází.

[4] sada 9 elementárních principů OO návrhu a rozdělení zodpovědností

[5] brainstorming technika, kdy se procházením scénářů definují třídy a jejich interakce, pomocí jistých karet

SW, který tento aspekt zcela opomine, je ohromným rizikem pro kompletní přepracování či zahození, a tím i ziskovou ztrátu. Návrh natolik obecného SW, že ho bude možné plně rozvíjet odpovídajícím způsobem, ale zároveň každá sebemenší změna nebude spojená s tisící řádky, zbytečně se opakujícího kódu, je nejlepším možným výsledkem.

1.2.4 Antivzory

Širokou oblastí a jakýmsi protipólem této práce jsou tzv. *antivzory*. Ačkoliv se nevztahují pouze na OO návrh stojí za zmínku. Jedná se o katalog běžných defektivních procesů a implementací, které jsou většinou zcela neefektivní a nesou vysoké riziko, že ve výsledku budou dokonce až kontraproduktivní — v podstatě ukázky, co a jak přesně nedělat. Jejich hlavní snahou je upozornit na tyto opakující se problémy, pomoci rozpoznat co je může způsobovat (ať už neznalost vývojáře či aplikace perfektně dobrého návrhového vzoru pouze ve špatném kontextu) a navrhnout možná řešení. Existují mnohé antivzory týkající se jak samotného vývoje, tak i celých architektur. S konkrétními příklady se může čtenář obeznámit třeba na webu *SourceMaking* [7].

1.3 Návrhové vzory obecně

Nyní když je čtenář seznámen se základními pojmy OOP a má jistou představu o problémech, které mohou v průběhu OO návrhu nastat, je na čase otevřít hlavní náplň této práce, tj. návrhové vzory a vše okolo nich. Na začátek je uvedena trocha historie o jejich vzniku, následuje formální definice, kterou musí každý vzor splňovat a nakonec formát, ve kterém budou individuální vzory popsány v tomto textu.

1.3.1 Původ

První zmínky okolo návrhových vzorů a s nimi úzce spojené náležitosti (návrhový jazyk) se začaly objevovat již koncem minulého století, v dobách kdy ještě ani pořádně neexistovaly osobní počítače, jak je známe dnes. Velkou popularitu získaly po vydání knihy *Design Patterns: Elements of Reusable Object-Oriented Software* [5] skupinou *GoF*[6]. Tato kniha popisuje 23 základních návrhových vzorů rozdělených do 3 kategorii[7], které samozřejmě nejsou jedinými co existují. Od jejího vydání bylo nalezeno a zformalizováno mnoho dalších a jejich počet se stále rozrůstá [8].

1.3.2 Definice

Návrhový vzor je obecné opakovaně použitelné řešení nějakého běžného problému, se kterým se lze v průběhu návrhu a vývoje SW setkat [9]. Jak název napovídá nejedná se přímo o kus kódu, který lze pouze zkopírovat a problém vyřešen. Jedná se spíše o generický koncept, který popisuje myšlenku za řešením nějaké překážky. Vývojář poté může následovat daný vzor a implementovat modifikované řešení, které bude zcela přizpůsobeno konkrétnímu SW. Vyjma popsané definice musí každý návrhový vzor formálně obsahovat 4 esenciální prvky [5]:

Název je nejtriviálnější, ale zároveň nejdůležitější součástí. Dovoluje nám vyjádřit problém, řešení a potenciální následky jedním nebo dvěma slovy. Zároveň rozšiřuje naši „vývojářskou slovní zásobu", což přináší vyšší úroveň abstrakce a umožňuje efektivnější komunikaci.

Problém popisuje, kdy je vhodné daný vzor použít. Vysvětluje potenciální překážku a její kontext — může popisovat algoritmický problém, problémové struktury nebo přímo poskytovat seznam nutných podmínek pro smysluplné využití vzoru.

[6] Gang of Four — Erich Gamma, Richard Helm, Ralph Johnson a John Vlissides

[7] poznámka pro čtenáře — ačkoliv tato práce z části využívá stejnou kategorizaci, individuální kategorie neobsahují pouze vzory uvedené v „*GoF* knize"

Řešení popisuje elementy, které dohromady tvoří abstraktní návrh řešící daný problém společně s jejich vztahy a odpovědnostmi. Co tato sekce neposkytuje je konkrétní návrh a implementaci, jelikož vzor je spíše jakási šablona, kterou lze využít ve více různorodých situacích.

Důsledky jsou výslednými kompromisy, které jsou nevyhnutelné při použití daného vzoru. Často mezi ně patří optimalizace typu paměť vs. čas. Jelikož znovupoužitelnost je ohromným faktorem OOP, často také odráží kompromisy spojené s dopadem na flexibilitu a rozšiřitelnost. Ačkoliv jsou tyto důsledky často přehlížené, jsou velmi důležitými pokud chceme předejít zbytečným chybám.

1.3.3 Struktura

Nalezení uceleného způsobu, jakým reprezentovat jednotlivé návrhové vzory je zcela zásadní pro zaujetí čtenáře a nematoucí výklad. Grafické notace, ač velice důležité a užitečné, jsou samy o sobě poměrně nedostatečné. Jak již bylo zmíněno v předchozí sekci, vzory s sebou nesou velké množství informací, ze kterých není dobrým nápadem cokoliv opomenout. Z tohoto důvodu budou jednotlivé vzory v následující kapitole popsány tímto formátem:

Pojmenování a klasifikace je první částí, kde je ve dvou až třech krátkých větách uveden seznam přidružených jmen, zařazení do příslušné kategorie a stručné shrnutí, co lze od vzoru očekávat.

Motivace je druhou částí, ve které je více přiblížen problém, který návrhový vzor řeší. Jsou zde zmíněny situace a případné okolnosti, které mohou sloužit jako indikátory možného využití rozebíraného vzoru. V neposlední řadě je zde daný problém předveden na fiktivním scénáři.

Řešení je další a diskutabilně nejdůležitější částí. Jak napovídá název, nachází se zde detailní popis řešení — konkrétně tu bývá uveden diagram zachycující veškeré objekty/třídy, které jsou podstatné v rámci návrhu jakožto celku. Součástí je též bližší popis jejich podstaty, odpovědností a vzájemných vztahů.

Zhodnocení je částí, ve které jsou uvedeny potenciální kompromisy, ke kterým může dojít s implementací daného vzoru. Respektive jsou zde bodově shrnuty hlavní výhody a nevýhody.

Zajímavosti jsou poslední a nepovinnou částí. Obsahově jsou tu potenciální dodatečné informace, které je dobré znát a brát v potaz. Těmi jsou zejména vztahy s ostatními vzory, úskalí konkrétních implementací a nápovědy, jak jim předejít.

Kapitola 2

Analýza návrhových vzorů

Tato kapitola je největší částí celé práce a její hlavní náplní je pečlivý rozbor individuálních vzorů. Výstupem této činnosti je mini-dokument, který zachycuje podstatu daného vzoru dle obecného formátu popsaného v sekci 1.3.

Práce čerpá velkou inspiraci z již zmíněné knihy *Návrh programů pomocí vzorů — stavební kameny objektově orientovaných programů* [5], která je považována za jakousi „bibli návrhových vzorů". Z tohoto důvodu je základem stejná kategorizace vzorů, tj. vzory týkající se tvorby objektů *(Creational)*, vzory týkající se struktury programu *(Structural)* a vzory týkající se chování *(Behavioral)*. Vyjma těchto se zde zavádí také další kategorie, do které spadají vzory týkající se souběžného řešení úloh *(Concurrent)*. V některých zdrojích informací je možné se setkat také s kategorií *architektonických vzorů*. Ta ovšem do značné míry koliduje s námětem architektonických stylů jako takových, což je ohromné téma samo o sobě, a proto není v rámci tohoto textu blíže probírána.

Dále je zde vhodné podotknout, že kategorizace vzorů nedefinovaných skupinou *GoF* je poměrně volatilní záležitostí a mezi vývojáři kolují rozporuplné názory, který z přístupů je ten správný. Někteří zastávají názor, že originální skupiny by se neměly měnit a všechny novější vzory označují jako „nezařazené", případně je škatulkují do zcela nových kategorií. Jiní naopak normálně obohacují již stávající skupiny. Jelikož kniha napsaná skupinou *GoF* [5] nikde neuvádí, že by jimi představené kategorie měly být konečnými uzavřenými množinami, tento text se staví k problému způsobem rozšiřování stávajících kategorií — tj. novější vzory zařazuje do vhodných skupin, které odpovídají jejich podstatě.

Obecné informace vyskytující se v následujících rozborech jsou založeny zejména na vědomostech získaných z těchto materiálů (není-li uvedeno jinak): *Refactoring.Guru* [10], *Návrh programů pomocí vzorů — stavební kameny objektově orientovaných programů* [5], *Java Design Patterns* [11], *Software-Pattern.org* [12].

2.1 Vzory týkající se tvorby objektů (Creational)

První z kategorií, kterou poprvé představili členové skupiny *GoF*. Pokrývá návrhové vzory, které různými způsoby a mechanizmy umožňují řídit a modifikovat proces vytváření konkrétních instancí tříd. Těchto cílů dosahují pomocí dvou poněkud abstraktních myšlenek — prvních z nich je zapouzdřování znalostí o tom, které konkrétní třídy systém používá, a druhou je skrývání detailů, jak jsou instance těchto tříd vytvářeny [13]. Většinou se ze vzory z této kategorie lze setkat zejména v robustních systémech, kde vynikají díky větší volnosti plynoucí z preference skládání objektů před vytvářením komplexních třídních hierarchií pomocí standardní dědičnosti.

2.1.1 Abstract Factory

Abstraktní továrna je *creational* návrhový vzor, pomocí kterého lze vytvářet různé varianty sady souvisejících objektů bez explicitního určení, o kterou z konkrétních variant se jedná. Vybrání jedné z existujících variant probíhá nepřímo až v *runtimu*[1] využitím odpovídající továrny.

Motivace

Obecnou situací vhodnou pro tento vzor je potřeba odstínit jisté klientské třídy od konkrétních implementací různých variant sady souvisejících objektů — ať už kvůli skrytí nepodstatných detailů implementace či zjednodušení nebo sjednocení logiky výběru jedné z existujících variant. Mezi hlavní indikátory, kdy je vhodné zvážit implementaci tohoto vzoru, patří:

- nutnost *runtime* hodnoty (např. uživatelský vstup) pro zvolení jedné z variant objektů
- existence pevně dané sady objektů, která je často rozšiřována o nové varianty
- vynucení podmínky, kdy pouze stejné varianty objektů musí fungovat společně
- požadavek na zveřejnění pouze rozhraní nikoliv implementace sady objektů

Dobrým demonstračním příkladem může být libovolná multiplatformní aplikace s GUI. Taková aplikace bude mít sadu objektů, které reprezentují jednotlivé grafické elementy (tlačítko, okno, …). Tyto objekty budou mít pevně daná rozhraní, ovšem podle toho, na jakém OS v daný moment aplikace poběží, je potřeba aby sada těchto objektů měla odpovídající implementace (Windows grafické elementy, Linux grafické elementy, …). Z pohledu spotřebitele jsou ale implementační detaily zcela nepodstatná informace — jemu záleží pouze na rozhraní jednotlivých grafických elementů. Zároveň verze aplikace určená pro Windows nebude nikdy pracovat s jinou implementací grafických elementů, než s tou pro Windows.

Řešení

Uvedený příklad vede na učebnicové vyřešení pomocí *abstraktní továrny*. Prvně se vytvoří obecná rozhraní pro továrny a každý z grafických elementů, což umožní jejich využití napříč aplikací bez vytvoření silné závislosti ke konkrétní implementaci. Dále se udělají implementace specifické pro každou z požadovaných variant (Window, Linux, …), s tím že třeba Windows továrna bude vytvářet jen Windows varianty grafický elementů. Nakonec je jen potřeba, aby každý spotřebitel grafických elementů delegoval jejích vytváření na továrnu. Nyní stačí na jednom místě vytvořit vhodnou továrnu, předat ji spotřebitelům a je hotovo.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.1. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

A, B, C je sada souvisejících objektů (ekvivalentní grafickým elementům z uvedeného příkladu) definující rozhraní společné pro všechny varianty

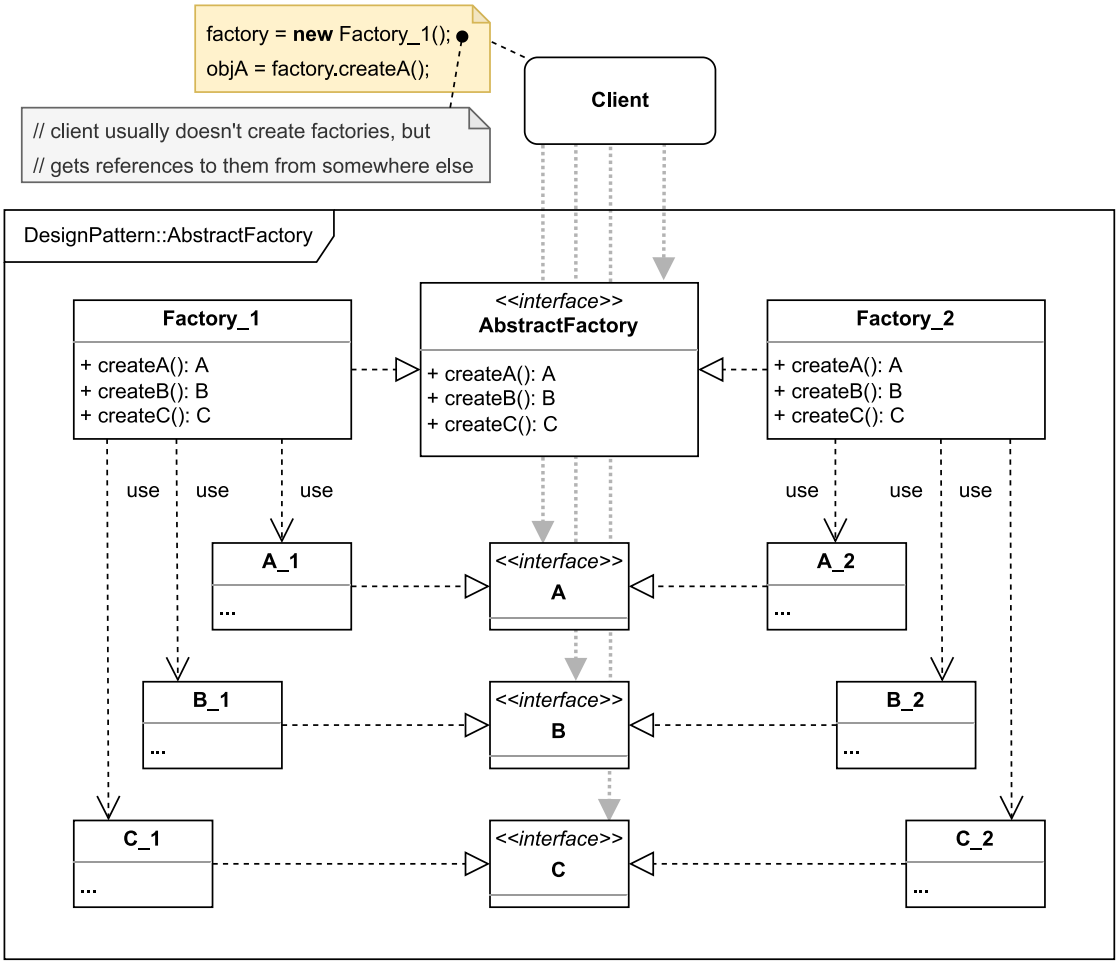
A_1, A_2, ... jsou konkrétní implementace jednotlivých variant sady objektů (ekvivalentní Windows/Linux/… grafickým elementům z uvedeného příkladu)

AbstractFactory je abstraktní továrna definující rozhraní pro vytváření jednotlivých objektů

Factory_1, Factory_2 jsou konkrétní implementace abstraktní továrny, které jsou pevně svázané s jednou z variant objektů (ekvivalentní továrně pouze na Windows/Linux/… grafické elementy z uvedeného příkladu)

Client je externí entita, která pracuje s objekty ze sady a jejich inicializaci deleguje na konkrétní továrnu, která je jí dosazena (ekvivalentní spotřebitelům z uvedeného příkladu)

[1] jedna z fází životního cyklu programu — běh



Obrázek 2.1 Třídní diagram obecné *abstraktní továrny*

Zhodnocení

Výhody:

- snadné přidání nových variant objektů bez nutnosti zásahu do klientského kódu
- konzistence a kompatibilita mezi konkrétními variantami objektů
- redukuje *tight-coupling*[2] mezi konkrétními objekty a klientem

Nevýhody:

- mírně složitější a méně čitelný kód (obsahuje spoustu nových rozhraní)
- přidání zcela nového objektu do sady vyžaduje mnoho úsilí (změna rozhraní definovaného abstraktní továrnou a tudíž i všech konkrétních továren) [14]

Zajímavosti

Jedná se o takovou „továrnu továren" — jinými slovy komplexnější nástavba nad *tovární metodou* (sekce 2.1.3), proto většina návrhu začíná s továrními metodami, ale časem se vyvinou směrem k abstraktním továrnám. Často bývají implementovány formou *singletonu* (sekce 2.1.5).

[2] vysoká závislost komponent

2.1.2 Builder

Stavitel je *creational* návrhový vzor specializující se na inicializaci komplexních objektů. To jsou třeba objekty s desítkami atributů, často ve formě dalších objektů, nebo objekty jejichž vytváření musí dodržovat jistá pravidla a sofistikovanější logiku.

Motivace

Za tímto vzorem typicky stojí problém, který vyžaduje oddělení inicializačního procesu objektu od jeho samotné reprezentace. Ve většině případu bude zároveň požadované, aby řešení takového problémů umožňovalo jednomu a tomu samému procesu produkovat rozdílné instance. Hlavním indikátorem jsou situace, kdy vytváření nějakého objektu pomocí obyčejného konstruktoru dělá potíže — typicky spojené s nepřehledností kódu kvůli mohutným konstruktorům (mají desítky parametrů s výchozími hodnotami), problémem tzv. *teleskopického konstruktoru* (výpis kódu 2.1) či složité rozhodovací logice.

```
class Foo
{
  public:
    Foo(string id_) { ... }
    Foo(string id_, bool attr1_) { ... }
    ...
    Foo(string id_, bool attr1_, ..., bool attr99_) { ... }
  private:
    string id;
    bool attr1;
    ...
    bool attr99;
};
```

Výpis kódu 2.1 Úryvek C++ pseudokódu demonstrující *teleskopický konstruktor*

Dobrým příkladem může být opět aplikace s grafickým rozhraním. Ta bude zřejmě obsahovat třídu reprezentující tlačítko, která bude zaobalovat mnoho informací počínaje rozměry, pozicí, přes barvu pozadí, font, až po definice chování při různých akcích. Mnoho z těchto parametrů bude nepovinných a nastavených s výchozími hodnotami. Také je možné, že bude požadované při vytváření provádět jistou logiku, například kontrolu zda jsou dané rozměry schopné pojmout vnitřní obsah nebo zda daná pozice není mimo vykreslovanou plochu.

Řešení

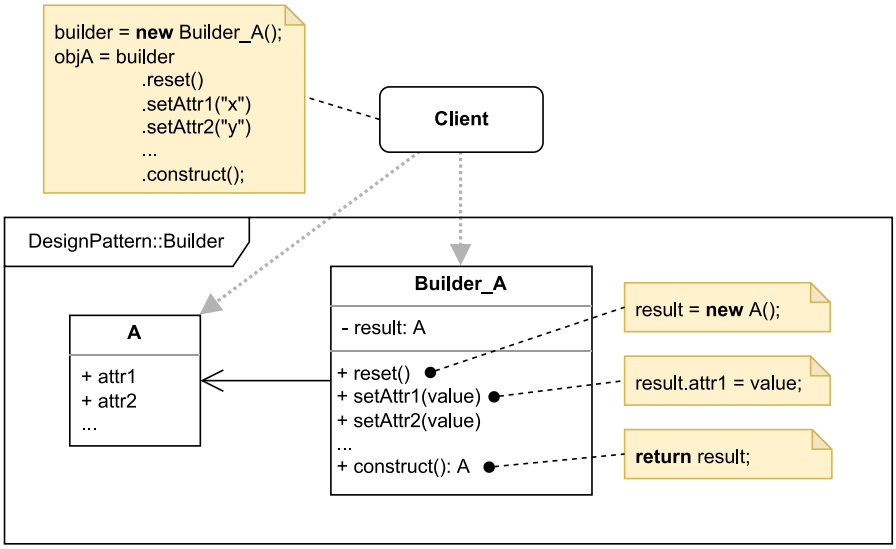
Uvedený příklad je ideální pro zneužití *stavitele*. Prvním krokem bude upravení třídy reprezentující tlačítko, tak aby se jednalo pouze o datovou třídu nesoucí informace. Dále se vytvoří nová třída reprezentující stavitele tlačítek, která bude zodpovědná za sestavování nových tlačítek dle poskytnutých parametrů. Nakonec každá klientská třída, která bude chtít nové tlačítko, využije stavitele, kterému postupně předá požadované vlastnosti. Stavitel bude průběžně sestavovat dané tlačítko a s vyvoláním ukončující akce poskytne sestavený objekt zpět klientovi.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.2. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

A je nějaký komplexní objekt (ekvivalentní tlačítku z uvedeného příkladu)

Builder_A je stavitel objektů typu A (ekvivalentní staviteli tlačítek z uvedeného příkladu)

Client je externí entita, která pracuje s objekty typu A



Obrázek 2.2 Třídní diagram obecného *stavitele*

Verze zobrazená na diagramu 2.2 je nejzákladnější variantou. Existuje i mírně rozšířená verze, který mezi klienta a stavitele dává jakéhosi režiséra (*Director*). Klient si poté neřídí kroky stavebního procesu sám, ale pouze předá odkaz na stavitele režisérovi a následně ho požádá o nějakou specifickou sestavu. Z toho je vidět, že hlavním přínosem režiséra je možnost zafixovat často používané konfigurace pro zvýšený komfort — například v návaznosti na zmíněný příklad by mohl existovat režisér se zafixovanými konfiguracemi pro potvrzovací a stornovací tlačítko (se zeleným resp. červeným pozadím).

Režisér jako takový není povinný a né vždy je nutná jeho implementace. Většinou se využívá ve chvíli, kdy existuje více komplexních objektů a tedy i více stavitelů, kteří sdílí nějaké společné rozhraní (resp. společné kroky stavebního procesu).

Zhodnocení

Výhody:

- zjednodušení kódu (odstranění teleskopického konstruktoru a konstruktoru s desítkami výchozích atributů)
- poskytuje dodatečnou kontrolu nad jednotlivými kroky inicializačního procesu komplexního objektu (možnost omezení pořadí kroků)
- izoluje logiku konstrukce do samostatné třídy (princip SRP)

Nevýhody:

- „nafouknutí" kódu (každý komplexní objekt musí mít svého stavitele)

Zajímavosti

Stavitel často bývá implementován formou *singletonu* (sekce 2.1.5). Funguje skvěle při vytváření složitých *kompozitních stromů* (sekce 2.2.3), jelikož je možné vychytrale zneužít jednotlivých kroků inicializačního procesu pomocí rekurze.

2.1.3 Factory Method

Tovární metoda (také *virtuální konstruktor*) je *creational* návrhový vzor, který popisuje způsob vytváření objektů bez nutnosti specifikace konkrétního typu. Pozorný čtenář si může povšimnout, že se jedná o velice podobný princip jako u *abstraktní továrny* (sekce 2.1.1).

Motivace

Jádro problému za tímto vzorem stojí na potřebě delegovat inicializační proces z nějaké rodičovské třídy na její podtřídy, tak aby každá z nich mohla vytvářet jiný typ objektu. Taková situace typicky vzniká v době, kdy existuje jisté obecné rozhraní a dopředu není jasné, s jakými konkrétními implementacemi bude muset aplikace fungovat či zda nebudou v budoucnu přibývat další.

Příkladem může být zákaznická aplikace, která bude umět vytvářet různé notifikace směrem k uživateli. V první verzi bude podporovat push[3] notifikace a notifikace přes email. Ovšem díky kvalitně udělané analýze je již dopředu jasné, že v dalších verzích bude následovat rozšíření funkcionality o notifikace pomocí SMS, iMessage a dalších kanálů.

Řešení

Zmíněná část aplikace z uvedeného příkladu je ideálním místem pro využití *tovární metody*. Na začátek bude potřeba navrhnout rozhraní společné pro všechny notifikace a vytvořit konkrétní implementace. Dále se vytvoří tovární metoda v třídě, která zprostředkovává práci s notifikacemi. Nakonec vzniknou podtřídy této třídy (tolik kolik je typů notifikací), které budou přepisovat tovární metodu, tak aby inicializovala vhodný typ notifikace. Zbylé metody mohou zůstat tak jak jsou v rodičovské třídě, jelikož pro práci s notifikacemi budou využívat jejich abstraktního rozhraní v kombinaci s právě přepsanou tovární metodu.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.3. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

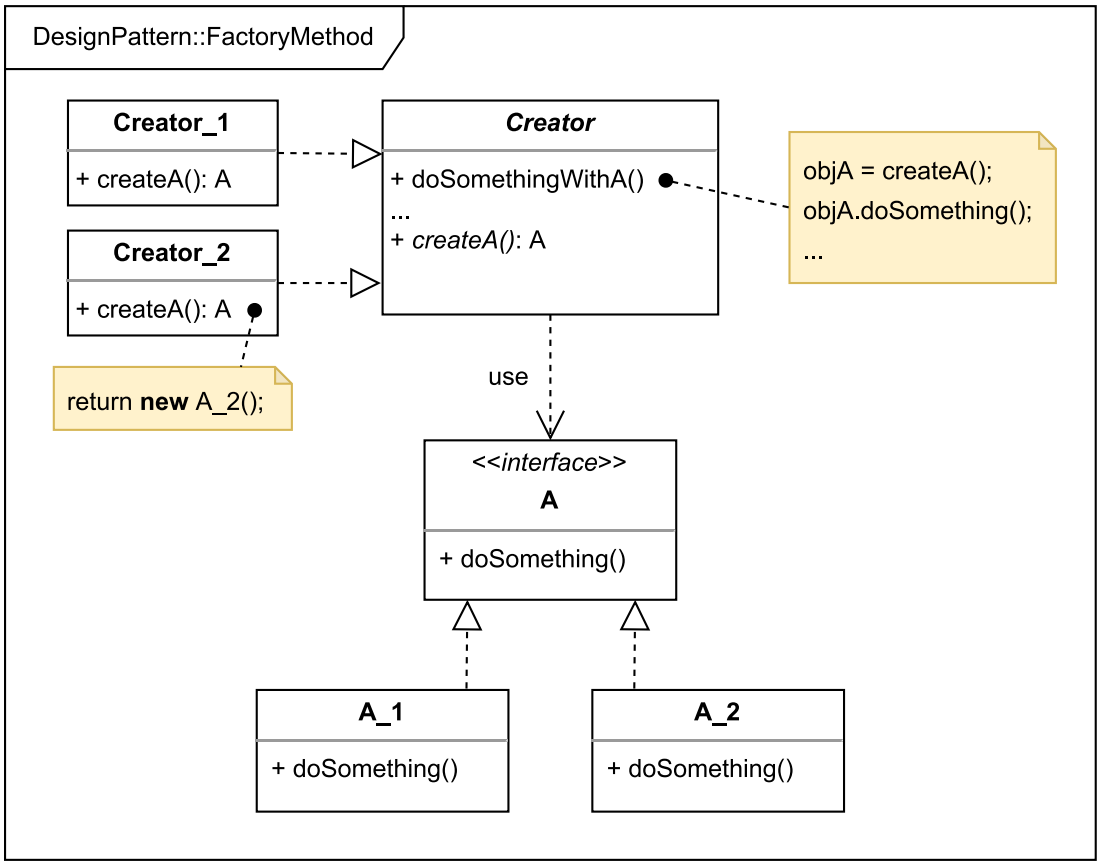
A je obecné rozhraní pro využívané objekty (ekvivalentní rozhraní notifikací ze zmíněného příkladu)

A_1, A_2 jsou konkrétní implementace využívaných objektů (ekvivalentní push/email/… notifikacím ze zmíněného příkladu)

Creator je třída pracující s objekty typu A pomocí tovární metody (createA) — zde je třeba podotknout, že vytváření objektů typu A není hlavní zodpovědností této třídy, ale pouze vedlejší produkt snahy oddělit *business logiku*[4] pracující s objekty typu A, od jejich konkrétních implementací

Creator_1, Creator_2 jsou podtřídy, které přepisují tovární metodu a vrací již konkrétní podtypy objektu A (tj. A_1, A_2)

[3] notifikace, které se zobrazují na ploše počítače nebo telefonu
[4] část programu, která implementuje nějaká pravidla ze skutečného světa



Obrázek 2.3 Třídní diagram obecné *tovární metody*

Zhodnocení

Výhody:

- redukuje *tight-coupling* mezi konkrétními objekty a business logikou
- podporuje zavedení nových typů objektů bez zásahu do stávajícího kódu

Nevýhody:

- složitější a méně čitelný kód (vyžaduje spoustu nových podtříd)

Zajímavosti

Tovární metoda je specializací *šablonové metody* (sekce 2.3.9) a často může posloužit jako jeden z jejich kroků.

2.1.4 Prototype

Prototyp (také *klon*) je *creational* návrhový vzor, jehož cílem je umožnit co nejjednodušší kopírování již existujících objektů.

Motivace

Hlavním problémem s kopírováním již existujících objektů, krom zdlouhavého procesu přiřazování jednotlivých atributů po jednom, je zapouzdření jistých vlastnosti. Tzn. objekty s privátními atributy vlastně ani nelze bez zásahu do jejich kódu efektivně kopírovat.

Pro příklad předpokládejme, že existuje aplikace, která se pracuje s různými distribucemi databází — třeba Oracle a Postgres. Taková aplikace bude mít obecné rozhraní pro jednotlivá připojení s konkrétními implementacemi pro každou distribuci (tj. OracleConnection, PostgresConnection). Zároveň uvažme, že aplikace potřebuje z nějakého důvodu umět tyto připojení kopírovat (třeba protože je chce předávat do subprocesů). Třída zaobalující připojení bude mít zřejmě několik privátních atributů, které by neměly být zvenčí viditelné. To znamená, že jakékoliv kopie stávajících objektů nebudou jednoduše realizovatelné. Zároveň by asi bylo vhodné vymyslet takové řešení, které umožní jednotné kopírování všech připojení, tj. i těch, které by mohly být v budoucnu přidány (např. pro MySQL distribuci).

