



## Zadání bakalářské práce

<b>Název:</b>	System pro generování tras pro využití v simulačních modelech jízdy vozidla
<b>Student:</b>	Lukáš Jílek
<b>Vedoucí:</b>	Ing. Marko Šidlovský
<b>Studijní program:</b>	Informatika
<b>Obor / specializace:</b>	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
<b>Katedra:</b>	Katedra softwarového inženýrství
<b>Platnost zadání:</b>	do konce letního semestru 2022/2023

### Pokyny pro vypracování

Cílem práce je vytvoření aplikačního softwaru pro generování jízdních tras, které budou využity v simulačních modelech jízdy vozidla. Software musí podporovat následující uživatelský proces generování tras:

1. výběr trasy z mapy včetně volby většího počtu průjezdních bodů
2. načtení dat o provozu v zadaný čas jízdy
3. načtení dat jízdního profilu (rychlost / čas jízdy, nadmořská výška)
4. ruční úprava úseků – zkrácení či prodloužení úseku, nastavení vlastní střední rychlosti
5. možnost definování dalších vlastních proměnných pro jednotlivé úseky
6. export výstupu do formátu GPX

Jednotlivé úkoly:

1. Analýza funkčních požadavků aplikace, výběr vhodného programovacího jazyka a následná rešerše existujících knihoven, jichž lze využít při implementaci.
2. Vytvoření návrhu aplikace v souladu s principy OOP.
3. Implementace aplikace.
4. Otestování jednotlivých částí aplikace.
5. Vytvoření aplikační dokumentace.



Bakalářská práce

**SYSTÉM PRO  
GENEROVÁNÍ TRAS  
PRO VYUŽITÍ  
V SIMULAČNÍCH  
MODELECH JÍZDY  
VOZIDLA**

**Lukáš Jílek**

Fakulta informačních technologií  
Katedra softwarového inženýrství  
Vedoucí: Ing. Marko Šidlovský  
12. května 2022

České vysoké učení technické v Praze

Fakulta informačních technologií

© 2022 Lukáš Jílek. Odkaz na tuto práci.

*Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.*

Odkaz na tuto práci: Jílek Lukáš. *Systém pro generování tras pro využití v simulačních modelech jízdy vozidla*. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

## Obsah

Poděkování	VII
Prohlášení	VIII
Abstrakt	IX
Seznam zkratek	X
<b>1 Úvod</b>	<b>1</b>
<b>2 Aplikační software</b>	<b>2</b>
2.1 Co je to aplikační software? . . . . .	2
2.2 Rozdělení aplikačního softwaru . . . . .	2
2.2.1 Rozdělení podle běhového prostředí . . . . .	2
2.2.1.1 Nativní aplikace . . . . .	2
2.2.1.2 Hybridní aplikace . . . . .	3
2.2.1.3 Webové aplikace . . . . .	4
2.2.2 Rozdělení podle typu licence . . . . .	4
2.2.2.1 Proprietární licence . . . . .	4
2.2.2.2 Svobodné licence . . . . .	4
2.2.3 Příklady aplikačního softwaru . . . . .	5
2.2.3.1 Správce souborů . . . . .	5
2.2.3.2 Kancelářské aplikace . . . . .	5
2.2.3.3 CAx aplikace . . . . .	5
2.2.3.4 Animační a vizualizační software . . . . .	6
2.2.3.5 Zábavní software . . . . .	6
2.2.3.6 Business management software . . . . .	6
2.2.3.7 Simulační software . . . . .	7
<b>3 API</b>	<b>8</b>
3.1 Co je to API? . . . . .	8
3.2 Příklady API . . . . .	8
3.2.1 Staticky linkované API . . . . .	8
3.2.1.1 API operačního systému . . . . .	9
3.2.1.2 Grafické API . . . . .	9
3.2.2 Webové API . . . . .	10
3.2.2.1 SOAP API . . . . .	10
3.2.2.2 REST API . . . . .	14
<b>4 Programovací jazyky</b>	<b>18</b>
4.1 Co je to programovací jazyk? . . . . .	18
4.2 Programovací paradigmata . . . . .	18
4.2.1 Imperativní a procedurální programování . . . . .	18
4.2.2 Objektově orientované programování . . . . .	20

