Využití návrhových vzorů pro vývoj naváděcích systému

František Vašica

Bakalářská práce
2019

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2018/2019

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**František Vašica**
Osobní číslo:	**A16196**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Bezpečnostní technologie, systémy a management**
Forma studia:	**kombinovaná**
Téma práce:	**Využití návrhových vzorů pro vývoj naváděcích systémů**
Téma anglicky:	**The Use of Design Patterns for the Development of Homing Systems**

Zásady pro vypracování:

1. **Popište obecně návrhové vzory.**
2. **Seznamte se s naváděcími systémy.**
3. **Popište metody navádění.**
4. **Analyzujte možnosti návrhových vzorů pro vývoj naváděcích systémů.**
5. **Zvolte vhodné návrhové vzory pro vývoj.**
6. **Demonstrujte využití návrhových systémů pro vývoj – vytvořte ukázku naváděcího systému.**

Rozsah bakalářské práce:

Rozsah příloh:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **FOWLER, Martin. Patterns of enterprise application architecture. Boston: Addison-Wesley, 2003, xxiv, 533 s. The Addison-Wesley Signature Series. ISBN 978-0-321-12742-6.**
2. **NAGEL, Christian, Jay GLYNN a Morgan SKINNER. Professional 5.0 and .NET 4.5.1. Indianapolis, IN: John Wiley and Sons, 2014, 1 online zdroj (1563 pages). ISBN 978-1-118-83294-3.**
3. **GAROFALO, Raffaele. Building enterprise applications with Windows Presentation Foundation and the model view ViewModel Pattern. Sebastopol, Calif:Reilly Media, 2011. ISBN 978-073-5650-923.**
4. **GALLOWAY, Jon. Professional ASP. NET MVC 5. Indianapolis, Indiana: John Wiley \& Sons, Inc., 2014, 1 online zdroj (622 pages). ISBN 978-1-118-79476-0.**

Vedoucí bakalářské práce:	**Ing. Lukáš Králík** Ústav počítačových a komunikačních systémů
Datum zadání bakalářské práce:	**21. prosince 2018**
Termín odevzdání bakalářské práce:	**15. května 2019**

Ve Zlíně dne 21. prosince 2018

doc. Mgr. Milan Adámek, Ph.D.
děkan

Ing. Jan Valouch, Ph.D.
ředitel ústavu

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne ……………………….

podpis diplomanta

ABSTRAKT

Cílem práce je seznámit čtenáře s principy navádění a návrhovými vzory, které se používají při implementaci objektivně orientovaných návrhů. Čtenáři budou vysvětleny principy objektivně orientovaných návrhů, pro pochopení výhod návrhových vzorů. Vybrané návrhové vzory jsou účelně popsány a graficky znázorněny. Každý vzor má soupis výhod a příkladů použití. Pro demonstraci je ve druhé části práce vytvořen objektový návrh dynamického naváděcího systému, ve kterém je použito návrhových vzorů. Cílem druhé části je vytvořit dynamický objektový návrh pro naváděcí systém, implementující rozhraní tak, aby si každý programátor mohl systém libovolně rozšířit při dodržení určitých pravidel.

Klíčová slova:

C#, .NET, Návrhový vzor, Jedináček, Tovární metoda, Abstraktní továrna, Prototyp, Stavitel, Adaptér, Dekorátor, Fasáda, Muší váha, Most, Skladba, Zástupce

ABSTRACT

The goal of this work is to familiarize the reader with guiding principles and design patterns, which are used in the implementation of objectively oriented proposals. The principles of objectively oriented proposals will be explained to readers to understand the benefits of design patterns. Selected design patterns are described and graphically illustrated. Each pattern has a list of benefits and examples of usage. For the demonstration, in the second part of the thesis, an object design of the dynamical guiding systém is created, which is using design patterns. The goal ofthe second part is to create a dynamic object-oriented design for a Guyance systém that implements the interface so that each programmer can expand the systém arbitrarily by following certain rules.

Keywords:

C#, .NET, Design pattern, Singleton, Factory method, Abstract factory, Prototype, Builder, Adapter, Decorator, Façade, Fly-Weight, Bridge, Composite, Proxy

Rád bych touto cestou poděkoval všem lidem, kteří mi při tvorbě práce vyšli vstříc ve volbě tématu, psychickou podporou i jejich časovou flexibilitou. Primárně mému vedoucímu práce, panu Ing. Lukáši Králíkovi, který byl vždy k dispozici, když jsem jej potřeboval.

Prohlašuji, že odevzdaná verze bakalářské/diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD 9
I TEOREТICKÁ ČÁST 10
1 NAVÁDĚCÍ SYSTÉMY 11
1.1 DETEKCE OBJEKTU V PROSTORU POMOCÍ RADAROVÝCH SYSTÉMŮ 11
1.1.1 Algoritmus pro princip činnosti radaru 11
1.2 METODY NAVÁDĚNÍ RAKET 12
1.2.1 Příkazová metoda 13
1.2.2 Inerciální metoda 13
1.2.3 Aktivní metoda 13
1.2.4 Semiaktivní metoda 14
1.2.5 Pasivní metoda 14
1.3 ZAMĚŘOVÁNÍ CÍLE POMOCÍ SIGNÁLU 14
1.3.1 Trilaterace signálu 15
1.3.2 Triangulace signálu 15
1.4 KVADROKOPTÉRY 17
1.5 BIOMETRIE 18
1.5.1 Koeficient nesprávného přijetí a odmítnutí 19
2 OBJEKTIVNĚ ORIENTOVANÉ PROGRAMOVÁNÍ 20
2.1 ZÁKLADNÍ POJMY 20
2.1.1 Polymorfismus 20
2.1.2 Zapouzdření 20
2.1.3 Dědičnost 20
2.2 ZÁSADY OBJEKTIVNĚ ORIENTOVANÉHO PROGRAMOVÁNÍ 21
2.2.1 Vysvětlení klíčových slov C# 21
3 NÁVRHOVÉ VZORY 22
3.1 KATEGORIZACE NÁVRHOVÝCH VZORŮ 22
3.2 VAZBY MEZI TŘÍDAMI UML 24
3.2.1 Asociace 24
3.2.2 Mnohočetnost 24
3.2.3 Směrová asociace 24
3.2.4 Reflexivní asociace 24
3.2.5 Agregace 25
3.2.6 Kompozice 25
3.2.7 Generalizace/ Dědictví 25
3.2.8 Realizace 26
II PRAKTICKÁ ČÁST 27
4 ANALÝZA NÁVRHOVÝCH VZORŮ 28

4.1 Tovární metoda (Factory Method) 28
4.2 Abstraktní továrna (Abstract Factory) 29
4.3 Jedináček (Singleton) 30
4.4 Prototyp (Prototype) 31
4.5 Stavitel (Builder) 32
4.6 Adaptér (Adapter) 33
4.7 Dekorátor (Decorator) 34
4.8 Fasáda (Facade) 34
4.9 Most (Bridge) 35
4.10 Muší váha (Fly-weight) 37
4.11 Skladba (Composite) 37
4.12 Zástupce (Proxy) 38
4.13 Interpret 39
4.14 Šablonová metoda (Template Method) 41
4.15 Iterátor 41
4.16 Návštěvník (Visitor) 43
4.17 Obnovitel (Memento) 43
4.18 Pozorovatel (Observer) 44
4.19 Prostředník (Mediator) 46
4.20 Příkaz (Command) 47
4.21 Řetězec odpovědnosti (Chain of responsibility) 47
4.22 Stav (State) 48
4.23 Strategie (Strategy) 49
4.24 Shrnutí 50
5 VYTVOŘENÍ NAVÁDĚCÍHO SYSTÉMU 51
5.1 Základnírozdělenísystému na subsystém 51
5.2 Hardware 51
5.2.1 Dynamické načítání hardware 53
5.2.2 Dostupné enumerace 55
5.2.3 Napojení na jádro 56
5.3 Grafické uživatelské rozhraní GUI 57
5.3.1 Modulární uživatelské rozhraní 57
5.3.2 Vytvoření pluginu 59
ZÁVĚR 60
SEZNAM POUŽITÉ LITERATURY 61
SEZNAM OBRÁZKŮ 65
SEZNAM TABULEK 67
SEZNAM PŘÍLOH 68

ÚVOD

Vytvoření kvalitní robustní aplikace, kterou bude možné v budoucnu nadále rozšiřovat a udržovat není jednoduchý úkol. V mnoha případech se lze setkat s konstrukcemi, které se stále opakují napříč aplikacemi. Řeč je o návrhových vzorech. Obecně ten, který se s návrhovými vzory ještě nesetkal, je při své práci může používat, aniž by o nich věděl anebo také ne a tak nemusí sáhnout po úplně optimálním řešení. V této práci demonstruji použití návrhových vzorů pomocí UML diagramů, implementace je pak už závislá na zvoleném jazyku. V příloze naleznete jednoduché demonstrativní kódy vybraných vzorů. Obsahem praktické části je obsahová analýza návrhových vzorů a následná demonstrace těchto vzorů pro sestavení objektového návrhu naváděcího systému. Navigace se zaměřuje na sledování pohybu určitého předmětu a jeho navedení do cílového bodu. Navádění pracuje na principu vysílač-přijímač, kdy je vše již automatizováno, případně poloautomatizováno.

V teoretické části jsou vysvětleny principy detekce objektů, které jsou nedílnou součástí pro navádění. Jsou zde popsány stávající techniky navádění raket, metody triangulace a trilaterace signálů. Součástí teoretické části je také kapitola věnující se vývoji dronů s myšlenkami pro uplatnění pro vojenský průmysl. Dále se lze v teoretické části dočíst o základech objektově orientovaného programování a výhodách při využití návrhových vzorů.

V praktické části je provedena obsahová analýza návrhových vzorů s jejich popisem a případech, kdy je vhodné je využít a následně demonstrace využití návrhových vzorů, pro vytvoření naváděcího střelného systému. Návrh systému je modulární, lze podle pravidel rozšiřovat část pro hardware, tak i pro uživatelské rozhraní. Systém může ovládat odpalování raket a stejně tak by mohl řídit chování dronů.

Systémy ovládající navádění raket používaných v armádě jsou i z důvodu bezpečnosti černou skříňkou, kdy by se jednalo o bezpečnostní díru, pokud by se jejich implementace dostala na veřejnost. Mít své vlastní byť i náhradní řešení obranného bezpečnostního systému (naváděcího systému) by mělo být státním zájmem, kdy se nespoléháme jen na externího dodavatele. Vyvstává zde bezpečnostní otázka, zda externí dodaný systém neobsahuje zadní vrátka a zda by v krajním případě potřeby nemohl být deaktivován. Konkurence schopnost je na straně druhé. Z těchto důvodů považuji za potřebné tuto problematiku řešit a ve své práci nabízím jednoduché řešení na konceptuální úrovni.

I. TEORETICKÁ ČÁST

1 NAVÁDĚCÍ SYSTÉMY

S rozvojem technologií se klade důraz na automatizaci procesů. Mnoho moderních strojů pro svou orientaci v prostoru využívá různé techniky navigace pro určení své polohy, tak aby mohl proces na základě těchto dat reagovat. Naváděcí systémy nelze jednoznačně kategorizovat. Rozdělení by mohlo být podle odvětví, dosahu nebo podle použitých technologií. Pro všechny navigační systémy ale platí, že mají část přijímací (radar, kamera, laser) a vysílací. Skrze jeden typ senzoru dostávám informace potřebné k vyhodnocování, zpravidla kde se hledaný nebo naváděný objekt nachází, a skrze druhý typ senzoru vysílám instrukce, co se má stát. S ohledem na bezpečnost v komerčních technologiích se nám nabízí využití kamerových systémů jako přijímačů a zároveň detektorů pro sledování objektů.

Kamery se začaly hojně využívat při programování robotů, plní zde úlohu očí a v tomto případě mluvíme o počítačovém vidění. Princip je velmi podobný jako u monitorovacích systémů. Monitorovací systémy se primárně zaměřovali na detekci obličeje, postavy nebo auta. Pokud mluvíme o počítačovém vidění, robot se snaží rozpoznat vše, co by pro něj mohlo znamenat překážku. Někteří mohou mít zaimplementovanou jednoducho umělou inteligenci, aby se dovedli po cestě učit, případně reagovat na vyhodnocovací odchylky.

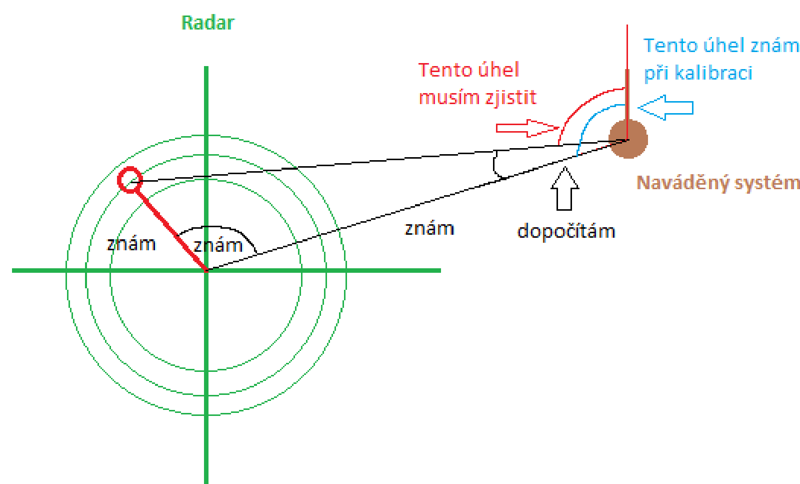
1.1 Detekce objektu v prostoru pomocí radarových systémů

S problematikou navádění na cíl je spojena problematika detekce objektu v zájmovém prostoru. Tento objekt lze detekovat pomocí různých prostředků. Mezi nejpoužívanější patří radary a kamerové systémy.

1.1.1 Algoritmus pro princip činnosti radaru

V případě ne-kamerového řešení je třeba zajistit získání souřadnic objektu v prostoru, tak aby námi naváděný systém mohl s těmito souřadnicemi pracovat. Pokud bychom uvažovali palný systém, musíme mu dát informaci o x a y poloze objektu, aby věděl úhel, o který se má natočit v jednotlivých osách. Stejnou logiku by měl například systém, který postupně otáčí kameru na pachatele (sleduje jej). Nutnost před spuštěním mít systém kalibrovaný a synchronizovaný. Problematika detekce objektu souvisí s měřením vzdálenosti objektu od detektoru. Pro tyto účely se dá využít různých typů infračervených (IR) nebo mikrovlnných vysílačů. [1]

Zde se objevuje druhý problém, kdy výše zmíněné detektory patří mezi lineární. Zjednodušeně lze říct, že například IR vysílač vysílá paprsek. To znamená, že údaj o poloze získáme jen v případě, že jej objekt výslovně protne. Pro získání polohy objektu pro *osu-x* i *osu-y* stáčí takový vysílač roztočit. Nutná informace v tomto případě je rychlost otáčení senzoru a frekvence vzorkování, zbytek už se dá dopočítat. [2][3]



Obr. 1 Princip zaměření pomocí radaru

Pro výpočet lze použít trigonometrii. Je třeba vyřešit dopočty úhlů trojúhelníku SUS (strana-úhel-strana) (1)(2)(3). Na to lze použít kosinovu větu.[4]

$$a^2 = b^2 + c^2 - 2bc * \cos \alpha \tag{1}$$

$$b^2 = a^2 + c^2 - 2ac * \cos \beta \tag{2}$$

$$c^2 = a^2 + b^2 - 2ab * \cos \gamma \tag{3}$$

1.2 Metody navádění raket

Pojem raketa se začal používat po druhé světové válce, jako synonymum k naváděné střele. Naváděná střela má typicky vlastní pohon a její trajektorie je někým nebo něčím řízena, oproti balistickým střelám. Naváděcí systém se pak skládá se ze dvou hlavních částí a to systému řízení polohy a systému řízení letu. Systém řízení polohy řídí stoupání a dráhu letu, včetně odchylek způsobenými okolními vlivy. Systém řízení letu má za úkol určovat dráhu nezbytnou pro zacílení. Dává příkazy systému řízení polohy ke směřování cíle. [5][6]

Naváděcí metody lze rozdělit do pěti kategorií:

1. Příkazová metoda
2. Inerciální metoda
3. Aktivní metoda
4. Semiaktivní metoda
5. Pasivní metoda [5][6]

1.2.1 Příkazová metoda

Umožňuje řízení rakety z místa odpálení. Přenos příkazů se odehrává na základě rádiové vln nebo laserových pulzů. Sledování může být prováděno i pomocí obrazového přenosu z rakety. Z počátku byly rakety ovládány ručně, později byly využity metody, kdy raketa snímala radarový paprsek ukazující na cíl a automaticky provedla korekci směru. Malé televizní kamery připevněné na hlaveň rakety umožňovali operátorovy vykreslit cíl, ten pak mohl provést autokorekci. Americký systém Patriot umožňoval zasílat z rakety relativní informace o rychlosti a směru cíle, kontrolnímu systému. Ten pak přepočítával optimální trajektorii pro průsečík střetu a upravoval směr řízené rakety. [5][6][7]

1.2.2 Inerciální metoda

Srdcem inerciální navigace jsou uspořádané akcelerometry a vysoce přesné gyroskopy. Této techniky se začalo využívat po roce 1970. Pomocí těchto senzorů lze nepřetržitě určovat polohu rakety v prostoru. Tyto senzory slouží jako vstupy pro výpočetní naváděcí jednotku, která pak na tomto základě udržuje kurz. Výpočetní systém navigace je umístněn na raketě. Výhoda této metody je, v tom, že nevysílá, ani nepřijímá žádné signály. Rušení na ni neplatí a je těžko odhalitelná nepřítelem. Některé protilodní a dálkové rakety používají inerciální metodu, aby se dostali co nejblíže k cíli, poté přepnou na aktivní radarové navádění. Rakety mohou mít v integrovaném obvodě paměť pro alternativní chování. [5][6][7]

1.2.3 Aktivní metoda

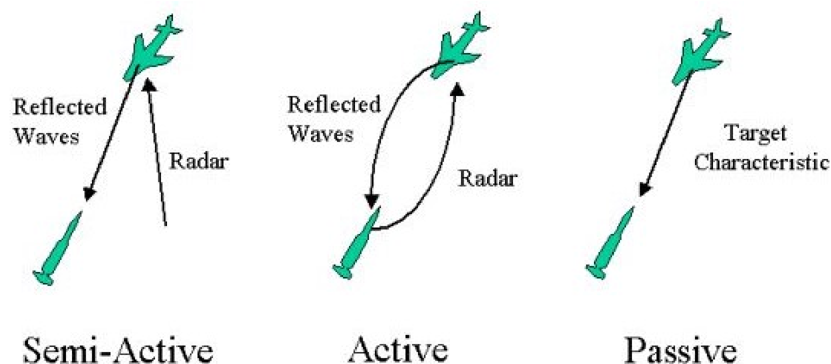
Raketa je vybavena vysílačem a přijímačem. Vzhledem k těsné vzdálenosti jsou měřené výsledky přesnější. Aktivně vysílá na svůj cíl a přijímá. Nevýhodou je právě vysílaný signál, protivník jej může rušit nebo jinak podvést. Přesto má aktivní metoda velkou úspěšnost. Kombinaci aktivní metody využívá technika, vystřel a zapomeň (pod čarou překlad). [5][6][7]

1.2.4 Semiaktivní metoda

Princip semiaktivní metody je ten, že cíl je označen někým jiným než raketou samotnou. V případě vzdušné obrany může být na cíl vysílán radarem signál buď z radarové stanice anebo stíhacího letounu. Tento signál se od cíle odráží a přijímá jej naváděcí raketa a na tomto základě upravuje souřadnice. V případě protitankových střel Hellfire se využívá této techniky také, ale cíl se zaměřuje pomocí laserové frekvence. Vrtulník pak může vystřelit mnoho mil od svého cíle. [5][6][7]

1.2.5 Pasivní metoda

Pasivní metoda oproti aktivní vůbec nevysílá. Prvotní pasivní systémy navádění cílily na vyzařované teplo proudových motorů pomocí detekce infračerveného záření. Pozdější systémy cílily na cíl pomocí detekce UV záření. Pokročilé pasivní naváděcí systémy využívají infračerveného záření k vytvoření obrazu skoro jako u lidského oka, kdy následně cílí na základě obrazu a ne jen odrazu signálu. [5][6][7]



Obr. 2 Naváděcí metody (převzato z [7])

1.3 Zaměřování cíle pomocí signálu

K zaměření nebo detekci cíle lze využít rádiových signálů. Pro uskutečnění zaměření se používají techniky, které jsou často mezi sebou zaměňovány. Jedná se trilateraci a triangulaci signálů. Techniky vychází ze znalostí polohy vysílačů a výpočty se opírají o trigonometrii. V případě přežívání v přírodě a potřebné lokalizace polohy se využívá triangulace s pomocí buzoly.