Řešení

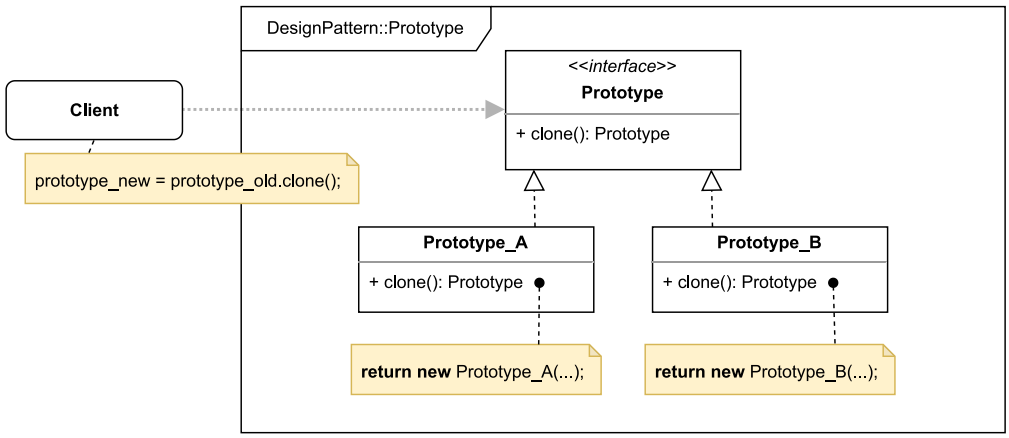
Uvedený příklad lze perfektně vyřešit využitím *prototypu*. Tím se stane obecné rozhraní pro všechny připojení, které bude nyní navíc definovat klonovací metodu. Tu následně všechny podtřídy s konkretními implementacemi přepíší odpovídajícím způsobem, tak že vrátí nový objekt svého typu se zkopírovanými atributy.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.4. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

Prototype je rozhraní prototypu s předpisem klonovací metody (ekvivalentní připojení z uvedeného příkladu)

Prototype_A, Prototype_B jsou konkrétní objekty s vlastní implementací klonovací metody (ekvivalentní Oracle/Postgres/MySQL připojení z uvedeného příkladu)

Client je externí entita, která pracuje s konkrétními objekty a vytváří si jejich kopie pomocí klonovací metody



Obrázek 2.4 Třídní diagram obecného prototypu

Zhodnocení

Výhody:

- vytváření kopií objektů bez závislosti na konkrétních třídách
- umožňuje jednoduše přidávat a odebírat objekty při *runtimu*
- odstranění opakujícího se inicializačního kódu

Nevýhody:

- klonování objektů s cyklickými závislostmi je náročné, někdy i nemožné (potřeba být na pozoru s mělkými kopiemi)
- vynutí zakrytí struktury skutečných objektů (né vždy je toto žádané chování)

Zajímavosti

Některé programovací jazyky definují již ve svých standardních knihovnách rozhraní, jehož implementací lze dosáhnout klonovací funkcionality — např. *java.lang.Cloneable* (Java), *System.ICloneable* (.NET) nebo *std::clone::Clone* (Rust).

2.1.5 Singleton

Singleton je *creational* návrhový vzor, který zaručuje že třída bude mít v jakýkoliv moment běhu aplikace pouze jednu globálně dostupnou instanci.

Motivace

Hlavní ideou tohoto vzoru je mít kontrolu nad počtem existujících instancí. Je důležité důkladně promyslet, zda je skutečně nutné využít tento vzor, jelikož kvůli jeho zbytečnému používání je mnohými považován za *antivzor* (sekce 1.2.4) — zejména díky mnohým restrikcím spojeným se zavedením globálního stavu do aplikace.

Typickým příkladem použití je nějaký *logger*[5] objekt. Ten většinou zapisuje do jednoho souboru, ale bývá využíván ve vícero částech aplikace zároveň — využiti singletonu zde eliminuje potřebu implementovat složité synchronizační mechanizmy, které by byly potřeba pokud by existovalo více asynchronně pracujících instancí, které by se snažili zapisovat do jednoho a toho samého souboru.

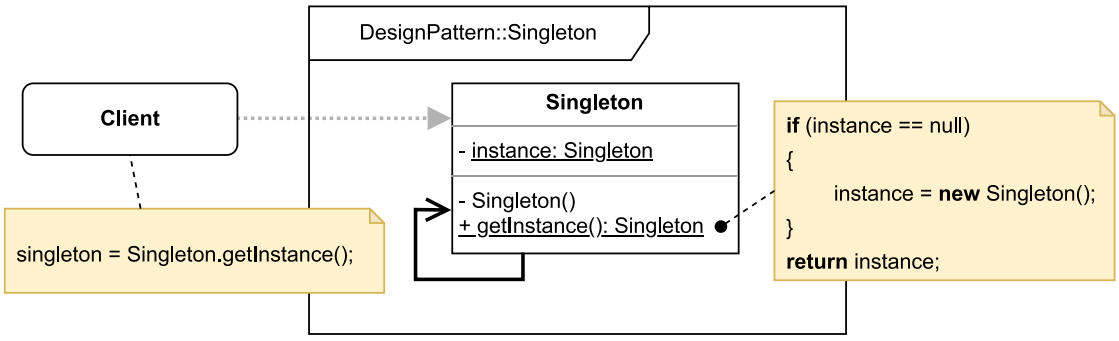
Řešení

Celé řešení uvedeného příkladu je v principu poměrně jednoduché a spočívá v následujících krocích. V daném *logger* objektu se vytvoří privátní statická proměnná, ve které bude uložená jediná existující instance. Dále se zprivatizuje výchozí konstruktor, aby se zamezilo vytváření nových instancí a nakonec se vytvoří statická metoda *getInstance()*, která bude obsahovat logiku zodpovědnou za omezení existujících instancí na pouze jedinou.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.5. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

Singleton je třída, která bude mít vždy jedinou instanci (ekvivalentní *logger* objektu z uvedeného příkladu)

Client je externí entita, která pracuje se Singletonem (ekvivalentní různým částem aplikace z uvedeného příkladu)



Obrázek 2.5 Třídní diagram obecného *singletonu*

Existuje i mírně rozšířená varianta nazývaná *multiton*, která povoluje existenci n instancí. Její implementace se liší víceméně pouze v tom, že třídní proměnná s uloženou instancí je nahrazena kolekcí (typicky mapou) a mírným upravením logiky *getInstance()* [15].

[5] objekt sloužící pro logování zpráv

Zhodnocení

Výhody:

- existence pouze jediné instance a její vytvoření pouze při první žádosti
- šetří paměť

Nevýhody:

- zanáší globální stav do aplikace (což vytváří skryté závislosti namísto jejich zobrazení skrze rozhraní)
- ohromně ztěžuje unit testy[6] jelikož většina testů se spoléhá na *mock objekty* (sekce 2.2.9) — což je v rozporu s privátním konstruktorem a nemožností přepisovat statické metody
- v jistých prostředích může být implementace skutečného singletonu záludná

Zajímavosti

Mnoho jiných vzorů jako třeba *abstraktní továrny* (sekce 2.1.1), *stavitelé* (sekce 2.1.2), *prototypy* (sekce 2.1.4) nebo i *fasády* (sekce 2.2.5) často bývá zkombinováno se singletonem.

Při vlastní implementaci singletonu je nutné být obzvláště na pozoru a brát v potaz aspekty konkrétního programovacího jazyka, aby výsledný plod práce byl skutečně singletonem — kupříkladu v Javě se nesmí opomenout případy využívající paralelní výpočty, serializaci/deserializaci objektů či reflexi[7].

[6] zautomatizované testy jednotlivých komponent bez závislostí na okolí
[7] abilita procesu nahlížet svou strukturu a chování, případně jej i modifikovat

2.1.6 Object Pool

Objektový fond (také *objektový bazén*) je návrhový vzor, který principiálně spadá do kategorie *creational* vzorů. Jeho podstatou je snaha eliminovat časté vytváření nových mohutných objektů přepoužíváním již existujících instancí [16].

Motivace

Problémy stojící za tímto vzorem jsou spojené s objekty, které jsou drahé na vytvoření a navíc často bývají potřeba po velmi krátký časový úsek v poměru k celkové době běhu aplikace. Častá inicializace takových objektů může velmi jednoduše způsobit výkonnostní problémy.

Vhodným příkladem může být opět aplikace s nějakým databázovým připojením, tentokrát ale bude vždy na jeden databázový server, který umí obhospodařovat více spojení zároveň. Navázání spojení na takový server není vůbec levná operace a tudíž není vhodné s každým dotazem vytvářet nové. Zároveň záplava několika požadavky na otevření nových spojení by mohla způsobit přetížení serveru.

Řešení

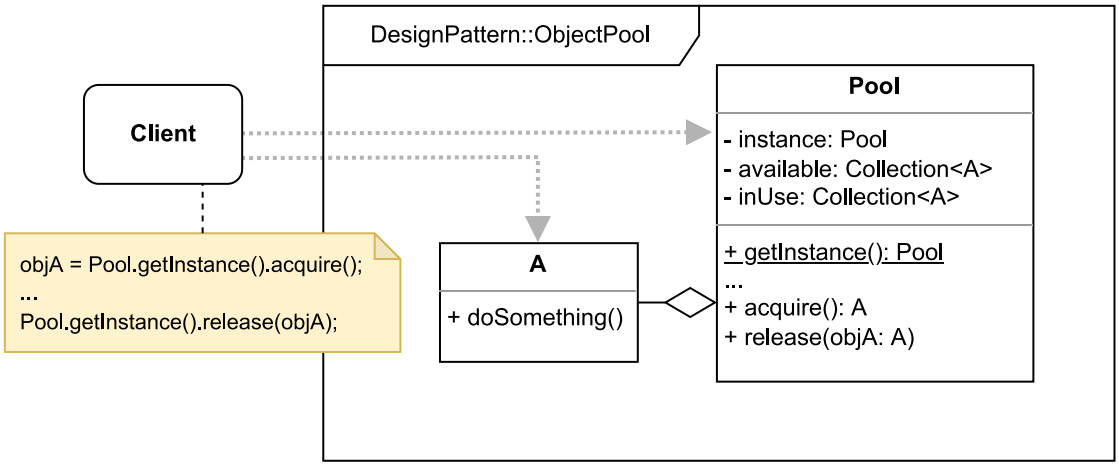
Poměrně efektivním řešením zmíněné situace je právě využít *objektového fondu*. Celé řešení spočívá pouze ve vytvoření fondu, který bude spravovat několik předem vytvořených připojení, která bude vždy propůjčovat cílovým klientům. Tímto zacachováním se omezí počet existujících spojení, a jelikož klient bude spojení vždy potřebovat pouze na vykonání nějaká operace načež ho vrátí zpět, dojde k velmi efektivní recyklaci mezi klienty.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.6. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

A je objekt, který je drahý na vytvoření (ekvivalentní databázovému připojení z uvedeného příkladu)

Pool je kontejner, který spravuje/cachuje objekty typu A (ekvivalentní fondu databázových připojení z uvedeného příkladu)

Client je externí entita, která pracuje s drahými objekty (ekvivalentní klientovi, který chce provést nějaký dotaz nad databází)



Obrázek 2.6 Třídní diagram obecného *objektového fondu*

Zhodnocení

Výhody:

- zlepšení výkonu díky opětovnému využití drahých objektů
- správa existujících objektů (včetně omezení počtu)

Nevýhody:

- vyžaduje dodatečnou synchronizaci v paralelních prostředích

Zajímavosti

Vláknový fond (Thread Pool) lze do jisté míry považovat za *concurrent* návrhový vzor, který ovšem funguje na zcela stejném principu, akorát namísto uchovávání objektů udržuje jednotlivá vlákna. Pokud jsou vlákna využívána na provádění často se opakujících úloh, dojde k velké úspoře výpočetního výkonu, který by normálně byl využit na vytváření a ničení příslušných vláken.

2.1.7 Lazy Load

Líné načítání (také *líná inicializace*) je návrhový vzor, který principiálně spadá do kategorie *creational* vzorů. Poskytuje nástroje pro oddálení inicializace objektu, provedení nějakého výpočtu nebo jakéhokoliv jiného náročného procesu, do doby kdy je to zcela nezbytné [17]. Jedná se spíše o optimalizační techniku.

Motivace

Myšlenkou za tímto vzorem je rozložení zátěže při startu aplikace. Jelikož aplikace, která se bude načítat několik minut (resp. v dnešní době i jen několik vteřin) neudělá na uživatele nejlepší první dojem a pravděpodobně nebude dlouho trvat, než přejde k nějaké konkurenci.

Příkladem, kde se velmi často využívá líného načítání jsou webové aplikace. Například bude-li součástí webu e-shop, ve kterém budou vypsané produkty ve formě karet s názvem a obrázkem. V takové aplikaci je poměrně nepraktické, při každém zpřístupnění stránky načítat klientovi obrázky všech produktů, když v daný moment jich vidí pouze omezený počet, kvůli velikosti jeho obrazovky.

Řešení

Web z uvedeného příkladu může pomocí líného načítání skvěle optimalizovat výkonnost stránky s produkty (zcela bez uživatelovi vědomosti). Řešení spočívá v tom, že aplikace bude hlídat pozici klienta na stránce (offset od vrchu stránky) a načítat pouze obrázky viditelných produktů. Následně při každém pohybu uživatele zkontroluje, zda nemá načíst další.

Obecně lze tento vzor implementovat několika způsoby dle dané potřeby. Mezi nejčastější varianty patří líná inicializace (*Lazy Initialization*) a virtuální proxy (*Virtual Proxy*) [18].

Líná inicializace je technika, která spočívá ve výchozím nastavení drahých hodnot na *null*. Následně v implementaci veškerého veřejně dostupného rozhraní provádí na prvním místě kontrolu, zda není zpřístupňovaná hodnota *null* a pokud je, tak jí nejdříve naplní skutečnou hodnotou.

```
class Foo
{
  public:
    void doSomethingWithFile()
    {
      if (file == nullptr)
      {
        file = new FILE(...);
      }
      ...
    }
  private:
    FILE * file = nullptr;
};

Foo f = Foo();            // file isn't instantiated at this point
f.doSomethingWithFile()   // this is where file init happens
...
f.doSomethingWithFile()   // file is already instantiated from prev. call
```

Výpis kódu 2.2 Úryvek C++ pseudokódu demonstrující *línou inicializaci*

Virtuální proxy je další přístup, který nad jistým objektem vytvoří tzv. „proxy objekt" se zcela stejným rozhraním, který ale obsahuje pouze referenci na originální objekt. Ten poté inicializuje až ve chvíli, kdy je ho skutečně třeba. Veškeré akce následně pouze deleguje na samotný originální objekt.

```
class FooInterface
{
  public:
    virtual void doSomething() = 0;
};

class Foo: public FooInterface
{
  public:
    void doSomething() { ... };
};

class FooProxy: public FooInterface
{
  public:
    void doSomething()
    {
      if (real_foo == nullptr)
      {
        real_foo = new Foo();
      }

      real_foo->doSomethingWithFile();
    };
  private:
    Foo * real_foo = nullptr;
};

FooProxy f = FooProxy();
f.doSomething();
```

Výpis kódu 2.3 Úryvek C++ pseudokódu demonstrující *virtuální proxy*

Zhodnocení

Výhody:

- zrychleni startu aplikace (lepší první dojem pro uživatele)
- úspora paměti (v daný moment nepotřebné objekty nebudou vůbec inicializovány)
- šetření uživatelských dat (u webových aplikací)

Nevýhody:

- první provolání bude oproti ostatním znatelně pomalejší

Zajímavosti

Výchozímu chování (v drtivé většině případů) a přesnému opaku k línému načítání se říká dychtivé načítáni (*eager loading*). Vyjma zmíněných implementací existují i další (*Value holder* a *Ghost*), který jsou ale méně používané.

2.2 Vzory týkající se struktury programu (Structural)

Další skupina návrhových vzorů, kterou zformulovalo seskupení *GoF*. Zabývá se principy a mechanizmy, které vysvětlují, jak pomocí stávajících objektů a tříd vytvořit nové sofistikované struktury. Zaměřují se na vymýšlení struktur, které si zachovají co nejvyšší flexibilitu a rozšiřitelnost. Individuální vzory zkoumají vztahy mezi entitami v daném prostředí, a následně popisují různé postoje vůči rozšiřování jejich funkcionalit. Tohoto typicky dosahují vhodným skládání již existujících objektů, což má za hlavní přínos to, že vytváření nových struktur nevyžaduje žádný zásah do těch již existujících. To je obzvláště užitečné v momentech, kdy se pracuje s nějakým kódem třetí strany, ke kterému není možné přistoupit a zanést modifikace přímo do něj (to je také nejčastější situace, kdy je těchto vzorů využíváno).

2.2.1 Adapter

Adaptér je *structural* návrhový vzor, zaměřený na zprostředkování komunikace mezi objekty s nekompatibilními rozhraními. Jako takový by neměl být používán při návrhu nových aplikací, ale spíše jen jako nástroj řešící problémy v již existujícím ekosystému.

Motivace

Problém za tímto vzorem je velice přímočarý. Existuje objekt, resp. nějaká entita, s omezenými funkčnostmi, která chce nově začít využívat služeb poskytovaných jinou entitou. Entita poskytující nějaké služby má ale pevně stanovené rozhraní, se kterým si v tomto případě první entita neumí poradit, kvůli svým omezeným funkčnostem. Důležitým detailem a hlavním důvodem vzniku tohoto vzoru je předpoklad nemožnosti zásahu do stávajícího kódu — jinak by samozřejmě stačilo upravit jednu z entit.

Reálnou situací, která může velmi jednoduše nastat je různorodost dat, resp. formátů ve kterých jsou uložena. Kupříkladu jedna entita budu umět poskytovat data pouze ve formátu XML a nově bude chtít využívat funkcionalit poskytovaných druhou entitou. Zádrhel spočívá v tom, že druhá entita umí pracovat pouze s daty v JSON formátu. Navíc zásah do kódu ani jedné z entit není možný — důvodu pro toto omezení může být několik, od nepřístupnosti protože se jedná o službu poskytovanou třetí stranou, po nevhodnost změny rozhraní, protože by s sebou nesla nereálný počet změn.

Řešení

S přihlédnutím na daná omezení a samotnou podstatu problému, lze opodstatnit využití *adaptéru*. Řešení bude spočívat ve vytvoření nové třídy, která bude stát mezi entitami a starat se o transformaci formátu. Konkrétně bude kopírovat rozhraní, pomocí kterého umí komunikovat první entita a obsahovat referenci na druhou entitu, na kterou bude delegovat konkrétní úkony s již transformovanými daty.

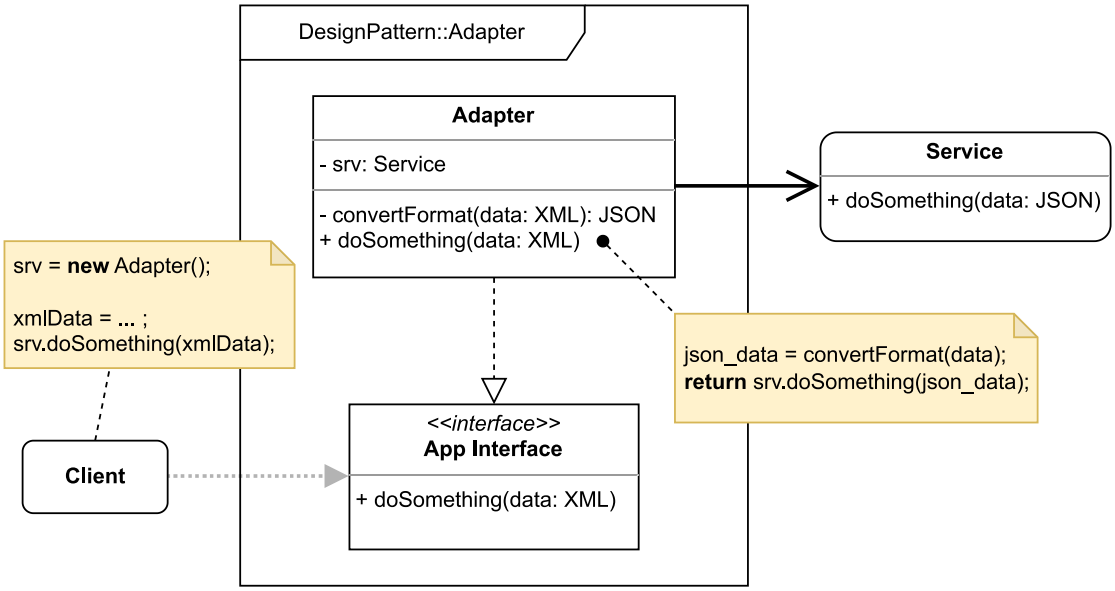
Obecná struktura tohoto vzoru je zobrazena na diagramu 2.7. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

App Interface je předpis rozhraním, se kterým umí klient s omezenými schopnostmi pracovat

Adapter je objekt, který dodržuje rozhraní se kterým umí pracovat první entita (Client) a nějak ho transformuje na rozhraní, které vystavuje druhá entita (Service)

Client je externí entita s omezenými funkčnostmi, která chce využívat služeb od druhé entity (ekvivalentní entitě, co umí poskytovat data pouze v XML formátu, z uvedeného příkladu)

Service je externí entita, která nabízí nějaké služby s pevně zafixovaným rozhraním (ekvivalentní entitě, co umí zpracovávat data pouze v JSON formátu, z uvedeného příkladu)



Obrázek 2.7 Třídní diagram obecného *adaptéru*

Zhodnocení

Výhody:

- umožňuje modifikovat nekompatibilní rozhraní bez zásahu do stávajícího kódu
- odděluje rozhraní od business logiky (princip SRP)

Nevýhody:

- přidává „zbytečný" kód (pokud je to možné, úprava stávajících rozhraní může být lepším řešením — zejména v raných fázích vývoje)

Zajímavosti

Adaptér, *Proxy* (sekce 2.2.7) a *Dekorátor* (sekce 2.2.4) jsou si velmi podobní tím, že zaobalují nějakou již existující entitu a tu následně modifikují. *Adaptér* transformuje rozhraní do jiné podoby. *Proxy* zachovává originální rozhraní, ale provádí nějaké před a/nebo po zpracování. *Dekorátor* staví na původním rozhraní, které následně rozšiřuje.

2.2.2 Bridge

Most je *structural* návrhový vzor, který je používán pro rozdělení ohromných monolitických tříd na dvě oddělené hierarchie, propojené pomocí vzájemných referencí. Hlavním přínosem tohoto přístupu je odstínění vývoje mezi zmíněnými hierarchiemi — tedy změna v jedné nebude vyžadovat zásah do druhé.

Motivace

Častým problémem, který se časem projeví u obřích monolitických tříd, je že použití standardní dědičnosti těžce svazuje rodičovské třídy s jejich podtřídami. Takže ve chvíli, kdy bude potřeba nějakou monolitickou třídu rozšiřovat podtřídami ve více jak jedné dimenzi, stane se ze systému naprostá noční můra na údržbu. To je protože přidání jedné nové podtřídy podle jedné dimenze bude vyžadovat vytvoření tolika podtříd, kolik variant jich bude existovat podle druhé dimenze.

Příklad. Existuje aplikace, která má v sobě třídu pro reprezentaci osob. Následně má podtřídy zachycující osoby podle dvou zcela nesouvisejících dimenzí — první dimenzí je pracovní poměr vůči zaměstnavateli (tj. zaměstnanec, zákazník) a druhou dimenzí je věrnostní program (tj. standard, premium, premium plus). Takže aplikace má třídy odpovídající všem kombinacím (tj. zákazník standard, zaměstnanec standard, zákazník premium, …). To ve stávajícím stavu není ideální, ale ani vyloženě problém. Trable začnou nastávat ve chvíli, kdy bude potřeba přidat novou variantu věrnostního programu nebo nový pracovní poměr. Taková změna bude vyžadovat zavést tolik nových podtříd, kolik je variant v druhé z dimenzí (tj. přidání brigádníka bude vyžadovat třídy: brigádník standard, brigádník premium, brigádník premium plus).

Řešení

Efektivní řešení zmíněné struktury pomocí *mostu* bude vypadat následovně. Struktura se rozdělí na dvě oddělené hierarchie, která na sebe budou pouze odkazovat. Konkrétně věrnostní programy se vyčlení bokem a individuální osoby poté budou mít odpovídající referenci. Zde je nutné podotknout, že rozdělení by mohlo být i opačným směrem (tj. pracovní poměr, na který by se odkazovalo z věrnostních programů). Toto rozdělení záleží na tom, která z hierarchií bude provádět reálnou práci (věrnostní program dle prvotního návrhu) a která ji bude pouze delegovat (pracovní poměr dle prvotního návrhu).

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.8. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

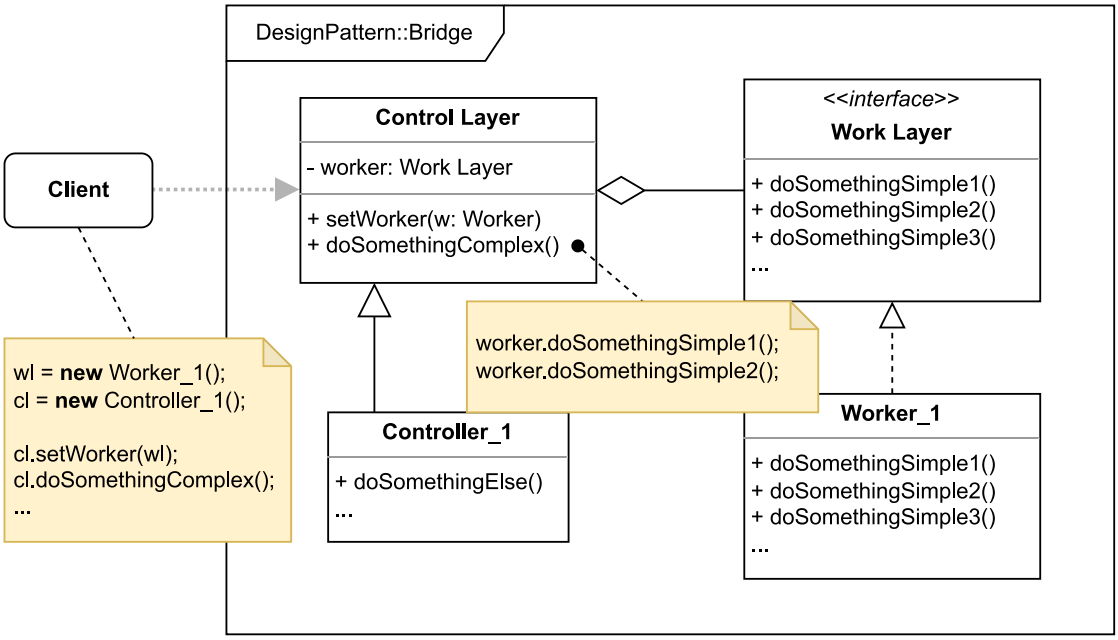
Control Layer je vysokoúrovňové rozhraní, které by samo o sobe nemělo dělat žádnou práci, ale mělo by jí delegovat na *Workers* (ekvivalentní osobě z uvedeného příkladu)

Controller_1 je nepovinná specializace *Control Layer* s extra funkčnostmi (ekvivalentní zaměstnanci/zákazníkovi z uvedeného příkladu)

Work Layer je rozhraní, které musí implementovat každý *Worker* (ekvivalentní věrnostnímu programu z uvedeného příkladu)

Worker_1 je konkrétní implementace nějakých funkčnosti (ekvivalentní individuálním variantám věrnostního programu z uvedeného příkladu — tedy standard, premium, premium plus)

Client je externí entita pracující s mostem



Obrázek 2.8 Třídní diagram obecného mostu

Zhodnocení

Výhody:

- optimalizuje vývoj vysoce závislých tříd (změna jedné hierarchie neovlivňuje druhou)
- izoluje klienta od implementačních detailů
- odstraňuje duplicitní kód

Nevýhody:

- zvýšení počtu nepřímostí ve zpracování (dopad na čitelnosti a v krajních případech i výkon)
- užitečný pouze pokud se očekává přidávání dalších podtříd (tj. variant jedné z dimenzí)

Zajímavosti

Most může být nádherně zapracován do rozšířené verze *stavitele* (sekce 2.1.2), která využívá režiséra — ten pak hraje roli řídící vrstvy (Control Layer) a jednotlivý stavitelé pak jsou ekvivalentní pracovníkům (Worker) z pracovní vrstvy (Work Layer). Také lze vynalézavě zkombinovat s *abstraktní továrnou* (sekce 2.1.1), je-li potřeba docílit omezení, že pouze jisté podtřídy řídící vrstvy (Control Layer) budou moci pracovat s jistými podtřídami pracovní vrstvy (Work Layer).

2.2.3 Composite

Kompozit (také *objektový strom*) je *structural* návrhový vzor postavený na jediné myšlence a tou je reprezentace objektů jakožto stromových struktur[8]. Smysluplné uplatnění nalezne pouze ve chvíli, kdy jádro aplikace tvoří skupina entit, které lze reprezentovat jakožto stromy.

Motivace

Dobrým indikátorem, že jistou část aplikace půjde reprezentovat stromovou strukturou, je existence takových dat, která se zanořují sama do sebe. Přesněji řečeno, když se vyskytuje objekt, jehož jedním atributem je kolekce objektů stejného typu. Problém s operacemi nad takto rekurzivně definovanou strukturou bývá, že neexistuje žádné jednoduché přímočaré řešení — v lepších případech povede k šíleně zanořeným cyklům se složitými přetypováními, v horších případech půjde o tímto stylem neřešitelný problém.

Pěkným příkladem může být aplikace pro HR[9] oddělení, ve které bude reprezentována struktura jisté společnosti. Hlavními stavebními bloky zde budou jednotlivá oddělení, která pod sebou mohou ale nemusí mít další oddělení. Koncovými entitami budou poté konkrétní zaměstnanecké pozice. A teď si představme, že členové představenstva budou periodicky požadovat zprávu s odhadem rozpočtu, který padne na veškeré výplaty. To je přesně případ, kdy by přímočarý výpočet vyžadoval nešikovně vnořené cykly, co by se postupně prokousávaly jednotlivými pododděleními. Ačkoliv by to byl asi dosažitelný cíl, výsledné řešení by nebylo úplně chvályhodné.