## Obsah

4.2.3	Deklarativní programování . . . . .	21
4.2.3.1	Funkcionální programování . . . . .	22
4.2.3.2	Logické programování . . . . .	23
<b>5</b>	<b>Vývoj aplikačního softwaru</b> . . . . .	<b>24</b>
5.1	Metodika vývoje aplikačního softwaru . . . . .	24
5.1.1	Vodopádový model . . . . .	25
5.1.2	Iterativní model . . . . .	25
5.1.3	Agilní model . . . . .	25
5.2	Požadavky aplikačního softwaru . . . . .	26
5.2.1	Požadavky . . . . .	26
5.2.2	Requirement engineering . . . . .	27
5.2.2.1	Proces requirement engineeringu . . . . .	28
5.3	Návrh aplikačního softwaru . . . . .	30
5.3.1	Výběr programovacího jazyka . . . . .	30
5.3.2	Způsob uložení dat . . . . .	30
5.3.3	Softwarová architektura . . . . .	30
5.3.3.1	Pohledy na softwarovou architekturu . . . . .	31
5.3.3.2	Architektonické vzory . . . . .	33
5.3.4	Návrhové vzory . . . . .	42
5.3.4.1	Vzory tříd . . . . .	42
5.3.4.2	Konstrukční objektové vzory . . . . .	46
5.3.4.3	Strukturální objektové vzory . . . . .	49
5.3.4.4	Behaviorální objektové vzory . . . . .	54
5.4	Implementace aplikačního softwaru . . . . .	60
5.4.1	Kvalitní kód . . . . .	60
<b>6</b>	<b>Výsledný aplikační software</b> . . . . .	<b>63</b>
6.1	Aplikační požadavky . . . . .	63
6.1.1	Funkční požadavky . . . . .	63
6.1.2	Nefunkční požadavky . . . . .	64
6.2	Návrh výsledné aplikace . . . . .	64
6.2.1	Typ aplikace a použitý programovací jazyk . . . . .	64
6.2.2	Výběr API . . . . .	64
6.2.2.1	Kritéria pro výběr . . . . .	64
6.2.2.2	APIs pro zobrazení mapy a jejích prvků . . . . .	65
6.2.2.3	APIs pro hledání informací o trase . . . . .	65
6.2.3	Použité architektonické vzory . . . . .	66
6.2.4	Způsoby importu a exportu trasy . . . . .	68
6.2.4.1	JSON import . . . . .	69
6.2.4.2	JSON export . . . . .	70
6.2.4.3	GPX export . . . . .	73
6.3	Implementace výsledné aplikace . . . . .	73
6.3.1	Jednotlivé součásti aplikace . . . . .	73
6.3.1.1	Zobrazení mapy . . . . .	73
6.3.1.2	Vytvoření, zobrazení a získání informací o trase . . . . .	75
6.3.1.3	Přidání nového bodu trasy . . . . .	79
6.3.1.4	Úprava vlastností jednotlivých úseků . . . . .	79
6.3.1.5	Získání informací o trase a jejích částech v zadaný čas . . . . .	80
6.3.1.6	Export ve formátu GPX a JSON . . . . .	81
6.4	Dokumentace výsledné aplikace . . . . .	82
6.5	Testování výsledné aplikace . . . . .	82

7 Závěr	83
A Náhled grafického uživatelského rozhraní	
B Dokumentace aplikace	

## Seznam obrázků

6.1	Vrstvy aplikace . . . . .	66
6.2	Struktura GUI . . . . .	67
6.3	Struktura CLI . . . . .	68
6.4	Mapa po inicializaci . . . . .	74
6.5	CLI tvorba trasy (1) . . . . .	75
6.6	CLI tvorba trasy (2) . . . . .	76
6.7	CLI import trasy . . . . .	76
6.8	GUI Okno pro tvorbu trasy . . . . .	77
6.9	GUI Okno pro import trasy . . . . .	77
6.10	GUI Okno pro přidání nového bodu . . . . .	79
6.11	GUI Okno pro úpravu úseku trasy . . . . .	79
6.12	GUI Okno pro nastavení časového intervalu . . . . .	80
6.13	CLI export trasy . . . . .	81
6.14	GUI Okno export trasy . . . . .	82