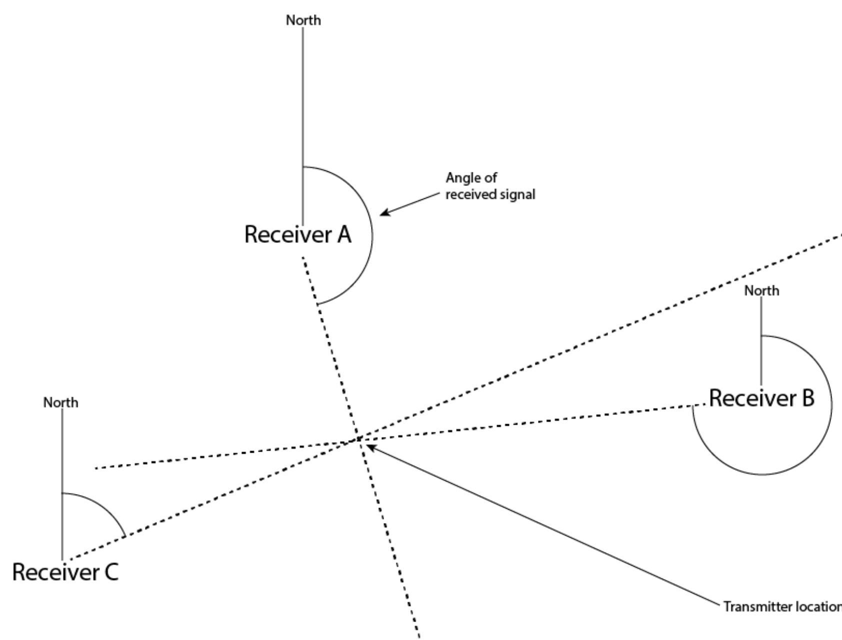
1.3.1 Trilaterace signálu

Trilateraci signálu využívá náš globální poziční systém GPS. Aby bylo možné řádně zaměřit cíl (například náš mobilní telefon), je k tomu třeba tří satelitů. Jeden satelit na jednu souřadnici, to znamená tři satelity pro souřadnice *x,y,z*. Trilaterace má základ v měření vzdáleností. GPS satelity konstantně vysílají (broadcast) dvě části informace a to jejich aktuální pozici a současný čas. Satelity v sobě mají zabudované velice přesné hodiny, kterým říkáme atomické hodiny. Jejich signál je šířen rychlostí světla. K dispozici jsou tedy všechny potřebné veličiny k určení vzdálenosti od daného satelitu a to rychlost a časový rozdíl vysílače a přijímače. Nalezení aktuální polohy pak spočívá v nalezení průsečíku tří kružnic o poloměru spočtené vzdálenosti. [9][10][11][12]

Pokud se budeme držet reálné situace, kdy není synchronizován čas na přijímači a vysílači, umožní nám tři satelity zaměřit cíl s přesností do 1 km^2. S přenosem signálu nic dělat nelze, blíží se rychlosti světla, kde může být jen nepatrná odchylka. Vysílač má v sobě atomické hodiny, u kterých se vylepšit také moc nedá, jediné co lze ovlivnit je čas na přijímacím zařízení. V tomto případě přichází na řadu čtvrtý satelit, který umožní korekci času tím, že se přidá nebo odečte od času na přijímači a pak dostaneme velmi precizní polohu. Pro precizní zaměření tedy potřebujeme čtyři satelity. [9][10][11][12]

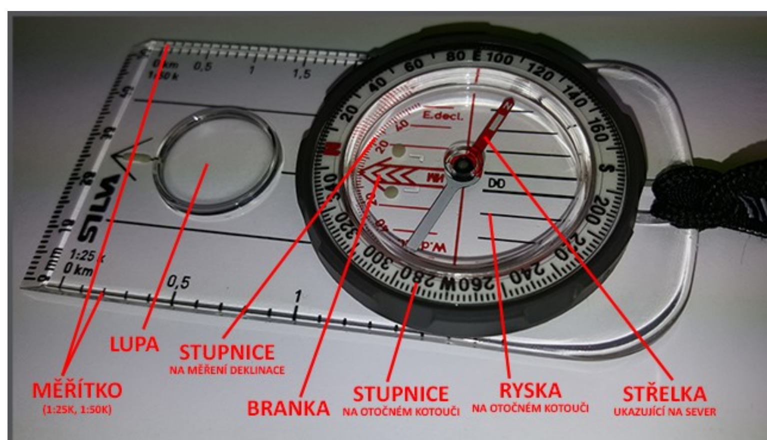
1.3.2 Triangulace signálu

Triangulace jak již název vypovídá, počítá polohu na základě úhlů. K uskutečněné této metody je třeba tří pevných přijímacích stanic nebo jedné pohyblivé (provede se jedno měření, přejde se na jiné stanoviště a měření se opakuje). Síla přijímaného signálu je vyšší v případě, že anténa ukazuje na vysílač. Stanice musí být vybavena směrovou anténou, která určuje úhel přijímaného signálu. V momentě, kdy všechny přijímací stanice určí úhel signálu, v místě, kde byl naměřen nejsilnější, lze pomocí trigonometrie dopočítat průnik těchto směrů. Výhodou této metody je, že není třeba synchronizace času a atomických hodin. Referenční bod stanic, pro měření úhlu může být magnetický sever. [12][13][14]



Obr. 3 Triangulace signálů (převzato z [12])

Stanovit tímto způsobem svou pozici lze i pomocí buzoly a mapy. Mapa je nejprve natočena tak, aby sever mapy, odpovídal severu střelky buzoly (referenční bod). Poté se hledá nejvíce specifická věc, pro dané místo, kde se nacházím, ve většině případů se jedná o nejvyšší kopec. Toto specifické místo je následně zaměřeno, skrze hranu buzoly a střelka je zafixována brankou buzoly. Následně se buzola přenese na mapu, celou buzolou je třeba otočit, tak aby se střelka dostala opět do branky a zároveň její hrana procházela specifickým místem na mapě. Na konec se udělá čára a celý proces se opakuje znovu pro jiné specifické místo. Další čára by nám měla protnout první a v místě průsečíku by měla být naše pozice. [13][14][15]



Obr. 4 Popis buzoly (převzato z [16])

1.4 Kvadrokoptéry

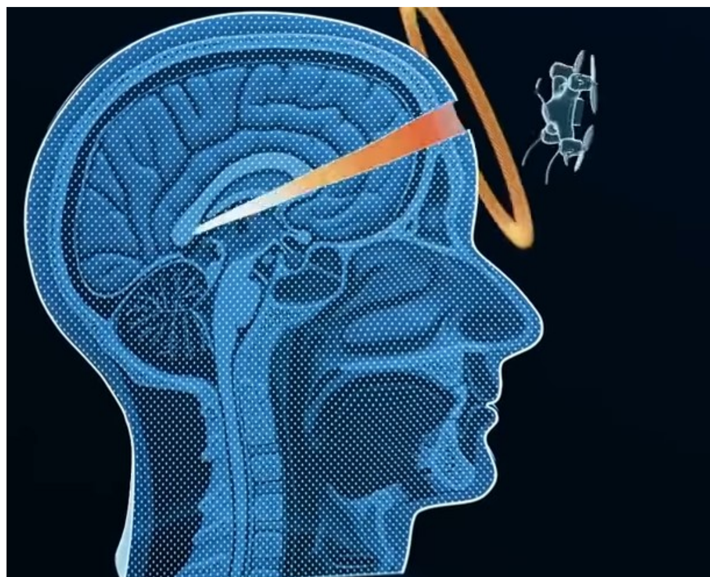
Patří mezi bezpilotní letouny, obecně známé pod pojmem drony. Kvadrokoptéra je nejvíc, veřejně využívaným typem bezpilotního letounu. Vyznačuje se svou vlastností vertikálního vzletu. Její součástí je kamera nebo fotoaparát, který umožňuje mapování terénu a proto láká spoustu příznivců, kteří si drony pořizují jako svou zálibu. Kvadrokoptéry o rozměrech kolem 15 centimetrů mají průměrnou délku letu okolo 20 minut. Mohou být vybaveny spoustou senzorů, od kterých se pak odvíjí i cena. Novější drony jsou vybaveny duálním globálním navigačním systémem, ale je možné je ovládat i bez použití satelitní navigace. Systém pro vyhýbání se kolizím je realizován pomocí kombinace následujících senzorů:

1. Optický senzor
2. Ultrazvukový
3. Infračervený
4. Lidar
5. ToF senzor (Time of Flight) [17][18]

Mezi pokročilé technologie v oblasti počítačového vidění pro drony, patří technologie SLAM a Omni-directionalObstacleSensing. Obě technologie používají algoritmy pro zpracování obrazu, vytvářejí obrazy 3D map. Video je přenášeno pomocí rádiového signálu, skrze vícepásmový bezdrátový vysílač. [17][18][19][20]

Při vývoji dronů je vytvořen počítačový model obdobný fyzickému. Vývoj probíhá za pomoci simulací. Principy detekcí a rozhodovacích algoritmů stojí na bázi umělé inteligence a učících se algoritmů, kdy se model dronu v simulačním prostředí sám učí podle nadefinovaných pravidel reagovat. Při začátku simulace se model pohybuje nejprve chaoticky a jeho přesnost narůstá se zvyšujícím se počtem cyklů. Obecně platí, že čím větší vzorek vstupních dat modelu předhodíme, tím přesnější ve výsledku je. Na tomto základě vznikají generace dronů, které jsou naučeny na velkém množství vstupních dat, ale třeba při odlišných vstupních podmínkách. Cílem učení je typicky vytvoření rozhodovacího stromu nebo neuronové sítě. Tento rozhodovací strom je implementován do fyzického dronu, očekávaný výsledek by pak měl být stejný jako u počítačového modelu. [20][21][22]

Tento systém vývoje značně usnadňuje vývoj bezpilotních letounů. V případě nevyužití simulací a algoritmů pro strojové učení by bylo třeba nespočet testů a přeprogramování zařízení.



Obr. 5 Princip útočného mikro-dronu (převzato z [23])

V současné době se objevují úvahy o nasazení umělé inteligence do útočných kvadrokoptér. Toto zařízení by se mohlo samo rozhodnout, kdo je nepřítel a neslo by na sobě jednu nábojnicí. Algoritmus pro detekci obličeje by byl použit pro nalezení hlavy nepřítele. Následně by dron provedl nálet a v těsné blízkosti by se vhodně natočil a vypálil nábojnici. [23]

1.5 Biometrie

Zaměřování objektů pomocí kamer se prolíná s disciplínou zvanou biometrie. Princip rozpoznávání pomocí kamer je založený na implementaci algoritmů, které vycházejí z biometrie. Biometrie se zabývá měřením fyziologických charakteristik jedinců a jejich chováním. Na základě měření fyziologických a behaviorálních znaků je možné určit unikátnost jedince. Pro vyhodnocení každého jedince, je nezbytné se zaměřit na znak, kterým disponují všichni jedinci. Za pomocí nastavení citlivosti se musí zajistit, aby nebyli dva jedinci vyhodnoceni jako jeden, zkoumaný znak by měl být měřitelný a neměnný (barva vlasů se dá snadno změnit). [24][25]

Aby mohla kamera vyhodnotit jeho pozici, je potřeba jej první vůbec detekovat. Můžeme si představit, že se kolem něj vykreslí obdélník a stanovuje se pak vzdálenost obdélníku od kamery. Pro tyto účely se hojně využívá knihovna OpenCV (open computer vision). Algoritmy jsou optimalizovány tak, aby byly schopné vyhodnocovat obrázky real-time. [26]

1.5.1 Koeficient nesprávného přijetí a odmítnutí

Koeficient nesprávného přijetí (FAR – False Accept Rate)(4) je bezpečnostní chyba, kdy vyhodnotíme nesprávnou osobu jako správnou. Tím můžeme do objektu vpustit potenciální hrozbu. Koeficient se udává v procentech jako pravděpodobnost nesprávného přijetí na počet pokusů. [24][25][27]

$$FAR = \left(\frac{N_{FA}}{N_{IIA}}\right) * 100 \quad (4)$$

N_{FA} – počet chybných přijetí

N_{IIA} – počet všech pokusů

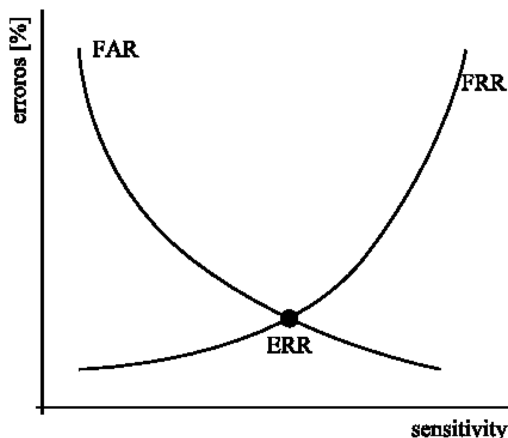
Koeficient nesprávného odmítnutí (FRR – False Rejection Rate)(5) jsou chyby, ne až tak kritické, kdy dojde k odmítnutí oprávněné osoby. Pokud jde o systém, kdy na základě FRR provádíme hlubší kontrolu sledovaného, například povinná prohlídka na letišti, jde spíše o komfort kontrolovaného. [24][25][27]

$$FRR = \left(\frac{N_{FR}}{N_{EIA}}\right) * 100 \quad (5)$$

N_{FA} – počet chybných odmítnutí

N_{IIA} – počet všech pokusů

Zpravidla platí, že pokud se FAR zmenší, FRR se zvětší. Pomocí těchto parametrů jde hodnotit spolehlivost tohoto systému. Stav kdy se FAR a FRR značíme ERR (Error Equal Rate) [27]



Obr. 6 Vztah mezi FRR a FAR (převzato z [27])

2 OBJEKTIVNĚ ORIENTOVANÉ PROGRAMOVÁNÍ

Objektivně orientované programování (OOP) je technika programování, kdy se programátor dívá na realitu jako na objekty. Pokud uvedu příklad na autě, můžu říct, že se jedná o určitý objekt, který se skládá, z dalších podobjektů, jako jsou kola, volant atd. Při psaní programu se tak snaží zabstraktnit realitu do té míry, do jaké ji potřebuje. Při psaní programu, který bude počítat rychlost auta, je jeho barva relevantní. Oproti procedurálnímu programování, využívá OOP, pojmy jako zapouzdření, dědičnost a polymorfismus.

2.1 Základní pojmy

Základní entitou v OOP je objekt. **Objekt** obsahuje atributy (vlastnosti) a metody (funkce). Atribut objektu, může být například *jméno, příjmení* a metoda například *JakSeJmenujes()*. Objekty jsou tvořeny na základě **tříd**. Říkáme pak, že objekt je **instancí** dané třídy. Mějme třídu *člověk*, která má atribut *jméno*. Nyní vytvoříme objekt třídy *člověk*, kterému dáme jméno *Josef*, a další objekt se jménem *Milan*. Nyní máme dva objekty, které mají stejné atributy i metody, ale hodnoty v nich se liší.[28]

2.1.1 Polymorfismus

S polymorfismem souvisí pojem *overriding*. U metod se jedná o klíčové slovo, které dovoluje přepsat metodu v nadřazené třídě, metodou z její podtřídy. Tím dosahujeme větší tvárnosti. S polymorfismem nesouvisí přetěžování metod/funkcí (*overloading*). Tato vlastnost nám umožňuje volat stejně pojmenovanou metodu s různými parametry nebo s různým návratovým typem. [29]

2.1.2 Zapouzdření

V souvislosti se zapouzdřením se bavíme o modifikátorech přístupu. Jsou to klíčové slova *private* (soukromý), *protected* (chráněný) a *public* (veřejný) před metodou nebo atributem. Tyto klíčové slova nám říkají, jak může být k těmto atributům přistupováno. **Soukromý** atribut je přístupný pouze vlastnímu objektu. **Chráněný** atribut je přístupný vlastnímu objektu a jeho potomkům. **Veřejný** atribut je přístupný odkudkoliv zvenčí. [30]

2.1.3 Dědičnost

Je vlastnost jazyka rozšiřovat nebo také specializovat už existující třídu. Jedná se o jednu z klíčových vlastností jazyka, která nám dává silný nástroj pro tvárnost aplikace.

Některé jazyky umožňují dědit z více tříd (vícenásobná dědičnost) například C++. Jiné zase mohou mít pouze jednoho dědice, ale můžou k tomu implementovat rozhraní, čímž vlastně mohou docílit stejného chování jako u vícenásobné dědičnosti.[28]

2.2 Zásady objektivně orientovaného programování

Obecně je dobré dbát při programování na určité zásady. Měli bychom se vyhýbat duplicitám kódu, dbát na zapouzdření, odpoutat část kódu, která by se mohla měnit, skrývat implementaci. Snažit se, aby jedna entita, řešila jeden problém. I když je to diskutabilní, tak nepodřizovat návrh snahám o maximální efektivitu (za cenu nepřehlednosti a složitosti). [31, s. 39-56]

2.2.1 Vysvětlení klíčových slov C#

Při programování tříd se lze setkat s různými typy modifikátorů. Názvy modifikátorů se mohou mezi různými programovacími jazyky lišit. Pro přehled je uvedena tabulka, která vysvětluje funkci daného modifikátoru pro programovací jazyk C#.