Řešení

Aplikaci z uvedeného příkladu lze poměrně pěkným způsobem navrhnout pomocí *kompozitu*. Stačí si uvědomit, že jednotlivá oddělení a zaměstnanecké pozice budou tvořit nádhernou stromovou strukturu. Nejdůležitější částí je, aby všechny objekty stromu implementovaly jedno rozhraní, pomocí kterého s nimi bude moci klient pracovat, aniž by musel rozlišovat jejich přesný typ. Ve zmíněném příkladu tedy třídy reprezentující oddělení a zaměstnanecké pozice musí mít jednotné rozhraní — zde s definovanou metodou pro zjištění výše výplat. Implementace se následně budou lišit pouze v tom, že koncové zaměstnanecké pozice vrátí vyšší výplaty, zatímco oddělení zavolá tuto metodu na všech vnořených objektech a vrátí jejich součet. Výpočet celkového rozpočtu potom vlastně bude znamenat jen provolání naimplementované metody na nejvýše postavené oddělení společnosti.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.9. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

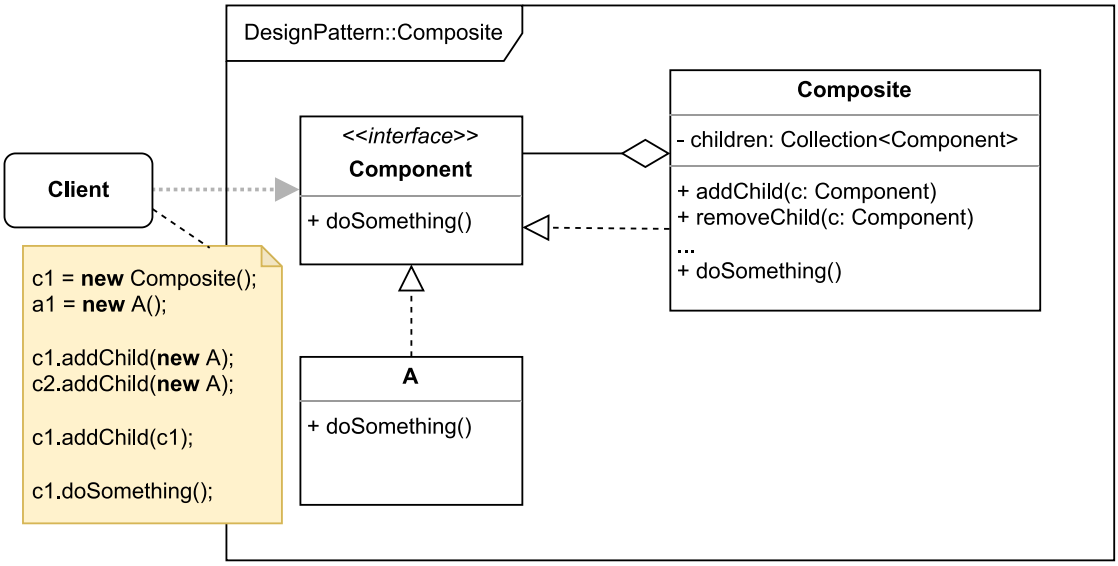
Component je rozhraní, které musí dodržovat všechny entity ve stromě (metoda *doSomething* by byla ekvivalentní výpočtu výše výplat z uvedeného příkladu)

A jsou koncové objekty, které efektivně ukončují rekurzi (ekvivalentní zaměstnanecké pozici z uvedeného příkladu)

Composite je objekt, resp. kontejner, který obsahuje kolekci s dalšími kontejnery a/nebo koncovými objekty typu A (ekvivalentní oddělení z uvedeného příkladu)

Client je externí entita, která chce nějak pracovat se stromovou strukturou (ekvivalentní entitě vyvolávající výpočet z uvedeného příkladu)

[8] datová struktura tvořená acyklicky propojenými uzly, viz. wiki článek [19]
[9] Human Resources — lidské zdroje



Obrázek 2.9 Třídní diagram obecného *kompozitu*

Zhodnocení

Výhody:

- optimalizuje práci s komplexními stromovými strukturami (pomocí polymorfismu a rekurze)
- zjednodušuje klientský kód (ten nemusí rozlišovat mezi jednoduchými a zanořenými objekty)

Nevýhody:

- vytváří poměrně neflexibilní hierarchii (všechny nové třídy musí dodržovat stejné stávající rozhraní)
- těžké vynucení použití pouze konkrétních komponent (vyžaduje runtime kontroly typu)
- těžká implementace operací specifických pouze pro jisté komponenty

Zajímavosti

Kompozit velmi benefituje ze spolupráce s dalšími vzory, pro zpříjemnění jeho použití: *stavitel* (sekce 2.1.2) lze využít pro sestavování komplexních stromu „zrekurzením" stavebních kroků, *iterátor* (sekce 2.3.3) lze využít pro postupný průchod stromovou strukturou, *flyweight* (sekce 2.2.6) může šetřit využívanou paměť přepoužitím jednoduchých objektů, *dekorátor* (sekce 2.2.4) lze použít pro rozšíření chování určitého objektu ve stromě, *prototyp* (sekce 2.1.4) může usnadnit kopírování složitých stromů.

2.2.4 Decorator

Dekorátor (také *wrapper*) je *structural* návrhový vzor, který umožňuje dynamicky rozšiřovat chování existujících objektů bez ovlivnění ostatních objektů stejného typu. Této funkcionality dosahuje zaobalením originálního objektu speciálním „wrapper" objektem.

Motivace

Tento vzor najde své využití zejména v momentech, kdy existuje třída požadující takový způsob rozšiřitelnosti, na který je na standardní dědičnost krátká. Jedním typickým případem bývá potřeba různými způsoby modifikovat chování objektu při runtimu, společně se zachováním kompatibility s klientským kódem. Dalším případem bývá existence vícero univerzálních funkcionalit, jejichž libovolná kombinace může být použita k obohacení nějakých objektů.

Příkladem může být třeba e-shop provozovaný malou rodinnou firmou, kde jsou nabízeny různé produkty a placené služby. Jelikož to pro rodinu není vůbec žádná priorita, vytvořil ho pro ně jejich známý poměrně jednoduchým způsobem, který nebral v potaz žádné vymoženosti ani budoucí rozšíření. E-shop se ovšem začal pomalu ale jistě rozrůstat a tak si rodinka řekla, že by bylo dobré začít poskytovat slevy na žhavý sortiment. Nicméně to je funkčnost, se kterou se nikdy nepočítalo a její implementace znamená rozsáhlý zásah do stávajícího kódu.

Řešení

Dotyčný problém lze vyřešit i mnohem jednodušším přístupem, než přepisovat polovinu aplikace. S využitím *dekorátoru* stačí vytvořit zaobalující objekt, který bude podporovat stejné rozhraní jako nabízené produkty a služby, a který bude modifikovat metodu vracející jejich cenu takovým způsobem, aby reflektovala požadovanou slevu. Následně stačí produkty nebo služby, které by měly být zrovna ve slevě, zaobalit do „slevového dekorátoru" a voilá. Skvělým vedlejším efektem tohoto řešení je, že zaobalování lze provádět při runtimu — tzn. velmi jednoduše může vzniknout administrační stránka, pomocí které si budou moci slevy nastavovat rodinní příslušníci sami.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.10. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

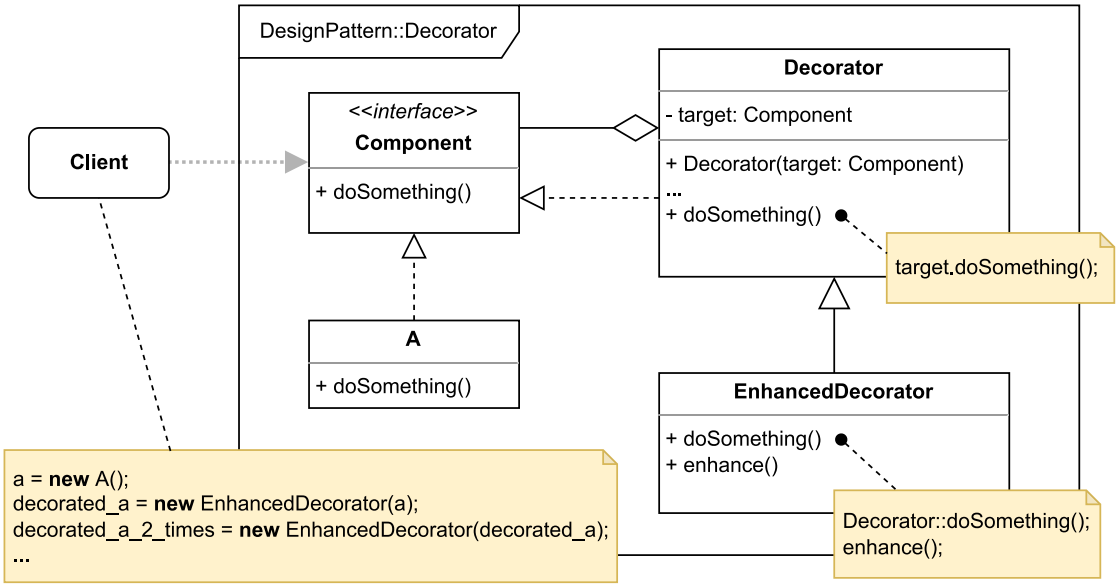
Component je rozhraní, které musí dodržovat všechny zaobalované objekty i dekorátory, aby nebyla ztracena kompatibilita s existujícím kódem

A je zaobalovaný objekty jehož funkčnost se rozšiřuje (ekvivalentní produktům a službám z uvedeného příkladu)

Decorator je objekt, který definuje společné rozhraní pro všechny dokorátory zaobalující konkrétní objekty (ten většinou nedělá žádnou extra práci, pouze udržuje referenci na zaobalený objekt a kopíruje jeho funkčnosti)

EnhancedDecorator je jeden z již rozšiřujících dekorátorů, který nějak modifikuje chování (ekvivalentní dekorátoru, který poskytuje slevu, z uvedeného příkladu; metoda *doSomething* by v tomto kontextu měla vracet cenu a metoda *enhance* by se starala o aplikaci slevy)

Client je externí entita, která pracuje s nějakým vylepšeným objektem (ekvivalentní e-shopu z uvedeného příkladu)



Obrázek 2.10 Třídní diagram obecného *dekorátoru*

Zhodnocení

Výhody:

- umožňuje modifikovat chovaní při runtimu
- díky své struktuře umožňuje zaobalovat již zaobalené objekty do nekonečna (resp. dokud to HW dovolí)
- umožňuje modifikovat pouze jednu konkretní instanci třídy — tzn. změny nijak neovlivní ostatní objekty stejného typu
- zvyšuje flexibilitu, jakožto alternativa k vytváření podtříd kvůli rozšíření funkcionality

Nevýhody:

- složité odebrání konkrétního dekorátoru, má-li jich objekt na sobě více

Zajímavosti

Při sestavování dekorovaných objektů je potřeba být obzvláště na pozoru, v jakém pořadí jsou jednotlivé dekorátory aplikovány — ve většině případu se různá pořadí budou chovat odlišnými způsoby a zamezení této vlastnosti bývá obtížným úkolem. Často bývá přirovnáván k *proxy* (sekce 2.2.7), jelikož oba zaobalují nějaký objekt, který je ale stále zodpovědný za hlavní zpracování požadavku — hlavním rozdílem mezi nimi je, že *proxy* většinou řídí životní cyklus zaobaleného objektu, zatímco *dekorátory* jsou zcela v moci klienta.

2.2.5 Facade

Fasáda je *structural* návrhový vzor s velmi jednoduchou přímočarou myšlenkou — schování složitých rozhraní a sad komplexních tříd za jedno ucelené a zjednodušené rozhraní.

Motivace

Motivací za tímto vzorem není nic složitého. Využívá se v situacích, kdy existuje nějaká vysokoúrovňová funkčnost, jejíž provedení od začátku do konce vyžaduje posloupnost více složitých kroku a součinnost několika komplexních tříd a rozhraní. Pokud by si měl klient sám provést jednotlivé kroky společně s udržováním potřebných závislostí, je vysoká šance, že do kódu zanese všelijaké chyby a pachy[10]. Typicky se využívají u knihoven či frameworků, za účelem poskytnutí co nejjednodušších *plug-n-play*[11] rozhraní pro vývojáře.

Perfektním příkladem může být třeba knihovna, pro převod obrázků z jednoho formátu do jiného. Z klientova pohledu se jedná o poměrně přímočarý úkon. Ten chce říct pouze „převeď mi tenhle obrázek z tohohle formátu do tohohle formátu" a mít hotovo. Pod pokličkou se ale bude dít několik na sebe navazujících kroků využívajících mnoho různých tříd — od načtení zdrojového obrázku do odpovídajících struktur, přes vhodné transformace uložených dat a hlaviček, až po finální uložení v požadovaném formátu.

Řešení

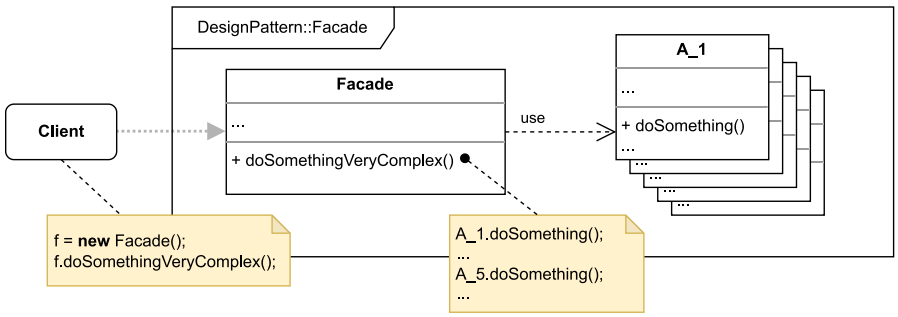
Raději než ponechávat orchestraci zmíněné procesu na klientovi, bude vhodnější využít fasády, ze kterou se tato logika schová. Celé řešení spočívá pouze ve vytvoření nové třídy, která bude poskytovat zjednodušené rozhraní — v tomto případě jednu metodu pro převod obrázku. Všechny potřebné kroky, struktury a závislosti si bude hlídat sama fasáda. Pro převod obrázku bude následně stačit vytvořit novou instanci fasády a zavolat odpovídající metodu.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.11. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

Facade je objekt, který odděluje nadbytečné složitosti od klienta a poskytuje zjednodušené rozhraní (ekvivalentní třídě, která bude poskytovat metodu pro převod obrázku, z příkladu)

A_1, ... je sada spolupracujících tříd, které vykonávají dílčí kroky nějakého složitějšího procesu (ekvivalentní třídám, které mají na starosti načtení obrázku, různé transformace a další úkony převodu obrázku, z uvedeného příkladu)

Client je externí entita, která chce využít funkcionalit zprostředkovaných sadou tříd A_1, ...



Obrázek 2.11 Třídní diagram obecné fasády

[10] charakteristiky v kódu indikující hlubší problémy, např. duplicitní kód, záhadné názvy, dlouhé metody, ...
[11] pojem popisující komponentu v počítačovém světě, kterou stačí tzv. „zapojit" a bude fungovat

Zhodnocení

Výhody:

- podporuje *loose-coupling*
- izoluje klienta od zbytečné komplexity
- zlepšuje přehlednost kódu

Nevýhody:

- velmi jednoduše se z fasády může stát objekty, který umí všechno (porušuje SRP, čímž ztěžuje rozšiřování a testování)

Zajímavosti

Častou chybou, zejména ve vrstvených architekturách, bývá vynucení objektů z jedné vrstvy, aby komunikovali s jinou vrstvou přes zprostředkovávající třídu — to ovšem není významově fasáda. S tímto fenoménem se lze nejtypičtěji setkat u UI elementů, kterým bývá omezována komunikace vůči zbytku aplikace.

2.2.6 Flyweight

Flyweight (také *cache*) je *structural* návrhový vzor, který míří na paměťovou optimalizaci robustních často se vyskytujících objektů, sdílením jejich společných atributů.

Motivace

Tento vzor najde své využití zejména ve chvíli, kdy se v systému vyskytuje ohromné množství objektů jednoho typu, které navíc sdílí atributy s většinou se opakujícími hodnotami. V takový moment je očividně zbytečné, aby každý objekt měl svou vlastní kopii jedněch a těch samých dat. Jedinou podmínkou je, že sdílená data budou vždy neměnná a jak si je jednou objekt nastaví, už je nebude moci změnit.

Příkladem z reálného světa je třeba způsob, jakým webový prohlížeč efektivně ukládá a zobrazuje obrázky na prohlíženém webu. Při vývoji vlastního prohlížeče je potřeba důkladně promyslet, jakým způsobem ukládat obrázky do paměti. Naivní přístup by mohl způsobit potíže v momentě, kdy by uživatel zkusil načíst web, který obsahuje tisíce kopií jednoho obrázku (např. nějaký obrazec pomocí kterého se dynamicky generuje pozadí). Přímočarý způsob, který by v takovém případě každý obrázek reprezentoval vlastní kopii všech dat, by velmi pravděpodobně „sežral" veškerou dostupnou paměť.

Řešení

Sofistikovanější řešení bude využívat právě mechanizmů *cache*. Spočívá ve vyčleněni společných dat, tj. samotného obrázku, do samostatné třídy, tak aby byla ve výsledku uložena v paměti pouze jednou. Druhou částí bude v třídě reprezentující prohlížečem zobrazované obrázky, společně s unikátními daty (pozice, orientace, měřítko, …), mít referenci na objekt s duplicitními (tj. obrázkovými) daty. Správu duplicitních dat bude mít na starosti továrna, jejíž hlavním úkolem bude ukládat nově se vyskytující informace a vracet reference na již známá data.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.12. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

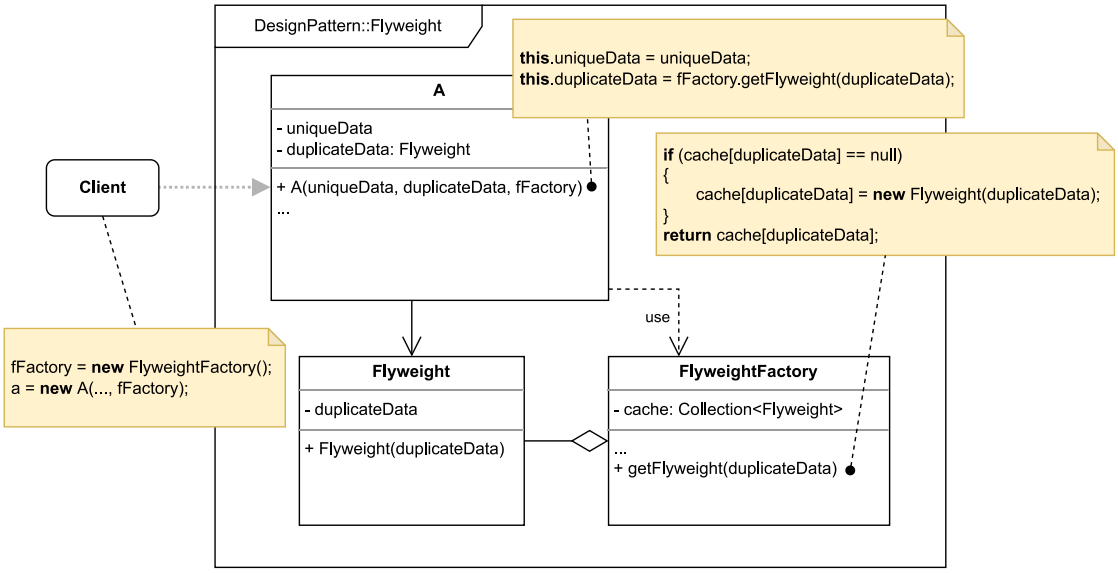
A je objekt s unikátními daty a referencí na duplicitní data (ekvivalentní třídě reprezentující prohlížečem zobrazovaný obrázek v uvedeném příkladu)

Flyweight je objekt úzce svázaný s A, který obsahuje jeho duplicitní data (ekvivalentní třídě se samotnými obrazovými daty z uvedeného příkladu)

FlyweightFactory je továrna zodpovědná za správu existujících Flyweight objektů a vytváření nových dle potřeby

Client je externí entita pracující s objekty typu A

Existuje i alternativní implementace, která chování originálního objektu (tj. *A*) přesouvá do *Flyweight* objektu, kterému jsou pak dle potřeby předávány unikátní data. Oproti vyobrazenému řešení, které používá *Flyweight* pouze jako datové úložiště, to má za výhodu, že oba objekty mají stejné rozhraní, ovšem na úkor větší nepřímosti při vyvolání jednotlivých metod.



Obrázek 2.12 Třídní diagram obecného *flyweightu*

Zhodnocení

Výhody:

- při vhodném využiti ohromná úspora paměti

Nevýhody:

- neintuitivní přístup (rozděluje ucelený objekt a vyžaduje dodatečnou logiku)
- úspora paměti vyžaduje obětovaní rychlosti

Zajímavosti

Tento vzor je docela specifický tím, že pokud je použit správně, vetšinou bude řešit problém, který nepůjde vymyslet jiným způsobem. Díky své komplexitě totiž není moc využíván v situacích, kdy to není zcela nutné. Poměrně těžce bývá využíván v počítačových hrách, pro úsporu při ukládání textur objektů.

2.2.7 Proxy

Proxy je *structural* návrhový vzor, který umožňuje nahradit jistý objekt zástupcem, který krom stejných funkčností, umí navíc provádět nějaké operace před a/nebo po vykonání originální akce. Jak napovídá název, nejčastěji se používá obdobně jako proxy ve smyslu síťování — tedy omezení přístupu, cachovaní, blacklisty[12], whitelisty[13] a obdobné vymoženosti.

Motivace

Obecně se tento vzor používá ve chvílích, kdy existuje nějaký objekt, ke kterému je potřeba vytvořit plně zaměnitelného sourozence, jehož základní funkcionalita zůstane víceméně stejná, ale bude nějak modifikovat vstupní/výstupní data. Proxy se v tu chvíli chová jako jakási mezivrstva, která požadovaným způsobem rozšiřuje funkce originálního objektu, jak již bylo řečeno buď před (např. zamezení přístupu neoprávněným entitám) vykonáním nějaké akce a/nebo po (např. cache výsledků pro další provolání).

Modelová situace — existuje aplikace poskytující úložiště pro uživatelská data, která navrch lokálního ukládání umožňuje zálohovat data do cloudu[14], poskytovaného třetí stranou. Jelikož pronájem cloudového úložiště není levnou záležitostí a často veškeré požadavky na tyto služby bývají zpoplatněny, je i jeho využití v rámci aplikace považováno za prémiovou funkcionalitu, ke které mají mít přístup pouze uživatelé, kteří si za něj připlatili.

Řešení

Vhodným způsobem, jak omezit přístup ke zmíněné funkcionalitě, z pohledu aplikační vrstvy, je právě využití *proxy*. Kód poskytnutý třetí stranou sloužící pro přístup ke cloudovému úložišti, resp. třída zprostředkovávající připojení, se zaobalí proxy objektem se zcela stejným rozhraním, který bude uvnitř uchovávat referenci na originální objekt, a před každým pokusem o zpřístupněním úložiště, bude provádět kontrolu, zda má právě přihlášený klient tuto službu zaplacenou.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.13. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

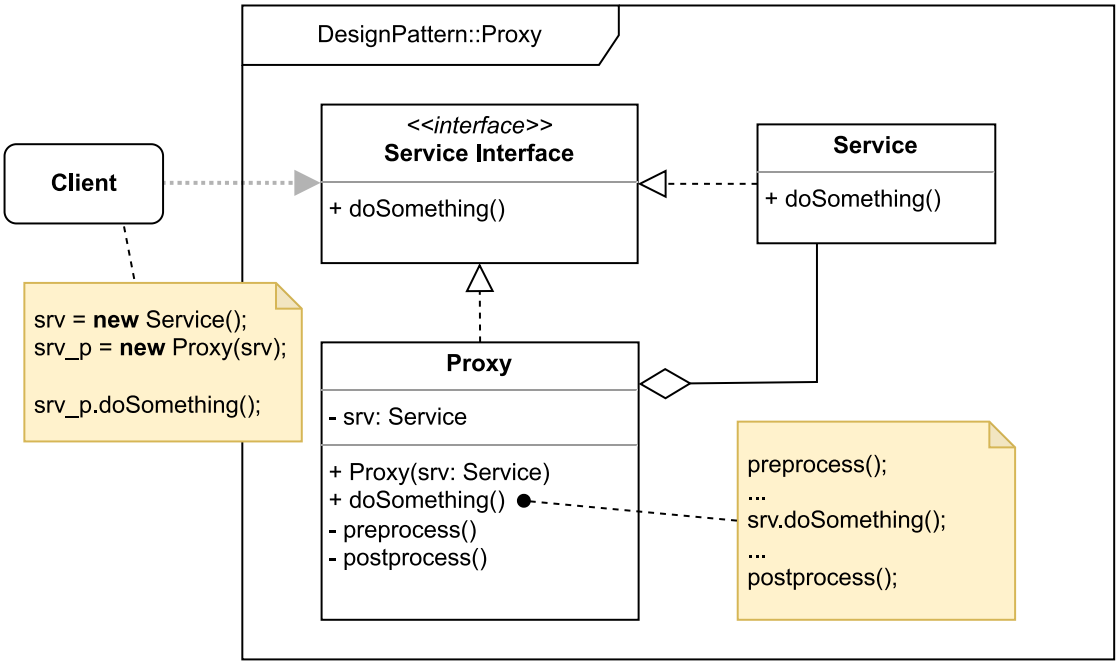
Service Interface je rozhraní nějaké služby, které musí dodržovat i proxy objekt, aby byl nerozlišitelný od originálu (ekvivalentní rozhraní, které má třída poskytnutá třetí stranou pro připojení ke cloudovému úložišti, z uvedeného příkladu)

Service je objekt poskytující nějakou funkčnost pro klienta (ekvivalentní třídě, poskytnuté třetí stranou pro připojení ke cloudovému úložišti, z uvedeného příkladu)

Proxy je objekt, který zaobaluje jistý existující objekt a provádí nějaké před/po zpracování (ekvivalentní proxy, která kontroluje zda má uživatel zaplacenou prémiovou funkcionalitu, v uvedeném příkladu)

Client je externí entita, která chce využívat poskytovaných služeb a zároveň nerozezná, jestli pracuje přímo s danou službou nebo proxy objektem (ekvivalentní části aplikace pracující s cloudovým úložištěm z uvedeného příkladu)

[12] seznam zakázaných hodnot, vše ostatní je povoleno
[13] seznam povolených hodnot, vše ostatní je zakázáno
[14] model, který je založen na službách a programech, běžících na vzdálených serverech přístupných z internetu



Obrázek 2.13 Třídní diagram obecné proxy

Zhodnocení

Výhody:

- umožňuje dodatečnou kontrolu bez povědomí klienta
- dovoluje libovolně zaměňovat různé proxy
- proxy funguje i bez dostupnosti dané služby

Nevýhody:

- při náročném před/po zpracování může dojít ke zpomalení celého procesu provolání služby

Zajímavosti

Jak již bylo zmíněno, *adaptér* (sekce 2.2.1), *dekorátor* (sekce 2.2.4) a *proxy* si jsou velmi podobní s drobnými odlišnostmi. *Adaptér* oproti *proxy* mění stávající rozhraní. *Dekorátor* je oproti *proxy* vždy zcela spravován klientem, zatímco *proxy* většinou řídí životní cyklus zaobaleného objektu.

2.2.8 Twin

Dvojče je návrhový vzor, který principiálně spadá do kategorie *structural* vzorů. Řeší velmi specifický druh problému, a to jakým způsobem lze nasimulovat vícenásobnou dědičnost v prostředích, které ji nativně nepodporují [20].

Motivace

U problému, který stojí za tímto vzorem, není moc co rozebírat — některé programovací jazyky (nejznáměji Java) nepodporují vícenásobnou dědičnost a dvojče umožňuje do jisté míry toto omezení obejít pouze pomocí jednoduché dědičnosti. Ačkoliv se jedná o poměrně kontroverzní vlastnost, kterou někteří považují za velmi šikovnou a jiní zase za extrémně problémovou, zcela jistě se naleznou případy, kdy s její pomocí lze ušetřit spoustu nepříjemností, které by s sebou neslo ekvivalentní řešení pomocí jednoduché dědičnosti. Mimo jiné je možné tento vzor také využít pro vyřešení typické problém, který se často vyskytuje u návrhů obsahujících vícenásobnou dědičnost, a tím jsou střety názvů.

Jako demonstrační příklad poslouží aplikace pro správu a přiřazování úkolů na vývojáře, nebo-li tzv. „issue tracking". V naší konkrétní aplikaci bude rozlišováno, zda je daný vývojář FE[15], BE[16] nebo FullStack[17]. Konkrétní úkoly vytvořené v aplikaci by poté podle jejich povahy měly jít přiřazovat pouze na vývojáře, který má odpovídající znalosti, aby je mohl zpracovat. Přesněji řečeno, FullStack vývojář by měl být schopen zastoupit jak FE, tak i BE protějšky.

Řešení

Uvedený příklad by zcela jistě šel vyřešit i jinak než pomocí vícenásobné dědičnosti, ale pro demonstraci bude řešení založeno právě na ní. Zároveň bude ale aplikace napsána v jazyce, který vícenásobnou dědičnost nativně nepodporuje, takže předpokladem je, že FullStack vývojář nemůže být zároveň potomkem FE vývojáře i BE vývojáře. Za pomoci *dvojčete* lze toto chování ovšem nasimulovat, a to tak že FullStack vývojáře budou ve skutečnosti reprezentovat dvě třídy — *FullStack_FE* a *FullStack_BE*, které budou potomky korespondujících tříd pro FE vývojáře a BE vývojáře. Mimo jiné budou tyto dvě třídy navíc mít vzájemné reference jedna na druhou, pomocí kterých budou moci využívat chování zděděné z obou rodičovských tříd.

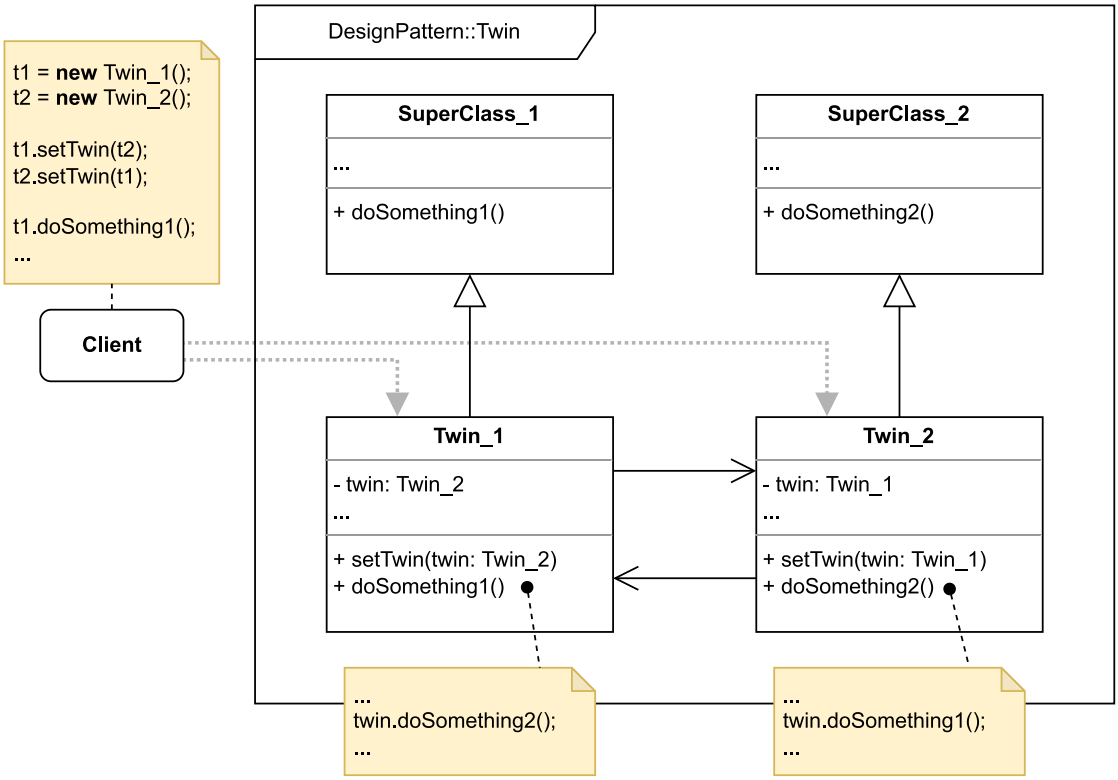
Obecná struktura tohoto vzoru je zobrazena na diagramu 2.14. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

SuperClass_1, SuperClass_2 jsou rodičovské třídy, ze kterých chce potomek zároveň dědit (ekvivalentní třídám pro FE vývojáře a BE vývojáře z uvedeného příkladu)

Twin_1, Twin_2 jsou části potomka, které společně simulují vícenásobnou dědičnost (ekvivalentní třídám FullStack_FE a FullStack_BE z uvedeného příkladu; společně reprezentují FullStack vývojáře)

Client je externí entita pracující s dvojčetem, která je zároveň zodpovědná za proces inicializace a provázání dílčích podtříd

[15] zabývá se klientskou částí aplikací
[16] zabývá se serverovou částí aplikací
[17] má kombinované znalosti FE i BE



Obrázek 2.14 Třídní diagram obecného *dvojčete*

Zhodnocení

Výhody:

- poskytuje vícenásobnou dědičnost v prostředích, kde není nativně podporována
- lze použít k eliminaci střetu názvů (častý problém vícenásobné dědičnosti)

Nevýhody:

- není přesnou kopií vícenásobné dědičnosti [20]
  - další dědění pouze jednoho z „twin objektu" vyžaduje, aby zduplikoval veškeré rozhraní a doplnil implementaci, která pouze provolává protějšek
  - přidávání více rodičovských tříd bude exponenciálně růst na komplexitě, protože každá vyžaduje vlastní „twin objekt", který musí obsahovat reference na všechny ostatní
- jednotlivé „twin objekty" jsou mezi sebou úzce svázány

2.2.9 Mock Object

Mock je návrhový vzor, který principiálně spadá do kategorie *structural* vzorů. Jádrem jeho podstaty je testovatelnost aplikací — konkrétně se snaží ulehčit a zlepšit testování tříd se závislostmi na jiné třídy.