## Seznam tabulek

6.1	Prvky JSON import souřadnic . . . . .	69
6.2	Prvky JSON import souřadnic . . . . .	69
6.3	Prvky JSON export souboru . . . . .	70
6.4	Prvky JSON export kroku trasy . . . . .	71
6.5	Prvky JSON export souřadnic . . . . .	71
6.6	Statusy odpovědi Google Directions API . . . . .	78
6.7	Statusy odpovědi Google Elevation API . . . . .	78
6.8	Statusy odpovědi Here Waypoints Sequence API . . . . .	78
6.9	Statusy odpovědi Google Distance Matrix API . . . . .	81

## Seznam výpisů kódu

3.1	Příklad zprávy SOAP, která obsahuje SOAP záhlaví a SOAP tělo . . . . .	14
6.1	Struktura JSON import souboru . . . . .	69



6.2	Struktura JSON import souřadnic . . . . .	69
6.3	Příklad JSON import souboru . . . . .	70
6.4	Struktura JSON export souboru . . . . .	70
6.5	Struktura JSON export kroku trasy . . . . .	71
6.6	Struktura JSON export souřadnic . . . . .	71
6.7	Příklad JSON export souboru . . . . .	72
6.8	Příklad GPX export souboru . . . . .	73
6.9	initMap() funkce . . . . .	74

*Chtěl bych poděkovat především panu Ing. Marku Šidlovskému za vedení mé bakalářské práce, za jeho cenné odborné rady při zpracování bakalářské práce a za ochotu a vlídnost během konzultací. Děkuji rovněž panu doc. Ing. Václavu Jírovskému, CSc. za jeho laskavý přístup a důležitá odborná doporučení. Nakonec děkuji také rodině za podporu během psaní této práce a během celého studia.*

## Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 12. května 2022

.....

## Abstrakt

Cílem praktické části této práce je návrh a následná implementace desktopové aplikace pro tvorbu tras, které budou následně použity v simulačních modelech jízdy vozidla. Teoretická část této práce se zabývá popisem současného aplikačního softwaru, API vzorů, programovacích jazyků a procesu implementace aplikačního softwaru. Zároveň popisuje výslednou aplikaci. Výstupem této práce je desktopová aplikace pro generování tras, jež splňuje všechny zadané požadavky a její kód dodržuje principy objektově orientovaného programování.

**Klíčová slova** aplikační software, vývoj softwaru, desktopová aplikace, generování tras, aplikační programovací rozhraní, API

## Abstract

The aim of the practical part of this thesis is the design and subsequent implementation of a desktop application for the creation of routes that will be used in vehicle driving simulation models. The theoretical part of this thesis covers the description of current application software, API patterns, programming languages and the process of implementing the application software. At the same time, it describes the resulting application. The output of this work is a desktop application for generating routes that meets all the requirements and its code follows the principles of object-oriented programming.

**Keywords** application software, software development, desktop application, routes generating, application programming interface, API

## Seznam zkratk

API	Application Programming Interface
BNF	Backusova–Naurova forma
C&C	component-and-connector struktura
CAD	Computer aided design
CAE	Computer aided engineering
CAM	Computer aided manufacturing
CAx	Computer aided technologies
CLI	Rozhraní využívající příkazový řádek
CNA	Cloudově Nativní Aplikace
CRUD	Create, Read, Update, Delete
CSS	Kaskádové styly
DPD	Digital Product Development
DU	Deployment Unit
EULA	End user license agreement
GPU	Grafický procesor
GRASP	General Responsibility Assignment Software Patterns
GUI	Grafické uživatelské rozhraní
HTML	Hypertext Markup Language
IS	Informační systém
JS	JavaScript
JVM	Java Virtual Machine
MVC	Model-View-Controller
OOP	Objektově orientované programování
OS	Operační systém
P2P	peer-to-peer
RAM	Random-access memory
RE	Requirements Engineering
REST	Representational State Transfer
RIA	Rich Internet Application
SDK	Software Development Kit
SDLC	Software Development Life Cycle
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
UI	Uživatelské rozhraní
URI	Uniform Resource Identifier
VR	Virtuální realita
WSDL	Web Service Description Language
WWW	World Wide Web