Tab. 1 Vysvětlení klíčových slov v C#

Abstract class	Třída nemůže vytvořit instanci, bude sloužit jako bázová třída.
Sealedclass	Ze třídy označené jako *sealed* není možné dědit, nebo ji rozšiřovat, jde o obdobou *final* v Javě. [32]
Partial class	Umožňuje implementaci jedné třídy ve více souborech.
Internal class	Třída je viditelná pouze v rámci své assembly, vhodné u wrapperů.
Abstract method	Musí být přepsány ve své podtřídě
Virtual method	Mohou být přepsány ve své podtřídě

Interface vsAbstractclass v C#

Interface (rozhraní) neumožňuje defaultní implementaci metod do verze C#8. Třída, která implementuje rozhraní, musí poskytnout implementaci všech metod v rozhraní. Abstraktní třída může obsahovat defaultní implementaci nebo stav. Abstraktní třída může být zděděna i bez implementace metod, jestliže je zděděná třída také abstraktní. Rozhraní může být děděno vícenásobně. Abstraktní třída ne.[33]

3 NÁVRHOVÉ VZORY

Při návrhu objektivně orientovaného softwaru předchází samotnému programování koncept nebo také návrh, vyjádřený nejčastěji pomocí diagramů. V první fázi návrháři softwaru dostanou k dispozici slovní popis problému, který má daná aplikace řešit. Je na návrháři, aby si zjistil k problematice co nejvíce detailů, případně mezních stavů. Formulovanou úlohu pak pomocí různých technik převedou na diagramy. Pro popis chování softwaru pomocí diagramů se často používají ER-diagramy, USE-CASE diagramy (případ užití), diagramy modelující popis komunikace tříd, objektová schémata a další. Pro převod textové úlohy na objektová schémata se často využívá techniky procházení textu a podtrhávání si jednotlivých podstatných jmen. Tyto podstatné jména budou z velké části znamenat objekty. Při dosažení konce se prochází celý text znovu a pro tentokrát se označují všechny slovesa. Slovesa značí, že nějaký objekt něco dělá, to znamená, že může být metodou určitého objektu.

Stejně jako u jiných profesí se i softwaroví návrháři zajímali o to, zda se dá jednou navržený projekt znovupoužít v jiném projektu. Pokud se podíváme například na klasické e-shopy na internetu, napadne nás samotné, zda mají nějaký stejný základ nebo zda neexistuje už nějaká šablona, podle které by mohl být e-shop vytvořen. V minulosti existoval i program DSSA, sponzorovaný Americkým Ministerstvem obrany, kde se měli vybraní softwaroví architekti soustředit na sběr architektonických informací a hledání podobností mezi těmito systémy. Bruece Anderson v roce 1991 vedl semináře s cílem vyvinout příručky pro softwarové architekty a mnoho dalších. To vše přispělo k rozvoji návrhových vzorů, které měli mapovat často opakující se problémy a způsob jejích řešení. Laicky řečeno, abychom znovu nevymýšleli kolo. [34, s. 344]

Základních návrhových vzorů je obecně uznáno 23. Setkat se lze i se vzory, které nejsou součásti obecně uznávaných, zde se už ale jedná o specializované vzory, které jsou kombinací několika uznávaných. Některé jsou zase tak jednoduché, že se nepovažují za návrhový vzor, ale spíše za idiom, příkladem může být vzor neměnný objekt (immutable-object). [31, s. 66-82]

3.1 Kategorizace návrhových vzorů

Návrhové vzory bychom mohly kategorizovat, na základě účelu, na základě, kterého byly navrženy. Jedná se o vzory tvořivé, strukturální a behaviorální (chování). Dále také

určujeme oblast, které se návrhový vzor týká. Jedná se o vzor třídní nebo objektový. Třídní vzory se zabývají vztahy mezi třídami a podtřídami, které se zavádějí prostřednictvím dědičnosti již při kompilaci. Objektové vzory se zabývají objektovými vztahy, které je možné měnit za běhu programu. [34, s. 29]

Tvořivé návrhové vzory zapouzdřují znalost o tom, které konkrétní třídy systém používá a skrývají to, jak se instance vytvářejí a dávají dohromady. Nabývají na důležitosti se vzrůstající závislostí na objektové skladbě, spíš než na dědičnosti. **Třídní** tvořivé vzory odkládají část své objektové tvorby na podtřídy. **Objektové** tvořivé vzory delegují tvorbu instancí na jiné objekty.*„Tvořivé návrhové vzory zabstraktňují proces tvorby instancí a pomáhají budovat systém, který je nezávislý na způsobu tvorby, skládání a vyjadřování jeho objektů.“*[34, s. 95]

Strukturální návrhové vzory se zabývají skládáním tříd a objektů a vytváření rozsáhlejších struktur. **Třídní** strukturální vzory používají dědičnost a skládání rozhraní nebo implementací. Užitečné při zajištěné spolupráce existujících třídních knihoven. **Objektové** strukturální vzory popisují převážně způsob skladby pro novou funkčnost. Výhoda objektové skladby je ve schopnosti měnit skladbu za běhu. [34, s. 143]

Vzory chování se zaměřují na algoritmy, dělbu povinností mezi objekty a komunikaci mezi nimi. **Třídní** vzory chování používají dědičnost k rozdělení chování mezi třídy. **Objektové** vzory chování popisují spolupráci objektů na určitém úkolu. U těchto vzorů je důležité sledovat vazby, jak o sobě objekty vědí. [34, s. 218]

Tab. 2 Kategorizace návrhových vzorů (převzato z [34, s. 29])

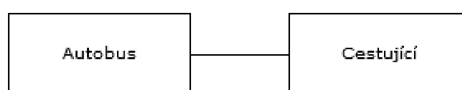
Oblast	**Tvořivé vzory**	**Strukturální**	**Chování**
Třída	Tovární metoda	Adaptér (třída)	Interpret
			Šablonová metoda
Objekt	Abstraktní továrna	Adaptér (objekt)	Iterátor
	Jedináček	Dekorátor	Návštěvník
	Prototyp	Fasáda	Obnovitel
	Stavitel	Most	Pozorovatel
		Muší váha	Prostředník
		Skladba	Příkaz
		Zástupci	Řetěz odpovědností
			Stav
			Strategie

3.2 Vazby mezi třídami UML

Dvě třídy mohou být naproti sobě v různém vztahu. Tento vztah se zapisuje jako určitý typ čáry mezi dvěma třídami. Například pokud jedna třída rozšiřuje druhou, jedná se o vazbu generalizace a tento vztah mezi třídami se zakresluje plnou čarou na konci se šipkou. [35][36]

3.2.1 Asociace

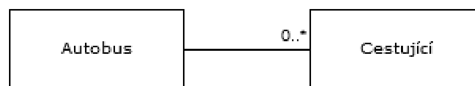
Jedná se o obecnou vazbu nebo logické spojení. Znázorňuje se tak, že jsou dvě třídy v určitém vztahu. Například Autobus a cestující. [35][36]



Obr. 7 Vztah asociace v UML

3.2.2 Mnohočetnost

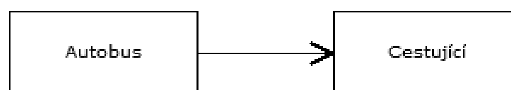
Vazba, pomocí které můžeme zaznamenat kardinalitu, nebo také četnost. Například u autobusu může nastoupit nula a více cestujících. [35][36]



Obr. 8 Vztah mnohočetnosti v UML

3.2.3 Směrová asociace

Vazba se značí přímkou se šipkou, která vychází směrem od kontejneru. Typicky se používá pokud, jedna třída obsahuje instanci jiné. Existuje i obousměrná asociace, v tom případě se kreslí šipky na obou stranách. [35][36]



Obr. 9 Vztah směrové asociace v UML

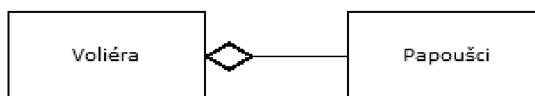
3.2.4 Reflexivní asociace

Reflexní asociace se použije v případě, kdy má jedna třída více rolí nebo zodpovědností. Příkladem může být žák, učitel a ředitel, kdy ředitel řídí učitele a tím vztah směřuje zpět do stejné třídy. [35][36]

Obr. 10 Vztah reflexe v UML

3.2.5 Agregace

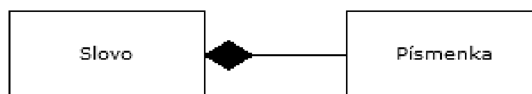
Je vazba kdy, vzniká třída na základě kolekcí nebo agregací tříd jiných. Vazba mezi nimi je slabá, jedna třída bez druhé dovede existovat. Například papoušek může existovat i bez voliéry, stejně tak voliéra může být bez papoušků. Značí se přímkou s kosočtvercem poblíž nadřazené třídy. [35][36]



Obr. 11 Vztah agregace v UML

3.2.6 Kompozice

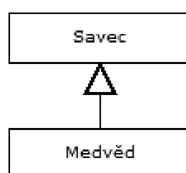
Jedná se o stejnou vazbu jako agregace s tím rozdílem, že má silnou vazbu. Jedna třída bez druhé nemůže existovat. Příkladem může být slovo a písmenka. [35][36]



Obr. 12 Vztah kompozice v UML

3.2.7 Generalizace/ Dědictví

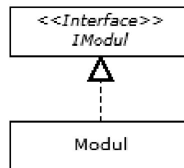
Tato vazba reflektuje vztah dědictví, nebo také rozšíření. Znázorňuje se plnou čárou se šipkou ve tvaru trojúhelníku. Příkladem může být savec a medvěd. O savci pak hovoříme jako o rodiči a medvědovi jako o potomkovi. [35][36]



Obr. 13 Vztah generalizace v UML

3.2.8 Realizace

Realizace se nejčastěji vyskytuje při implementaci interface. Říká se také, že tato vazba implementuje kontrakt, kdy se jedná o dohodu, které metody a atributy musí být implementovány. [35][36]



Obr. 14 Vztah realizace

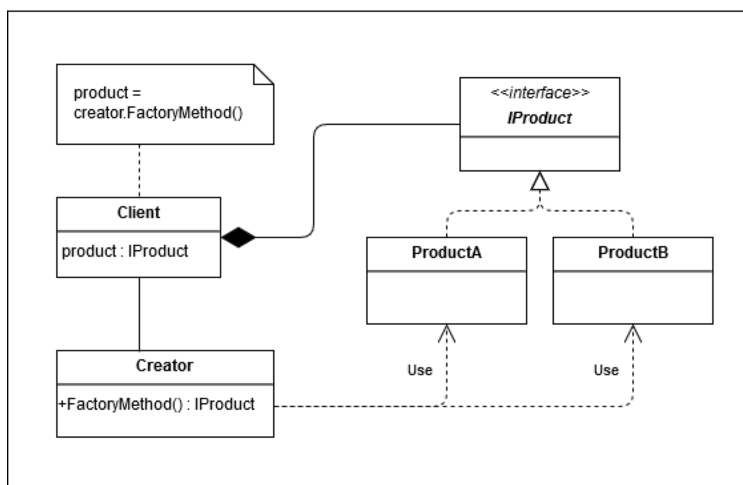
II. PRAKTICKÁ ČÁST

4 ANALÝZA NÁVRHOVÝCH VZORŮ

V této kapitole je shrnuto 23 návrhových vzorů s jejich popisem a případy, kdy se hodí konkrétní návrhový vzor použít. Byla provedena jejich obsahová analýza. Vysvětlení vazeb mezi jednotlivými třídami je znázorněné pomocí UML diagramů.

4.1 Tovární metoda (Factory Method)

Tovární metoda definuje rozhraní pro vytváření objektu. Návrh tohoto vzoru umožňuje přenechat rozhodnutí o vytvoření instance na podtřídu. Klient může mít více tvůrců pro rozdílné produkty. Rozdíl mezi konstruktorem a tovární metodou je ten, že konstruktor hned po zavolání vytvoří svou instanci, kdežto tovární metoda se může rozhodnout, zda vrátí novou instanci nebo už existující a zda vrátí instanci daného typu, nebo instanci některého ze svých potomků. Ve srovnání s abstraktní továrnou je tovární metoda, jak už název vypovídá jen metoda. Abstraktní továrna je objekt. Nevýhodou tohoto návrhového vzoru je, že může vyžadovat vytvoření nové třídy při nepatrné změně produktu (mnoho tříd). Vhodnější pro rozšíření stávajícího systému než abstraktní továrna. Přidáme ji formou metody potřebné třídě. [[31, s. 279-284][34, s. 116-124][37, s. 110-114]



Obr. 15 Návrhový vzor Tovární metoda (převzato z [37, s. 111])

Jiný název

Virtual Constructor (virtuální konstruktor)

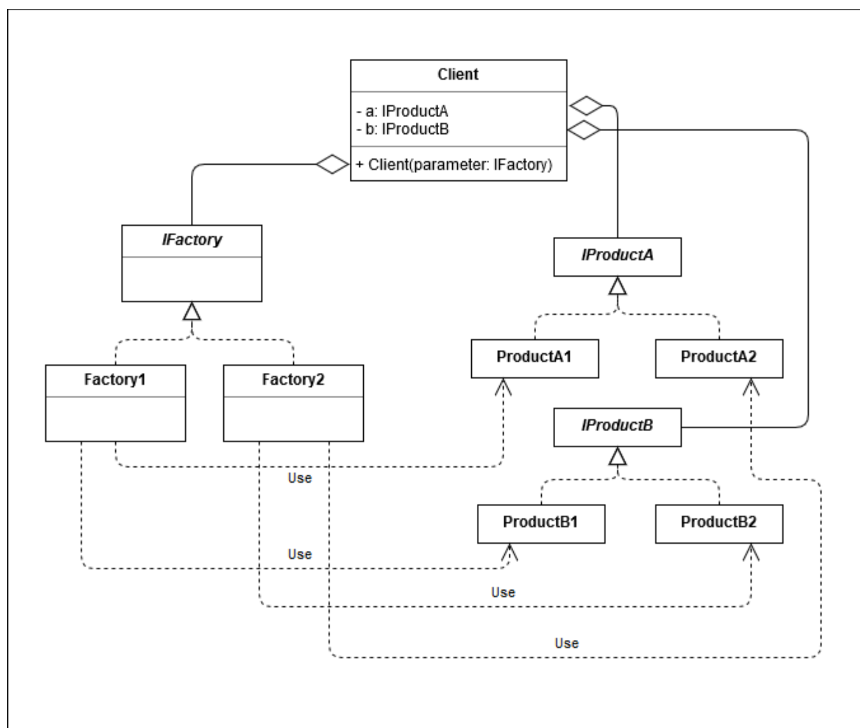
Použití

- Flexibilita
- Deleguje zodpovědnost za tvorbu instance na podtřídu

- Třída nemůže dopředu vědět, který objekt se má vytvořit
- Objekt může být rozšířen v podtřídách[31, s. 279-284][34, s. 116-124][37, s. 110-114]

4.2 Abstraktní továrna (Abstract Factory)

Poskytuje rozhraní pro inicializaci příbuzných nebo závislých objektů. Umožňuje zastínění jména třídy, odkud je objekt inicializován. Třída konkrétní továrny se vyskytuje v aplikaci jen jednou, v místě instance. Záměnou továrny lze použít různé produktové konfigurace. Zvyšuje konzistenci mezi produkty. Rozšiřování je obtížné. Změna se promítne v rozhraní i ve všech jeho podtřídách. Řešení tohoto problému je možné v kombinaci s návrhovým vzorem *Stavitel*. Instance vracené jednotlivými metodami mohou mít stejný nebo různý typ (na základě potomka). Abstraktní továrny jsou často *jedináčci*. [31, s. 329-339][34, s. 100-107][37, s. 122-128]



Obr. 16 Návrhový vzor Abstraktní továrna (převzato z [37, s. 124])

Jiný název

Kit (souprava)

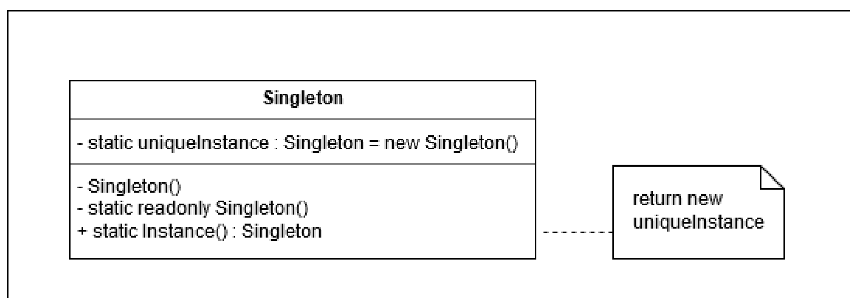
Použití

- Když je algoritmus tvoření produktů nezávislý na produktu samotném

- V případě kontroly konstrukčního procesu
- Pokud chceme vytvořit knihovnu, která má poskytnout jen rozhraní
- V případě konfigurace systému podle určité produktové řady [31, s. 329-339][34, s. 100-107][37, s. 122-128]

4.3 Jedináček (Singleton)

V mnoha případech potřebujeme, aby vznikla jediná instance dané třídy. *Abstraktní továrny* bývají často *Jedináčci*. Stejně tak různé *loadery* a *bootstrapy* mohou být *Jedináčci*. U návrhového vzoru *Jedináček* je dobré myslet při návrhu na multithreading, a při dotazování zda instance už existuje, jej zaopatřit *lock-objektem*. V tomto případě bychom měli být na pozoru, protože hodně záleží na verzi .NET, kterou používáme, případně jakou verzi Javy. Obecně se nedoporučuje použít *double-checklocking,*protože ve starších verzích Java nezajišťovala, že se konstruktor dokončí před referencí na instanci nového objektu. Mělo by pomoci deklarovat instanci, jako *volatile*, ale dal bych raději přednost bezpečnosti před optimalizací a techniku *double-checklocking* nepoužil. [31, s. 107-120][34, s. 134-140][37, s. 115-119]



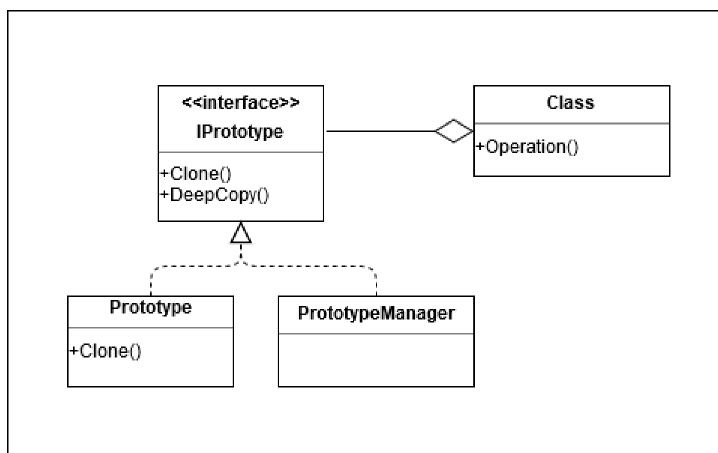
Obr. 17 Návrhový vzor Jedináček (převzato z [37, s. 115])