Motivace

Průběžné testování aplikací pomocí automatizovaných testů je zcela nezbytnou, ale poměrně zanedbávanou, součástí moderního vývoje. Mnohé problémy zde vznikají s existencí tříd, které jsou úzce svázány s dalšími objekty — testy takových tříd reálně kontrolují moc velkou část kódu najednou a jejich případné selhání neposkytne takřka žádné relevantní informace o tom, kde přesně se vykytuje jádro problému [21]. V praxi platí, že čím jemnější granularita testů, tj. každý test testuje co nejmenší možný kus chování, tím lepší výsledky.

Demonstračním příkladem může být vlastně libovolná třída se závislostmi. Víceméně každá aplikace bude podporovat nějaký druh logování a třída zprostředkovávající tuto funkčnost (tzv. *logger*) poslouží jako skvělá ukázka. Univerzálnější *logger* by mohl mít referenci na třídu, která bude zprostředkovávat přístup k médiu, kam se mají dané logy ukládat (tj. konzole, soubory, ElasticSearch[18], …). Najednou otestování takového *logger* objektu není jen tak, protože pro svou činnost potřebuje další třídu, a to má za důsledek to, že jeho jednotlivé testy budou nepřímo testovat i v daný moment nainstancovanou třídu zprostředkovávající uložení samotných dat (resp. logů). To není zrovna žádoucí vedlejší efekt, protože selhání takových testů neřekne nic o tom, zda je chyba přímo v *loggeru* nebo podpůrné třídě ukládající data.

Řešení

Uvedený příklad je přesně momentem, kdy se při testování dá nádherně využít *mock objektů*. Přesněji u testů *loggeru* se vytvoří třída se stejným rozhraním jako mají všechny třídy zprostředkovávající ukládání logů do různých destinací, jejíž implementace bude simulovat uložení třeba do paměti. Objekt této třídy je následně v jednotlivých testech používán v kombinaci s testovaným *logger* objektem, čímž je eliminována potřeba reálných implementací a s nimi spojených výskytů parazitního chování.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.15. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

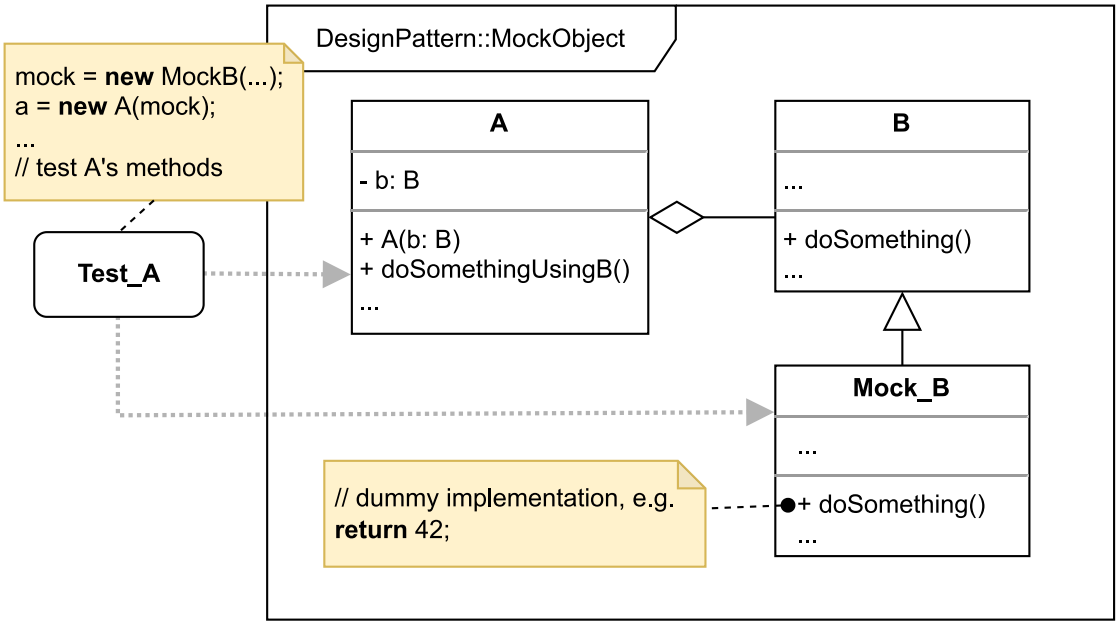
A je testovaná třída, která obsahuje závislost na třídu B (ekvivalentní *logger* třídě z uvedeného příkladu)

B je třída poskytující nějakou funkcionalitu třídě A (ekvivalentní třídám, které zajišťují uložení logů na různá média, z uvedeného příkladu)

Mock_B je „mockovací" třída, která simuluje chování třídy B a v testech je předhazována objektu typu A, který ji nějak využívá (ekvivalentní *mocku*, který simuluje ukládání logů do paměti, z uvedeného příkladu)

Test_A je test jednotlivých funkčností třídy A (ekvivalentní jednotlivým testům *logger* třídy z uvedeného příkladu)

[18] nástroj pro ukládání JSON dokumentů s fulltextovým vyhledáváním



Obrázek 2.15 Třídní diagram obecného *mock objektu*

Zhodnocení

Výhody:

- zmenšuje rozsah testů (to znamená užitečnější výsledky)
- eliminuje potřebu dodatečných prostředků (úložišť, přístupu na internet, …)
- zrychluje testy
- v *TDD*[19] často odhalují podobu potřebných rozhraní

Nevýhody:

- zanáší do aplikace extra třídy, které se musí udržovat
- vyžaduje znalost funkčnosti, kterou mají simulovat (aby jí mohly simulovat alespoň vzdáleně podobným způsobem)

Zajímavosti

Mock objekty se do jisté míry staly nedílnou součástí unit testů, takže v dnešní době je pomalu nevídané narazit na strohé implementace jako jsou uvedeny zde — většinou se s nimi vývojář setká nepřímo v podobě *mockování* pomocí testovacích frameworků.

[19] postoj k vývoji SW, který je založený na myšlence prvotního sepsání testů, které následuje samotné naprogramování funkcionalit

2.3 Vzory týkající se chování (Behavioral)

Poslední ze skupin návrhových vzorů, které představila skupina *GoF*, jsou vzory zabývající se interakcemi mezi entitami. Blíže zkoumají algoritmy jako celky, při čemž se zaměřují na vnitřní přerozdělení zodpovědností a způsoby komunikace mezi jednotlivými třídami. Jejich hlavní snahou je přesunout vývojářovu pozornost od složitých programových cest (tj. rozvětvování, cykly, …) směrem k propojování objektů [5]. Mnoho vzorů z této kategorii vypadá na první pohled velmi podobně (např. *stav* a *strategie*), ovšem s hlubším pohledem jsou často vidět fundamentální rozdíly v interakci dílčích objektů.

2.3.1 Chain of Responsibility

Řetěz zodpovědnosti (také *CoR*) je *behavioral* návrhový vzor, který popisuje způsob zpracování nějaké zprávy vícero po sobě jdoucími objekty. Přesněji řečeno, definuje fiktivní řetěz, přes jehož jednotlivé články je postupně předávána žádost, která může být jaksi zpracována každým z nich.

Motivace

Tento vzor je vhodný zejména pro aplikace, které pracují s různými druhy zpráv, jejichž zpracování lze modularizovat na více přepoužitelných kroků. S dostatečnou rozmanitostí takových zpráv by se naivní řešení mohlo velmi jednoduše rozrůst do nemyslitelných rozměrů. CoR poskytuje velmi elegantní a flexibilní způsob, pomocí kterého se lze vyhnout zbytečným detailům a dojít k poměrně efektivnímu východisku.

Dobrým příkladem, ve zmenšeném měřítku, mohou být bankovní aplikace pro řízení schvalovacího procesu, požádá-li si klient o úvěr, hypotéku či jinou obdobnou půjčku. Taková žádost musí projít několika kroky, kde každý z nich kontroluje jistým způsobem všelijaké informace o klientovi a rozhoduje, zda je způsobilý pro danou výpůjčku. Zároveň různé typy půjček budou vyžadovat rozdílné mezi kroky a budoucí rozvoj nabízených produktů pravděpodobně přinese nové druhy kontrol.

Řešení

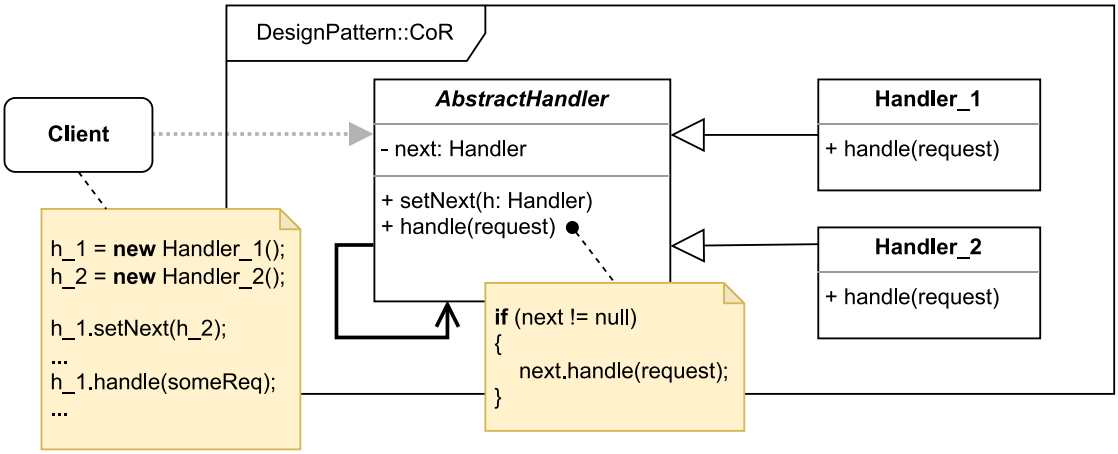
Aplikaci z uvedeného příkladu lze šikovně navrhnou právě pomocí *CoR*. Řešení bude spočívat v tom, že se vytvoří třída pro každý ověřovací krok, načež schvalovací proces jako celek bude reprezentován pomocí CoR seznamu, který bude odlišný pro různé typy úvěrů. Žádost potom bude postupně předávána tímto seznamem, kde ji každý článek zpracuje, a pokud bude v pořádku, tak ji předá dalšímu v pořadí. Toto řešení je skvěle připraveno na budoucí rozšiřování ve formě nových ověřovacích kroku, a tedy i rozšiřování portfolia nabízených úvěrů.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.16. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

AbstractHandler je obecný předpis rozhraní pro všechny články v řetězu (nejdůležitější je odkaz na další článek v pořadí)

Handler_1, Handler_2, ... jsou konkrétní články, které provádí nějaké zpracování předávané zprávy/žádosti (ekvivalentní jednotlivým ověřovacím krokům žádosti z uvedeného příkladu)

Client je externí entita vyvolávající proces zpracování provoláním prvního článku v řetězci (mimo jiné je zodpovědná za poskládání řetězu, který se skládá z jednotlivých článků)



Obrázek 2.16 Třídní diagram obecného CoR

Zhodnocení

Výhody:

- extrémně flexibilní vůči budoucím rozšířením
- snižuje úroveň závislosti mezi klientem a pracujícími třídami
- poskytuje kontrolu nad pořadím zpracovávaných operaci
- umožňuje dynamicky modifikovat proces zpracování

Nevýhody:

- vyšší úroveň abstrakce (z kódu nemusí být na první pohled jasné, co se bude dít, což s sebou nese těžší odhalování chyb)
- chyba v jednom článku způsobí přerušení celého procesu

Zajímavosti

Principem se velmi podobá *dekorátoru* (sekce 2.2.4) — hlavní rozdíl je, že *CoR* je plně lineární a může kdykoliv přerušit zpracování, zatímco *dekorátory* ne. *CoR*, *příkaz* (sekce 2.3.2), *prostředník* (sekce 2.3.4) a *pozorovatel* (sekce 2.3.6) všichni nějakým způsobem propojují dvě strany (odesílatele a příjemce) každý s jinou záminkou — *CoR* předává zprávu seznamem příjemců, *příkaz* vytváří obousměrné propojení, *prostředník* se staví mezi dvě komunikující strany a *pozorovatel* umožňuje příjemcům rozhodovat, zda chtějí dostávat novinky od odesílatelů.

2.3.2 Command

Příkaz (také *akce* nebo *transakce*) je *behavioral* návrhový vzor, který stojí na myšlence reprezentování veškerých operací formou objektů. Tyto objekty poté zaobalují všechny potřebné informace, díky čemuž lze s operacemi pracovat ve zcela novém rozměru.

Motivace

Zásadní motivací za tímto vzorem jsou problémy, které spočívají v potřebě dodatečně manipulovat s procesem vyvolávání funkčností jako takovým, namísto manipulace s konkrétní logikou. Nejtypičtějšími případy bývají implementace reverzibilních akcí nebo front — obecně se dají shrnout jako případy, které vyžadují zachycení a reprezentaci „provolání metody nebo funkce".

Skvělou demonstrační aplikací může být jednoduchá kalkulačka. Jedna z velmi užitečných funkcionalit je anulace předešle aplikovaných matematických operací nebo jejich případné zopakování. Jinými slovy řečeno, bylo by vhodné ukládat posloupnost provedených výpočtů.

Řešení

Kalkulačku se zmíněnou logikou je vhodné nadesignovat právě pomocí *příkazů*, jelikož je potřeba zachytit celou historii postupně prováděných výpočtů a zároveň poskytnout mechanizmus, pomocí kterého bude možný „pohyb do minulosti a zpět". Přesná implementace by obsahovala: příkazovou třídu, která zaobalí informace o prováděném výpočtu (matematickou operaci a číselnou hodnotu), pracovní třídu, která bude vykonávat výpočty vůči dosavadnímu výsledku a kolekci, která bude udržovat historii příkazů. Nyní když se klient bude chtít pohybovat po časové ose, aby mohl zvrátit/zopakovat zaregistrované výpočty, bude se pouze posouvat ukazatel na aktuální stav do kolekce s historii a přeposílat příkaz pracovní třídě o provedení odpovídajícího výpočtu — krok zpět bude aplikace inverzní matematické operace vůči uloženému záznamu a krok kupředu bude pouhé zopakování uloženého záznamu.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.17. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

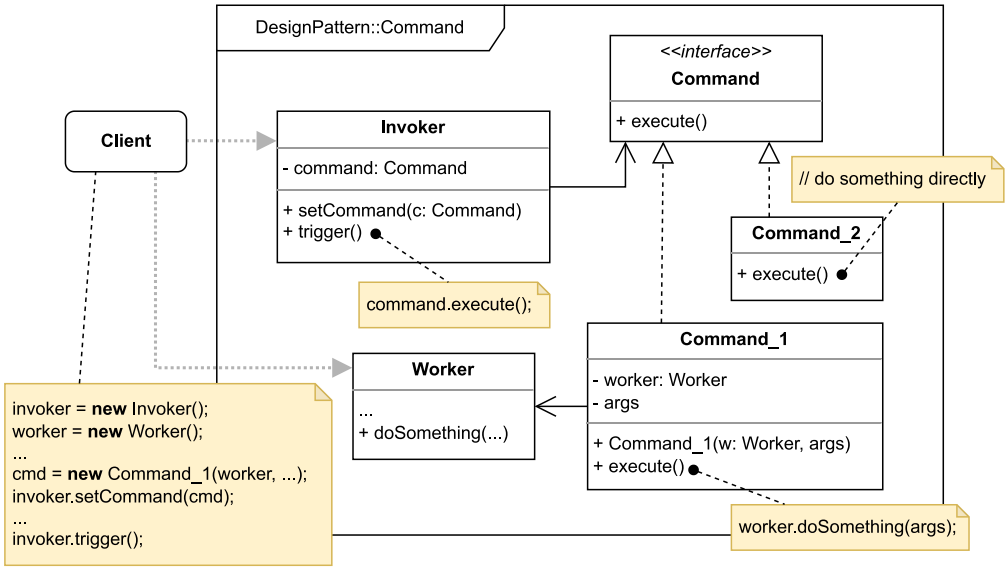
Invoker je entita, která vyvolává nějakou akci (bylo by ekvivalentní uživatelskému rozhraní kalkulačky z uvedeného příkladu)

Command je společné rozhraní, které musí dodržovat všechny příkazy (většinou bude definovat jedinou metodu *execute*)

Command_1, ... jsou konkrétní příkazy (ekvivalentní výpočtům z příkladu) — mohou nabývat dvou variant: častěji příkaz pouze zaobaluje informace a práci deleguje na pracovníka (*Command_1*), ale je i možné, že sám o sobě vykoná nějakou logiku (*Command_2*)

Worker je entita s nějakou business logikou, která je vyvolávána odpovídajícími příkazy (ekvivalentní třídě provádějící výpočty z uvedeného příkladu)

Client je externí entita, která většinou provádí prvotní nastavení (tj. hlavně nastavení, jaké události budou produkovat jaké příkazy — provázání *Invoker* s konkrétním *Command*)



Obrázek 2.17 Třídní diagram obecného *příkazu*

Zhodnocení

Výhody:

- zmenšuje provázanost objektů, které vyvolávají akci a objektů, které ji zpracovávají
- realizuje jinak těžce zhotovitelné funkcionality (ukládání do front, reverzibilní akce, …)
- vysoká flexibilita vůči přidávání nových příkazů

Nevýhody:

- „nafouknutí kódu" (každý příkaz vyžaduje vlastní třídu)

Zajímavosti

Při použití na reverzibilní akce se vyplatí využít společně s *mementem* (sekce 2.3.5). Implementaci historie provedených operací může ulehčit zkombinovaní s *prototypem* (sekce 2.1.4).

2.3.3 Iterator

Iterátor je *behavioral* návrhový vzor s jediným účelem, kterým je poskytnutí prostředků pro procházení elementů různých kolekcí (list, strom, mapa, …) pomocí různých algoritmů.

Motivace

Kolekce jsou nedílnou součástí naprosté většiny programů, díky čemuž je i tento vzor jedním z důležitějších, se kterými by měl být každý vývojář seznámen. Používá-li klient komplexnější datové struktury, může být iterace přes jejich elementy poměrně náročným úkolem a hlavně zbytečně se opakujícím kódem napříč aplikací. Zde přichází na scénu iterátor, který vyčleňuje daný iterační algoritmus na jedno místo, čímž zároveň nádherně odstiňuje klienta od specifických implementačních detailů kolekce.

Častým důvodem pro implementaci vlastních iterátorů je požadavek nestandardní průchod již existujících kolekcí. Tím může být například funkce kaskádových stylů *nth-child* ze světa webových stránek. Jedná se o funkci, která umožňuje uživateli definovat libovolnou posloupnost, pomocí které se následně vybere každý n-tý prvek ze seznamu daných elementů. Přesněji řečeno, uživatel může třeba vhodným selektorem vybrat všechna tlačítka na webové stránce a následně pomocí funkce *nth-child* každému čtvrtému nastavit jinou barvu pozadí než má zbytek.

Řešení

Zmíněná funkce je ideálním adeptem na využití vlastního *iterátoru*, jelikož pracuje s předem stanovenou kolekcí elementů a pouze přes ně potřebuje iterovat velmi specifickým způsobem. Konkrétní implementace by vypadala tak, že by se vytvořil nový typ iterátoru pro kolekci zaobalující vybrané elementy, který by obsahoval referenci na konkrétní kolekci, předpis posloupnosti a číslo aktuální iterace. Metoda, která vrací další prvek potom pouze vezme číslo iterace, dopočítá pomocí posloupnosti pozici, na tu si sáhne do kolekce a vrátí odpovídající prvek.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.18. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

Collection je obecný předpis rozhraní pro všechny konkrétní kolekce, které budou podporovat vytváření vlastních iterátorů

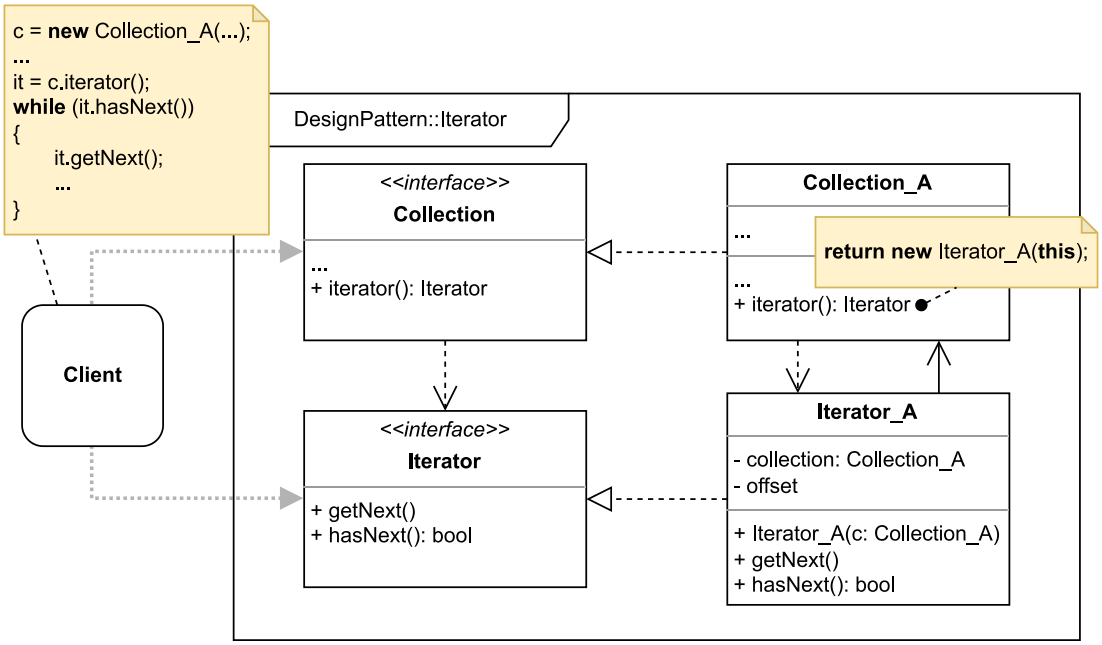
Collection_A je konkrétní kolekce s daty (ekvivalentní seznamu elementů z uvedeného příkladu)

Iterator je společné rozhraní všech iterátorů (většinou bývá předdefinováno konkrétním programovacím jazykem)

Iterator_A je konkrétní iterátor provázaný s konkrétní kolekcí (ekvivalentní iterátoru, který vrací n-té prvky ze seznamu elementů dle nějaké posloupnosti, z uvedeného příkladu)

Client je externí entita, která vlastní konkrétní kolekci s daty, ke které si vytváří iterátor, pomocí kterého přes ní bude iterovat

Ačkoliv vyobrazená struktura je obecným řešením, jedná o natolik používaný vzor, že naprostá většina programovacích jazyků předdefinovává obecná rozhraní, která následně stačí naimplementovat. V praxi se tedy vývojáři většinou nesetkají s vytvářením všech rozhraní/tříd, ale spíše budou implementovat pouze konkrétní iterační algoritmus — to s sebou ponese výhodu symbiózy s konstrukty daného programovacího jazyka (např. *foreach* cykly).



Obrázek 2.18 Třídní diagram obecného *iterátoru*

Zhodnocení

Výhody:

- vysoká podpora pro přidáváni nových iterátorů a podporovaných kolekcí (bez rozbití stávajícího kódu)
- umožňuje libovolné varianty procházení (od začátku, od konce, každý druhý, …)
- lze zachovat aktuální stav (tzn. umožňuje „pozastaveni")
- každý iterátor je samostatnou instancí, takže lze paralelně provádět vícero iterací nad jednou kolekcí, aniž by se jakkoliv ovlivňovali

Nevýhody:

- horší výkonnost a paměťová efektivita oproti přímočarému přístupu k prvkům kolekce

Zajímavosti

Často se využívá k postupnému procházení komplikovaných *kompozitních stromů* (sekce 2.2.3) a v‚kombinaci s *mementem* (sekce 2.3.5) pro jednoduché navrácení do uložených stavů.

2.3.4 Mediator

Prostředník (také *zprostředkovatel* nebo *kontrolor*) je *behavioral* návrhový vzor, který zabraňuje chaotickým závislostem mezi několika přímo komunikujícími objekty, tím že se mezi ně postaví a diriguje jejich vzájemnou výměnu informací.

Motivace

Poměrně vídaným problémem u špatně navržených obsáhlých systémů bývají zmateně provázané objekty, což bývá důsledkem jejich vzájemných interakcí. Pokud tento problém není včasně adresován, dojde velmi jednoduše ke zvýšení nároku na údržbu a ztrátě flexibility. Prostředník zakládá právě na tom, že vnese určitý řád do kooperací jednotlivých objektů, tím že veškerá komunikace „poteče" přes něho. Díky této myšlence nebudou všechny entity zmateně závislé mezi sebou, ale pouze na jednom centrálním objektu.

Zajímavým demonstračním příkladem může být webová aplikace se zákaznickou linkou ve formě online chatu. V dnešní době je poměrně běžné, že větší společnosti využívají *chatbotů*[20], kteří komunikují s klienty o méně komplexních tématech. Až v době, kdy si bot neví rady, je komunikace předána skutečné osobě, resp. zaměstnanci. Tuto část aplikace je nevhodné řešit přímým přístupem a zanášet do odpovídajících objektů logiku předávání zpráv, a tím je zbytečně svazovat k sobě.

Řešení

Vhodnějším řešením zmíněného chatu je právě vyseparování komunikačních detailů do samostatného objektu, který se bude starat o přijetí zpráv z jedné strany a následně jejich předání příslušné protistraně. Jinými slovy přesně přístup, který popisuje *prostředník*. Ohromným přínosem tohoto řešení je, že klientská logika (myšleno z pohledu kódu), a obecně logika všech komunikujících stran, nebude znečišťována rozhodováním jak a komu poslat zprávu — všichni budou komunikovat pouze se zprostředkovatelem a ten už si zbytek zařídí sám.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.19. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

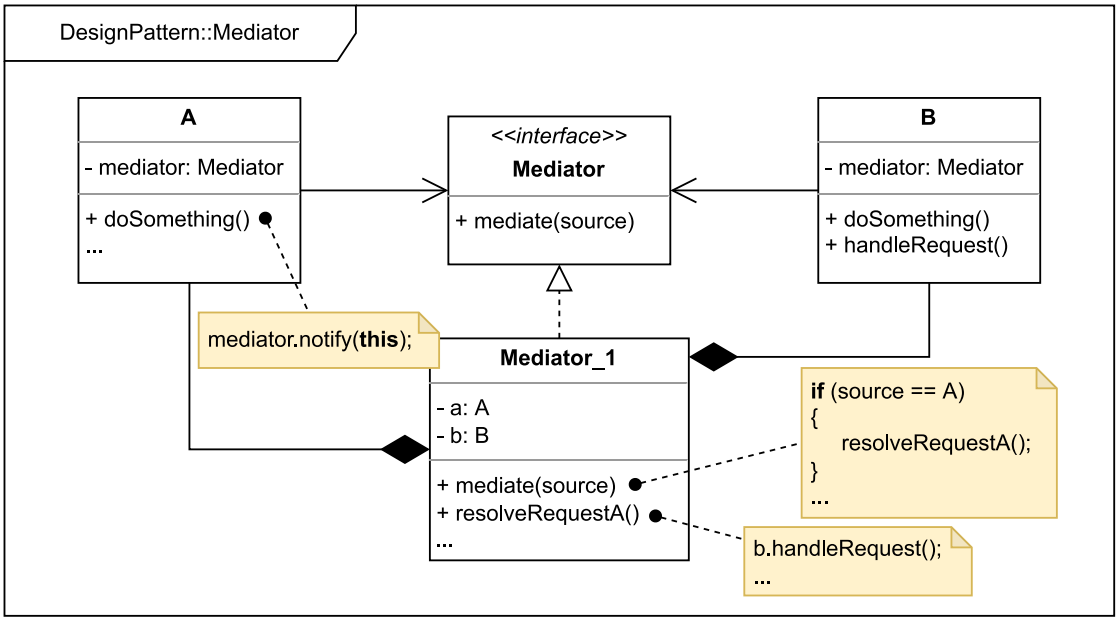
Mediator je obecný předpis rozhraní pro všechny prostředníky (většinou bude obsahovat pouze jednu metodu, pro dirigaci komunikace — „odkud kam co poslat")

Mediator_1 je již konkrétní prostředník, který řídí komunikaci mezi objekty A, B (ekvivalentní objektu, který se stará o předávání zpráv, z uvedeného příkladu)

A, B jsou konkrétní objekty, které spolu komunikují, avšak mezi sebou se vůbec nevidí a veškerá komunikace je řízena prostředníkem (ekvivalentní zákazníkům/zaměstnancům/botům z uvedeného příkladu)

Vyjma zmíněných entit je ještě potřeba nepřímo se zúčastňující klient, který je zodpovědný za životní cykly individuálních komunikujících objektů a jejich prvotní provázání s daným prostředníkem.