Použití

- V případě, že musí existovat jediná instance třídy
- Řízený přístup k instanci oproti globálním proměnným
- Vzor jedináček lze použít v kombinaci s tovární metodou a na tomto základě může vzniknout jediná instance různého typu, závislá na implementaci.[31, s. 107-120][34, s. 134-140][37, s. 115-119]

4.4 Prototyp (Prototype)

Návrhový vzor *Prototyp* vytváří nové instance klonováním prototypické instance. Výhodou je rychlost tvoření instancí, které mohou být velmi velké a vytvořené pomocí dynamicky načítaných tříd. Při klonování rozlišujeme mělké kopie a kopie hloubkové. Mělké kopie známe jako *shallow copy* kopírují hodnotové typy. U referenčních typů, za což můžeme považovat námi vytvořené objekty, dojde jen ke kopii reference. Takže výsledný klon ukazuje na ten samý objekt jako objekt originální. Při změně objektu u originálu se změna promítne i u klonu, což je ve většině případů nežádoucí. Hloubková kopie, známá pod názvem *Deep copy*, tento problém řeší. V jazyce C# se pro *shallow copy*používá metoda *MemberwiseClone*. Tuto metodu můžeme použít i u *Deep copy*, ale vytváření a kopii referenčních objektů zůstane v naší režii. [31, s. 285-305][34, s. 125-133][37, s. 101-109]



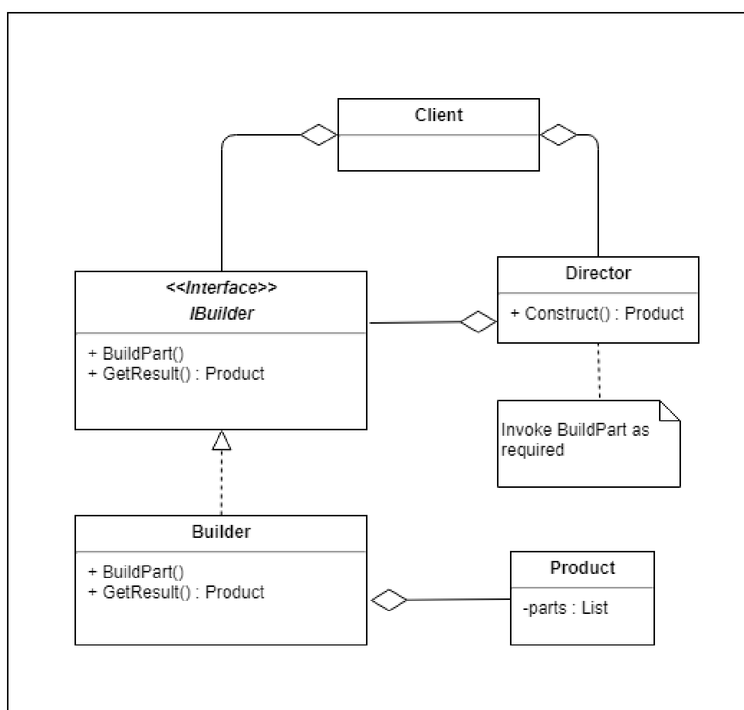
Obr. 18 Návrhový vzor Prototyp (převzato z [37, s. 101])

Použití

- Přidávání a mazaní objektů za běhu programu
- Pokud chceme klientu skrýt konkrétní třídu
- Pokud jsou třídy, které se mají vytvořit známé až za běhu programu pomocí dynamického načítání [31, s. 285-305][34, s. 125-133][37, s. 101-109]
- Adaptace na měnící se struktury dat za běhu programu
- Zvažme použití s návrhovým vzorem skladba pro archivování
- Zvažme použití výhod *Prototypu* místo *Tovární metody* [31][34][37]

4.5 Stavitel (Builder)

Návrhový vzor *Stavitel* odděluje specifikaci komplexního objektu od jeho aktuálního konstrukčního procesu. Stejný konstrukční proces může vytvářet různé reprezentace. Jeho použití je vhodné u různých tříd s podobným procesem konstrukce. Příkladem v jazyce C# může být *StringBuilder*. Tento vzor je založen na *Ředitelích (Director)* a *Stavitelých (Builder)*, kde může být velký počet stavitelů, kteří jsou voláni skrze rozhraní ředitelem. Často se používá v kombinaci s návrhovým vzorem *Skladba*. [31, s. 309-327][34, s. 108-115][37, s. 129-136]



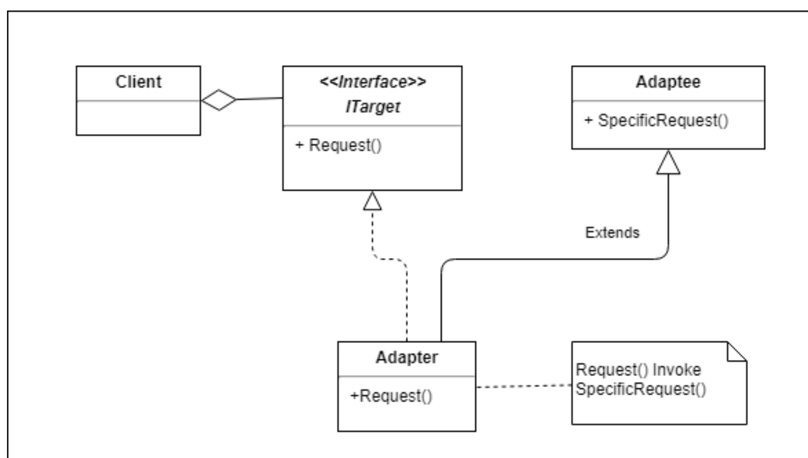
Obr. 19 Návrhový vzor Stavitel (převzato z [37, s. 130])

Použití

- Algoritmus vytváření produktů je nezávislý na produktech samotných
- Tam kde je třeba mít kontrolu nad tvorbou procesu
- Vytvářený objekt může mít různou reprezentaci
- *Stavitel* má oproti *Abstraktní továrně* výhodu, že tvorbu složitých objektů může realizovat postupně po krocích, *abstraktní továrna* vrací produkt v jednom kroku.[31, s. 309-327][34, s. 108-115][37, s. 129-136]

4.6 Adaptér (Adapter)

Převádí rozhraní třídy na rozhraní, které očekává klient. Adaptér použijeme v případě, že potřebujeme, aby třída měla jiné rozhraní, než které právě má. Mezi uživatele a nekompatibilní třídu vložíme třídu adaptér, která konvertuje rozhraní naší třídy na rozhraní požadované. Rozhraní starších systémů se často liší od rozhraní systémů nových. Přepisování starých programů bývá často velmi obtížné i nákladné. Řešením je použití adaptéru, který využije již napsaný starý program. [31, s. 261-269][34, s. 145-155][37, s. 74-92]



Obr. 20 Návrhový vzor Tovární metoda (převzato z [37, s. 76])

Jiný název

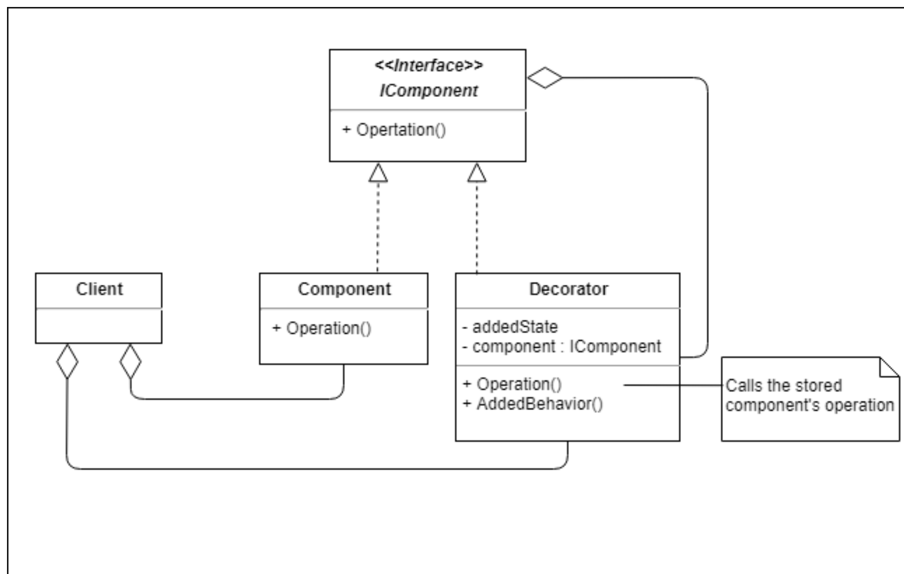
Wrapper - obal

Použití

- V případě, že máme existující třídu, u které se neshoduje rozhraní
- Chceme vytvořit třídu, která pracuje s neznámými nebo nepříbuznými třídami
- Chceme změnit názvy volajících metod od originálu
- *Adaptér* je možné implementovat čtyřmi způsoby
- Třídní adaptér - neviditelný pro klienta
- Objektový adaptér - rozšiřitelný na podtřídy adaptéru
- *Two-Way* adaptér - umožňuje různým klientům se různě dívat na objekt
- *Pluggable* adaptér - umožňuje připojovat a odpojovat různé adaptéry[31, s. 261-269][34, s. 145-155][37, s. 74-92]

4.7 Dekorátor (Decorator)

Dekorátor dynamicky připojuje k objektu další funkcionalitu, dekorovaný objekt neví o tom, že byl dekorován. Rozšíření funkcionality pomocí *dekorátoru*, se liší oproti klasické dědičnosti tím, že přidá funkcionalitu jen jedné instanci. Klasická dědičnost by rozšířila funkcionalitu všech instancí podtříd. Přidaná funkcionalita může být případně odebrána. [31, s. 343-360][34, s. 176-184][37, s. 9-21]



Obr. 21 Návrhový vzor Dekorátor (převzato z [37, s. 11])

Jiný název

Wrapper (obal)

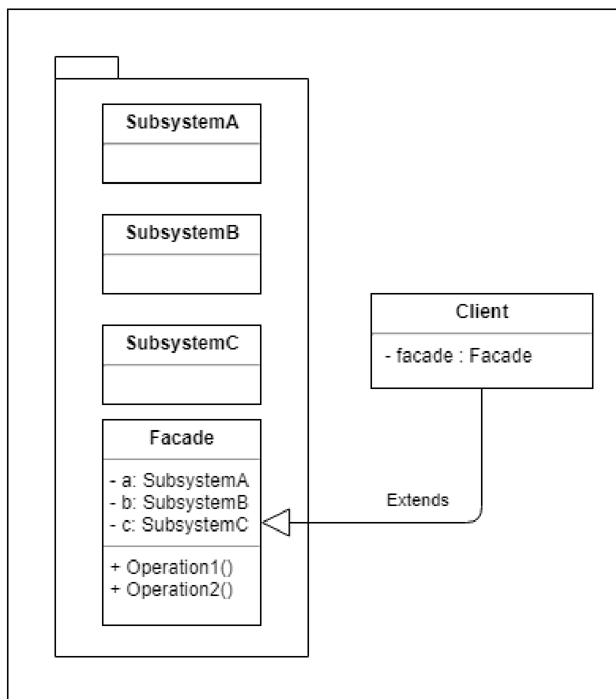
Použití

- Chceme dynamicky připojit funkcionalitu nějakému objektu, bez ovlivnění jiných
- Pokud není rozšíření pomocí podtříd praktické
- Pro každé rozšíření funkcionality je definován samostatný *dekorátor*[31, s. 343-360][34, s. 176-184][37, s. 9-21]

4.8 Fasáda (Facade)

Návrhový vzor *Fasáda* nám slouží k tomu, aby nám poskytl unifikované rozhraní nad mnoha subsystémy. Klient může operace provádět výběrem z různých podsystémů. Fasáda nám zpřehledňuje pohled na systém jako celek, v momentě, kdy začíná být příliš složitý. Je definováno jako rozhraní vyšší úrovně, které usnadní využití subsystému. Snižu-

je počet tříd, se kterými musel klient komunikovat. *Fasáda* může být definována jako *Jedináček* (jeden vstupní bod). [31, s. 255-259][34, s. 185-192][37, s. 93-98]



Obr. 22 Návrhový vzor Fasáda (převzato z [37, s. 94])

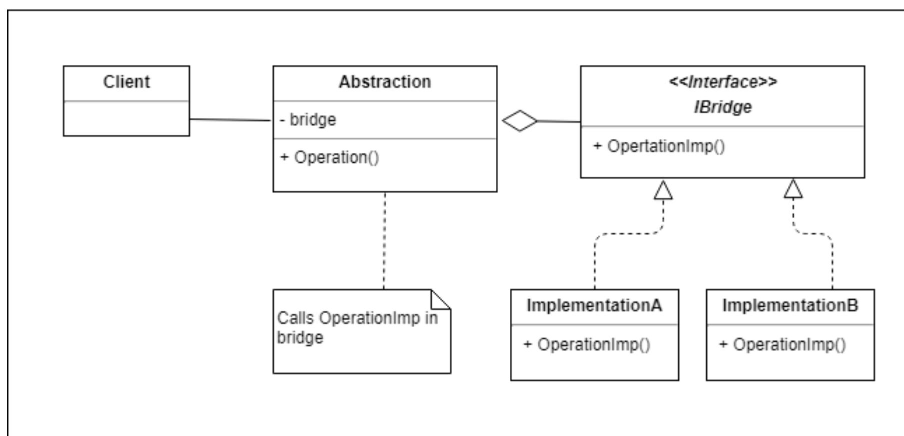
Použití

- Chceme zjednodušit rozhraní celého systému
- Potřebujeme vstupní bod nad systémem
- Pokud mezi klienty a třídami existuje mnoho závislostí
- Chceme uspořádat systém do vrstev
- Chceme poskytnout alternativní rozhraní (pro začátečníky)
- Abstrakce a implementace subsystému jsou pevně spojeny
- Zvažme před použitím výhody a nevýhody použití abstraktní továrny[31, s. 255-259][34, s. 185-192][37, s. 93-98]

4.9 Most (Bridge)

Odděluje abstrakci od implementace a umožňuje je měnit nezávisle. Užitečný například při uvolnění nového softwaru, který by měl nahradit starou verzi. Stará verze ale musí stále běžet pro klienty, kteří novou verzi z nějakého důvodu nepovolili. Kód klienta se nebude muset měnit, protože odpovídá dané abstrakci. Řeší situaci, kdy klient komunikuje se třídou, která je abstrakcí nabízené služby, implementace má na starosti jiná třída. Objekty

třídy (nemusí být nutně třída abstraktní nebo rozhraní), které zprostředkovávají služby, označujeme jako abstrakci. Třída, která tyto služby doopravdy poskytuje, označujeme jako implementaci. [31, s. 399-413][34, s. 156-165][37, s. 36-39]



Obr. 23 Návrhový vzor Most (převzato z [37, s. 38])

Jiný název

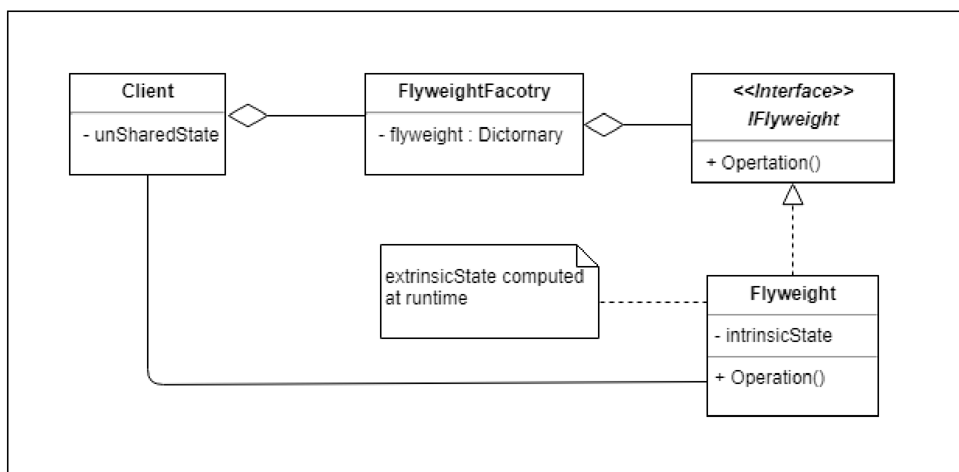
Handle/Body (držák/tělo)

Použití

- Identifikujeme, že existují operace, které nemusí být vždy prováděny stejným způsobem
- Chceme úplně skrýt implementaci od klienta
- Chceme kombinovat různé části systému za běhu
- Chceme změnit implementaci bez rekompilování abstrakce
- Všechny abstrakce i implementace by měli být rozšiřitelné pomocí podtříd [31, s. 399-413][34, s. 156-165][37, s. 36-39]

4.10 Muší váha (Fly-weight)

Návrhový vzor *Muší váha* je efektivní cesta jak sdílet informace v malých/zrnitých objektech. Pomáhá tak snížit požadavky na úložný prostor, v případě duplikace hodnot. Návrhový vzor rozlišuje mezi vnějším a vnitřním stavem objektu. Vnitřní stav lze sdílet a minimalizovat požadavky na úložný prostor. Interní charakteristiky jsou pro všechny virtuální instance z dané skupiny společné. Vnější stav lze vypočítat za *letu*. [31, s. 171-186][34, s. 194-205][37, s. 61-71]



Obr. 24 Návrhový vzor Muší váha (převzato z [37, s. 62])

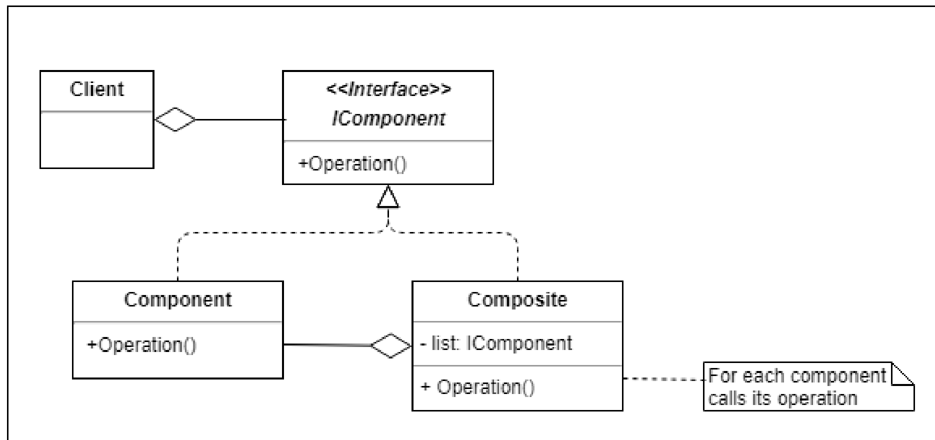
Použití

- Je-li účelné zastoupit velký počet virtuálních instanci jednou reálnou, při odstranění vnějšího stavu
- V aplikaci se používá velký počet objektů
- Velké náklady na úložný prostor
- Většinu objektového stavu lze určit vnějším
- Pokud chceme nějaké stavy objektu počítat za běhu[31, s. 171-186][34, s. 194-205][37, s. 61-71]