[20] počítačový program určený k automatizované komunikaci s lidmi (nejčastěji zákazníky)



Obrázek 2.19 Třídní diagram obecného *prostředníka*

Zhodnocení

Výhody:

- snižuje provázanost komunikujících komponent (vyšší flexibilita a znovupoužitelnost)
- přesouvá detaily komunikace z konkrétních objektů na separátní místo s jediným úkolem (princip SRP)

Nevýhody:

- prostředník se velmi lehce stane vysoce provázaným s většinou tříd systému (antivzor, viz. sekce 1.2.4)

Zajímavosti

Z jistého pohledu si je podobný s *fasádou* (sekce 2.2.5) — rozdílem je, že podpůrné objekty *fasádu* vůbec nevidí a mohou si komunikovat mezi sebou, zatímco *prostředník* je přesným opakem, kde podpůrné objekty vidí pouze jeho a o sobě navzájem nemají vůbec tušení. Svou implementací je v mnoha případech je zaměnitelný s *pozorovatelem* (sekce 2.3.6) — rozdíl je viditelný spíše jen v mezních případech, kde *pozorovatel* je natolik dynamický, že nesplňuje definici *prostředníka* a nebo naopak *prostředník* je natolik statický, že nesplňuje definici *pozorovatele*.

2.3.5 Memento

Memento (také *snapshot*) je *behavioral* návrhový vzor s velice přímočarou myšlenkou, a to umožnit ukládání/načítání vnitřního stavu objektů.

Motivace

Podstata tohoto vzoru je velmi jednoduchou záležitostí na pochopení — v podstatě umožňuje vzít libovolný objekt s nějakým vnitřním stavem a rozšířit ho o funkcionalitu ukládání kopií, které lze v budoucnu využít pro návrat do stavu, ve kterém se objekt vyskytoval v době uložení. Hlavně řeší problém ukládání objektů s privátními atributy — ty nelze ručně překopírovat při pohledu zvenčí, bez porušení významu jejich privatizace. Využívá se zejména při udržování „interaktivní historie", tj. styčných bodů v minulosti, do kterých je možné se kdykoliv navrátit.

Například může poměrně zjednodušit implementaci zvratných operací. Třeba v kontextu takového e-shopu, je velmi pravděpodobná existence nějaké administrační komponenty, pomocí které bude možné editovat stávající produkty. Sofistikovanější verze bude u této funkčnosti zobrazovat dialogové okno se seznamem provedených změn a tlačítko pro návrat do stavu před editací, pro jednoduchou opravu nežádoucích změn provedených omylem.

Řešení

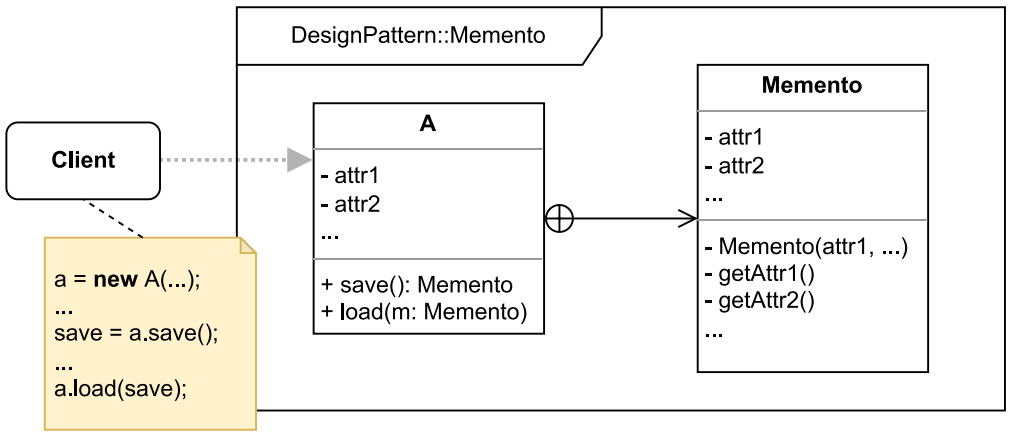
Ačkoliv by navrácení změn šlo udělat několika způsoby, použití *mementa* je jedním z objektivně atraktivnějších přístupů. Z pohledu kódu řešení vypadá tak, že třída reprezentující produkty se rozšíří o vnořenou neměnnou třídu, která bude pouze zaobalovat všechny atributy produktu. Dále se produktová třída rozšíří o dvě metody pro vytváření a načítání instancí právě zmíněné vnořené třídy. Na závěr editace produktů bude probíhat tak, že před jejím potvrzením se vytvoří memento s aktuálním stavem, které se v případě potřeby využije pro zvrácení provedených změn.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.20. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

A je objekt, který umí ukládat/načítat svůj stav (ekvivalentní produktům z uvedeného příkladu)

Memento je neměnný objekt s uloženým stavem objektu typu A (ekvivalentní mementům produktů z uvedeného příkladu)

Client je externí entita, která pracuje s objekty typu A a spravuje jejich mementa (ekvivalentní části kódu, která orchestruje samotnou editaci produktů, z uvedeného příkladu)



Obrázek 2.20 Třídní diagram obecného mementa

Uvedená a zároveň i nejpoužívanější varianta využívá vnořených tříd, které jsou pevně svázané s nadřazenou třídou. Existují ovšem programovací jazyky, které tuto funkčnost neumí zreplikovat (např. PHP), pro něž byly vymyšleny i alternativní implementace, které vyčleňují memento do normální třídy a privatizaci stavu zachovávají osekáním dostupného rozhraní — blíže popsáno třeba na webu *refactoring.guru* [10].

Zhodnocení

Výhody:

- jednoduchá obnova objektu do stabilního stavu
- zachovává principy zapouzdření (oproti alternativním řešením stejného problému)

Nevýhody:

- dynamické programovací jazyky neumí zaručit neměnnost mement (takže uložený stav je teoreticky možné zmodifikovat před jeho opětovným načtením)
- klienti, kteří spravují mementa by měli mít přehled o životním cyklu odpovídajících objektů (aby drželi pouze relevantní kopie a zbytečně „nežrali" paměť)

2.3.6 Observer

Pozorovatel (také *naslouchač*) je *behavioral* návrhový vzor, který v podstatě poskytuje mechanizmus, pomocí kterého lze sjednoceně upozorňovat sadu objektů o změnách, akcích a podobných událostech provedených jinou řídící entitou. Jedná se o obdobu pravděpodobně mírně známějšího *publish-subscribe* modelu.

Motivace

Obecně se tento vzor využije ve chvíli, kdy je v aplikaci *one-to-many* vztah a objekty na straně „many" je nutno automaticky notifikovat o změnách objektu na straně „one". Lépe řečeno, jedná se o problém, kdy v doméně existuje jedna produkční entita, která vykonává různé činnosti nebo produkuje nějaký obsah, a vedle ní sada pozorujících objektů, které musí reagovat na právě provedené změny.

Skvělým příkladem praktického využití tohoto vzoru ve větším měřítku jsou třeba aplikace železničních společností, které zobrazují plánovaná vlaková spojení a zároveň slouží jako nákupní portál na jízdenky. Pokud taková aplikace bude co k čemu, měla by uživatelům aktivně poskytovat informace o vlacích, do kterých mají zakoupeno jízdenku — informace jako případná zpoždění, přesměrování nebo zrušení příslušných spojení.

Řešení

Zmíněná aplikace by mohla nádherně zužitkovat *pozorovatele* a to tak, že třída reprezentující daný vlak bude hrát roli produkční entity, která bude rozesílat oznámení o důležitých upozorněních. Tyto informace poté budou chodit pouze objektům, které se u ní zaregistrují, že je chtějí dostávat. Těmito objekty budou jednotlivý uživatelé a proces registrace na dané spojení bude probíhat v momentě, kdy si uživatel koupí odpovídající jízdenku. Na konec odhlášení uživatele proběhne ve chvíli, kdy vlak dorazí do stanice, do které měl cestující zakoupenou jízdenku.

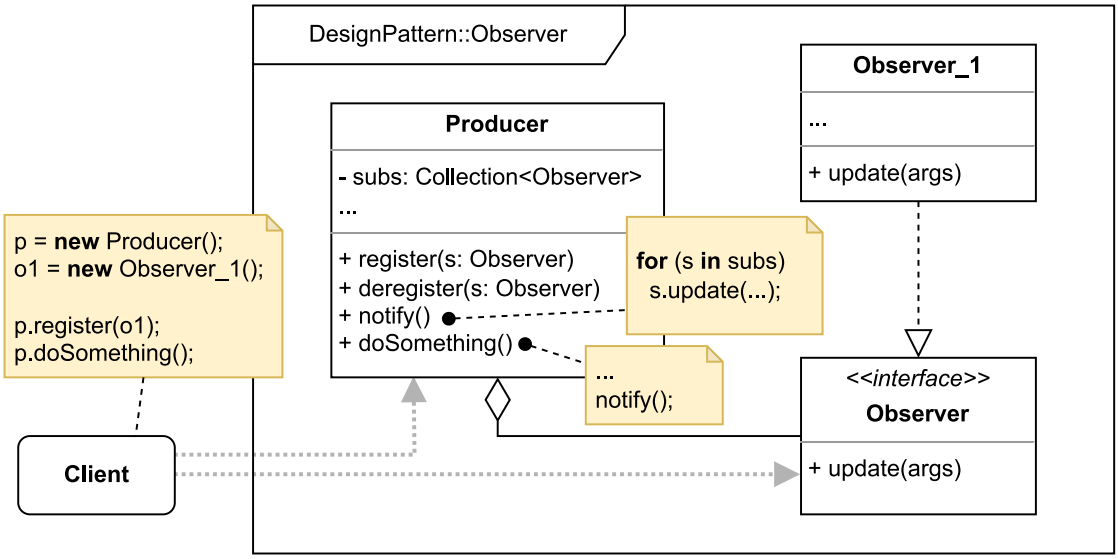
Obecná struktura tohoto vzoru je zobrazena na diagramu 2.21. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

Producer je produkční entita (*producent*) o jejichž akcích budou notifikovány okolní objekty (ekvivalentní vlaku z uvedeného příkladu)

Observer je obecné rozhraní pro všechny pozorující objekty daného *producenta*, které chtějí dostávat notifikace (zde je potřeba důkladně promyslet, jaké informace budou předávány)

Observer_1 je konkrétní *pozorovatel*, který dostává notifikace od *producenta* a nějak na ně reaguje (ekvivalentní uživatelům, kteří mají jízdenku na nějaký vlak, z uvedeného příkladu)

Client je externí entita, která spravuje vytvoření/zrušení registrace *pozorovatelů* k odpovídajícím *producentům* (ekvivalentní nákupu jízdenky a dojezdu do cílové stanice z uvedeného příkladu)



Obrázek 2.21 Třídní diagram obecného *pozorovatele*

Zhodnocení

Výhody:

- dynamické přidávání a odebírání pozorovatelů při runtimu
- flexibilní rozšiřování typů pozorovatelů, které nevyžaduje zásah do stávajícího kódu
- podporuje nízkou provázanost mezi komunikujícími objekty (producenty a pozorovateli)

Nevýhody:

- pořadí vyslaných upozornění je náhodné (resp. v pořadí, v jakém se registrovali pozorovatelé)
- nedostatečný výkon může způsobit nežádoucí chování (dříve zaregistrovaní pozorovatelé dostanou upozorněni mnohem dříve, než jedni z posledních)

Zajímavosti

V poslední době možná používanější *publish-subscribe* vzor je téměř ekvivalentním s hlavním rozdílem, že producenti a pozorovatelé o sobě vůbec neví a mají mezi sebou tzv. *brokera*, který je zodpovědný za správnou distribuci notifikací [22]. Pozorovatel také dobře spolupracuje s *řetězem zodpovědnosti* (sekce 2.3.1) a *prostředníkem* (sekce 2.3.4), jak je zmíněno mezi jejich zajímavostmi.

2.3.7 State

Stav je *behavioral* návrhový vzor, který umožňuje objektům pozměnit své chování na základě jejich vnitřního stavu. V jistém slova smyslu se dá říci, že objekty potom fungují na obdobném principu jako *konečné automaty*[21].

Motivace

Tento vzor staví na objektech, které se po dobu jejich životního cyklu nachází v různých logických stavech, které diktují, jakým způsobem se mají chovat. V daný moment aktivní stav poté určuje, jak se zrovna objekt zachová při vyvolání některé z jeho metod. Typickým indikátorem vhodných stavových objektů bývají metody s robustními opakujícími se *if-else* podmínkami, které provádějí rozhodování na základě třídních atributů.

Příkladem, kde se často využívá *stavů*, může být třeba počítačová hra. Přesněji odlišné vlastnosti entit ovládaných hráčem. Obyčejná FPS[22] hra bude pravděpodobně hráči umožňovat měnit styl jeho pohybu (tj. chůze, běh, sprint, dřep, …). V konkrétní moment aktivní stav by měl zjevně ovlivňovat další akce, jako například rychlost pohybu, nemožnost ADS[23] při sprintu či všelijaká obdobná omezení.

Řešení

Naivní řešení takové mechaniky by obnášelo právě konstantně se opakující *if-else* podmínky nějakého indikátoru aktivního stavu napříč celým kódem. Nejhorší vlastností takového přístupu je, že jakákoliv změna by znamenala ohromné množství práce. Mnohem lepší variantou bude využít *stavového vzoru*, který proměnlivé chování vyseparuje do stavových objektů a v hlavní třídě bude definovat pouze referenci na zrovna aktivní stav, který bude zodpovědný za provedení vhodné akce. Konkrétně třída reprezentující ovládanou postavu bude mít svůj stav, na který budou delegovány stavově specifické úkony (např. získání rychlosti pohybu).

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.22. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

A je objekt, který může nabývat různých logických stavů a na jejich základě mění své chování (ekvivalentní postavě z uvedeného příkladu)

State je obecné rozhraní společné pro všechny stavy svázané s konkrétní třídou typu A (musí definovat vše co bude stavově specifické)

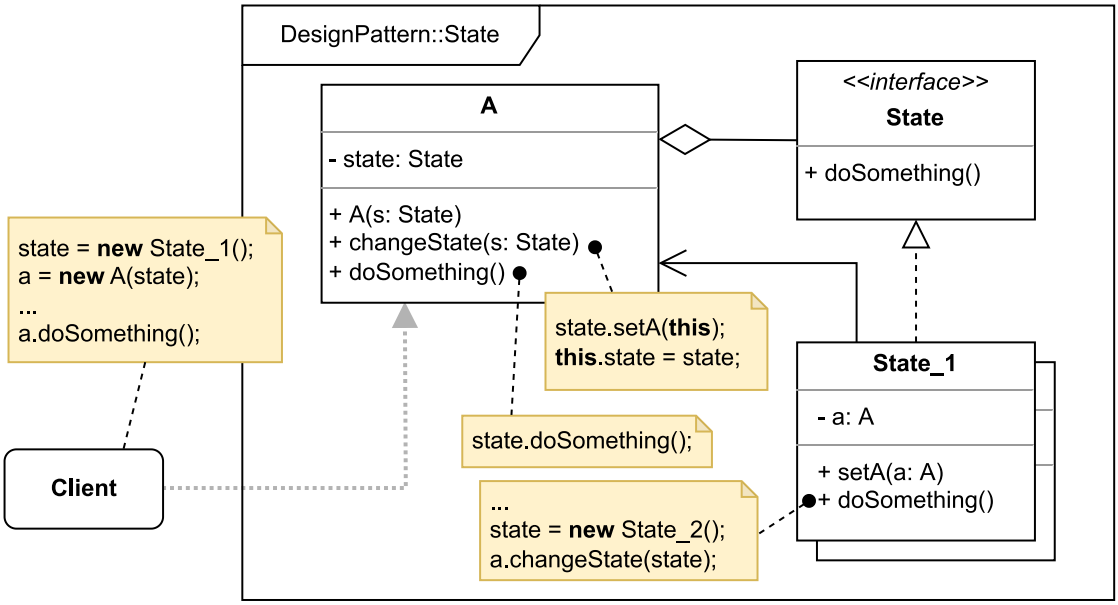
State_1, ... jsou konkrétní stavy obsahující odlišná chování (ekvivalentní stavům chůze/dřep/… z uvedeného příkladu); reference na hlavní objekt (A) není povinná, ale často se používá, aby sám stav mohl vyvolat přechod do stavu jiného

Client je externí entita, která pracuje s objekty typu A (na diagramu nastavuje prvotní stav, ovšem v praxi se většinou o stavy objektu A nebude vůbec zajímat)

[21] výpočetní model, který se vždy nachází v jednom z vícera stavů, mezi kterými přechází na základě přečtených vstupů, čímž reflektuje různá chování

[22] First-Person Shooter — střílečka z pohledu první osoby

[23] Aim Down Sights — změna pohledu v FPS hrách na míření skrze zaměřovač zbraně



Obrázek 2.22 Třídní diagram obecného stavu

Zhodnocení

Výhody:

- snižuje provázanost mezi objekty a jejich stavově závislým chováním
- přidaní nového stavu nevyžaduje zásah do stávajícího kódu
- chování lze měnit dynamicky při runtimu
- odstraňuje robustní *if-else* podmínky

Nevýhody:

- vhodné pouze za přítomnosti většího počtu stavů
- zhoršení paměťové efektivity

Zajímavosti

Stav a *strategie* (sekce 2.3.8) jsou si velmi podobnými, s tím že *stav* může být považován za lehké zobecnění — oba vlastně mění chování objektu s využitím pomocných tříd s tím rozdílem, že jednotlivé *strategie* vůbec netuší o vzájemné existenci, zatímco stav toto omezení nevynucuje a často i implementuje možnost, že jeden stav vyvolává přechod do jiného. Obecně *strategie* by měly zaobalovat jednu konkrétní věc a být jistému kontextu předávány, zatímco *stavy* postihují širší spektrum funkcionalit a bývají zcela v režii daného kontextu.

2.3.8 Strategy

Strategie (také *politika*) je *behavioral* návrhový vzor, který umožňuje aktivně vyměňovat používané algoritmy za běhu aplikace.

Motivace

Překážka stojící na pozadí tohoto vzoru je velmi přímočará. V podstatě řeší, jak v rámci runtimu, lehce zaměňovat různé algoritmy ze stejné rodiny. Rodinou je zde myšleno seskupení takových algoritmů, které řeší jeden a ten samý problém — například řadící algoritmy. Typická potřeba zmíněného chování vyvstává ve chvíli, kdy se v aplikaci nachází rodina algoritmů, mezi jejíž členy je potřeba přepínat, protože každý z jich je třeba jinak vhodný nebo efektivní pro jistá vstupní data.

Perfektním místem, kde by *strategie* nalezla své využití jsou třeba navigační aplikace. Přesněji aplikace na styl *Google Maps*, která umí vyhledat cestu z bodu A do bodu B pomocí nějakého algoritmu, ale zároveň umožňuje uživateli si vybrat z několika způsobů přepravy (tj. auto, MHD, chůze, …). S trochou zamyšlení začnou na povrch vybublávat problémy, které znemožní (resp. znepříjemní) implementaci jediného algoritmu pro všechny varianty. Každý dopravní prostředek totiž může pouze na jisté typy cest, MHD musí brát v potaz jízdní řády a tady by seznam omezujících podmínek rozhodně neskončil.

Řešení

Daný problém by možná šlo vyřešit jedním univerzálním algoritmem, ovšem takové řešení by zcela jistě bylo velmi komplexní a tudíž nebylo moc přívětivé vůči budoucím rozšířením. Elegantnějším řešením bude právě vydefinovat si bokem rodinu algoritmů, které budou vždy zodpovědné za nalezení cesty, ale s ohledem na zvolený dopravní prostředek budou cesty hledat rozdílnými způsoby. Nejlepší vlastnosti tohoto přístupu je, že v aplikaci poté stačí pouze dosazovat správný algoritmus na základě uživatelovi volby.

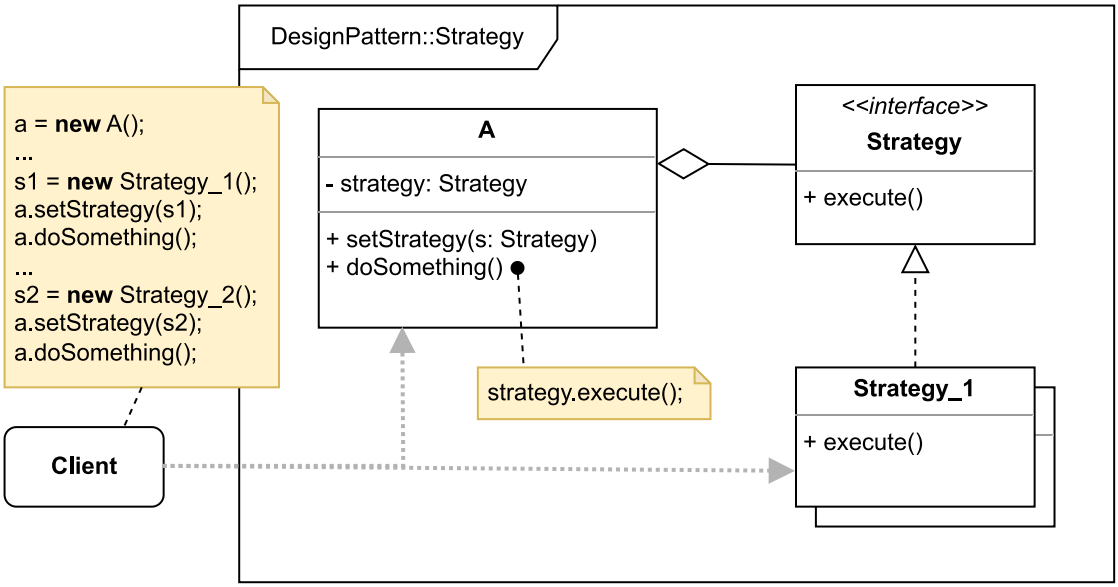
Obecná struktura tohoto vzoru je zobrazena na diagramu 2.23. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

A je objekt, který zaobaluje a spouští daný algoritmus (ekvivalentní komponentě, která zodpovídá za nalezení cesty, z uvedeného příkladu)

Strategy je obecný předpis rozhraní pro všechny algoritmy z jedné rodiny (ekvivalentní rozhraní algoritmů, který hledá cestu z bodu A do bodu B, z uvedeného příkladu)

Strategy_1, ... jsou konkrétní implementace algoritmů z jedné rodiny (ekvivalentní implementacím algoritmů, které hledají cestu autem, MHD a nebo chůzí, z uvedeného příkladu)

Client je externí entita, která spravuje objekt typu A a dosazuje mu v daný moment vhodné strategie (ekvivalentní nějakému GUI, pomocí kterého lze vybírat způsob přepravy, z uvedeného příkladu)



Obrázek 2.23 Třídní diagram obecné *strategie*

Zhodnocení

Výhody:

- dynamická záměna algoritmů za běhu aplikace
- flexibilní přidávání nových algoritmů
- zjednodušení kódu (oproti komplexním univerzálním algoritmům)

Nevýhody:

- většina moderních programovacích jazyků umožňuje docílit stejného výsledku pomocí standardních funkcí, díky jejich podpoře jakožto *„prvořadých občanů"* [24]
- klient musí rozumět rozdílům mezi strategiemi a umět je správně použít

[24] známé pod pojmem *first-class citizen* — entita programovacího jazyka podporující stejné operace jako všechny ostatní entity, tj. předávání pomocí argumentů, ukládání do proměnných, návratové hodnoty funkcí, …

2.3.9 Template Method

Šablonová metoda je *behavioral* návrhový vzor, který definuje kostru skupiny algoritmů, kterou mohou navazující implementace rozšiřovat za předpokladu zachování její obecné struktury.

Motivace

Tento vzor buduje na situacích, kdy se v aplikaci nachází vícero téměř identických algoritmů, které se ale přeci jen mezi sebou liší natolik, aby je nebylo možné spojit do jednoho. Často to jsou vysoce modularizovatelné algoritmy, jejichž průběh lze rozdělit do několika logických kroků. Prostý přístup spočívá v implementaci každé varianty zcela separátně, nicméně takové řešení je poměrně primitivní a zároveň jednoduše zdokonalitelné zapojením šablonové metody.

Zajímavým využitím bývají kupříkladu různé obtížnosti AI oponentů v počítačových hrách. S opominutím variant, které provádí pouze náhodné akce, budou mít veškeré úrovně nadefinovaný nějaký minimální předpis chování, aby herní entity nebyly zcela nečinné či sebepoškozující.

Řešení

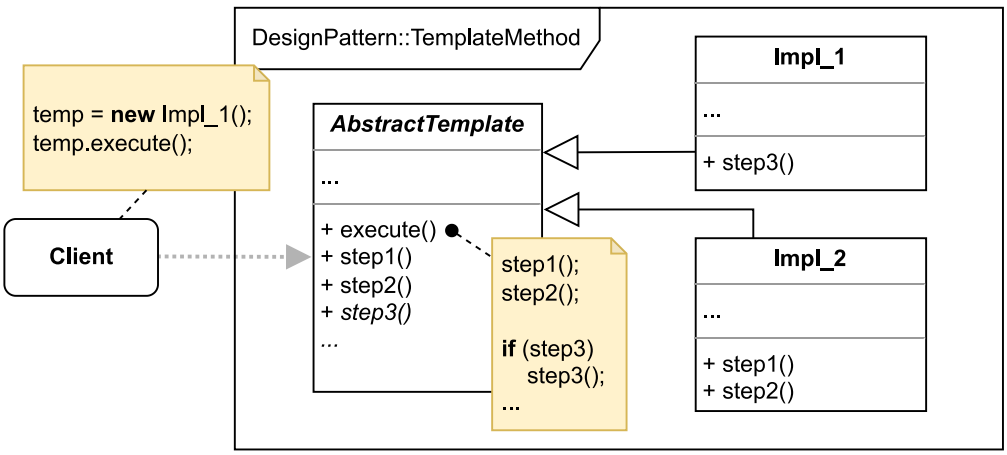
Jak již bylo zmíněno, zcela jistě by šlo uvedený příklad řešit separátní implementací algoritmu pro každou obtížnost. Ovšem hra s desítkami úrovní obtížnosti, s co nejjemnější granularitou, by takto obsahovala ohromné množství duplicitního kódu. Lepším východiskem bude využít *šablonové metody*. Nejprve se nadefinuje kostra chování (tj. neměnné rozdělení algoritmu na posloupnost jednotlivých kroků) společně s implementací esenciálních částí, která bude sloužit jako nejjednodušší obtížnost. Následně se s každou vyšší úrovní přidá nová třída, která základní kostru obohatí dalšími kroky nebo případnou modifikací již stávajících.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.24. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

AbstractTemplate je třída se šablonovou metodou (*execute*), která definuje strukturu algoritmu, a případně implementuje základní kroky (ekvivalentní nejjednodušší úrovni z příkladu)

Impl_1, Impl_2, ... jsou implementace již konkrétních algoritmu, které upravují a rozšiřují jednotlivé kroky (ekvivalentní implementacím jednotlivých obtížnosti z uvedeného příkladu)

Client je externí entita, která vybírá a spouští s konkrétní algoritmus



Obrázek 2.24 Třídní diagram obecné *šablonové metody*

Zhodnocení

Výhody:

- redukuje duplicitní kód
- flexibilita vůči přidávání nových variant
- umožňuje kontrolovat rozsah prováděných změn (šablonová metoda, tj. *execute*, by měla být nepřepisovatelná — tzn. něco jako *final*)

Nevýhody:

- nedostatečné promyšlení struktury algoritmu v nejvýše postavené třídě, může vést k budoucím omezením
- přepsání výchozího kroku může porušit *Liskov substitution principle*[25]

Zajímavosti

Jak bylo řečeno u *tovární metody* (sekce 2.1.3) jedná se v podstatě o její zobecnění. Velmi se podobá *strategii* (sekce 2.3.8) s hlavním rozdílem, že *strategie* se spoléhá na skládání objektů (flexibilnější), zatímco *šablonová metoda* staví na dědičnosti (efektivnější).

[25] princip, který říká: pokud S je podtypem typu T, potom libovolný objekt typu S by měl být schopen nahradit libovolný objekt typu T, a to bez jakýchkoliv změn

2.3.10 Visitor

Návštěvník je *behavioral* návrhový vzor, který předkládá způsob, kterým lze zavést nové chování do sady existujících tříd s minimálními zásahy do stávajícího kódu — jinými slovy řečeno, popisuje postup pro oddělení prováděných algoritmů od objektů, nad kterými jsou vykonávány.

Motivace

Hlavní problém, který řeší tento vzor, je potřeba obohatit sadu existujících objektů o stejné chování. Typicky se jedná o případy, kdy je v aplikaci nějaká komplexnější struktura entit a nově přidávaná funkcionalita se týká pouze některých tříd v dané hierarchii. Případně lze využít také na učesání business logiky, tzn. vyčlenění podpůrných funkčností do samostatných entit bokem, tak aby se hlavní třídy mohly plně soustředit pouze na své nejhlavnější zodpovědnosti.

Častým případem užití bývá přidání exportace dat do stávající aplikace. Třeba takový e-shop, který umožňuje zákazníkovi nahlížet na své předešlé objednávky, a nově vznikl požadavek na rozšíření jeho schopností o export těchto údajů v různých formátech.

Řešení

Nepromyšlené řešení by spočívalo pouze v přidání exportační metody do stávájících tříd (tj. produkt, objednávka, zákazník) a bylo víceméně hotové, to by ovšem vyžadovalo velký zásah do existujícího kódu, nehledě na porušení SRP principu. Správnější přístup pomocí *návštěvníků* bude vypadat následovně. Vytvoří se různí návštěvníci pro export do různých formátů, kteří budou obsahovat metody s konkrétní logikou pro export stávajících tříd. Tyto metody návštěvníků budou následně vyvolávány z konkrétních instancí stávajících tříd s předáním referencí na sebe sama.

Toto řešení má na první pohled velmi komplikovanou souslednost prováděných operací, a to protože obecný návštěvník staví na principu *double dispatch*, což je vlastně trik, který umožňuje zkombinovat tzv. *dynamic binding*[26] společně s jejich přetěžováním [23].