4.11 Skladba (Composite)

Skládá objekty do stromových struktur, pro vyjádření co je část a co celek. Umožňuje klientům zacházet stejně se skupinami objektů i s jednotlivými objekty. Typické operace nad komponentou jsou přidání, odebrání, zobrazení, hledání a skupinové operace. Návrhový vzor se potýká s pojmy komponenta (Component) a skladba (Composit). Oba typy im-

plementují stejnou metodu rozhraní a to nám zajistí jednotnou práci jak se skladbou, tak s konkrétní komponentou. [31, s. 271-276][34, s. 166-175][37, s. 49-60]



Obr. 25 Návrhový vzor Skladba (převzato z [37, s. 51])

Jiný název

Strom

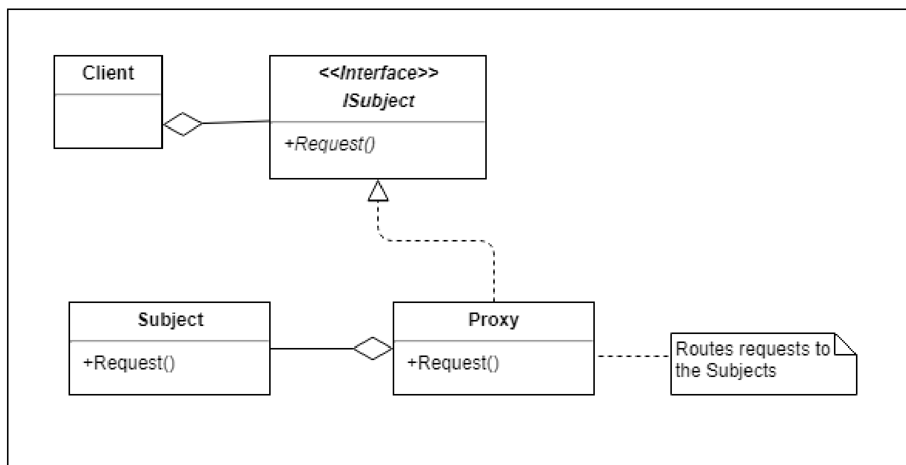
Použití

- Pokud chceme, aby mohli klienti používat všechny objekty ve složené struktuře stejným způsobem
- Chceme vyjádřit několik objektových hierarchií
- Měli bychom zvážit, zda není výhodné použít *dekorátor* pro operace přidat, odebrat a hledání
- Měli bychom zvážit použití *Muší váhy*, pro sdílené komponenty, kdy nepotřebujeme vědět, kde se momentálně nacházíme a pokud všechny operace začínají na kořeni stromu
- Měli bychom zvážit použití *Návštěvníka*, ke zjištění kde právě jsem[31, s. 271-276][34, s. 166-175][37, s. 49-60]

4.12 Zástupce (Proxy)

Návrhový vzor zástupce má za úkol odstínit objekt od jeho uživatelů. Sám řídí přístup k danému objektu. Často se jedná o malý veřejný objekt, který stojí v cestě komplexnímu soukromému objektu. Je použitelný tam, kde je třeba důmyslnějšího odkazu, než pouhý ukazatel. *Vzdálený zástupce* zastupuje objekt, umístněný někde jinde a poskytuje nám s ním komunikaci. *Virtuální zástupce* vytváří nákladné objekty, až když je třeba, do té

doby můžeme využívat jiný náhradní. *Ochranný zástupce* je výhodné použít, tam, kde určitý objekt potřebuje k přístupu různá práva. *Chytrý odkaz* je náhradou za ukazatel, který nám umožňuje poskytnout nějakou přídavnou funkcionalitu, když je k objektu přistupováno. [31, s. 189-194][34, s. 206-215][37, s. 22-35]



Obr. 26 Návrhový vzor Zástupce (převzato z [37, s. 23])

Jiný název

Surrogate (náhradník)

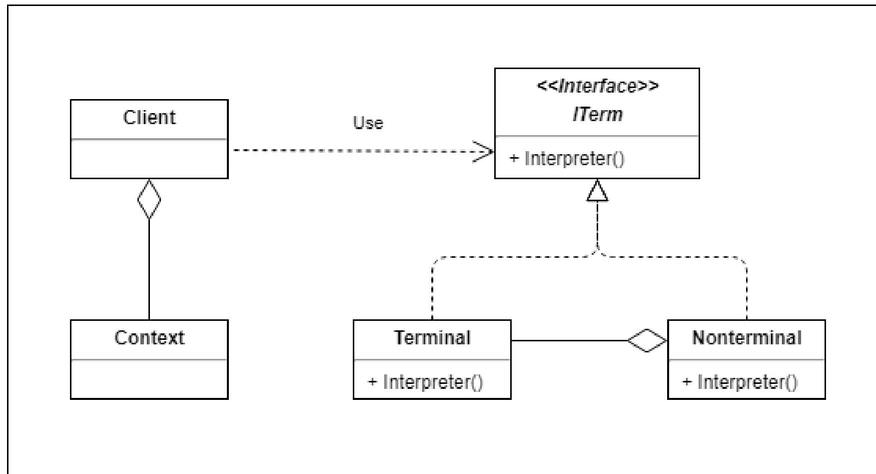
Použití

- Pro zlepšení funkčnosti
- Ochranný zástupce zamezuje přímé komunikaci s objektem
- Slouží k lepšímu zapouzdření implementace
- Když potřebujeme provést nějaké akce, když je k objektu přistupováno
- V případě, že chceme vytvořit objekt, jen když jsou jeho operace požadovány
- V případě, že máme objekt, který odkazuje na objekt vzdálený[31, s. 189-194][34, s. 206-215][37, s. 22-35]

4.13 Interpret

Jedná se o třídní návrhový vzor. Vzor interpret podporuje interpretaci pokynů napsaných v jazyce nebo notaci definované pro konkrétní účel. Někdy bývá vhodnější navrhnout si vlastní programovací jazyk, k popisu daného problému. Interpret pak interpretuje tento popis a tím problém vyřeší. Napsat si vlastní programovací jazyk není triviální disciplína. Programátor musí být obeznámen a chápat pojmy jako gramatiky, lexikální, syntaktická a sémantická analýza. To vše totiž předchází interpretu. Vstup do Interpreta nemusí být nut-

ně textový. Spíše přebírá binární stromy - abstraktní syntaktický strom. Vzor Interpret nepopisuje tvorbu abstraktních syntaktických stromů. [31, s. 475-507][34, s. 237-248][37, s. 233-241]



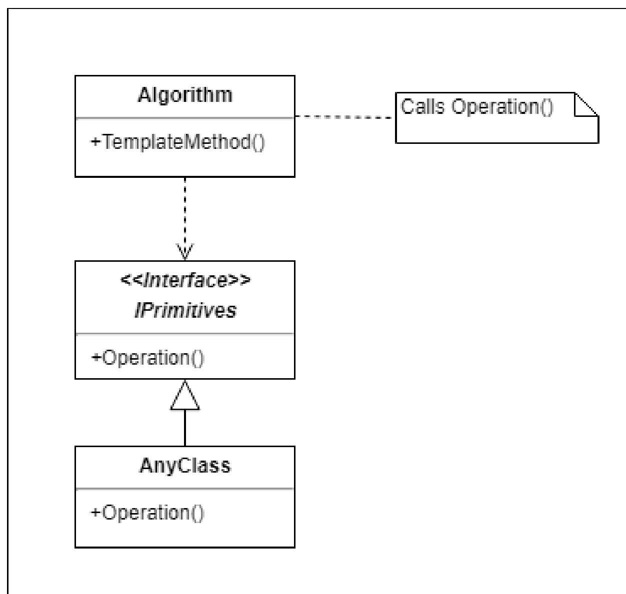
Obr. 27 Návrhový vzor Interpret (převzato z [37, s. 235])

Použití

- Vzor interpret definuje minimálně jednu třídu pro každé gramatické pravidlo
- Nejúčinnější interpreti jsou u převodu úvodního stavu do jiného
- Použijeme v případě, kdy interpret není příliš rozsáhlý
- Máme omezené nástroje pro parsování
- Kde je jedna z možností použití pro specifikaci je XML
- Účinnost není kritická
- Lze použít na substituce proměnných [[31, s. 475-507][34, s. 237-248][37, s. 233-241]

4.14 Šablonová metoda (Template Method)

Šablonová metoda umožňuje algoritmům přenechat provádění určitých částí na podtřídu. Podtřídám umožňuje předdefinovat určité kroky algoritmu, aniž by se změnila jeho struktura. Šablonová metoda je velmi užitečná při použití s návrhovým vzorem strategie. [31, s. 239-251][34, s. 306-310][37, s. 158-162]



Obr. 28Návrhový vzor Šablonová metoda (převzato z [37, s. 158])

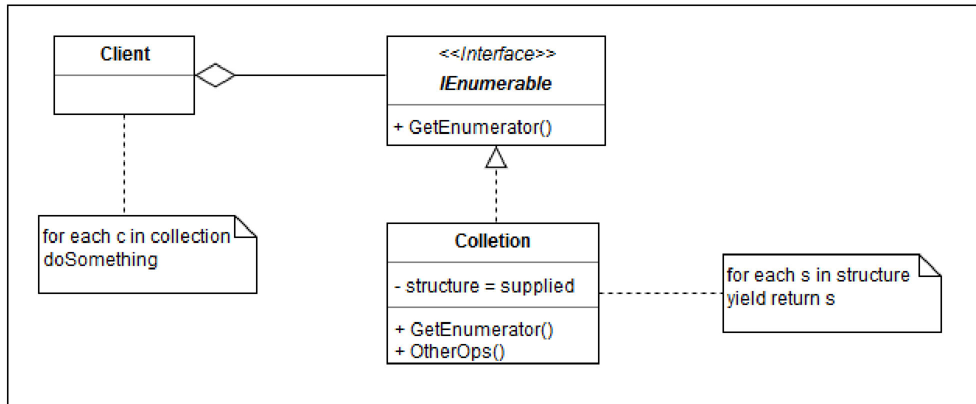
Použití

- Jedna z nejčastěji používaných
- Implementuje v rodičovské metodě algoritmus, který používají potomci
- Často implementovaná v abstraktní třídě
- Počet abstraktních metod, které implementují potomci, by měl být co nejmenší
- Šablonovou metodou nemůže být konstruktor
- Výsledné chování se může lišit podle potomka[31, s. 239-251][34, s. 306-310][37, s. 158-162]

4.15 Iterátor

Iterátor poskytuje sekvenční přístup k prvkům kolekce (kontejneru). Vnitřní implementace zůstane před klientem skryta. *Iterátor* se používá hlavně při procházení a hledání v seznamech. Typicky by měl *Iterátor* zajistit, aby měl metodu, která vrátí následující prvek, aby byl schopen říct, zda už dosáhl konce, přidání, změnu, odebrání prvku. Může

umožňovat změnu směru procházení nebo jinou práci s prvky. Bývá dobrým zvykem implementovat prázdný objekt, aby se nám při dosažení konce nevracel *null*. [31, s. 203-220][34, s. 249-261][37, s. 188-199]



Obr. 29 Návrhový vzor Iterátor (převzato z [37, s. 191])

Jiný název

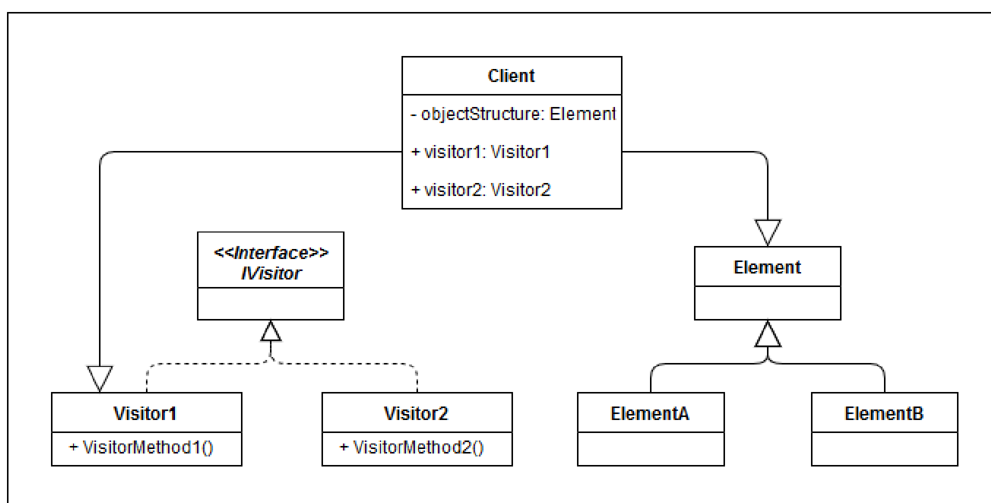
Ukazatel (Cursor)

Použití

- Když je více možností jak procházet
- Máme různé kolekce pro procházení stejným způsobem
- Pokud bychom chtěli řadit a filtrovat různým způsobem
- bývá definován jako soukromá vnitřní třída
- *Iterátor* pracuje jako specializovaný zástupce
- Lze použít jako nekonečný *Iterátor*[31, s. 203-220][34, s. 249-261][37, s. 188-199]

4.16 Návštěvník (Visitor)

Návrhový vzor *Návštěvník* umožňuje rozšíření skupiny tříd o nové operace. Ve třídách se nemusí měnit jejich kód. V případě, že máme skupinu tříd, kdy nepočítáme s tím, že ji budeme ubírat nebo přidávat členy, ale máme na paměti, že se to v průběhu života aplikace může stát. Rozšiřované třídy ale musí být předem připravené a přijímání návštěvníka. Vzor *Návštěvník* často nutí poskytovat veřejné operace pro přístup k vnitřnímu stavu elementu, to může vést k porušení zapouzdření. [31, s. 453-466][34, s. 311-323][37, s. 220-232]



Obr. 30 Návrhový vzor Návštěvník (převzato z [37, s. 220])

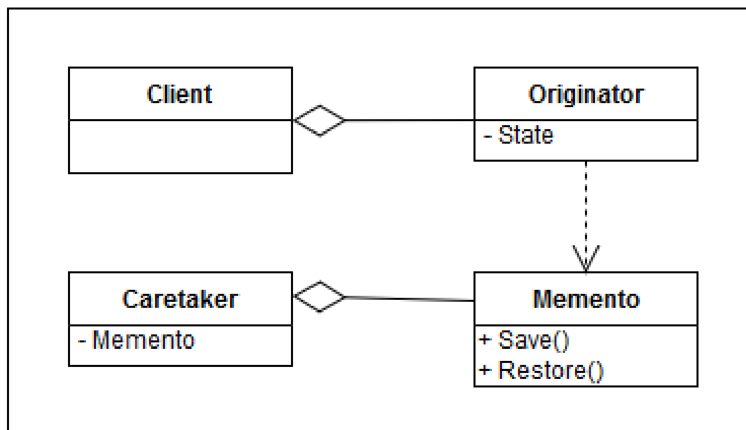
Použití

- Když se struktura tříd mění málo často, ale často potřebujeme přidat novou metodu
- Nad objekty struktury se provádí rozdílné operace
- Pro každou metodu je třeba definovat novou třídu *Návštěvníka*
- V kombinaci s návrhovým vzorem *Adaptér*, lze definovat rozhraní, které bude zjednodušovat definici návštěvních metod
- V těle metody je jediný příkaz, předává řízení metodě *Návštěvníka*[31, s. 453-466][34, s. 311-323][37, s. 220-232]

4.17 Obnovitel (Memento)

Návrhový vzor *Obnovitel*, řeší problém jednoduchého uchování stavu objektu, aniž bychom odkryli implementaci. Tento stav je možné obnovit. Typické pro použití funkčnosti *zpět (undo)*, *vpřed (redo)*. Základní problém je v předání svého stavu tak, aby nebyla

odhalena implementace, která má být maximálně utajena. V tomto případě je *Originator* objekt, jehož stav má být uložen. *Memento* provádí ukládání a *CareTaker* uchovává přehled o různých uložených stavech. [31, s. 467-474][34, s. 271-279][37, s. 242-251]



Obr. 31 Návrhový vzor Obnovitel (převzato z [37, s. 243])

Jiný název

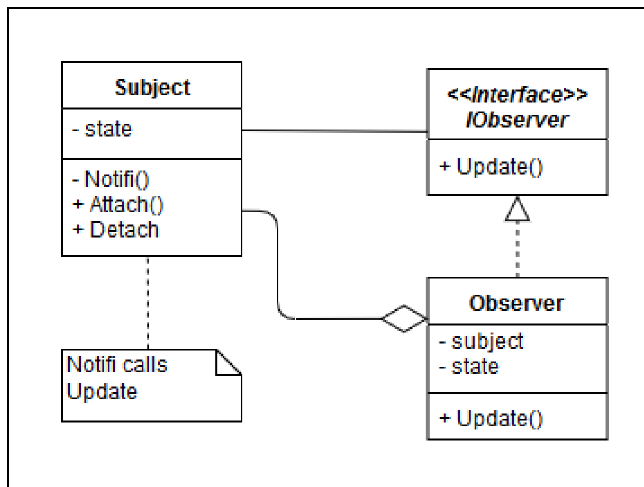
Pamětník, Token

Použití

- Přímé rozhraní k získání stavu by mohlo narušit zapouzdření objektu
- V případě, že chceme zachytit aktuální stav objektu, pro pozdější použití
- Možnost implementace seznamu stavů (historie)
- Obnovitel bývá často implementován jako vnitřní třída (internal)
- Je možné uchovávat inkrementální změny[31, s. 467-474][34, s. 271-279][37, s. 242-251]

4.18 Pozorovatel (Observer)

Návrhový vzor *Pozorovatel* definuje vztah mezi objekty tím způsobem, že pokud dojde ke změně stavu pozorovaného objektu, pozorující objekty jsou na tuto změnu upozorněni. Běžně se využívá u *blogerů*, kdy můžete přijít na stránky, které vás zajímají a přihlásit se k odběru (*subscribe*), pokud daný *bloger* na svůj web něco přidá. V jazyce *Java* tomuto návrhovému vzoru říkají *listener* (posluchač). Při tvorbě uživatelského rozhraní je možné na tlačítko přihlásit tento *posluchač*, aby se program nemusel dokola vyptávat, zda nebylo tlačítko stisknuto. [31, s. 375-385][34, s. 280-289][37, s. 210-217]



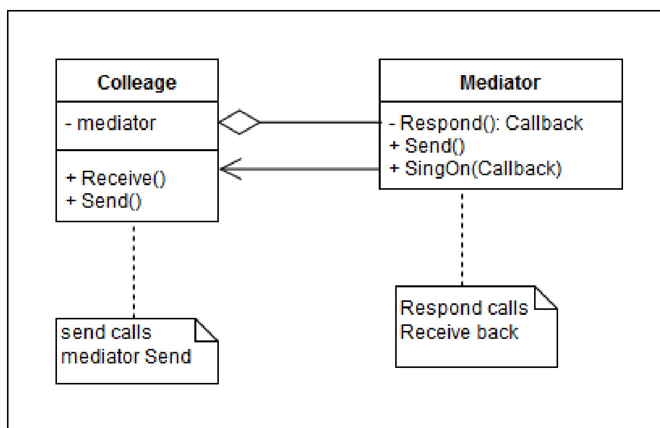
Obr. 32 Návrhový vzor Pozorovatel (převzato z [37, s. 211])

Použití

- V jazyce Java je pro aplikaci tohoto vzoru připravena třída *Observable*
- Vzor zabezpečuje včasnou notifikaci o události, aniž by musel obtěžovat periodickými dotazy iniciátora
- V případě, že změna jednoho objektu vyžaduje změnu druhého
- V případě, že chceme upozornit objekty, o kterých nic nevíme[31, s. 375-385][34, s. 280-289][37, s. 210-217]

4.19 Prostředník (Mediator)

Prostředník umožňuje objektům navzájem mezi sebou komunikovat bez nutnosti, aby se navzájem znaly. Umožňuje měnit chování objektů nezávisle na ostatních. Ruší mezi nimi přímou vazbu, tím že vstoupí mezi ně. *Prostředník* zasílá zprávy všem přihlášeným Kolegům pomocí delegovaného respondenta. *Kolegové*, se registrují k *Prostředníkovi* poskytnutím přijímací metody. Objekt, kterému se má změnit definice a komunikuje s jiným napřímo, zapříčiní to, že se bude muset změnit i druhý objekt. V případě, že by těch objektů bylo mnoho a komunikovaly každý s každým, změna by byla příliš nákladná. Proto tuto situaci řeší návrhový vzor *Prostředník*, který zpřetrhá přímé vazby. [31, s. 387-396][34, s. 261-271][37, s. 200-210]



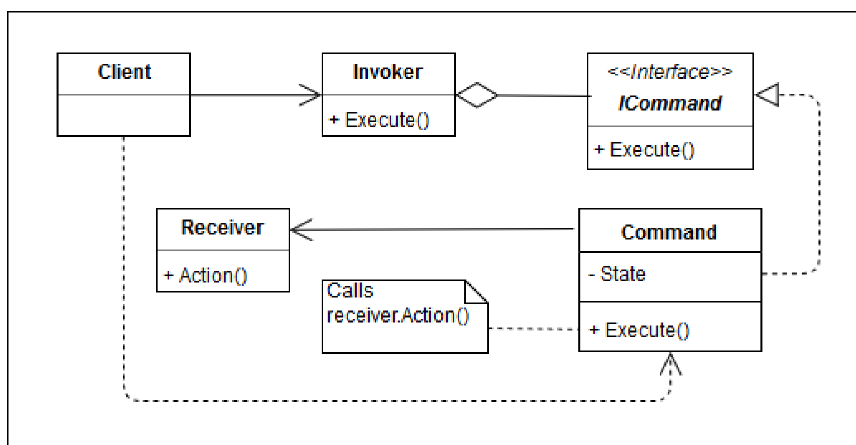
Obr. 33 Návrhový vzor Prostředník (převzato z [37, s. 201])

Použití

- V mnoha systémech reflektuje použití protokolu *send/recieve* (odešli/příjmy)
- V případech kdy objekty komunikují složitým způsobem
- V případě kdy chceme měnit komunikaci objektů beze změn na mnoha místech
- Posílání zpráv je definováno jako metoda *Prostředníka*, součásti zprávy se posílají skrze parametry
- Součástí zprávy bývá odkaz na příjemce, případně odesílatele
- *Prostředník* bývá často použit s návrhovým vzorem *Pozorovatel*, kdy pozorovaným objektem je náš *Prostředník* a zpráva nějakého objektu je událost, kterou je třeba pozorovatelům (komunikujícím objektům), kteří jsou k odběru přihlášeni předat.[31, s. 387-396][34, s. 261-271][37, s. 200-210]

4.20 Příkaz (Command)

Návrhový vzor *Příkaz*, zabaluje metodu do objektu, můžeme ji pak používat jako objekt a dle potřeby měnit. Návrhový vzor je univerzální a dovoluje zasílat požadavky různým příjemcům, funkcionalitu vpřed a zpět, řazení do front, logování a zamítnutí operací. Klient vytváří a spouští příkazy. Interface *ICommand* specifikuje spouštěcí operaci. Třída *Command* implementuje spouštěcí operaci za pomocí invokace *Reciever*. *Invoker* požádá objekt *Commanad* o provedení operace *Action*. *Reciever* je třída, která může provést požadovanou akci. *Action* je operace, kterou je nutno provést. [31, s. 195-202][34, s. 228-236][37, s. 175-185]



Obr. 34 Návrhový vzor Příkaz (převzato z [37, s. 177])

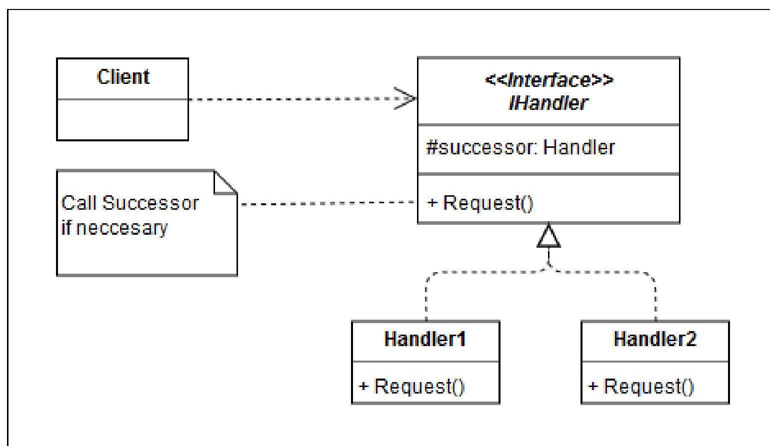
Použití

- Můžeme přidat nové operace bez narušení existujících
- Pokud chceme spouštět a dávat do fronty příkazy v různém čase
- Pro podporu příkazů funkcionalitu zpět/vpřed
- Pro podporu logování změn skrz příkazy
- V případě, že mohou různí příjemci reagovat na příkazy různě
- Chceme vysokoúrovňové operace implementovat pomocí primitivních, běžné v systémech podporující transakce.[31, s. 195-202][34, s. 228-236][37, s. 175-185]

4.21 Řetězec odpovědnosti (Chain of responsibility)

Řetězec odpovědností je návrhový vzor, který nám pomůže v situaci, kdy chceme, aby na požadavek zareagoval někdo jiný než objekt, kterému byl původně určen. Často je reprezentován pomocí stromových struktur, kdy uzel, který nebude vědět jak na požadavek

reagovat, předá tuto žádost svému rodičovskému objektu. Toto chování by se dalo nazvat zřetězení, a proto o návrhovém vzoru mluvíme jako o *řetězci odpovědností*. Příkladem může být nápověda, pokud prvek nemá implementovanou nápovědu, zobrazí se nápověda nadřazeného okna. *Successor* je odkaz na následujícího zpracovatele (*Handler* objekt). [31, s. 361-373][34, s. 219-227][37, s. 164-175]



Obr. 35 Návrhový vzor Řetězec odpovědností (převzato z [37, s. 165])

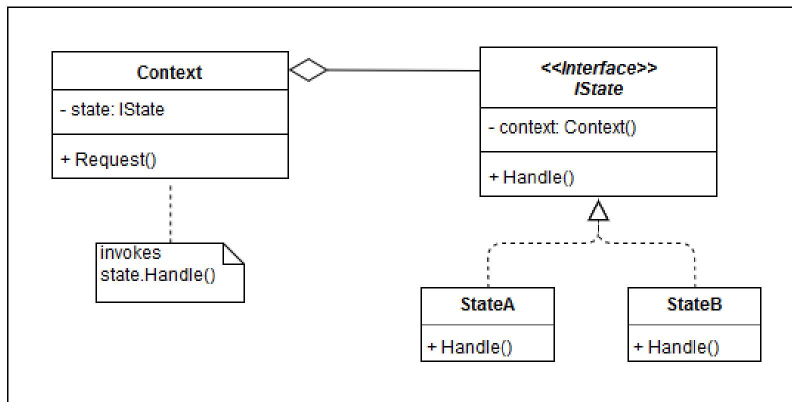
Použití

- V případě kdy pro požadavek potřebujeme více než jednoho zpracovatele (handler)
- V případě, že máme důvod, proč by měl zpracovatel předat požadavek dál
- Chceme zachovat flexibilitu přiřazování požadavků
- Chceme vystavit žádost jednomu z mnoha objektů, aniž bychom určili příjemce
- V případě, že objekty, které mohou žádost zpracovat, mají být určeny dynamicky[31, s. 361-373][34, s. 219-227][37, s. 164-175]

4.22 Stav (State)

Na návrhový vzor *Stav* se lze dívat jako na dynamickou verzi návrhového vzoru Strategie. Změna vnitřního stavu objektu může změnit chování přepnutím na jinou kolekci operací. Změna vnitřního stavu se řeší výměnou objektu reprezentujícího stav. Třída *Context* obsahuje instanci stavu. Třídy *StateA* a *StateB* implementují chování asociované s instancí třídy *Context*. V případě mnohostavové třídy, je vhodné tuto třídu dekomponovat na třídu, kdy její instance představuje původní objekt (jedná se o více-stavovou třídu) a její

jednotlivé stavy implementovat také jako samostatné třídy, které řeší chování pouze chování svého (jednoho) stavu. Definice je pak přehlednější. Přidání dalšího stavu se řeší jako přidání další třídy. [31, s. 221-237][34, s. 290-297][37, s. 148-157]



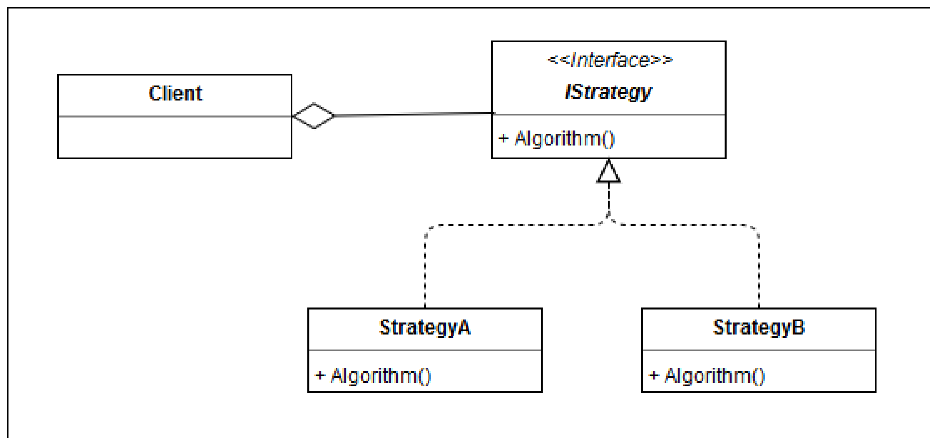
Obr. 36 Návrhový vzor Stav (převzato z [37, s. 150])

Použití

- Pokud se bude na základě kontextu měnit chování za běhu programu
- Pokud existuje mnoho objektů v různých stavech často reprezentovaných výčtovými typy
- Podle potřeby lze předávat více-stavový objekt metodám jednostavových objektů
- Lze implementovat metody vracející odkaz na jiný jednostavový objekt
- V rozhraní se implementují metody, odpovídající zprávám pro reakci více-stavového objektu[31, s. 221-237][34, s. 290-297][37, s. 148-157]

4.23 Strategie (Strategy)

Návrhový vzor *Strategie* umožňuje dynamickou zaměnitelnost algoritmů, nezávislou na klientech. Tento návrhový vzor umožňuje separovat algoritmus z hostitelské třídy, do separátních tříd. Pokud existuje hostitelská třída, která obsahuje různé prováděcí algoritmy, může být výsledkem jedna velká podmínka. V tomto případě je dobré uplatnit návrhový vzor *Strategie*. Tento návrhový vzor se podobá vzoru *Most*, s tím rozdílem, že počítá se změnou za doby běhu programu a soustřeďuje se na chování aplikace, ne na architekturu. [31, s. 415-424][34, s. 298-305][37, s. 139-147]



Obr. 37 Návrhový vzor Zástupce (převzato z [37, s. 141])

Použití

- V případě existence mnoha podobných tříd, které se liší jen v chování
- V případě potřeby variant algoritmu
- Návrhový vzor *Strategie*, pracuje na požadavek klienta oproti vzoru *Stav*, kde je přepínání interní záležitost.
- V případě, že algoritmus používá data, ke kterým by neměl mít klient přístup[31, s. 415-424][34, s. 298-305][37, s. 139-147]

4.24 Shrnutí

Z popsaných návrhových vzorů se pro implementaci naváděcího systému bude dát využít návrhového vzoru *Řetězec odpovědností*, ať už z důvodu specifikace chyb nebo pro komunikaci mezi komponentami. Návrhový vzor *Jedináček* se dá uplatnit všude, kde bude potřeba maximálně jedné instanci třídy. V jazyce C# se hodně využívá třída *StringBuilder*, k postupnému skládání textových řetězců, je implementovaná podle vzoru Stavitel. Pro načítání hardware se dá použít *Tovární metoda* nebo *Abstraktní továrna*. Pomocí *tovární metody* lze volat konkrétní konstruktor v závislosti na splněných podmínkách. Pro řešení naváděcího systému by mohl být použit také návrhový vzor *Příkaz*, jeho uplatnění by se dalo využít při komunikaci s fyzickým zařízením na bitové úrovni.

5 VYTVOŘENÍ NAVÁDĚCÍHO SYSTÉMU

Použití návrhových vzorů bude demonstrováno na naváděcím střelném systému. Systém bude navržen jako objektový návrh spolupracujících objektů a tříd. Systém bude umožňovat přepínání uživatelského rozhraní. Systém bude umět dynamicky načítat hardwarové komponenty. Systém bude navržen jako otevřený, při dodržení rozhraní bude možné systém rozšířit o chování a komponenty.

5.1 Základní rozdělení systému na subsystém

Naváděcí systém bychom mohli dekomponovat na jednotlivé funkční části. Pro ovládání bude třeba vytvořit uživatelské rozhraní. Logiku pro přenos zpráv bude realizovat jádro programu. Konkrétní hardwarové komponenty by neměli mít přímé vazby na uživatelské rozhraní. Ke komunikaci bude docházet skrze jádro a to nám zajistí lepší udržovatelnost aplikace. Naváděcí systém bude mít předdefinované vzory chování, které bude možné rozšiřovat.

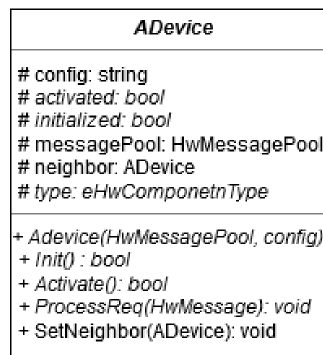
Výsledné subsystémy:

1. Jádro
2. Uživatelské rozhraní
3. Hardware
4. Logika

5.2 Hardware

Při načítání knihoven pro hardware je nutné předdefinovat obecné zařízení. Toto zařízení reprezentuje abstraktní třída *ADevice,* která obsahuje metody a atributy, které jsou společné pro všechna námi načítaná zařízení, a rovněž stanovuje předpis, co musí být implementováno. Třída *ADevice* nese informaci o svém typu, zda bylo zařízení správně inicializováno, umožňuje zařízení aktivovat a deaktivovat, pomocí atributu *config* je možné zařízení konfigurovat. Pro komunikaci už s konkrétním zařízením se používá metoda ProcessReq(). Zasílání zpráv od konkrétního hardware směrem k jádru je realizováno skrze atribut *HwMessagePool*, který je nastaven při volání přetíženého konstruktoru. Atribut *neighbor* nám pak slouží pro zřetězení konkrétních jednotlivých zařízení, za účelem komunikace. Zprávy jádra zpracovává metoda *ProcessReq()*. Jádro zasílá zprávu prvnímu zařízení. Pokud první zařízení identifikuje, že zpráva není pro něj, zavolá metodu *Pro-*

cessReq() svého následníka – *neighbor.ProcessReq()*. Pro zřetězení je užit návrhový vzor **Řetězec odpovědnosti**.

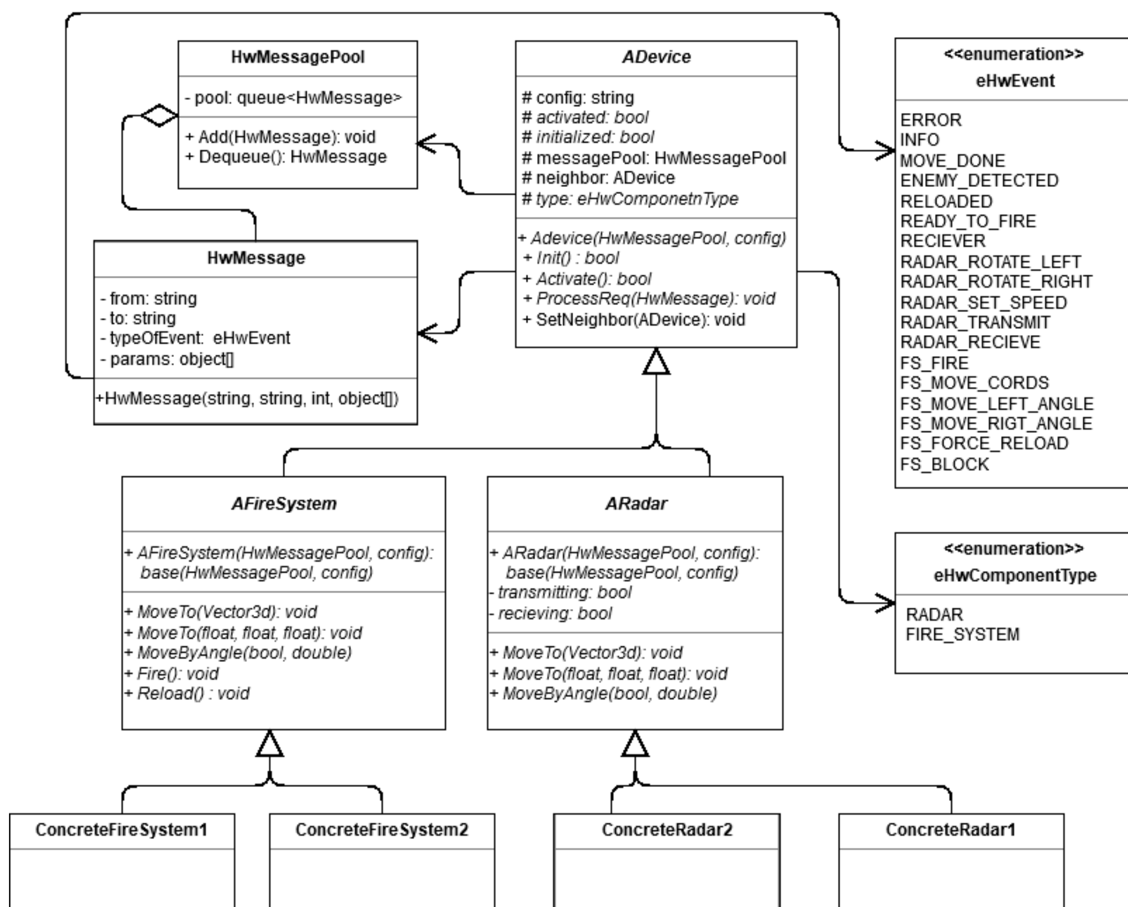


Obr. 38 Popis třídy pro obecné zařízení

Jakmile je obecné zařízení hotové, přichází na řadu konkretizace zařízení. Pro demonstraci jsem zvolil u naváděcího systému dvě základní zařízení. Jedná se o radar a střelný systém. Lze definovat i jiná zařízení, například *ADron, AReciever, ATransmitter*. Pro popis obecného radaru je vytvořena abstraktní třída *ARadar*, která dědí od *ADevice*. V této třídě je definováno chování, společné pro všechny radary, které by mohly být v budoucnu implementovány.