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.25. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

Element je obecné rozhraní pro všechny objekty, kteří musí umět spolupracovat s daným typem návštěvníků (jinými slovy, jejichž chování bude rozšiřováno)

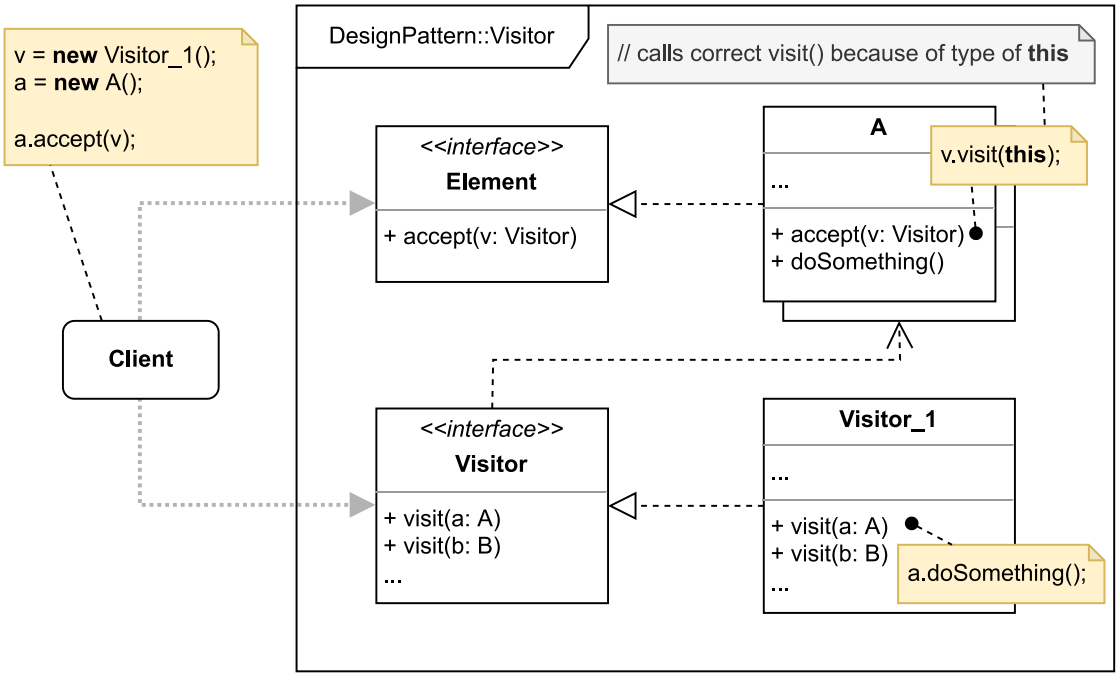
A, ... jsou konkrétní objekty, jejichž funkcionality jsou rozšířeny konkrétními návštěvníky (ekvivalentní produktu/objednávce/zákazníkovi z uvedeného příkladu)

Visitor je obecné rozhraní pro všechny návštěvníky s danou funkčností (definuje rozhraní metod, které poskytují nové chování, pro každý z podporovaných elementů)

Visitor_1 je konkrétní implementace nějakého návštěvníka, který implementuje rozšiřující chování pro dané elementy (ekvivalentní XML/JSON/... exportéru z uvedeného příkladu)

Client je externí entita, která již pracuje s konkrétními elementy a pomocí konkrétních návštěvníků nad nimi provádí nové funkčnosti

[26] dynamické vyhledávání volaných metod za běhu aplikace



Obrázek 2.25 Třídní diagram obecného návštěvníka

Zhodnocení

Výhody:

- jednoduché přidání nových rozšiřujících funkčností do vybraných tříd (které spolu navíc vůbec nemusí souviset, ani sdílet společnou nadtřídu, ani nic podobného)
- vyčlenění nesouvisejícího chování do jedné nové třídy (podpora SRP)
- podporují uchování stavu mezi průchody jednotlivých objektů

Nevýhody:

- zpočátku komplexní na porozumění (kvůli matoucí interakci mezi objekty)
- přidáni nového elementu vyžaduje rozšíření všech návštěvníků (vysoká provázanost)
- „nafouknutí kódu"

2.3.11 Null Object

Nulový objekt je návrhový vzor, který principiálně spadá do kategorie *behavioral* vzorů. Používá se spíše jako refaktorovací[27] technika, která odstraňuje potřebu kontroly nulových ukazatelů, a obecně nahrazuje významově prázdné chování [24].

Motivace

Myšlenka za tímto vzor je poměrně prostá — v rámci vývoje je poměrně vídané při psaní kódu narážet na konstantně se opakující kontroly, zda se má provolat nějaká metoda nebo dokonce zda daná reference vůbec ukazuje na skutečnou instanci třídy. Provádění takových kontrol není striktně řečeno špatně, ale může vést na zbytečně duplicitní kód a méně přívětivé rozhraní.

Užitek může být nádherně vidět třeba na implementacích složitějších datových struktur. Takový generický binární strom bude pravděpodobně umět vrátit počet uložených prvků. Ze struktury binárních stromů vylívá, že nejvhodnější implementace této operace budu pracovat rekurzivně přes jednotlivé uzly. Nicméně ne každý uzel musí mít levého a pravého potomka, a tudíž je potřeba ve všech obdobných operacích ručně kontrolovat, kdy rekurzi zastavit.

Řešení

Elegantnějším řešením může být právě vytvoření *nulového objektu* ve formě nulového uzlu, který bude nahrazovat neexistující potomky a poskytovat významově „prázdné" chování. Přesněji řečeno nulový uzel bude kopírovat stejné rozhraní jako má normální uzel, ale jeho implementace nebude nic dělat, případně bude vracet neutrální hodnoty — tzn. třeba u metody vracející počet prvků ve stromě zastaví rekurzi a bude vracet nulu, protože svou existencí nebude navyšovat celkový počet prvků.

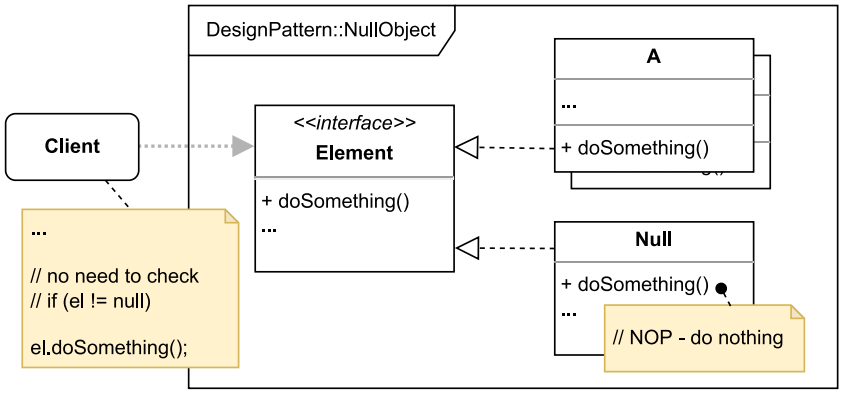
Obecná struktura tohoto vzoru je zobrazena na diagramu 2.26. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

Element je obecný předpis rozhraní pro svázané entity, ke kterým vznikne nulový objekt

A, ... jsou konkrétní implementace entit (ekvivalentní uzlům bin. stromu z uvedeného příkladu)

Null je významově prázdný objekt, jehož implementace operací nic nedělají, případně vrací neutrální hodnoty (ekvivalentní nulovému uzlu z uvedeného příkladu)

Client je externí entita, která pracuje s konkrétními elementy



Obrázek 2.26 Třídní diagram obecného *nulového objektu*

[27] proces transformace struktury stávajícího kódu bez změny jeho chování

Zhodnocení

Výhody:

- zjednodušuje klientský kód
- odstraňuje nutnost kontrol vedoucích na neprovedení akcí, ale zároveň zachovává možnost na ně reagovat

Nevýhody:

- horší výkonnost a paměťová náročnost
- může maskovat jinak viditelné chyby v programu

Zajímavosti

Často bývá implementován jakožto *singleton* (sekce 2.1.5) kvůli snížení paměťových nároků. Z jistého úhlu pohledu se dá považovat za velmi zjednodušený *mock objekt* (sekce 2.2.9). Při jeho využití je potřeba být extrémně na pozoru, jelikož může zamaskovat chyby, které by jinak způsobily pád aplikace a tím byly ihned odhaleny.

2.4 Vzory týkající se souběžného řešení úloh (Concurrent)

Další, již neoficiálně formulovanou kategorií, jsou vzory zabývající se vícevláknovým programováním. Vývoj korektně fungujících plně paralelních aplikací vyžaduje nemalou zkušenost a vzory z této skupiny se snaží pomoci právě s tímto úkonem. Blíže zkoumají a vylepšují základní synchronizační mechanizmy, doposud zmíněné vzory a optimalizační techniky, které snižují režii spojenou se správou vláken.

2.4.1 Active Object

Aktivní objekt (také *actor*) je *concurrent* návrhový vzor (významem spadající i do kategorie *behavioral*), který odděluje vyvolání metod od samotné exekuce, která je následně prováděna zcela asynchronně, čímž zjednodušuje synchronizovaný přístup k danému objektu [25].

Motivace

Hlavním cílem tohoto vzoru je snaha usnadnit souběžnou exekuci metod různých objektů takovým způsobem, že v ideálním případě nebude její provolání externími zdroji vyžadovat žádnou dodatečnou synchronizaci. Celý princip poté stojí na myšlence, že pod pokličkou má daný objekt kontrolu nad svým vlastním vláknem, které využívá k provedení operací. Typickou charakteristikou pro využití aktivního objektu je přítomnost objektů, jejichž funkčnost je zcela nezávislá na běhu hlavního vlákna a tedy může být vykonána plně asynchronně.

Pro příklad lze využít třeba aplikace z bankovního sektoru, která „scrapuje" data o klientech z různých webových zdrojů. Proces „oscrapování" běží od začátku do konce synchronně, protože jednotlivé kroky mohou potřebovat stažené údaje z předchozích kroků. Nicméně kvůli náhodně prováděným auditům je potřeba veškeré získané informace odlévat do bokem ležícího datového skladu, což je kvůli ohromnému množství dat časově náročná operace.

Řešení

Z uvedeného příkladu lze vycítit, že provádět zálohování dat do datového skladu v hlavním vlákně aplikace je nesmysl, protože by to byla zbytečná brzda celého procesu. Zde lze nádherně využít *aktivního objektu*, který se bude starat o postupné odlévání dat ve vlastním vlákně, a tím nebude zatěžovat hlavní logiku. Skvělá věc na řešení pomocí tohoto vzoru je, že objekt zodpovědný za zálohu dat si zachová synchronní rozhraní, takže komunikace z hlavního vlákna bude zcela standardní, ale „pod kapotou" se vykonávání bude dít asynchronně.

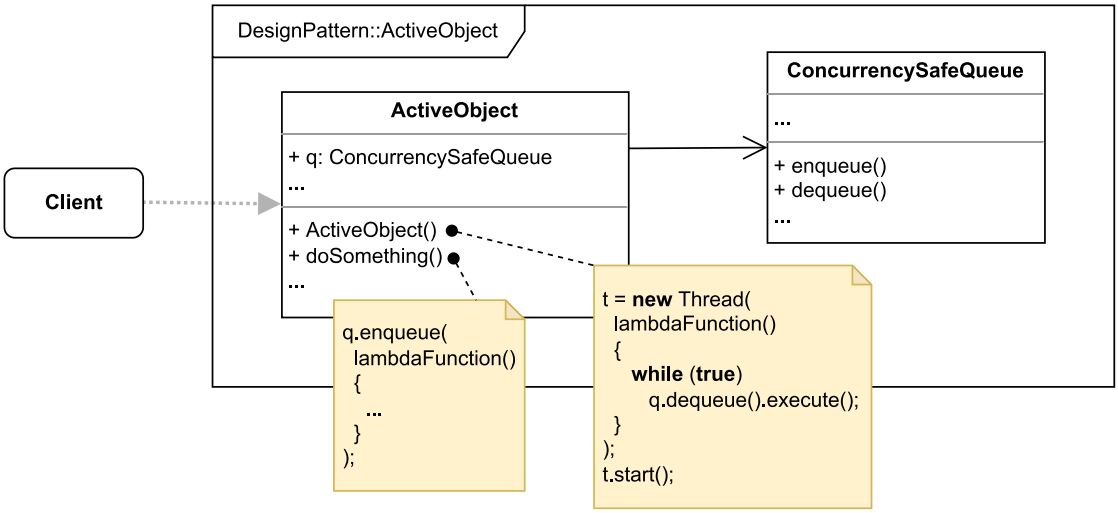
Na pozadí aktivního objektu se nachází šest komponent, které zprostředkovávají veškerou funkcionalitu na nejnižší úrovni implementace. Tato práce se jimi blíže nezabývá, protože v dnešní době programovací jazyky obsahují propracované knihovny, které zaobalují vícero těchto nízko úrovňových komponent do jedné a tedy standardní implementace aktivního objektu vypadá poněkud jednodušeji. V případě potřeby může čtenář nahlédnout do citované práce *Active Object by Douglas C. Schmidt* [25], ve které je blíže popsána implementace s použitím pouze základních konstruktů.

Zjednodušená struktura tohoto vzoru [26] je zobrazena na diagramu 2.27. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

ActiveObject je hlavní objekt, který vykonává funkčnosti asynchronně ve vlastním vlákně, ale navenek je možné ho používat jako zcela normální objekt (ekvivalentní objektu, který odlévá data do datového skladu, z uvedeného příkladu)

ConcurrencySafeQueue je fronta ošetřující souběh[28], do které se vkládají požadavky na exekuci metod, které jsou následně odbavovány v interním vlákně

Client je externí entita, která využívá rozhraní aktivního objektů a vůbec se nemusí starat o jakoukoliv synchronizaci (ekvivalentní hlavnímu vláknu, ze kterého se postupně „scrapují" webová data, z uvedeného příkladu)



Obrázek 2.27 Třídní diagram obecného *aktivního objektu*

Zhodnocení

Výhody:

- zjednodušuje kód (umožňuje programovat jak kdyby s jedno vláknovým přístupem)
- nevyžaduje žádnou dodatečnou synchronizaci
- umožňuje poměrně lehce zanést „paralelizmus" do starých projektů

Nevýhody:

- vyžaduje existenci sofistikovaných mechanizmů (viz. *ConcurrencySafeQueue*), jinak se musí implementovat všechny základní komponenty od nuly (v podstatě taková mini knihovna)
- zhoršuje odhalování chyb

Zajímavosti

Jakýmsi významovým opakem je *monitor objekt*, který naopak pomocí různých synchronizačních mechanizmů (mutex a podmíněná proměnná) vynucuje, aby určitá metoda byla exekuována pouze jediným vláknem v danou chvíli [27]. Zjednodušeně řečeno je to v podstatě vláknově bezpečná třída, která zaobaluje mutex, pomocí kterého řídí přístup k jednotlivým metodám.

[28] *race condition* — nepředvídatelná chyba způsobená nesprávným pořadím/načasováním prováděných operací

2.4.2 Proactor

Proaktor je *concurrent* návrhový vzor (významem spadající i do kategorie *behavioral*), který popisuje jak efektivně strukturovat souběžné aplikace, které zpracovávají více procesů plně asynchronním neblokujícím způsobem [28]. Obvykle je implementován pomocí OS mechanizmů (Windows IOCP, Linux AIO)[29].

Motivace

Tento vzor nalezne své využití zejména v aplikacích, které pracují souběžně s několika často tisíci I/O požadavky každou vteřinu. Pro takové aplikace není zcela vhodné využívat přístupu „co požadavek, to nové vlákno", jelikož I/O operace často čekají na nějaká data a tedy efektivně představují blok. Mimo jiné, vlákno stojí okolo 1MB paměti a nese s sebou režii spojenou s konstantním zaměňováním kontextu[30] mezi čekajícími a připravenými jedinci. Největším indikátorem je, když aplikace musí být vysoce responzivní vůči vysokému počtu klientů, ale může si dovolit obětovat trochu času na samotné zpracování jejich požadavků.

Perfektním příkladem jsou webové servery, u kterých je extrémně důležité, aby zvládaly kontinuálně komunikovat s klienty a přijímat jejich požadavky, tak aby nedocházelo k tomu, že nově příchozí spojení budou muset čekat, než se plně obslouží již připojení jedinci. Jinými slovy je od nich požadováná vysoká responzivita.

Řešení

Uvedený příklad by bylo v menších rozměrech možné řešit synchronním vícevláknovým přístupem. Ovšem to by s sebou neslo vysoké režijní nároky a v momentě, kdy by takový server měl obhospodařovat desetitisíce klientů zároveň, tak by umřel na samotné správě ekvivalentního počtu vláken. Vhodnější řešení bude založeno právě na *proaktorovi*. V první části existuje naslouchač, který v nekonečném cyklu akceptuje nová spojení a vyvolává asynchronní operace pomocí nějakého exekutora (tuto roli typicky zastupuje OS; často se využívá *vláknový fond*, viz. zmínka v sekci 2.1.6-Zajímavosti). V druhé části je potom dispečer, který přebírá výsledky provedených operací a ty předává různým správcům, kteří s nimi následně něco provádí, typicky je vrací klientovi.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.28. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

Proactor je objekt běžící v hlavním vlákně, který vyvolává asynchronní operace pomocí asynchronního exekutora

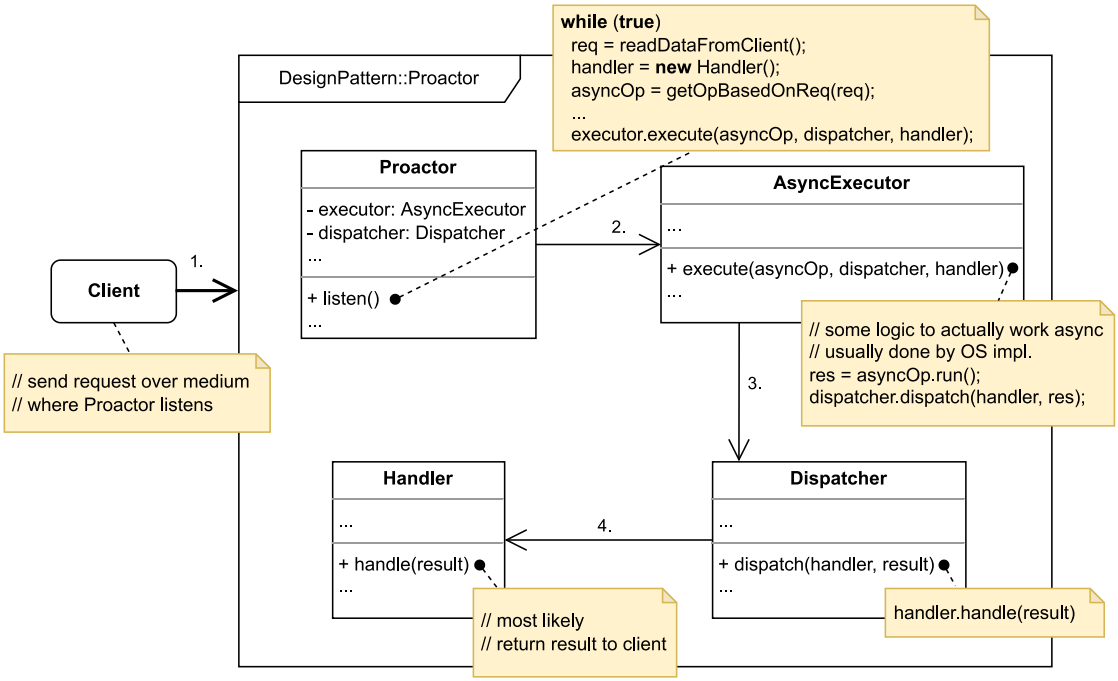
AsyncExecutor je entita asynchronně zpracovávající operace vyvolané proaktorem (typicky zastřešuje OS, který používá pracovní vlákna; avšak klidně může být vlastní výroby)

Dispatcher je objekt, kterému asynchronní exekutor předává výsledky operací a ten je následně přeposílá odpovídajícím správcům

Handler je správce zodpovědný za zpracování výsledků jednotlivých asynchronně prováděných operaci (v podstatě ekvivalentní *callback* funkci)

Client je externí entita, která chce pouze nějakou službu, ale vůbec nemá vhled do toho jak funguje (jde jí spíše o responzivitu)

[29] API operačních systému pro asynchronní zpracování I/O operací
[30] zaměňování kontextu popisuje proces přesouvání vláken z/na CPU, kde jsou exekuována



Obrázek 2.28 Ilustrační diagram entit a toku informací *proaktora*

Zhodnocení

Výhody:

- téměř nulová reakční doba
- odděluje asynchronizační logiku od business logiky (snižuje provázanost)
- lepší výkon oproti čistě vícevláknovému přístupu (v jistých situacích)

Nevýhody:

- nevhodné pro malé aplikace kvůli vysoké komplexnosti
- složité hledání chyb dané časovou a místní separací mezi vyvoláním a dokončením operací

Zajímavosti

Za zmínku stojí návrhový vzor *reaktor*, který se s jistým nadhledem dá považovat za jednoduššího sourozence, jelikož také zpracovává současně přijaté požadavky od několika klientů, ale dosahuje toho plně synchronním přístupem [29].

2.4.3 Master/Slave

Master/Slave (také *map-reduce*) je *concurrent* návrhový vzor, který funguje na stejném principu jako stejnojmenný komunikační model pouze v kontextu souběžného programování — tj. existuje nějaký *master* objekt, který rozděluje práci mezi několik *slave* objektů.

Motivace

Tento vzor stojí na idee, že jisté problémy lze nádherně rozložit na menší nezávislé problémy, pro jejichž vyřešení je možné zužitkovat vícevláknového přístupu, který s sebou ponese znatelné zvýšení výkonnosti. Obecně z *master/slave* designu budou nejvíce těžit aplikace, u kterých dává smysl mít nějakou řídící entitu, která pouze diriguje set paralelně pracujících objektů.

Perfektním problémem pro tento vzor je kupříkladu jeden z paralelních algoritmů pro řešení problému obchodního cestujícího, který hledá nejkratší možnou cestu, kterou lze navštívit set daných míst a navrátit se do startovního bodu.

Řešení

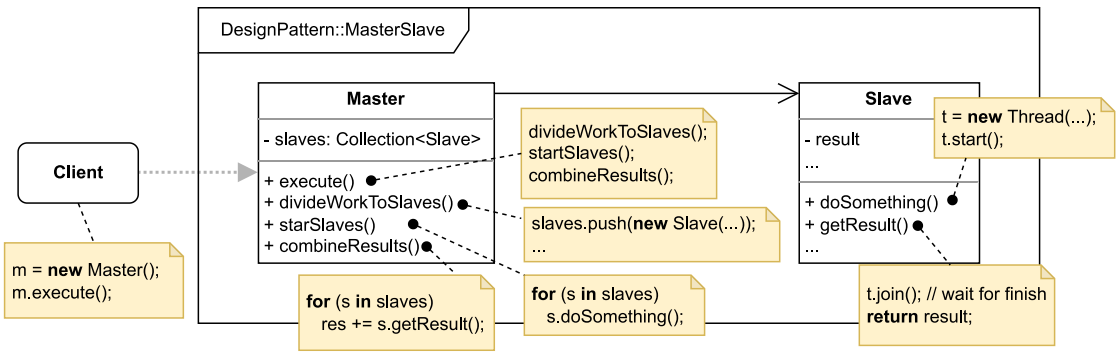
Ač zmíněný problém zní poměrně jednoduše, ve skutečnosti jeho vyřešení není žádnou triviální záležitostí — jedná se totiž o NP-těžký[31] problém. Jeden ze sofistikovanějších algoritmů využívá právě paralelního přístupu pomocí *master/slave*, kde přesně *master* rozděluje problém na menší podproblémy, které následně předává *slave* objektům na vyřešení, a z jejich výsledků nakonec sestavuje celkové řešení (přesné vysvětlení tohoto algoritmu je mimo rozsah této práce, ale čtenář se s ním v případě zájmu může seznámit v práci *TSP in Distributed Environment* [30]).

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.29. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

Master je řídící objekt (v hlavním vláknu), který rozděluje problém na menší podproblémy, které následně deleguje na jednotlivé *slave* objekty (ve svých vlastních vláknech)

Slave je pracovní objekt plně řízený svým *masterem*, který řeší nějaký konkrétní problém

Client je externí entita vyvolávající řešení nějakého problému (komunikuje pouze s *masterem*)



Obrázek 2.29 Třídní diagram obecného *master/slave*

[31] množina problému, které jsou alespoň natolik těžké jako nejtěžší z množiny NP (problémy řešitelné nedeterministickým Turingovým strojem v polynomiálním čase), ale zároveň ani nemusí být řešitelné

Zhodnocení

Výhody:

- rozdělení odpovědností — *master* se stará o koordinaci a *slave* o práci (princip SRP)
- navýšení rychlosti díky paralelismu

Nevýhody:

- použitelný pouze na specifické problémy (takové co lze rozdělit na menší podproblémy)

Zajímavosti

Jak již bylo bryskně zmíněno, s pojmem *master/slave* je typičtější se setkat v kontextu celých architektur, kde bývá velmi často využívána pro rozprostření systémů přes několik navzájem komunikujících zařízení. Také je možné se s nimi setkat, s jakožto jedním možným řešením problému rozložení zátěže mezi vícero identických serverů (tzv. *load balancing*).

2.4.4 Thread-Local Storage

TLS (také *thread-specific storage*) je *concurrent* návrhový vzor, který umožňuje vícero vláknům využívat globálního úložiště svých dat bez nutnosti jakékoliv dodatečné synchronizace [31].

Motivace

Motivace za tímto vzor je velmi přímočará — snaží se odstranit režii spojenou s uzamykáním sdílených úložišť při jejich zpřístupnění různými vlákny, která typicky nejvíce ubližuje výkonnosti vícevláknových aplikací. Bohužel se nejedná o zcela univerzální řešení takových problémů. *TLS* je funkční pouze v případech, kdy má sdílené úložiště jeden logicky globální přístupový bod pro všechna vlákna, ale samotná uložená data jsou specifická pro každé vlákno (při mezi-vláknovém sdílení dat, musí být přístup synchronizován mutexy, aby nedocházelo k souběhu).

Zajímavým příkladem může být třeba implementace nízko režijní knihovny pro aplikační logování. Zaznamenávání sekundárních informací při běhu aplikace málokdy slouží jako hlavní záměr, a tudíž by nemělo nijak výrazně zpomalovat její chod. To by ovšem mohl být problém v momentě, kdy bude v aplikaci více paralelně běžících vláken, které se budou snažit logovat současně — jednoduché řešení prostou synchronizací přes nějaký globální zámek by způsobovalo právě vysoký pokles výkonu.

Řešení

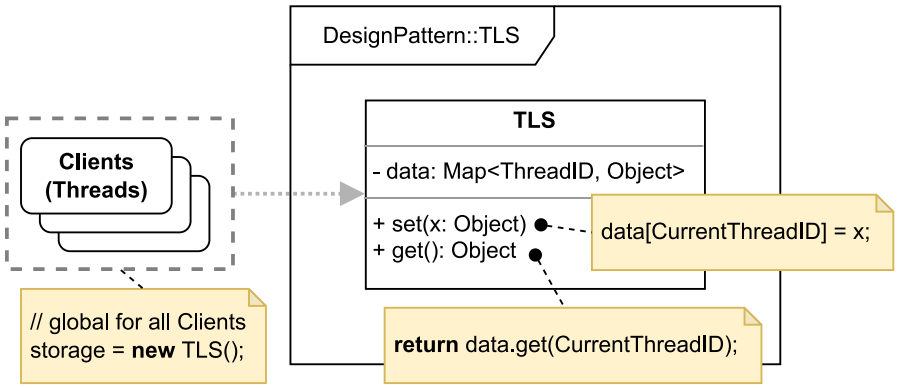
Jeden z možných přístupů by mohl využívat právě *TLS*. Na konkrétním řešení není nic složitého, spočívá v podstatě pouze v tom, že objekty zprostředkovávající logování z různých vláken, nebudou využívat plně sdílenou paměť, ale právě nějaký buffer na bázi *TLS*, díky čemuž budou vždy pracovat se svým vlastním kusem fyzické paměti, nad kterou nebude muset probíhat žádná synchronizace.

Zde je dobré zmínit, že většina programovacích jazyků má opět pro tuto funkčnost předdefinové konstrukty, které stačí v případě potřeby pouze využít — *thread_local keyword* (C++), *java.lang.ThreadLocal* (Java), *threading.local* (Python) a mnoho dalších.

Obecná struktura tohoto vzoru je zobrazena na diagramu 2.30. Jednotlivé subjekty hrají následující role v rámci celého kontextu:

TLS je objekt, který zastřešuje TLS (v podstatě jen ukládá data do mapy, kde klíčem záznamu je unikátní ID vlákna a hodnotou jsou data specifická pouze pro dané vlákno)

Clients jsou jednotlivá vlákna, která pracují s globálně viditelnou TLS, ve které jsou data všech vláken, avšak logicky mají přistup pouze k těm svým



Obrázek 2.30 Ilustrační diagram obecné *TLS*

Zhodnocení

Výhody:

- zvýšená efektivita způsobená absencí synchronizační režie
- po iniciální implementaci velmi jednoduché použití napříč aplikací

Nevýhody:

- zanáší do aplikace globální stav, který je málokdy skutečně nezbytný (vlákna mající pouze svou lokální kopii bývají čistším řešením)

Zajímavosti

V rámci drtivé většiny aplikací je poměrně nepravděpodobné se s tímto vzorem setkat. Většinou bývá součástí spíše letitých aplikací, extrémně specializovaných programů nebo velmi nízkoúrovňových API (Windows implementace *errno*, cache pro paměťové alokátory, …) [32].

Kapitola 3
Návrh prototypu

Tato kapitola se věnuje SW prototypu, jehož kontext je využit k demonstraci možných způsobů využití jednotlivých návrhových vzorů. Nejprve je čtenář seznámen s myšlenkami, které doprovázely volbu dané tématiky, společně s krátkým popisem odvětví trhu, pro které je tento prototyp zamýšlen, a motivací, která za ním stojí. Následuje hrubý odhad požadované funkcionality, od které je následně odvíjen samotný návrh fiktivní architektury. Na závěr jsou stručně popsány moderní technologie, které jsou součástí vysokoúrovňového designu.

3.1 Motivace

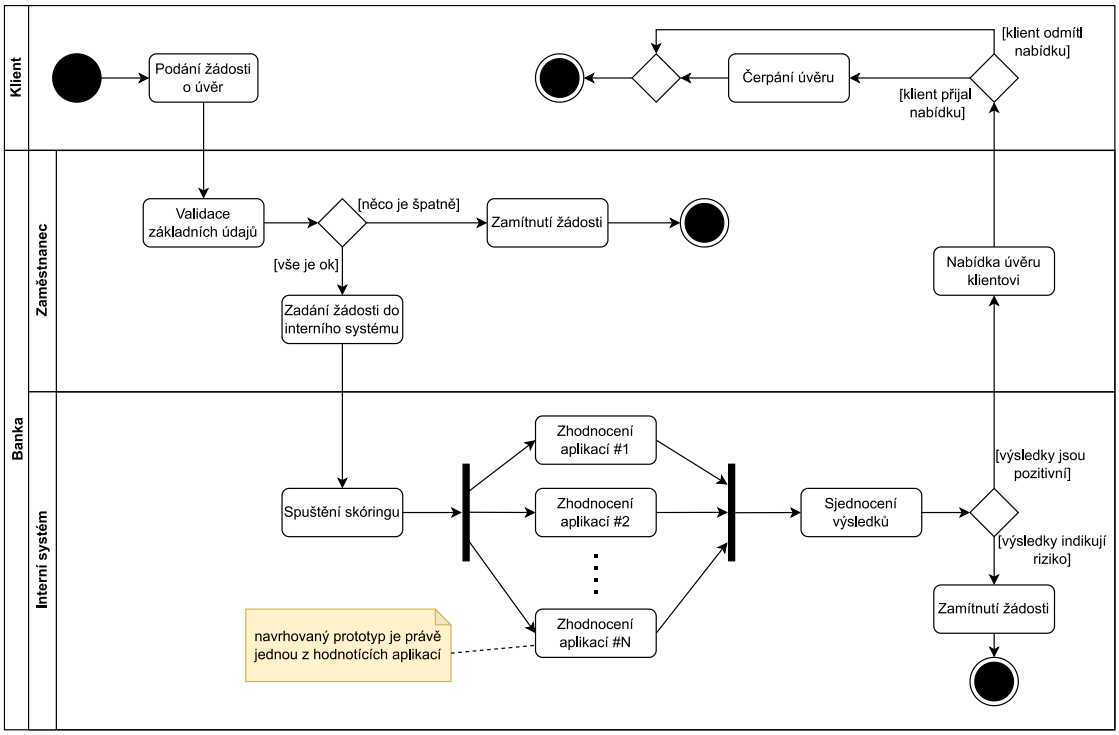
Celá podstata zde navrhovaného prototypu, vyjma vytvoření demonstračních příkladů, které budou reflektovat pro vývojáře relevantní situace, je zaobalit veškeré ukázky návrhových vzorů do jedné škatulky. Myšlenka za tímto přístupem je, že jisté případy budou vyzdvihovat, jak lze občas principiálně stejné problémy řešit různými vzory, za účelem dosažení lehce odlišných výsledků (typicky daných mikroskopickými rozdíly v prerekvizitách a hraničních podmínkách). Případně také nepřímo vyplyne na povrch, jak některé vzory umí spolupracovat s jinými.