Demonstrativně je třída *ARadar* doplněna o metody pro pohyb radaru a obsahuje atributy pro povolení a zakázání vysílání a přijímání signálu. *AFireSystem* je také abstraktní třída, která popisuje společné chování střelný systém. V tomto případě se předpokládá, že střelný systém se bude umět hýbat, střílet a nabíjet.

Třídy *ConcreteFireSystem* a *ConcreteRadar* musí implementovat předdefinované chování abstraktních tříd. Při dodržení tohoto předpisu mohou vznikat nové zařízení, kdy s nimi bude jádro umět komunikovat. Pro rozšíření funkcionality na úrovni HW by bylo nutné rozšířit jádro a podporované zprávy. Například pokud by byl požadavek na zapojení nového radaru, který se umí pohybovat nahoru a dolu, muselo by se jádro rozšířit o volání nové zprávy a to by mělo rovněž dopad na obecnou abstraktní třídu *ARadar*, která by se musela také rozšířit o novou metodu. Následně by to ovlivnilo existující konkrétní radary, které by na novou abstraktní metodu musely také reagovat, byť jen příkazem *return*.



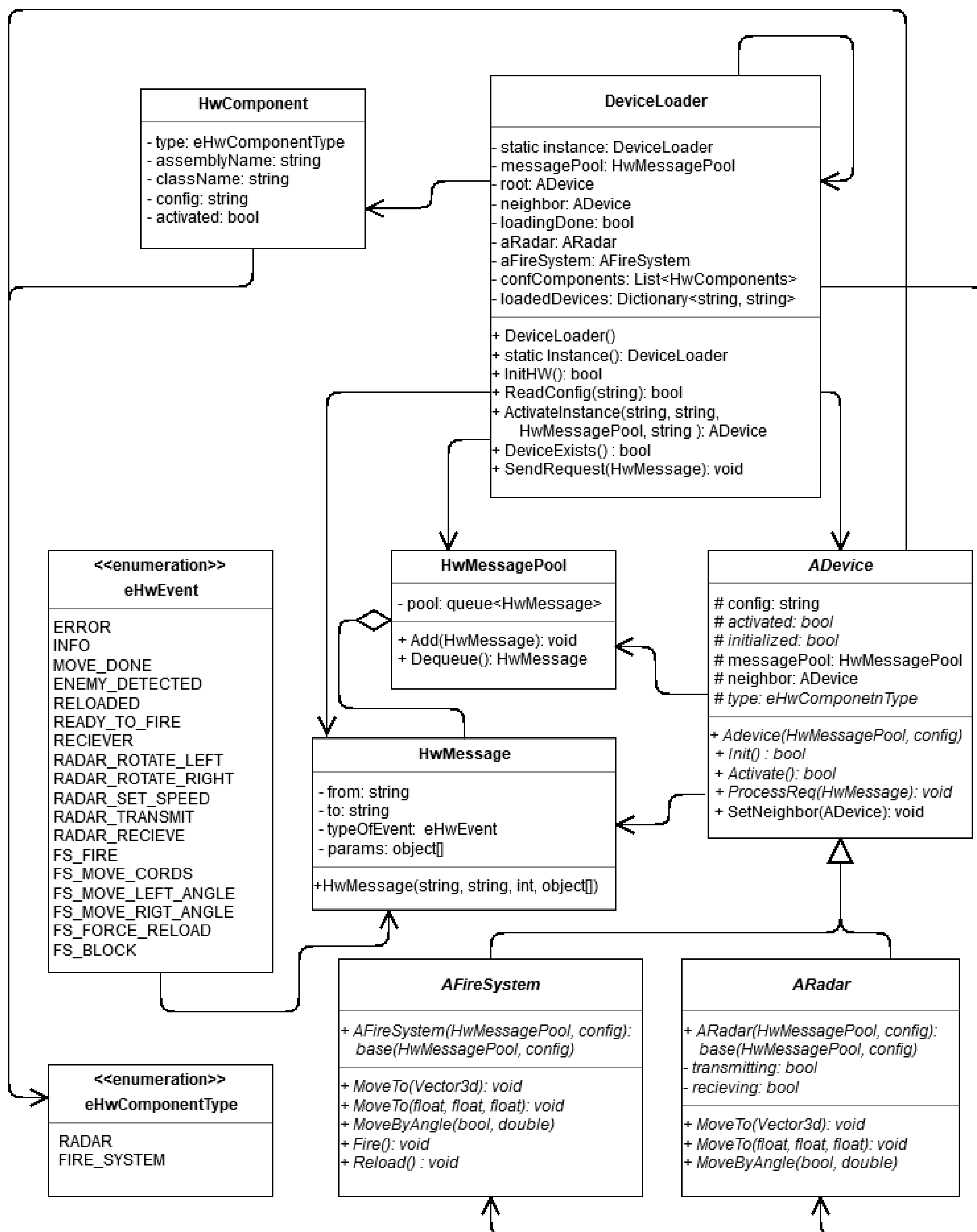
Obr. 39 Demonstrace konkretizace jednotlivých zařízení

5.2.1 Dynamické načítání hardware

Dynamické načítání je realizováno skrze třídu *DeviceLoader*. Načítat Hardware je potřeba jen jednou, při inicializaci celého programu, proto je tato třída navržena jako **Singleton**, jedná se o jeden z návrhových vzorů. Díky němu je zaručeno, že vznikne jediná instance této třídy. Po vytvoření instance se volá metoda *InitHW()*. Uvnitř této metody proběhne celé načtení hardware. Nejprve se z konfiguračního souboru vyčte, který hardware se má načíst, může se jednat například o *json* soubor, ve kterém je pole odpovídající třídě

HwComponent. Pomocí json knihoven je možné toto textové pole převést rovnou na pole objektů daného typu. Vzniklé objekty *HwComponent* jsou uchovány v atributu *confComponents.* Při procházení tohoto listu se volá metoda ActivateInstance, která je **tovární metodou**. Objekt *HwComponent* nese informace o cestě k potřebné *.dll* knihovně a také konfiguraci komponenty. Pomocí knihovny *System.Reflection* pak můžeme vytvořit instanci

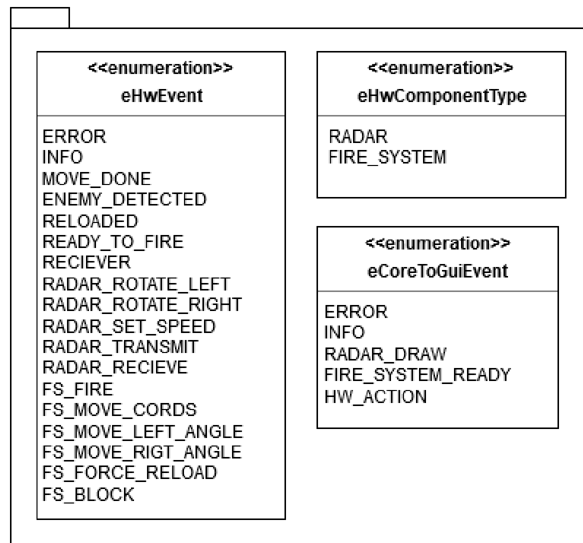
dané *.dll*.Metoda ActivateInstance() vrací objekt *ADevice*, který pak může být násilně přetypován například na *ARadar.* Při procházení a tvoření instancí se zároveň jednotlivé komponenty zřetězí. První komponenta se uchová v atributu *root*, a následující v *neighbor*. Volá se nad nimi pak metoda *SetNeighbor()*, kdy se ukáže na předcházejícího. Při validním načtení komponenty a validním vytvoření její instance se tato komponenta zapíše do atributu *loadedDevices* a následně je možné se na tuto komponentu doptat, zda existuje, pomocí *DeviceExists().*



Obr. 40 Demonstrace načítání hardware

5.2.2 Dostupné enumerace

Z důvodu dostupnosti enumerací potřebných v celém projektu je na místě je umístnit do samostatného jmenného prostoru *namespace*, který se pak použije pomocí *using,* v místech kde bude vazba na enumeraci zapotřebí.

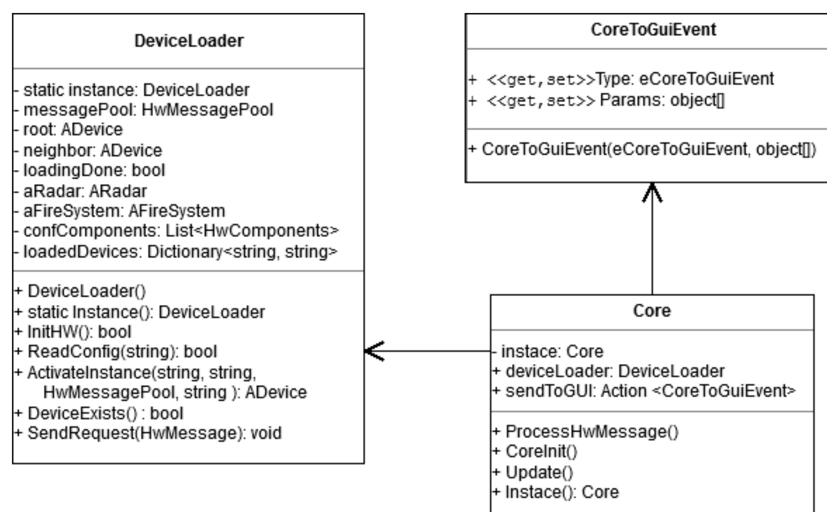


Obr. 41 Zaobalení enumerací do jednoho namespace

5.2.3 Napojení na jádro

Jádro programu slouží jako rozcestník v komunikaci mezi uživatelským rozhraním a hardware. Jádro obsahuje instanci třídy *DeviceLoader* a komunikace s hardware probíhá skrze tuto instanci. Instance Jádra je zapotřebí také jedna, rovněž je zde využit návrhový vzor singleton. Po vytvoření instance Jádra je zavolána jeho metoda *CoreInit(),* kde proběhne inicializace jádra a rovněž inicializace *DeviceLoader* a načtení všeho hardware.

S každým cyklem smyčky se volá metoda *ProcessHwMessage()*, která zpracovává zprávy z instance třídy *DeviceLoader* a reaguje na ně. Jádro obsahuje atribut *sendToGUI* typu *Action*, díky kterému můžeme invokovat metodu s parametry v uživatelském rozhraní GUI.

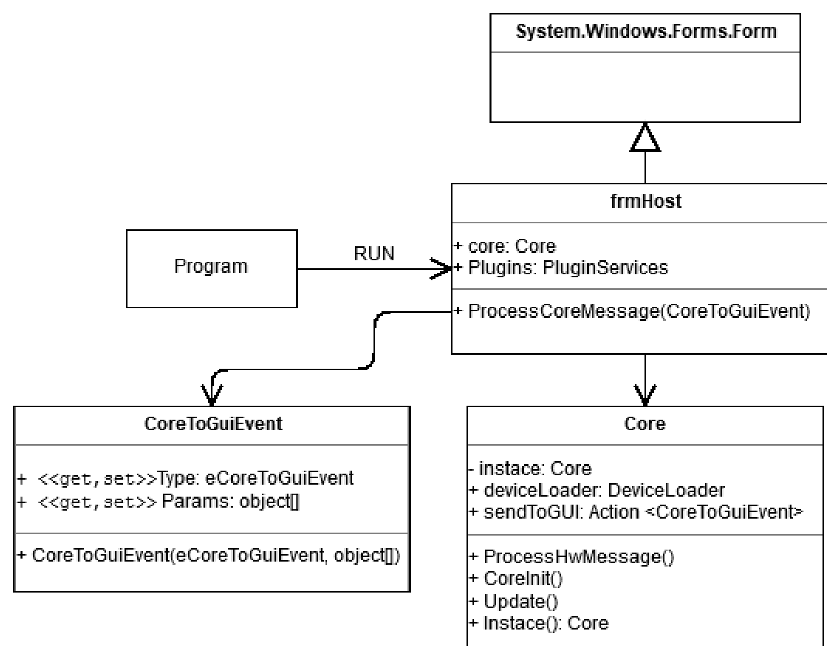


Obr. 42 Napojení hardware na jádro

5.3 Grafické uživatelské rozhraní GUI

Z uživatelského pohledu se jedná o formulářovou aplikaci. Při návrhu je použito formulářů ze jmenného prostoru *System.Windows.Forms*a také funkcionalita pro dynamické načítání *System.Reflection*. Uživatelské rozhraní je navrženo jako modulární aplikace, kterou je možné rozšiřovat. Systém je možné rozšiřovat pomocí pluginů, které implementují definované rozhraní.

Při startu programu je vytvořena instance formuláře *frmHost*. Tento formulář pak obsahuje atribut *core*, tím je zajištěna vazba mezi jádrem a uživatelským rozhraním UI. Pro zajištění chodu jádra je vytvořen Thread, který periodicky volá metodu jádra *core.Instance.Update()*. V metodě *Update()* dochází ke zpracování zpráv, proto je nutné tuto metodu periodicky volat. Při inicializaci je svázán *ActionsentToGUI* z jádra s metodou formuláře *ProcessCoreMessage().*



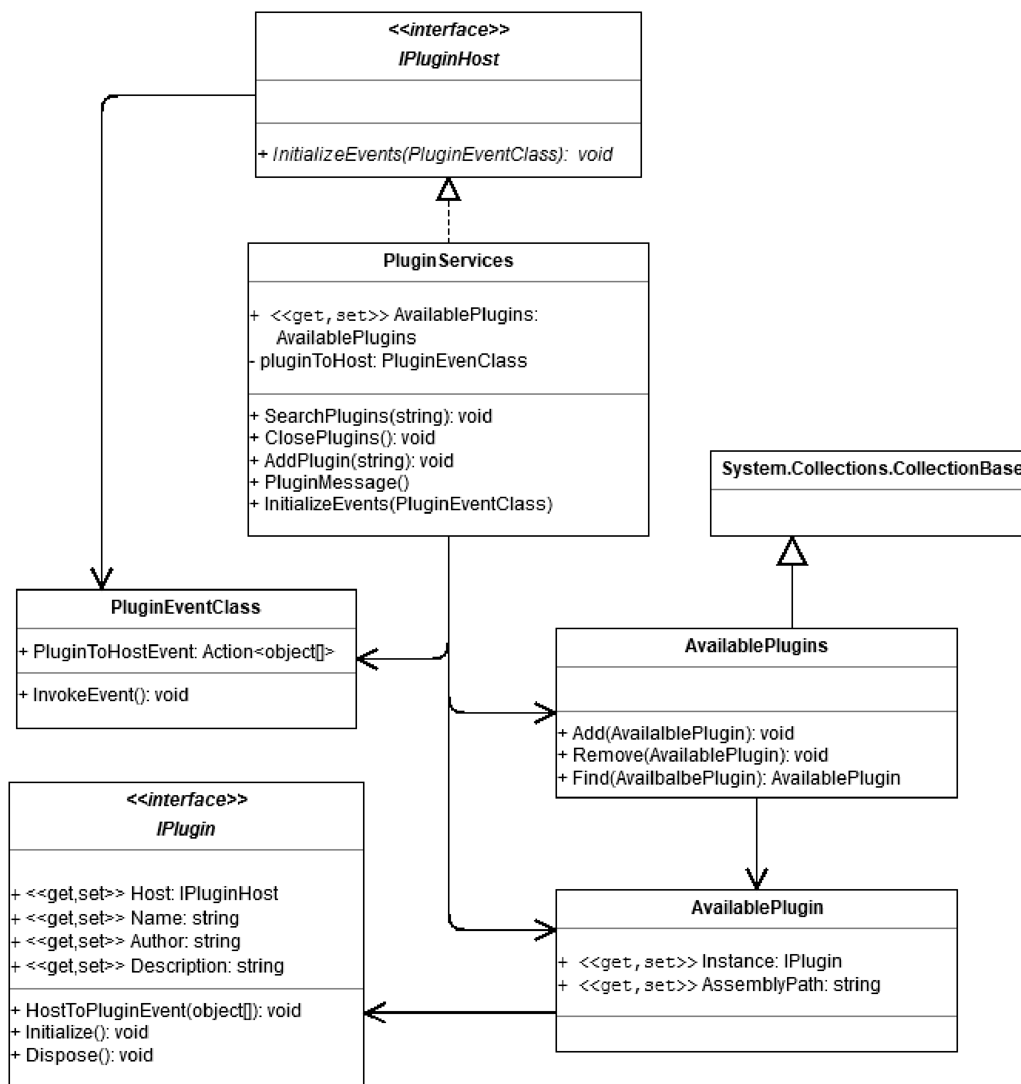
Obr. 43 Napojení jádra na uživatelské rozhraní

Formulář *frmHost* obsahuje atribut *Plugins*, který nese informace o zaregistrovaných pluginech pomocí třídy *PluginServices*.

5.3.1 Modulární uživatelské rozhraní

Třída *PluginServices* implementuje rozhraní *IPluginHost.* Toto rozhraní obsahuje metodu *InitializeEvents()*, která přebírá přes parametr objekt *PluginEventClass.* Třída*PluginEventClass* je zodpovědný za komunikaci od pluginu směrem k hostu. Tato třída

obsahuje atribut *ActionPluginToHostEvent,* který musí být svázán s hostovou částí – *PluginServices*. Toto spojení se realizuje metoda *InitializeEvents(),* kde dojde ke spojení akce s konkrétní metodou hostu, v tomto případě s metodou *PluginMessage()*.Metoda InvokeEvent() třídy *PluginEventClass*, zde byla ponechána, pokud by došlo k rozhodnutí, že místo *Action* použijeme *EventHandler*.