Z doposud nabytých informací z předchozí kapitoly, která měla za úkol prozkoumat právě několik individuálních vzorů, je poměrně očividné, že veškeré demonstrace nezvládne pojmout jen tak každá aplikace, kvůli ohromné rozmanitosti problémů, které jsou řešeny. Z tohoto důvodu je potřeba vymyslet takový prototyp, který bude dostatečně robustní, aby zvládl pokrýt všechna zákoutí, ale zároveň nebyl přehnaně komplexní, jelikož má sloužit jako výukový materiál. Zde je nutné podotknout, že aplikací splňujících tato kritéria by šlo vymyslet nespočet, a zde zvolený exemplář, byl vymyšlen s ohledem na praktické zkušenosti autora tohoto textu z daného odvětví (tj. bankovnictví).

Bez zabíhání do zbytečných detailů je všem asi velmi jasné, že banky jako takové jsou nedílnou součástí stávající ekonomie. Víceméně nejhlavnější aktivita jakékoliv banky je poskytování různých úvěrů všelijakým subjektům — obyčejným lidem, malým firmám, obřím korporacím nebo dokonce i státním institucím. S trochou racionálního myšlení je zřejmé, že takové „rozhazování peněz" nemůže provádět jen tak jak se jí zachce. Přesně z tohoto důvodu provází každou žádost o půjčku důkladné prověrky, které se snaží odhalit rizikové klienty (tj. takové klienty u kterých hrozí, že by vypůjčené peníze nesplatili — typicky protože jsou zadlužení u jiných banek, mají problémy s hazardními hrami, jsou nezaměstnaní, apod.).

S rozmachem technologií a neustále rostoucím využitím internetu se již bankám nevyplatí prověřovat každého klienta ručně, což vede ke snaze celý proces schvalování úvěrů co nejvíce automatizovat. V praxi to znamená, že zaměstnanec banky pouze zkontroluje nějaké základní informace a zbytek zajistí sada interních aplikací. Procesu prověřování a hodnocení klienta se říká

„skóring klienta" a jeho průběh typicky obnáší součinnost desítek aplikací, které mají na starosti konkrétní pod-úkoly v rozmezí kdekoliv od navrhování maximálních výší půjček po odhalování potenciálně podvodných aktivit. Zjednodušený proces celého zpracování žádosti může být viděn na diagramu č.3.1.



Obrázek 3.1 Diagram aktivit zachycující hlavní kroky žádosti o úvěr

3.2 Obecný popis

Dotyčný SW prototyp bude imaginární aplikace, která by v reálném prostředí byla součástí již zmíněného „skóringu klienta". Do celkového výsledku tohoto procesu by přispívala zhodnocením klientovy veřejné identity.

Konkrétně tato činnost bude vyvolána s každým „skóringem", tj. s každou novou žádostí nebo změnou již stávajícího úvěru, a její hlavní náplní bude pomocí základních informací o klientovi (jméno, rodné číslo, IČO, …) provést tzv. *webový scraping*[1] různých zdrojů (Google, živnostenský rejstřík, Facebook, …). Získání rozmanitých dat bude pouze jednou z částí. Nasbíraná fakta se následně přetransformují do numerických hodnot, které budou v neposlední řadě vloženy do jistého statistického modelu, pomocí kterého se vypočte výsledné skóre reprezentující klientovu veřejnou identitu. Poslední již „mimo prototypovou" části by bylo zohlednění zkalkulované hodnoty v rámci rozsahu celého „skóringu klienta".

Přesným významem vypočtené hodnoty je zbytečné se zabývat, mohla by indikovat pravděpodobnost, že daný klient zkrachuje (tj. nebude si moci dovolit splátky daného úvěru), nebo kupříkladu pravděpodobnost, že se jedná o klienta, který se angažuje v nelegálních činnostech. Možných využití je zkrátka mnoho a volba konkrétního z nich není pro účely tohoto projektu zapotřebí.

[1] získávání strukturovaných dat z webových stránek

3.3 Požadavky

V této sekci je hrubě odhadnut seznam významných funkčních a nefunkčních požadavků, s ohledem na sektor, ve kterém by prototyp fungoval. Zhodnocení předpokládaných vlastností vyvíjeného systému je důležitým aspektem vývoje, jelikož veškerá další činnost staví právě na této analýze — v našem případě je výčet požadavků zohledňován při volbě vhodné architektury celé aplikace.

Bankovní sektor je známý pro své vysoké nároky na dostupnost a bezpečnost. Nejedná se úplně o života ohrožující ekosystém, ale každý výpadek některé z interních aplikací, obzvláště v úředních hodinách, znamená pro banku potenciální ztrátu klientů, a tedy i finanční újmu v řádech statisíců, ne-li milionů korun. V závislosti na těchto okolnostech a obecném popisu chování z minulé sekce (č. 3.2), lze odvodit následující sadu (tabulka 3.1) nejdůležitějších relevantních požadavků, které by měl navrhovaný prototyp splňovat.

Pro bližší klasifikaci je využit model kombinující dva koncepty — *FURPS*[2] s *MoSCoW*[3]. Ten nádherně zachycuje jednotlivé nároky na systém do matice s několika kategoriemi a různými prioritami [33].

	M	S	C	W
F	*scraping* odlišných webových zdrojů	dočasná *cache* předešlých výsledků	monitorování stavu aplikace	
	transformace dat na numerické hodnoty	záloha všech stažených dat	dynamicky definované transformace	
	vyhodnocení statistickým modelem		dyn. volba aktivních web. zdrojů	
	zpracování více žádostí najednou			
U	API endpoint (pro okolní systémy)	dokumentace pro správce		GUI — grafické rozhraní
	CMD rozhraní (pro správce a logy)			
R	zotavení se z datově závislých chyb		návaznost na přerušené procesy	zotavení se z kritických chyb
	zopakování přerušených procesu			
P	zpracování žádosti do limitu (~10s)	škálovatelnost výkonnosti (instancí)		rozložení zátěže síť. provozu
S	nasazení bez odstávek	častá záměna *scrapovaných* zdrojů	zaměnitelné statistické modely	
		pravidelné bezpečnostní aktualizace knihoven		

Tabulka 3.1 Očekávané funkční a nefunkční požadavky SW prototypu

[2] pohled na kvalitu SW dle 5 kategorií: funkčnost (*Functionality*), užitečnost (*Usability*), spolehlivost (*Reliability*), výkon (*Performance*) a rozšiřitelnost (*Supportability*)

[3] prioritizační technika dle 4 kategorií: musí mít (*Must have*), měl by mít (*Should have*), mohl by mít (*Could have*) a momentálně nebude mít (*Won't have this time*)

3.4 Architektura

Na základě doposud získaných poznatků ohledně ekosystému, kterého má být prototyp součástí, a přibližného odhadu požadavků, lze začít uvažovat nad vhodnou architekturou. Z nároků na vysokou odolnost vůči chybám a škálovatelnost systému, bude volba směřovat vůči jedné konkrétní architektuře, která je mimo jiné u bankovních aplikací velmi populární. S bližším pohledem na její vlastnosti zároveň zjistíme, že nádherně pokryje také některé ze sekundárních vlastností.

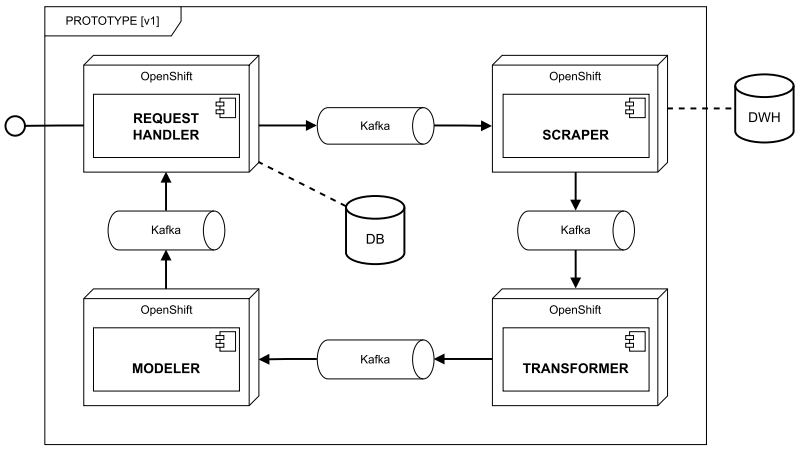
Rýsující se volbou je architektura mikroslužeb, která staví na rozdělení systému do několika nezávisle běžících komponent. Tyto komponenty spolu poté komunikují pomocí odlehčených komunikačních protokolů, pro které je charakteristická co nejmenší možná režie (tj. nejméně přidávaných metadat). Konkrétní design, zobrazený na diagramu 3.2, vychází zejména z funkčních požadavků z předešlé sekce, v závislosti na kterých je prototyp rozdělen na 4 logicky nezávislé komponenty, kde každá vykonává určitou část daného procesu — konkrétní popis jednotlivých komponent následuje v dalších pod-sekcích.

Rozdělení systému do oddělených logických celků (resp. tento styl architektury) s sebou nese několik podstatných výhod:

- kritická chyba jedné z komponent nijak neovlivní komponenty ostatní
- vysoce flexibilní vůči vývoji (komponenty lze vyvíjet nezávisle na sobě)
- jednoduché nasazování verzí komponent (většinou je zcela nezávisle od zbytku aplikace)
- nesrovnatelná škálovatelnost (s využitím vhodných technologií lze velmi jednoduše přidávat/odebírat instance individuálních komponent)
- vnitřní přenosové kanály mohou sloužit jako dočasná záloha (pro návaznost při přerušení)

Ovšem žádný architektonický styl nebude bez svých nedostatků a architektura mikroslužeb není žádnou výjimkou. Mezi největší nevýhody patří:

- zvýšený síťový provoz (potenciální omezení výkonnosti — tzv. *bottleneck*)
- efektivní vývoj se těžce spoléhá na důkladné navržení mezi-komponentních rozhraní



Obrázek 3.2 Diagram vysokoúrovňového pohledu na architekturu SW prototypu

3.4.1 Request Handler

První z dílčích součástí architektury prototypu (diagram 3.2) je komponenta zodpovědná za zpracování příchozích požadavků. Přesněji řečeno, jedná se o část aplikace, která vystavuje veřejně dostupné API směrem do banky a následně řídí komunikaci s okolním světem. Její hlavní činnost se skládá z přijetí požadavku, správy interní databáze, přeposílání klientských informací další komponentě a čekání na dokončení zbývajících kroků celého procesu následované vrácením vypočtené hodnoty systému, který zaslal originální požadavek. Mimo jiné je zodpovědná za dočasné *cachování* výsledných hodnot v interní databázi, díky kterému identické požadavky v určitém časovém intervalu, zbytečně nevytěžují zbylé komponenty.

3.4.2 Scraper

Druhá z dílčích součástí architektury prototypu (diagram 3.2) je komponenta zodpovědná za vytěžování dat o jistém klientovi z různých webů, s využitím jeho základních údajů (jméno, rodné číslo, IČO, …). Přesněji řečeno, jedná se o část aplikace, která řídí několik paralelně běžících *crawlerů*[4], jejichž úkolem je přistupovat k internetu a dotahovat informace z různých webových stránek. Hlavními kroky jsou: vyzvednutí údajů o klientovi poskytnutých předchozí komponentou, následné spuštění *crawlerů* pro různé datové zdroje (Google, Facebook, Živnostenský rejstřík, osobní webové stránky, …) a nakonec sjednocení získaných dat, včetně jejich předáním další komponentě. Mimo jiné zde probíhá zálohování veškerých „mimo-bankovních" dat do externího datového skladu — pro případné zpětné kontroly vyvolané interními audity, při nalezení jakýchkoliv nesrovnalostí. Potřeba zálohovat vše je dána tím, že vytěžované webové stránky se budou zpravidla s postupem času měnit.

3.4.3 Transformer

Třetí z dílčích součástí architektury prototypu (diagram 3.2) je komponenta zodpovědná za přetransformování strukturovaných dat do numerických hodnot. Na této funkcionalitě není nic zákeřného, přesněji řečeno, spočívá pouze ve vyzvednutí vytěžených dat poskytnutých předchozí komponentou a přiřazení číselných hodnot konkrétním záznamům podle setu nějakých pravidel. Výstupem transformace je sada několika proměnných, které jsou následně předány další komponentě, která s jejich pomocí provádí jisté výpočty.

3.4.4 Modeler

Čtvrtá a poslední z dílčích součástí architektury prototypu (diagram 3.2) je komponenta zodpovědná za výpočet výsledného skóre na základě několika desítek až stovek proměnných. Přesněji řečeno, vezme přetransformovaná vytěžená data, která vloží do předem definovaného statistického modelu, pomocí kterého spočítá pravděpodobnost, zda daný klient nebude schopen splácet poskytovaný úvěr či dokonce zkrachuje. Zde opět připomínáme, že význam napočteného skóre je pouze demonstrativní, protože jak bylo řečeno v sekci 3.2, může reprezentovat všelijaké možné charakteristiky a jeho konkrétní podstata je pro účely této práce zcela irelevantní informací. Posledním krokem je předání vypočtené hodnoty zpět *Request Handleru*, který ji po přijetí navrátí dotazujícímu systému.

[4] obecný název pro specializované internetové boty

3.4.5 Úložiště

Krom uvedených komponent jsou součástí prototypu také dvě, resp. tři, datová úložiště, která byla doposud zmíněna pouze velmi povrchově. Prvním je interní databáze, ke které má přistup pouze *Request Handler*. Tato databáze je využívána zejména pro ukládání napočtených výsledků, které jsou zužitkovány pro odlehčení pracovní zátěže u duplicitních požadavků. Druhým je externí datový sklad, ke kterému má přístup pouze *Scraper*, který je využíván pro dlouhodobé zálohování vysokých objemů dat — zde veškerých *scraperem* zpřístupněných webů. V reálu bývají obdobné subjekty zpravovány jiným zcela samostatným oddělením. Posledním úložištěm jsou mezi-komponentní datové kanály realizované *Kafkou* (sekce 3.6). Zde ukládání dat není hlavní pointou, ale je využíváno pro vytvoření dočasného časového okna, ve kterém se při pádu jedné z komponent dá navázat na předešle rozpracovaný proces.

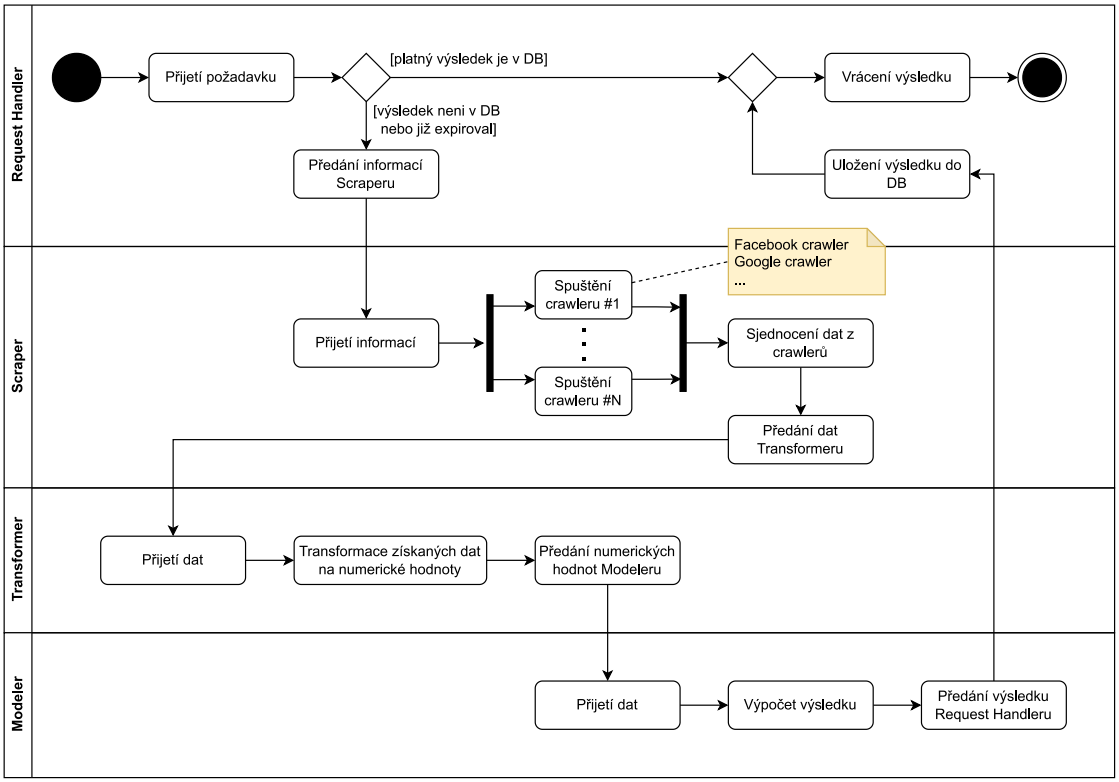
3.5 Tok informací

Z diagramu architektury (č. 3.2) a samotného popisu je asi vidět, že celý proces „oscrapování klienta", tj. vše od iniciálního přijetí požadavku, až po navrácení jistého výsledku, je takovým sériovým pochodem přes jednotlivé komponenty, s jistými paralelními prvky, které urychlují celkovou dobu běhu. Pro lepší ilustraci lze celý průběh sumarizovat v několika následujících hlavních krocích, které jsou také zachyceny na diagramu 3.3.

Proces „oscrapování" klienta

1. *Request Handler* přijme požadavek na zhodnocení veřejného profilu nějakého klienta
2. *Request Handler* prohledá interní DB vůči „nacachovaným" výsledkům
   a. pokud najde platný výsledek, vrátí ho a proces končí
   b. jinak vezme základní informace o klientovi, předá je *Scraper* komponentě, a čeká
3. *Scraper* vyzvedne klientská data, načež s jejich pomocí začne prohledávat web a ukládat veškeré relevantní informace
   a. každou zpřístupněnou webovou stránku průběžně odesílá do datového skladu k záloze
4. *Scraper* po prohledání požadovaných datových zdrojů vše sjednotí a přepošle *Transformer* komponentě
5. *Transformer* převezme vytěžená data a podle sady nějakých pravidel je přetransformuje do číselných proměnných, které následně předá *Modeler* komponentě
6. *Modeler* využije numerických hodnot, které nasype do jistého statistického modelu a vypočte výsledné skóre, které následně vrátí *Request Handleru*
7. *Request Handler* nakonec vezme napočtenou hodnotu, uloží ji do interní DB a vrátí ji systému, který vytvořil původní požadavek

Tento seznam pokrývá souslednost hlavních kroků jednoho provolání prototypu, tj. zhodnocení veřejné identity jediného klienta, které vyvolal jeden okolní systém. Nicméně jak bylo uvedeno mezi nutnými funkčními požadavky (tabulka 3.1), takovýchto požadavků bude umět finální aplikace obhospodařovat několik najednou (v závislosti na velikosti banky, ve které by prototyp běžel, by se mohlo jednat o rozmezí, kdekoliv od nižších jednotek až po vyšší stovky).



Obrázek 3.3 Diagram aktivit zachycující posloupnost hlavních kroků prototypu

3.6 Technologie

V průběhu popisu prototypu a návrhu jeho architektury (diagram 3.2) byly zmíněny následující technologie, které nejsou nutně pevně dány, ale jsou použity jako demonstrace možných variant, vycházející z moderních a v praxi často používaných produktů. V realitě by nám nic nebránilo nahradit je jinými alternativami, s největší pravděpodobností takovými, které by již byly zavedeny v dané společnosti (resp. bance).

OpenShift [34] je platforma od společnosti *RedHat* pro hostování aplikací. Je postavena nad Kubernetes s Linuxovými kontejnery a mezi její hlavní funkčnosti patří správa životních cyklů aplikací, automatizované instalace/nasazení, monitorování a mnohé další. V rámci projektu je zamýšlena pro cloudové hostování nad vlastními prostředky (tzv. „on premise cloud") a správu individuálních komponent.

Kafka [35] je platforma od společnosti *Apache* pro streamování událostí. Jedná se o otevřený software, postavený nad *publish-subscribe* modelem, který využívají stovky společností po celém světě. V rámci projektu je zamýšlena jakožto vysokorychlostní datový kanál pro přenos informací mezi jednotlivými komponentami.

Kapitola 4

Realizace demonstračních příkladů

Poslední kapitola prozkoumává způsob, jakým jsou řešeny demonstrační ukázky jednotlivých návrhových vzorů takovým stylem, aby mohly sloužit jako hodnotný výukový materiál. Nejdůležitější myšlenkou je zde volba vhodné struktury a způsob propojení s kontextem prototypu, který byl popsán v předchozí kapitole (č. 3).

Prvně jsou krátce popsány hlavní nástroje, které byly použity v průběhu vývoje. Následuje volba vhodného postoje vůči uchování zhotovených příkladu a zveřejnění tím vytvořeného katalogu. Dále je popsána myšlenka a struktura celého projektu společně s ilustrací jednoho příkladu konkrétního vzoru. Závěrem jsou shrnuty pozitivní i negativní výsledky, které vyplynuly jak z průběžné práce, tak z představení dokončeného díla skupině programátorů.

4.1 Nástroje

Tato sekce je zaměřena na stručné shrnutí hlavních nástrojů, které byly použity při zpracování praktických ukázek. Tyto nástroje, konkrétně jazyk *C++*, vývojové prostředí *CLion* a služba *GitHub* (resp. *Git*), poskytly možnost efektivní práce a snadné manipulace s daty, což bylo nezbytné pro dosažení stanovených cílů, a tedy i významným krokem k úspěchu.

4.1.1 C++

C++ je poměrně univerzální programovací jazyk, který vznikl téměř před 40 lety jako rozšíření jazyka *C*. Jeho hlavní předností je robustnost funkcí s ohledem na neuvěřitelné rychlosti, což jej činí vhodným pro široké spektrum uplatnění, od herního průmyslu až po celé operační systémy.

Pro demonstraci návrhových vzorů na volbě jazyka zase až tak moc nezáleží díky jejich samotné podstatě (sekce 1.3.2). Jedinou okolností je, že u staticky typovaných jazyků jsou lepé viditelné jejich přínosy než u dynamických typovaných — zejména přínosy týkající se flexibility.

4.1.2 CLion

Jedná se o vývojové prostředí od společnosti *JetBrains* pro programovací jazyky C/C++ [36]. Hlavním důvodem volby tohoto prostředí je zejména jeho multiplatformní podpora a funkčnost chytrého doplňování kódu, která v mnoha případech urychluje proces psaní kódu, pomocí sofistikované analýzy aktuálního kontextu a předešlé historie.

4.1.3 Git

Pro verzování zdrojových kódů byl využit velmi populární nástroj *Git* s repozitářem[1] hostovaným službou *GitHub*. Volba této konkrétní služby je otázkou čistě osobní preference a bez větších problému by mohla být nahrazena libovolnou konkurencí, jako je třeba *GitLab* nebo *BitBucket*.

4.2 Dostupnost

Jednou z úplně prvních věcí, nad kterými je potřeba se zamyslet, je výběr vhodného prostředku pro zpřístupnění vytvořených výsledků. Pozornému čtenáři bude odpověď na tuto otázku asi již jasná, nicméně i tak bude dobré ji zde zdůvodnit a blíže popsat.

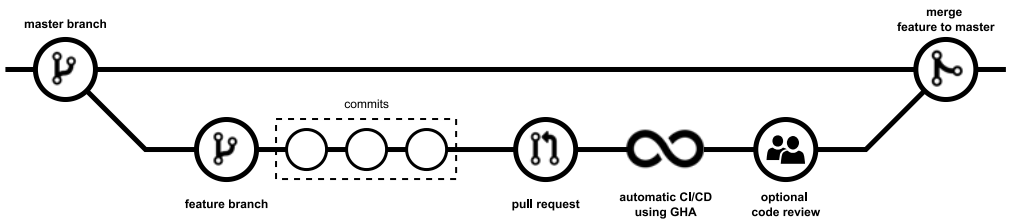
Hlavní charakteristika, kterou je potřeba zohlednit, je jak bylo již mnohokrát zmíněno, že výsledný produkt by měl být vhodnou alternativou vůči jiným výukovým materiálům — tzn. měl by být snadno dostupný, přehledný, podporovat jistou formu zpětné vazby (požadavky na rozšíření, opravy, …) a v ideálním případě i kontrolovaný *crowdsourcing*[2]. První z nápadů spočíval ve vytvoření vlastního zcela na míru ušitého webu. Nicméně tento přístup byl zavrhnut v poměrně raných fázích, jelikož zakomponování podpory všech zmíněných vlastností by zabralo nemalé úsilí, a navíc v rámci průzkumu trhu bylo odhaleno, že jedna velmi populární platforma již podporuje vše potřebné — tou je, jak už bylo nepřímo prozrazeno, *GitHub*.

Všechny praktické ukázky jsou tedy uloženy a zpřístupněny skrze veřejný repozitář na *GitHubu* (`https://github.com/safar-jn/design-patterns`). Vyjma pokrytí všech základních požadavků je velikou výhodou to, že naprostá většina vývojářů (tj. hlavní cílené skupiny) se již někdy s *GitHubem* (nebo jinou distribucí) setkala, a tudíž se jedná o pro mnohé familiární prostředí. V neposlední řadě *GitHub* umožňuje nastavení několika velmi šikovných funkcionalit, které jsou dále stručně popsány.

První část nastavení spočívá v sadě základních ochranných pravidel, které jsou nastaveny vůči hlavní větvi, jelikož se jedná o veřejně přístupný projekt, u kterého je potřeba zamezit, aby libovolný uživatel mohl provádět libovolné změny. Tyto pravidla obnáší zejména zamezení přímých změn obsahu a seběhnutí různých testů v rámci *pull requestů*[3].

Mimo jiné byly při zpracování využity tzv. *conventional commits*, což je zjednodušeně řečeno konvence, která říká jakým způsobem strukturovat zprávy jednotlivých *commitů*[4], aby nad nimi bylo možné provádět strojově automatizované operace (např. sledování problémů/úkolů).

V neposlední řadě stojí za zmínku využití *GitHub Actions* [37]. Jedná se o platformu, která pomocí *GitHubem* hostovaných serverů (i.e. *runnerů*) provádí nadefinované úkony ve virtuálních prostředích. Nejtypičtěji se tato platforma používá pro automatizované CI/CD procesy (viz. součást *Git* workflow na diagramu 4.1). V projektu jsou využity v rámci *pull requestů* pro automatické zkompilování a kontroly úniků paměti pomocí nástroje *Valgrind*. Jinými slovy kontrola, že jakákoliv nová změna nerozbila stávající části kódu.



Obrázek 4.1 Diagram zachycující standardní *Git* workflow (převzato z *GitHub* dokumentace [37])

[1] datové úložiště systému správy verzí
[2] označení dělby práce, na které se podílí veřejnost (tedy i jiní vývojáři budou moci přispívat)
[3] mechanizmus *GitHubu* pro notifikaci ohledně dokončení nějaké logické funkčnosti
[4] *snapshot* repozitáře v daný moment — tj. stav všech spravovaných souborů

Následující úryvek YAML[5] souboru (č. 4.1) zachycuje právě zmíněné nastavení automatických kontrol, které probíhají s každým *pull requestem* vytvořeným vůči hlavní větvi. Zkráceně řečeno, dynamicky poskládá seznam dostupných ukázek (*collect-design-patterns*), pomocí kterého následně spustí souběžnou kompilaci a kontrolu (*compile-all*), a nakonec počká na úspěšné ukončení všech spuštěných procesu (*compile-all-success*). Definice kompilačního (resp. kontrolního) kroku je separátní *GitHub Action* (viz. *compile-all* položka *uses*), která je nastavena obdobným způsobem, akorát je poněkud jednodušší — pouze přes vstupní parametr dostane název vzoru, pomocí kterého si vleze do odpovídající složky, kde zkompiluje veškeré soubory a prožene výstup skrze *Valgrind*.

```
name: Pull Request checks

on:
 pull_request:
  branches: master

jobs:
 collect-design-patterns:
  runs-on: ubuntu-latest

  outputs:
   design-patterns: ${{ steps.set-list.outputs.list }}

  steps:
   - name: Checknout most recent commit
     uses: actions/checkout@v3

   - name: Create list with design pattern folders
     id: set-list
     run: echo "list=[$(ls -m -Q -d */ | tr '\n' ' ')]" >> $GITHUB_OUTPUT

 compile-all:
  needs: collect-design-patterns

  strategy:
   matrix:
    design-patterns: |
      ${{ fromJSON(needs.collect-design-patterns.outputs.design-patterns) }}

  uses: safar-jn/design-patterns/.github/workflows/compile.yml@master
  with:
   path: ${{ matrix.design-patterns }}

 compile-all-success:
  needs: compile-all

  runs-on: ubuntu-latest

  steps:
   - name: Wait for compile-all success
     run: echo "completed"
```

Výpis kódu 4.1 Ukázka části nastavení *GitHub Actions* pro automatické kontroly při *pull requestech*

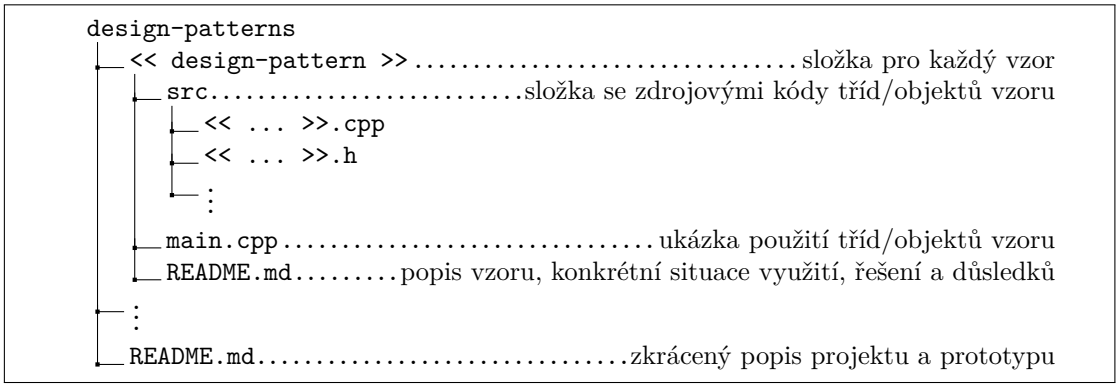
[5] formát pro serializaci strukturovaných dat

4.3 Struktura

Objektivně nejpodstatnější částí celé poslední kapitoly a praktické části této práce (z pohledu hlavního textu) je opodstatnění zvolené struktury pro zpracování jednotlivých demonstračních příkladů návrhových vzorů. V zájmu skutečného porozumění je potřeba přiblížit ideu, na které je vše založeno, a která vychází z doposud nasbíraných vědomostí — tj. znalost obecných problémů, které inspirovaly vznik návrhových vzorů jako takových (sekce 1.2), znalost konkrétních vzorů (kapitola 2) a znalost připraveného kontextu ve formě fiktivní aplikace (kapitola 3).