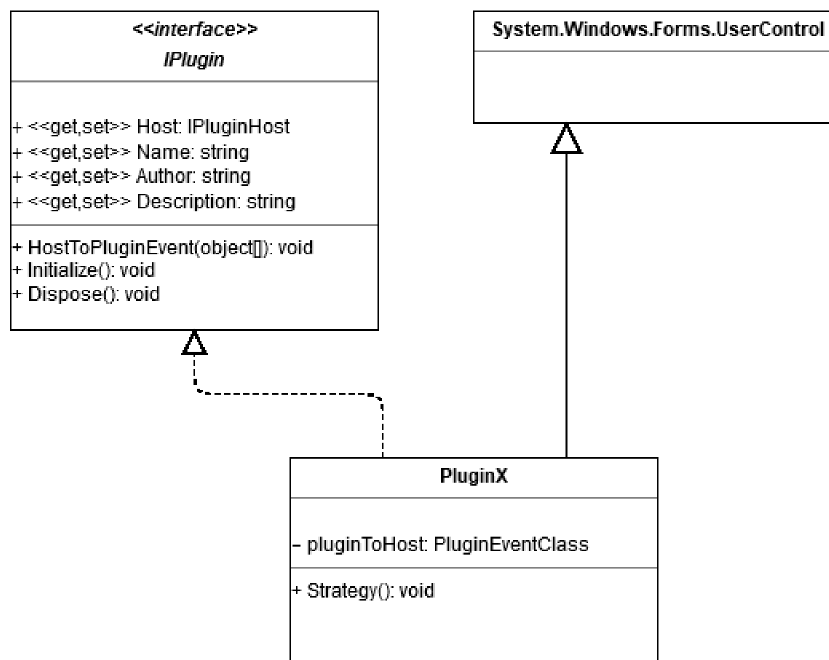


Obr. 44 Modulární uživatelské rozhraní

Třída *PluginServices*dále obsahuje vlastnost (property) *AvailablePlugins* podle stejnojmenné třídy, která je kolekcí načtených pluginů. Definuje své interní metody pro hledání, zavírání a přidávání pluginů. Třída *AvailablePlugin* pak uchovává odkaz na instanci pluginu, pro zpřístupnění.

5.3.2 Vytvoření pluginu

Uživatelské rozhraní rozšiřujeme pomocí pluginů. Konkrétní plugin musí implementovat rozhraní *IPlugin*. V tomto rozhraní se mohou nacházet jakékoliv metody, na které mají jednotlivé pluginy reagovat. Pro názornost je v rozhraní *IPlugin* definována metoda *HostToPlugin()*, která přes parametr posílá pole objektů. Tímto způsobem je zajištěna komunikace od hosta směrem k pluginu s neomezeným počtem parametrů. Dále je to už jen otázka definovaného protokolu.



Obr. 45 Vytvoření konkrétního modulu/pluginu

Na obrázku je znázorněn konkrétní pluginPluginX, který dědí od UserControl a zároveň implementuje rozhraní IPlugin. Pokud bude uvnitř vytvořena instance třídy *PluginEventClass*, může plugin zavolat *someClass.PluginToHostEvent(newobject[] { param})*a následně host tuto zprávu obdrží. Plugin samozřejmě může obsahovat instance jiných tříd, tím se zde nabízí možnost rozšířit i logiku programu, proto je zde znázorněna metoda *Strategy()*, která má poukázat na možnost vlastní implementace logiky.

ZÁVĚR

Tato práce se zabývá využitím návrhových vzorů pro vývoj naváděcích systémů. V práci je popsán princip a metody navádění na cíl, čtenář je seznámen s problematikou dronů na technické úrovni. Dále je čtenář seznámen se základy objektivně orientovaného programování, jsou zde vysvětleny základní principy třídní dědičnosti a použití modifikátorů tříd. V práci je shrnuto použití UML diagramů na úrovni tříd a vysvětleno použití vzájemných vazeb mezi nimi.

V praktické části je analyzováno 23 návrhových vzorů, které jsou obecně uznané. Pomocí UML diagramů jsou znázorněny vazby mezi třídami a objekty. Každý návrhový vzor je popsán, včetně výpisu jeho možného použití. V přílohách se nachází implementace vybraných návrhových vzorů od každé kategorie (strukturální, behaviorální a tvořivé). Dále je pak demonstrováno použití návrhových vzorů při tvorbě naváděcího systému. V popsaném návrhu figurují tři přímo viditelné návrhové vzory a další, které jsou obsaženy v použitých knihovnách.

Navržený systém je modulární a rozšiřitelný jak na úrovni hardware, tak i na úrovni grafického uživatelského rozhraní. Oddělení logiky od ostatních subsystému je realizováno uvnitř jednotlivých pluginů.

Navržený systém nemusí nutně předpokládat použití pro naváděcí střelný systém. Pokud budou uvnitř systému figurovat komponenty jako vstupně/výstupní deska, můžeme pomocí této aplikace ovládat například Arduino modul, modul od Jablotron, prakticky jakýkoliv hardware a aplikace může být použita k úplně jinému účelu. Jedním z nich může být pro využití v komerční bezpečnosti sytém na řízení dronů. Mezi systémem a drony by mohla být realizována komunikace (rádiová), dron by vysílal základní informace (svou polohu a například obrazový záznam z kamery). Systém by prováděl vyhodnocování a pomocí umělé inteligence by mohl řídit roj těchto dronů. Mohl by vzniknout systém s vysokým výpočetním výkonem, který by přebral roli vyhodnocování, tím pádem by se o vyhodnocování nemuseli starat drony samotné. To by znamenalo, že by použité drony mohly být ve výsledku menší, protože by nemuseli být vybaveny velkými pamětmi, bateriemi a dalšími komponentami, které jsou pro rozhodovací algoritmy potřebné. Toto jsou možnosti kudy je možné navržený systém rozšířit.

SEZNAM POUŽITÉ LITERATURY

[1] What type of sensor forthreedimensionalpositiontracking.*Metafilter* [online]. United Kingdom: MetaFilter Network, 2011 [cit. 2018-11-19]. Dostupné z: https://ask.metafilter.com/203358/What-type-of-sensor-for-three-dimensional-position-tracking

[2] DetectingPositionwith a Standard IR Receiver.*Arduino* [online]. New York: Arduino, 2018, 2015 [cit. 2018-11-19]. Dostupné z: https://forum.arduino.cc/index.php?topic=300686.0

[3] Calculatenew X,Y positionbased on angle center x,yisfacingfor a radar.*StackOverflow* [online]. New York: Stack Exchange, 2018, 2014 [cit. 2018-11-19]. Dostupné z: https://stackoverflow.com/questions/27664171/calculate-new-x-y-position-based-on-angle-center-x-y-is-facing-for-a-radar

[4] Výukový materiál zpracovaný v rámci projektu „Učíme moderně “. *SPŠ strojnická Olomouc* [online]. Olomouc: Střední průmyslová škola strojnická Olomouc, 2012 [cit. 2018-11-19]. Dostupné z: https://www.spssol.cz/rsimages/DUM/MAT/20_Reseni_obecneho_trojuhelniku_prezentace.pdf

[5]DURANT, Frederick C. a John F. GUILMARTIN. Rocket and missilessys-tem.*EncyclopaediaBritannica* [online]. London: EncyclopaediaBritannica, 2018 [cit. 2019-05-19]. Dostupné z: https://www.britannica.com/technology/rocket-and-missile-system/Tactical-guided-missiles#ref57320

[6] Chapter 15 Guidance and Contro. *Fundamentals of Naval Weapons Systems: FederationofAmericanScientists* [online]. USA: FederationofAmericanScientists, 2016 [cit. 2019-05-19]. Dostupné z: https://fas.org/man/dod-101/navy/docs/fun/part15.htm

[7] MissileGuidance.*MissileGuidance* [online]. USA: aerospaceweb.org, 2018 [cit. 2019-05-19]. Dostupné z: http://www.aerospaceweb.org/question/weapons/q0187.shtml

[8] SEEKER. HowSatellites Track YourExactLocation.*Youtube* [online]. California: youtube, 2015 [cit. 2019-05-19]. Dostupné z: https://www.youtube.com/watch?v=04VK5XscxB4

[9] BRAIN, Marshall a Tom HARRIS. How GPS ReceiversWork.*HowStuffWorks* [online]. California: HowStuffWorks, 2006 [cit. 2019-05-19]. Dostupné z: https://electronics.howstuffworks.com/gadgets/travel/gps1.htm

[10] UNFA. How GPS works? Trilaterationexplained.*Youtube* [online]. California: youtube, 2014 [cit. 2019-05-19]. Dostupné z: https://www.youtube.com/watch?v=4O3ZVHVFhes

[11] BIEZEN, Michiel van. SpecialTopics - GPS (6 of 100) TriangulationWithSatellites.*Youtube* [online]. California: youtube, 2016 [cit. 2019-05-19]. Dostupné z: https://www.youtube.com/watch?v=QK1lDsinMwk

[12] HANDLEY, Phil. RadioDirectionFindingTechniques.*Hackaday* [online]. USA: Hackaday, 2017 [cit. 2019-05-19]. Dostupné z: https://hackaday.io/project/25995-bloodhound-autonomous-radiolocation-drone/log/63866-radio-direction-finding-techniques

[13] TrilaterationvsTriangulation – How GPS ReceiversWork.*GISGeography* [online]. California: GISGeography, 2019 [cit. 2019-05-19]. Dostupné z: https://gisgeography.com/trilateration-triangulation-gps/

[14] BUSHCRAFT, Coalcracker. Navigation: Triangulation.*Youtube* [online]. California: youtube, 2018 [cit. 2019-05-19]. Dostupné z: https://www.youtube.com/watch?v=QWHRyHZbXsw

[15] Findyourlocationusing a map and compasswithtriangulation.*Youtube* [online]. California: youtube, 2017 [cit. 2019-05-19]. Dostupné z: https://www.youtube.com/watch?v=-Ak86suJFjo

[16] DOMLÁTIL, Jan. Základy orientace v terénu a topografie – 2. část – Buzola a azimut. *Rychlá rota testuje* [online]. ČR: rychlarota.akademie-shield.cz, 2016 [cit. 2019-05-19]. Dostupné z: http://rychlarota.akademie-shield.cz/zaklady-orientace-v-terenu-a-topografie-2-cast-buzola/

[17] How long candronesflyfor?.*Quora* [online]. California: Quora, 2019 [cit. 2019-05-19]. Dostupné z: https://www.quora.com/How-long-can-drones-fly-for

[18] CORRIGAN, Fintan. How Do DronesWork And WhatIsDrone Technology.*Dronezone* [online]. New York: Dronezone, 2019 [cit. 2019-05-19]. Dostupné z: https://www.dronezon.com/learn-about-drones-quadcopters/what-is-drone-technology-or-how-does-drone-technology-work/

[19] CORRIGAN, Fintan. 12 Best Photogrammetry Software For 3D MappingUsingDrones.*Dronezone* [online]. New York: Dronezone, 2019 [cit. 2019-05-19]. Dostupné z: https://www.dronezon.com/learn-about-drones-quadcopters/drone-3d-mapping-photogrammetry-software-for-survey-gis-models/

[20] Meetthedazzlingflyingmachinesofthefuture | RaffaelloD'Andrea.*Youtube* [online]. California: youtube, 2016 [cit. 2019-05-19]. Dostupné z: https://www.youtube.com/watch?v=RCXGpEmFbOw

[21] SCHROTH, Lukas. Drones and ArtificialIntelligence.*Droneii* [online]. Německo: Droneii, 2018 [cit. 2019-05-19]. Dostupné z: https://www.droneii.com/drones-and-artificial-intelligence

[22] GOHL, Pascal, Dominik HONEGGER, Sammy OMARI, Markus ACHTELIK, Marc POLLEFEYS a Roland SIEGWART. OmnidirectionalVisualObstacleDetectionusingEmbedded FPGA. *ETH Zurich* [online]. Švýcarsko: ETH Zurich, 2019 [cit. 2019-05-19]. Dostupné z: https://inf.ethz.ch/personal/marc.pollefeys/pubs/GohlIROS15.pdf

[23] MICRO DRONES KILLER ARMS ROBOTS - AUTONOMOUS ARTIFICIAL INTELLIGENCE - WARNING !!.*Youtube* [online]. California: youtube, 2017 [cit. 2019-05-19]. Dostupné z: https://www.youtube.com/watch?v=TlO2gcs1YvM

[24] LAPKOVÁ, Dora. *Technologie komerční bezpečnosti II: Řízení rizik*. Univerzita Tomáše Bati ve Zlíně, 2016.

[25] TypesofBiometrics.*Biometricsinstitute* [online]. Londýn: Biometric Institute, 2017 [cit. 2018-11-19]. Dostupné z: https://www.biometricsinstitute.org/types-of-biometrics

[26] About.*OpenCV* [online]. California: OpenCV Team, ©2018 [cit. 2019-05-19]. Dostupné z: https://opencv.org/about.html

[27] Autentizační metody založené na biometrických informacích. *ČVUT Fakulta Elektrotechniky* [online]. Brno: Vysoké učení technické v Brně, 2010 [cit. 2018-11-19]. Dostupné z: http://access.feld.cvut.cz/view.php?cisloclanku=2010110002

[28] MENCL, Michal. Základní principy a pojmy objektově orientovaného programování. *Pehapko.cz* [online]. ČR: pehapko.cz, 2018 [cit. 2019-05-19]. Dostupné z: http://pehapko.cz/oop/uvod

[29] Polymorfismus (Průvodce programováním v C#). *Docs.microsoft.com* [online]. USA: Microsoft, 2015 [cit. 2019-05-19]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/csharp/programming-guide/classes-and-structs/polymorphism

[30] Modifikátory přístupu (Referenční dokumentace jazyka C#). *Docs.microsoft.com* [online]. USA: Microsoft, 2015 [cit. 2019-05-19]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/csharp/language-reference/keywords/access-modifiers

[31] PECINOVSKÝ, Rudolf. *Návrhové vzory: [33 vzorových postupů pro objektové programování]*. Brno: ComputerPress, 2007. ISBN 978-80-251-1582-4.

[32] WHITLOCK, Byron. Whatisaninternalsealedclass in C#?.*StackOverflow* [online]. New York: StackOverflow, 2011 [cit. 2019-05-19]. Dostupné z: https://stackoverflow.com/questions/4619185/what-is-an-internal-sealed-class-in-c

[33] Interface vsAbstractClass (general OO).*StackOverflow* [online]. NewYork: StackOverflow, 2017 [cit. 2019-05-19]. Dostupné z: https://stackoverflow.com/questions/761194/interface-vs-abstract-class-general-oo

[34] GAMMA, Erich. *Návrh programů pomocí vzorů: stavební kameny objektově orientovaných programů*. Praha: Grada, 2003. Moderní programování. ISBN 978-802-4703-022.

[35] UML Class Diagram Relationships in ExplainedwithExamples.*Creately* [online]. Australia: CinergixPty., 2018 [cit. 2019-05-20]. Dostupné z: https://creately.com/blog/diagrams/class-diagram-relationships/

[36] Understanding UML Class Diagram Relationships.*Creately* [online]. Australia: CinergixPty., 2018 [cit. 2019-05-20]. Dostupné z: https://creately.com/blog/diagrams/understanding-the-relationships-between-classes/

[37] BISHOP, J. M. *C# 3.0 design patterns*. Sebastopol, CA: O'Reilly, c2008. ISBN 978-0-596-52773-0.

SEZNAM OBRÁZKŮ

Obr. 1 Princip zaměření pomocí radaru 12
Obr. 2 Naváděcí metody (převzato z [7]) 14
Obr. 3 Triangulace signálů (převzato z [12]) 16
Obr. 4 Popis buzoly (převzato z [16]) 16
Obr. 5 Princip útočného mikro-dronu (převzato z [23]) 18
Obr. 6 Vztah mezi FRR a FAR (převzato z [27]) 19
Obr. 7 Vztah asociace v UML 24
Obr. 8 Vztah mnohočetnosti v UML 24
Obr. 9 Vztah směrové asociace v UML 24
Obr. 10 Vztah reflexe v UML 25
Obr. 11 Vztah agregace v UML 25
Obr. 12 Vztah kompozice v UML 25
Obr. 13 Vztah generalizace v UML 25
Obr. 14 Vztah realizace 26
Obr. 15 Návrhový vzor Tovární metoda (převzato z [37, s. 111]) 28
Obr. 16 Návrhový vzor Abstraktní továrna (převzato z [37, s. 124]) 29
Obr. 17 Návrhový vzor Jedináček (převzato z [37, s. 115]) 30
Obr. 18 Návrhový vzor Prototyp (převzato z [37, s. 101]) 31
Obr. 19 Návrhový vzor Stavitel (převzato z [37, s. 130]) 32
Obr. 20 Návrhový vzor Tovární metoda (převzato z [37, s. 76]) 33
Obr. 21 Návrhový vzor Dekorátor (převzato z [37, s. 11]) 34
Obr. 22 Návrhový vzor Fasáda (převzato z [37, s. 94]) 35
Obr. 23 Návrhový vzor Most (převzato z [37, s. 38]) 36
Obr. 24 Návrhový vzor Muší váha (převzato z [37, s. 62]) 37
Obr. 25 Návrhový vzor Skladba (převzato z [37, s. 51]) 38
Obr. 26 Návrhový vzor Zástupce (převzato z [37, s. 23]) 39
Obr. 27 Návrhový vzor Interpret (převzato z [37, s. 235]) 40
Obr. 28Návrhový vzor Šablonová metoda (převzato z [37, s. 158]) 41
Obr. 29 Návrhový vzor Iterátor (převzato z [37, s. 191]) 42
Obr. 30 Návrhový vzor Návštěvník (převzato z [37, s. 220]) 43
Obr. 31 Návrhový vzor Obnovitel (převzato z [37, s. 243]) 44
Obr. 32 Návrhový vzor Pozorovatel (převzato z [37, s. 211]) 45

Obr. 33 Návrhový vzor Prostředník (převzato z [37, s. 201]) 46
Obr. 34 Návrhový vzor Příkaz (převzato z [37, s. 177]) 47
Obr. 35 Návrhový vzor Řetězec odpovědností (převzato z [37, s. 165]) 48
Obr. 36 Návrhový vzor Stav (převzato z [37, s. 150]) 49
Obr. 37 Návrhový vzor Zástupce (převzato z [37, s. 141]) 50
Obr. 38 Popis třídy pro obecné zařízení 52
Obr. 39 Demonstrace konkretizace jednotlivých zařízení 53
Obr. 40 Demonstrace načítání hardware 55
Obr. 41 Zaobalení enumerací do jednoho namespace 56
Obr. 42 Napojení hardware na jádro 56
Obr. 43 Napojení jádra na uživatelské rozhraní 57
Obr. 44 Modulární uživatelské rozhraní 58
Obr. 45 Vytvoření konkrétního modulu/pluginu 59

SEZNAM TABULEK

Tab. 1 Vysvětlení klíčových slov v C# 21
Tab. 2 Kategorizace návrhových vzorů (převzato z [34, s. 29]) 23

SEZNAM PŘÍLOH

Příl. 1 Obsah přiloženého CD

PŘÍLOHA I: OBSAH PŘILOŽENÉHO CD

\Navrhove_vzory\

\Navrhove_vzory\Adapter\

\Navrhove_vzory\Builder\

\Navrhove_vzory\ChainOfResponsibility\

\Navrhove_vzory\Command\

\Navrhove_vzory\Decorator\

\Navrhove_vzory\FactoryMethod\

\Navrhove_vzory\Singleton\

\UML_navrhove_vzory_cpp\

V bázi CD se nachází dva adresáře. Adresář *Navrhove_vzory* a adresář *UML_navrhove_vzory_cpp*. Adresář *Navrhove_vzory* obsahuje osm podadresářů pojmenovaných podle jména návrhového vzoru. Uvnitř se nacházejí zdrojové kódy, pro jednoduchou demonstraci těchto návrhových vzorů. V adresáři *UML_navrhove_vzory_cpp* lze najít UML diagramy návrhových vzorů při nevyužití interface.