Při vymýšlení, jakým způsobem vyhotovit jednotlivé příklady a zachytit jejich esenci, bylo bráno v potaz dvou hlavních intencí. První z nich je, že se snažíme vytvořit materiál, který když před sebe dostane někdo, kdo nikdy neslyšel o návrhových vzorech, tak i přes jeho naprostou neznalost bude schopen uchopit, o co se jedná a jak to vlastně funguje. Druhá byla již krátce zmíněna u motivace za navrhovaným prototypem (sekce 3.1), konkrétně že veškeré ukázky by měly odrážet nějakou specifickou funkčnost namodelované aplikace, s tím že přidanou hodnotou tohoto přístupu bude částečné vyzdvižení jak lze principiálně identické problémy řešit různými vzory, případně jak si některé vzory „jdou ruku v ruce". S těmito cíli na mysli by mělo být ihned jasnější, proč kupříkladu samotná implementace není zpracována způsobem naprogramování celého prototypu od A do Z s pouhým zakomponováním jistých vzorů — řešení jednoho a toho samého problému různými způsoby by s touto variantou bylo nepředstavitelně komplexní, ne-li nemožné (v nejlepší případě by takový přístup vyžadoval desítky diametrálně rozdílných verzí celé aplikace). Jak se říká „v jednoduchosti je krása" a přesně proto byl také zvolen následující přístup a struktura (obrázek 4.2).

Každý nasimulovaný vzor stojí zcela nezávisle na ostatních a vždy řeší jeden specifický problém či funkčnost, kterou by za daných podmínek měl popsaný prototyp umět. V kořenovém adresáři celého repozitáře se nachází obecný popis projektu a prototypu v *markdown*[6] formátu společně se složkami pro každý z demonstrovaných vzorů. Struktura veškerých ukázek je následně vždy stejná. Skládá se z krátkého obecného popisu vzoru, popisu fiktivního problému nebo situace, kdy by mohl být daný vzor využit, odpovídajícího řešení a spustitelných úryvků kódu, které zmíněné řešení demonstrují. Implementace řešení vždy pouze simulují nějaká chování (nejčastěji pouze výpisem do konzole a uměle vytvořením uspáváním vláken), jelikož jak bylo řečeno celá pointa ukázek není naprogramovat zbytečně komplexní funkcionality složité na pochopení, které by následně odváděly pozornost od samotného významu a hlavních principů návrhových vzorů.



Obrázek 4.2 Struktura GitHub repozitáře s praktickými ukázkami

Tento text již dále nezachází do velkých detailů jednotlivých ukázek, jelikož veškeré potřebné informace jsou součástí jejich zpracování. Jedinou výjimkou je následující sekce, která popisuje jeden konkrétní příklad, a stručné shrnutí všech smyšlených případů užití (tabulka 4.1).

[6] značkovací jazyk sloužící pro úpravu prostého textu

VZOR	KOMPONENTA	SITUACE
Vzory týkající se tvorby objektů (Creational)		
Abstract Factory	Modeler	*binning* (obecnější zaokrouhlování) proměnných používaných ve statistických modelech
Builder	Scraper	sestavování profilu klienta (datová třída)
Factory Method	-	vytváření *logger* objektů ve vlastní logovací mini-knihovně
Lazy Load	Scraper	připojení k externímu DWH (s jistým omezením)
Object Pool	Scraper	připojení k externímu DWH (s jistým omezením)
Prototype	Modeler	proměnné používané ve statistických modelech
Singleton	Request Handler	připojení k databázi (s jistým omezením)
Vzory týkající se struktury programu (Structural)		
Adapter	Scraper	rychlé nahrazení kompromitované knihovny
Bridge	Request Handler	připojení k databázi (s jistým omezením)
Composite	Transformer	vlastní struktura reprezentující profil klienta
Decorator	Scraper	dynamické rozšiřování webových *crawlerů*
Facade	-	rozhraní vlastní logovací mini-knihovny
Flyweight	Scraper	vyseparování nastavení webových *crawlerů*
Mock Object	-	testování vysoce provázaných tříd
Proxy	Scraper	modifikace webových *crawlerů* poskytovaných knihovnou třetí strany
Vzory týkající se chování (Behavioral)		
CoR	Request Handler	zpracování příchozích požadavků
Command	Scraper	pokyny ke spuštění různých webových *crawlerů*
Iterator	Transformer	procházení vlastní struktury (profil klienta)
Mediator	Scraper	řízení komunikace mezi webovými *crawlery*
Memento	Transformer	ulehčené zvracení změn ve vlastní dat. struktuře
Null Object	Modeler	logicky nulové proměnné používané při výpočtech
Observer	Scraper	automatické spouštění webových *crawlerů* po přijetí dat přes Kafku
State	Request Handler	stavy žádostí
Strategy	Modeler	různé výpočetní algoritmy
Template Method	Modeler	různé výpočetní algoritmy (sdílející jisté kroky)
Visitor	-	export vlastní dat. struktury v různých formátech
Vzory týkající se souběžného řešení úloh (Concurrent)		
Active Object	Scraper	souběžně běžící webové *crawlery*
Master/Slave	Scraper	souběžně běžící webové *crawlery* (s dodatečnou řídící logikou)
TLS	-	ukládání zpráv ve vlastní logovací knihovně

Tabulka 4.1 Přehled zrealizovaných ukázek návrhových vzorů

4.4 Příklad

Pro lepší představu se podíváme na jeden konkrétní příklad — *singleton* (sekce 2.1.5). První velmi důležitý soubor je *README.md* (webové rozhraní *GitHubu* ho pěkně zobrazí při otevření složky — obrázek 4.3), který obsahuje shrnutí vzoru a důležitěji popis konkrétního problému/řešení v kontextu našeho prototypu. Pročtení těchto sekcí je naprosto stěžejní, jelikož jejich součástí jsou různé prerekvizity a předpoklady, které vyzdvihují, proč zrovna daný vzor je vhodným řešením — v tomto případě se jedná o funkcionalitu připojení k databázi, která je ale limitovaná HW výkonem náležitých DB serverů. Na závěr jsou uvedeny významné výhody či nedostatky.

README.md

SINGLETON

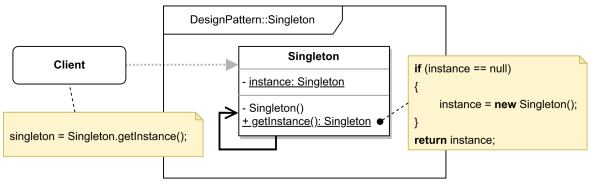
GENERAL

Creational GoF design pattern that lets you ensure only one instance of given class will exist at any point in time during application's runtime.

USAGE

Main use case is pretty self-explanatory - simply put there's a need for having only one globally accessible instance of some class. Although usage of this pattern should be carefully wighted as it introduces often times unnecessary global state which is by its nature in conflict with more than few principles of good OO code.

STRUCTURE



TL;DR behaviour - basically the class that's supposed to be a **Singleton** (i.e. have only one instance) firstly privatizes its constructor and all alternative construction mechanisms, then defines private static instance variable and lastly exposes public static method used for creating/accessing the instance. The **getInstance()** method is where most of the magic happens because it acts as a constructor that's also responsible for the "uniqueness control".

NOTE: depending on your needs and programming language you use, you have to be careful when implementing this pattern to really ensure there will always be only one instance - most common problem is forgetting multithreading (but also serialization/deserialization, reflexion, ...)

EXAMPLE

Let's imagine following situation in the context of aforementioned prototype. The **request handler** saves processed requests in database for caching purposes (calculated score is valid for certain amount of time). This DB is provided by another dept. responsible for all DBs hosted on company's internal server and because they have limited computing power, they restrict the number of active connections to only one per application.

SOLUTION

One way to meet the set criteria is to use a **Singleton**. Basically, we just have to ensure that at any given time the **request handler** will have only one active connection to the DB, which is pretty much exactly what the **Singleton** pattern does.

Dummy implementation of this example/solution and how to use it is part of this directory.

SUMMARY

Best part about this approach is that it provides one global access point that can be reused throughout the whole app without any additional need for uniqueness checks. Everything is handled internally by the object.

Worst thing is that **Singleton** violates several OO principles, complicates unit testing and might mask otherwise easily visible shortcomings in application's design.

Obrázek 4.3 Ukázka zobrazeného *README.md* konkrétního příkladu

Samotný popisek (obrázek 4.3) poté doprovází zdrojové kódy simulující danou funkčnost. Soubory s kódem se dělí na dvě části — soubor *main.cpp* a soubory ve složce *src*. Soubor *main.cpp* (výpis kódu 4.2) je jakýmsi vstupním bodem, který ukazuje jakým způsobem se využívají dílčí třídy/objekty, které odráží chování demonstrovaného vzoru — v tomto případě simuluje vytvoření databázového připojení ze dvou různých vláken. Pro čtenáře, kteří si důkladně prošly kapitolu s rozbory návrhových vzorů (č. 2), se dá říci, že obsah tohoto souboru je ekvivalentní *client* entitám, které jsou součástí drtivé většiny diagramů. Soubory uložené ve složce *src* jsou poté právě jednotlivými třídami, které zaobalují simulované chování popsané danou situací — zde se jedná o třídu zaobalující fiktivní připojení k nějaké databázi (výpisy kódu 4.3 a 4.4). Ve výpisech kódu je zde vidět, jakým zhruba stylem probíhá simulace funkčností ve všech příkladech — kód se nikam nepřipojuje, pouze si ukládá nějaká data a v metodě *execute*, která by měla spouštět nějaký dotaz, probíhá pouze výpis do konzole.

```
void t1Func()
{
  // simulate getting DB connection
  // (URIs are different only to see that there'll be only one instance)

  std::shared_ptr<DatabaseConnection> conn =
    DatabaseConnection::getInstance("localhost:8080", "admin", "1234");

  conn->execute("SELECT * FROM users");
}

void t2Func()
{
  // simulate getting DB connection
  // (URIs are different only to see that there'll be only one instance)

  std::shared_ptr<DatabaseConnection> conn =
    DatabaseConnection::getInstance("remotehost:8080", "admin", "1234");

  conn->execute("SELECT * FROM products");
}

int main(int argc, char **argv)
{
  // simulate creating DB connection from 2 different threads
  //  - should always be same 'cause of conn being Singleton

  std::cout << "[main] | both URIs should always be same" << std::endl;

  std::thread t1(t1Func);
  std::thread t2(t2Func);

  t1.join();
  t2.join();

  return 0;
}
```

Výpis kódu 4.2 Úryvek kódu ze souboru *main.cpp* (demonstrace použití Singleton vzoru)

```
/// Singleton - restricts existence of only one connection object at a time
class DatabaseConnection
{
  public:
    static std::shared_ptr<DatabaseConnection>
      getInstance (std::string uri, std::string usr, std::string pwd);

    bool execute (const std::string &query) const;
  private:
    // privatize constructor
    DatabaseConnection (std::string uri, std::string usr, std::string pwd);

    // forbid cloning
    DatabaseConnection (DatabaseConnection &other) = delete;

    // forbid assigning
    void operator = (const DatabaseConnection &) = delete;

    // only existing instance
    static std::shared_ptr<DatabaseConnection> _instance;
    static std::mutex _mutex;

    std::string _uri;
    std::string _usr;
    std::string _pwd;
};
```

Výpis kódu 4.3 Úryvek kódu ze souboru *DatabaseConnection.h* (demonstrace Singleton vzoru)

```
std::shared_ptr<DatabaseConnection> DatabaseConnection::getInstance
  (std::string uri, std::string usr, std::string pwd)
{
  // create new instance only if it doesn't exist yet,
  // otherwise just return the existing one

  std::scoped_lock<std::mutex> lock(DatabaseConnection::_mutex);

  if (!_instance)
    _instance.reset(
      new DatabaseConnection(std::move(uri), std::move(usr), std::move(pwd))
    );

  return _instance;
}

bool DatabaseConnection::execute (const std::string &query) const
{
  // simulate executing database query

  std::cout << " |- [DatabaseConnection] | " << _uri
       << " | executing '" << query << "'" << std::endl;

  return true;
}
```

Výpis kódu 4.4 Úryvek kódu ze souboru *DatabaseConnection.cpp* (demonstrace Singleton vzoru)

Za poslední zmínku stojí zpětné dodělání „univerzálního" *makefile*[7] souboru (výpis kódu 4.5), který má potenciálním recipientům usnadnit kompilaci a spuštění. Tento soubor je díky jeho obecnější struktuře ve všech příkladech stejný a podporuje standardní příkazy: *make all* (příkaz pro zkompilování celého příkladu), *make run* (příkaz pro spuštění výstupu kompilace) a *make clean* (příkaz pro promazaní dočasných a zkompilovaných souborů).

```
CXX := g++
CXXFLAGS := -std=c++17 -Wall -pedantic -g
LIBS := -pthread

TARGET := main
OBJDIR := .obj

SRC := $(shell find . -name "*.cpp" | cut -c3-)
OBJ := $(addprefix $(OBJDIR)/, $(patsubst %.cpp, %.o, $(SRC)))

all: $(OBJ)
  $(CXX) $(CXXFLAGS) $(LIBS) $^ -o $(TARGET)

$(OBJ): $(OBJDIR)/%.o: %.cpp
  mkdir -p $(@D)
  $(CXX) $(CXXFLAGS) -c $< -o $@

run:
  ./$(TARGET)

clean:
  rm -rf $(OBJDIR)
  rm -f $(TARGET)
```

Výpis kódu 4.5 Kód obecnějšího makefile

4.5 Shrnutí

Posledním zdrojem informací je souhrn postupně sesbíraných poznatků, dosažených úspěchů, známých nedostatků a obecně všelijakých detailů stojících za vyzdvižení. Prvně je krátce zhodnoceno celé „praktičtější" snažení, tj. demonstrace návrhových vzorů na navrženém prototypu. Následně je zohledněna zpětná vazba poskytnutá skupinou vývojářů, kterým byl projekt představen. A na úplný závěr je předloženo subjektivní zhodnocení jednotlivých vzorů.

Celý projekt v momentě dokončení této práce obsahuje téměř třicet návrhových vzorů ze čtyř různých kategorií. Vesměs se dá říci, že nádherně splnil očekávání (sekce 3.1), která ho měla odlišit od jiných obdobných prací, jak může být vidět z přehledu všech zrealizovaných příkladů (tabulka 4.1) — kupříkladu dvojice *composite/iterator* přesně demonstruje, jak některé vzory skvěle navzájem spolupracují, zatímco třeba dvojice *object pool/lazy load* ukazuje, jak některé principiálně ekvivalentní problémy lze řešit různými vzory v závislosti na jistých okolnostech konkrétní situace. Takovýchto interakcí lze mezi příklady nalézt několik, což zároveň navazuje na jeden z největších nedostatků, kterým je že tato vlastnost vzájemných vztahů není nikde explicitně zachycena a tedy je pouze na čtenářovi, aby si o ní udělal vlastní názor.

Součástí zakončení této práce bylo mimo jiné také představení praktické části skupině programátorů. Daná skupina čítala jedince v rozmezí od naprostých nováčků až po seniory se značným množstvím zkušeností. Zde je nutné podotknout, že zpětná vazba poskytnutá tímto kolektivem nebude perfektní, jelikož se jedná o poměrně specifický vzorek — tím že všichni dotyční pracují

[7] nástroj, který umožňuje blíže specifikovat, jakým způsobem mají být zkompilovány a provázány jisté soubory

ve stejném sektoru, kterého se týká navržený prototyp (tj. bankovnictví). Bohužel tedy nevyřeší jednu z obav, kterou je že ačkoliv příklady mají sloužit jako univerzální studijní materiál, tak prototyp je poměrně specializovaným produktem a porozumění jeho podstatě by nemuselo být pro každého jednoduchým úkolem. Nicméně co se týče samotné zpětné vazby na konkrétní příklady, tak celkově se shledaly s poměrným úspěchem — nejčastěji byla oceněna jednoduchost ukázek a četnost představených vzorů. Každopádně máloco bude na první pokus perfektní a ani tento projekt se neobešel bez svého podílu konstruktivní kritiky. Mezi hlavní nápady na potenciální vylepšení, které byly vzneseny, spadá: implementace příkladů ve vícero programovacích jazycích (díky jejich jednoduchosti), doplnění *readme* souborů o třídní diagramy specifické dané situaci/problému a doplnění portfólia o některé významné vzory, které byly opomenuty (např. z právě vynechané kategorie architektonických vzorů).

Zcela posledním námětem je shrnutí subjektivních pozorování, která vychází z práce s jednotlivými návrhovými vzory. Přesněji řečeno, jedná se o srovnání zpracovaných vzorů na základě dvou zajímavějších metrik — míra použitelnosti a náročnost na pochopení/implementaci. Zde je opět nutné podotknout, že čistě objektivní zhodnocení není úplně možné, vzhledem k tomu, že praktická část práce vyžadovala vymyšlení vlastních situací, aby bylo vůbec možné daný vzor nademonstrovat. Tudíž pohled na použitelnost v reálném světě může být mírně zkreslený, nicméně tuto vlastnost lze odhadnout pomocí subjektivního pohledu na náročnost vymyšlení konkrétních případů užití (tj. snazší vymyšlení situace implikuje vyšší míru použitelnosti). Stejně tak náročnost na implementaci může být v některých případech ovlivněna volbou programovacího jazyka. S těmito polehčujícími okolnostmi se ale musíme smířit a bez další obezličeek následují jednotlivá srovnání.

POUŽITELNOST[8]

1: Prototype, Bridge, Flyweight, Mediator, Visitor, TLS

2: Factory Method, Composite, Facade, Proxy, Command, Observer, Template Method

3: Abstract Factory, Builder, Lazy Load, Object Pool, Singleton, Adapter, Decorator, Mock Object, CoR, Iterator, Memento, Null Object, State, Strategy, Active Object, Master/Slave

x: Twin (spíše nízká použitelnost), Proactor (spíše střední použitelnost)

NÁROČNOST[8]

1: Builder, Lazy Load, Object Pool, Singleton, Adapter, Decorator, Facade, Mock Object, Proxy, CoR, Iterator, Memento, Null Object, State, Strategy, Template Method, TLS

2: Abstract Factory (zejména principem), Factory Method, Prototype, Composite, Command (zejména implementací), Mediator (zejména implementací), Observer, Active Object, Master/Slave

3: Bridge, Flyweight (zejména implementací), Visitor

x: Twin (principem lehký), Proactor (principem těžký)

[8] 1 = nízká; 2 = střední; 3 = vysoká, x = nehodnoceno/neimplementováno

Závěr

Tato práce měla za cíl prozkoumat problematiku OO návrhových vzorů a důkladně rozebrat několik konkrétních exemplářů. Následně měl být navržen vhodný prototyp SW a na něm ukázáno, jak by bylo možné jednotlivé vzory využít v rámci vývoje. Pointou tohoto snažení mělo být vytvoření jakéhosi interaktivního katalogu vzorů, který bude demonstrovat individuální případy nad jedním a tím samým kontextem. Zmíněný katalog by následně měl být veřejně dostupný, aby mohl sloužit jako jedna z alternativ pro rozvoj vědomostí mladých vývojářů.

V kapitole *rešerše* (č. 1) jsou uvedeny základní informace a předpoklady nutné k porozumění, jak tomuto textu, tak i abstraktnějším důvodům pro existenci samotných návrhových vzorů. Prvně jsou zde popsány základny pojmy a principy OOP. Následuje přehled nejtypičtějších problémů, se kterými je možné se setkat při návrhu OO systémů. Závěrem je představeno několik obecností ohledně návrhových vzorů, coby odvětví SI, a zvolena obecná struktura, pomocí které jsou jednotně zhotoveny všechny rozbory v následující kapitole.

Kapitola *analýza návrhových vzorů* (č. 2) je hlavní náplní této práce. Jak již bylo zmíněno, zabývá se rozdělením návrhových vzorů do několika kategorii a jejich následným rozborem.

Kapitola *návrh prototypu* (č. 3) popisuje fiktivní aplikaci, jež je následně využita jako základní stavební kámen pro praktické demonstrace jednotlivých vzorů. Nejpodstatnější úvahou je zde vylíčení problematiky, kterou prototyp řeší, společně s navržením a příslušným odůvodněním vysokoúrovňové architektury.

Poslední kapitola *realizace demonstračních příkladů* (č. 4) objasňuje, jakým způsobem byly vyhotoveny ukázky probraných vzorů za pomoci navrženého prototypu. Její podstatou není opětovný popis každého z již jednou zpracovaných vzorů, ale spíše obecné vysvětlení myšlenky, kterou se řídí veškeré příklady. Přesné detaily jsou již poté součástí veřejně dostupného katalogu, který je jedním z výstupů této práce. Na závěr kapitoly je shrnuto několik poznatků, které vyplynuly, jak ze samotného snažení, tak i odezvy od expertů v příslušném oboru.

Jak bylo právě řečeno, jedním z výstupů je katalog praktických ukázek, který slouží jako alternativní výukové médium. Konkrétní implementace se snaží reflektovat reálná úskalí, avšak s pomyslnými funkčnostmi — tento přístup byl zvolen z podstaty celého projektu, kterou je co nejpříznivěji obeznámit čtenáře s návrhovými vzory. Zbytečně komplexní funkcionality by tedy byly poněkud kontraproduktivní, jelikož by jen strhávaly pozornost na sebe.

Mimo rozsah stanovených cílů byla praktická část práce poskytnuta týmu vývojářů, jehož členové pokrývají široké spektrum zkušeností, od poměrně neznalých jedinců až po seniorní programátory, kteří se pohybují v oboru několik dekád. Ačkoliv je zpětná vazba poskytnutá touto skupinou mírně zkreslená, jelikož se jedná o vývojáře ze stejného sektoru jako je navrhovaný prototyp, její celkový tón byl pozitivní a z konstruktivní kritiky vyplynulo několik nápadu na potenciální vylepšení (součást shrnuti na konci poslední kapitoly — č. 4).

Sečteno podtrženo, všechny cíle práce byly splněny a tento projekt se snad stane dobrým rozcestníkem pro uvedení mladých vývojářů do problematiky návrhových vzorů.

Bibliografie

1. WEISFELD, Matt A. *The object-oriented thought process*. 3rd ed. Upper Saddle River: Addison-Wesley, 2009. ISBN 978-0-672-33016-2.
2. DOHERTY, Erin. What is object-oriented programming? OOP explained in depth. *Educative* [online]. 2020 [cit. 2022-04-26]. Dostupné z: `https://www.educative.io/blog/object-oriented-programming`.
3. PARR, Kealan. The Four Pillars of Object-Oriented Programming. *freeCodeCamp* [online]. 2020 [cit. 2022-04-30]. Dostupné z: `https://www.freecodecamp.org/news/four-pillars-of-object-oriented-programming/`.
4. BROOKS, Frederick P. *No Silver Bullet: Essence and Accident in Software Engineering*. Chapel Hill, 1986. Tech. zpr. UNC.
5. GAMMA, Erich. *Návrh programů pomocí vzorů: stavební kameny objektově orientovaných programů*. 1. vyd. Praha: Grada, 2003. ISBN 80-247-0302-5.
6. CLEM, Roy. Identifying Object-Oriented Classes. *CodeProject* [online]. 2005 [cit. 2022-05-06]. Dostupné z: `https://www.codeproject.com/Articles/9900/Identifying-Object-Oriented-Classes`.
7. *AntiPatterns* [online]. SourceMaking, 2022 [cit. 2022-05-05]. Dostupné z: `https://www.sourcemaking.com/antipatterns`.
8. SCOBEY, Porter. *Design Patterns: Overview and BackGround* [online]. Saint Mary's University, 2022 [cit. 2022-05-10]. Dostupné z: `https://cs.smu.ca/~porter/csc/465/notes/design_patterns.html#:~:text=A%5C%20Brief%5C%20History%5C%20of%5C%20Design,(They%5C%20had%5C%20253%5C%20patterns.)`.
9. SINGLA, Lalit. What's a Software Design Pattern? *Net Solutions* [online]. 2022 [cit. 2022-05-12]. Dostupné z: `https://www.netsolutions.com/insights/software-design-pattern/#7-best-software-design-patterns`.
10. *Design Patterns* [online]. Refactoring.Guru, 2022 [cit. 2022-09-14]. Dostupné z: `https://refactoring.guru/design-patterns`.
11. SEPPÄLÄ, Ilkka. *Java Design Patterns* [online]. 2021 [cit. 2022-09-14]. Dostupné z: `https://java-design-patterns.com`.
12. SUTER, Rico. *Software Pattern* [online]. 2022 [cit. 2022-09-14]. Dostupné z: `http://software-pattern.org`.
13. *GoF Patterns* [online]. DistributedNetworks LLC, 2021 [cit. 2022-09-14]. Dostupné z: `https://www.gofpatterns.com/creational/index.php`.
14. *Abstract Factory* [online]. OODesign.com, 2006 [cit. 2022-09-18]. Dostupné z: `https://www.oodesign.com/abstract-factory-pattern`.

15. MURUGAN, Ramachandran. Design Pattern Series: Singleton and Multiton Pattern. *CodeProject* [online]. 2017 [cit. 2022-09-19]. Dostupné z: https://www.codeproject.com/Articles/1178694/Singleton-and-Multiton-Pattern.
16. KUMAR, Saket. Object Pool Design Pattern. *GeeksforGeeks* [online]. 2020 [cit. 2022-09-21]. Dostupné z: https://www.geeksforgeeks.org/object-pool-design-pattern/.
17. LABS, Escape Velocity. Software Design Pattern 4: Lazy Initialization. *Medium* [online]. 2021 [cit. 2022-09-24]. Dostupné z: https://medium.com/geekculture/software-design-pattern-4-lazy-initialization-35f606f1ddf3.
18. *Lazy Loading Pattern* [online]. Source Code Examples, 2018 [cit. 2022-09-24]. Dostupné z: https://www.sourcecodeexamples.net/2018/05/lazy-loading-pattern.html.
19. Tree (data structure). *Wikipedia* [online]. 2022 [cit. 2022-09-30]. Dostupné z: https://en.wikipedia.org/wiki/Tree_(data_structure).
20. MÖSSENBÖCK, Hanspeter. *Twin — A Design Pattern for Modeling Multiple Inheritance*. Altenbergerstraße 69, A-4040 Linz, 2000. Tech. zpr. Johannes Kepler University Linz.
21. *The Mock Object Pattern* [online]. Project Management Institute, 2022 [cit. 2022-10-05]. Dostupné z: https://www.pmi.org/disciplined-agile/the-design-patterns-repository/the-mock-object-pattern.
22. HASSAN, Ahmed Shamim. Observer vs Pub-Sub Pattern. *Better Programming* [online]. 2017 [cit. 2022-10-07]. Dostupné z: https://betterprogramming.pub/observer-vs-pub-sub-pattern-50d3b27f838c#:~:text=In%5C%20the%5C%20observer%5C%20pattern%5C%2C%5C%20the,message%5C%20queues%5C%20or%5C%20a%5C%20broker..
23. CHOVATIYA, Vishal. Double Dispatch in C++. *DZone* [online]. 2020 [cit. 2022-10-11]. Dostupné z: https://dzone.com/articles/double-dispatch-in-c.
24. BAIN, Scott. The Null Object Pattern. *Project Management Institute, Inc.* [Online]. 2022 [cit. 2022-10-12]. Dostupné z: https://www.pmi.org/disciplined-agile/the-design-patterns-repository/the-null-object-pattern.
25. SCHMIDT, Douglas C. *Active Object, an Object Behavioral Pattern for Concurrent Programming*. St. Louis, 1998. Tech. zpr. Washington University. Dostupné také z: https://www.dre.vanderbilt.edu/~schmidt/PDF/Active-Objects.pdf.
26. ESHLEMAN, Matthew. What is an Active Object? *Cove Mountain Software* [online]. 2021 [cit. 2022-10-16]. Dostupné z: https://covemountainsoftware.com/2021/04/20/what-is-an-active-object/.
27. SUN, Hong. CONCURRENCY PATTERNS — ACTIVE OBJECT AND MONITOR OBJECT. *TopCoder* [online]. 2019 [cit. 2022-10-16]. Dostupné z: https://www.topcoder.com/thrive/articles/Concurrency%5C%20Patterns%5C%20-%5C%20Active%5C%20Object%5C%20and%5C%20Monitor%5C%20Object.
28. SCHMIDT, Douglas C. *Proactor, an Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events* [online]. St. Louis, 1998 [cit. 2022-10-30]. Tech. zpr. Washington University. Dostupné z: https://www.dre.vanderbilt.edu/~schmidt/PDF/Proactor.pdf.
29. *CppCon 2017: Sean Bollin "Reactor vs. Proactor"* [online]. YouTube, 2017 [cit. 2022-11-02]. Dostupné z: https://www.youtube.com/watch?v=iMRbm32O0ws.
30. DINH, Lau Nguyen; CHIEN, Tran Quoc. *Traveling Salesman Problem in Distributed Environment* [online]. Vietnam, 2015 [cit. 2022-11-03]. Tech. zpr. University of Da Nang. Dostupné z: https://www.researchgate.net/publication/301454434_Traveling_Salesman_Problem_in_Distributed_Environment.

31. SCHMIDT, Douglas C. *Thread-Specific Storage for C/C++, an Object Behavioral Pattern for Accessing per-Thread State Efficiently* [online]. St. Louis, 1998 [cit. 2022-11-03]. Tech. zpr. Washington University. Dostupné z: `https://www.dre.vanderbilt.edu/~schmidt/PDF/TSS-pattern.pdf`.

32. *Use Cases for Thread-Local Storage* [online]. Paul E. McKenney a JF Bastien, 2014 [cit. 2022-12-26]. Dostupné z: `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4324.html`.

33. DYSON, Jonathan. Conjoining FURPS and MoSCoW to Analyse and Prioritise Requirements. *LinkedIn* [online]. 2019 [cit. 2022-12-26]. Dostupné z: `https://www.linkedin.com/pulse/conjoining-furps-moscow-analyse-prioritise-jonathan-dyson`.

34. *Red Hat OpenShift* [online]. Red Hat, 2022 [cit. 2022-12-27]. Dostupné z: `https://www.redhat.com/en/technologies/cloud-computing/openshift`.

35. *Apache Kafka* [online]. Apache Software Foundation, 2022 [cit. 2022-12-27]. Dostupné z: `https://kafka.apache.org/`.

36. *CLion* [online]. JetBrains s.r.o., 2022 [cit. 2022-12-28]. Dostupné z: `https://www.jetbrains.com/clion/`.

37. *GitHub Actions* [online]. GitHub, Inc., 2022 [cit. 2022-12-28]. Dostupné z: `https://docs.github.com/en/actions`.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `design-patterns` ..... zdrojové kódy demonstračních ukázek vzorů
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text`
  - `thesis.pdf` ..... text práce ve formátu PDF