# Kapitola 1

## Úvod

Simulační software je software modelující reálné děje pomocí sady matematických rovnic, jež uživateli poskytuje vhled na danou problematiku bez nutnosti nákladného sběru dat ze skutečného pozorování. Simulace jsou v současné době široce používány v průmyslových procesech k odhadu konečné ceny či chování výrobku za určitých podmínek. Zároveň jsou používány i k odhadu rizik spojených s užíváním konkrétního výrobku, nebo procesu.

Jednou z možných aplikací simulací jsou simulace jízdy vozidla, pro něž výsledný software této práce bude generovat trasy. Simulace jízd mohou být například použity pro snížení nákladů při doručování zboží, kontrole chování řidičů při samotném doručování, nebo odhadu budoucích cen poskytovaných služeb.

Hlavním vstupem do takového simulačního modelu jsou trasy vozidla, jež mohou mít různé parametry jako je např. průměrná rychlost, převýšení, doba trvání, atd. Cílem této práce je navrhnout a implementovat aplikační software pro návrh, úpravu a následný export ve formátu GPX těchto parametrizovaných tras. Parametr odhadovaného času jízdy je získáván pomocí webové API, jež zohledňuje provoz v zadaném časovém intervalu jízdy. Výsledná aplikace je napsaná v programovacím jazyce Java s využitím knihovny JavaFX pro grafické uživatelské rozhraní a Javascriptu, jež zjednodušuje práci se zvolenými webovými API pro zobrazení tras.

Práce se skládá z následujících hlavních částí:

### Aplikační software

Tato část definuje, co je aplikační software, jeho rozdělení a uvádí některé v současnosti používané typy a jejich konkrétní příklady.

### API

Část definující pojem API a popisující současné trendy v této oblasti.

### Programovací jazyky

Kapitola zabývající se rozdělením a popisem typů programovacích jazyků.

### Vývoj aplikačního softwaru

Kapitola, jež se věnuje procesu vývoje aplikačního softwaru se zaměřením především na architektonické a návrhové vzory softwaru.

### Výsledný aplikační software

Přehled výsledné aplikace zahrnující seznam požadavků, použité technologie a popis klíčových objektů společně s jejich testy.

# Aplikační software

## 2.1 Co je to aplikační software?

Na aplikační software, nebo také *aplikaci* můžeme definovat jako na ucelenou strukturu informací a alternativ řízení, jejichž prostřednictvím může uživatel provést požadovaný úkol, jež přímo nesouvisí se zařízením samotným.<sup>1</sup>[1] Jako příklad lze uvést textové editory, webové prohlížeče, hry, přehrávače videa a mnoho dalších aplikací, které uživatelé na svých zařízeních používají.

## 2.2 Rozdělení aplikačního softwaru

Jelikož jako lidé máme přirozenou potřebu u pozorovaných fenoménů hledat sdílené vlastnosti a následně je podle těchto vlastností třídít, neboť kategorizace je jednou z hlavních součástí lidské kognitivní schopnosti[2], lze i aplikace kategorizovat podle určitých parametrů. Bohužel ani u tak deterministické věci jako jsou aplikace, není pouze jediný způsob, jak mezi jednotlivými aplikacemi rozlišovat a ne každý způsob dělení je jednoznačný. Tato část se tedy zabývá problematikou dělení aplikací podle nejčastěji používaných dělicích vlastností.

### 2.2.1 Rozdělení podle běhového prostředí

První vlastností, podle které můžeme aplikace rozdělit, je jejich běhové prostředí. V tomto případě je dělení velmi jednoznačné, bohužel současný vývoj v odvětví cloudových aplikací, přináší nové typy aplikací, kterým začíná být těžké jednoznačně přiřadit jejich typ. I přesto jsme schopni bez větších problémů většinu aplikací rozdělit do kategorií popsanych v následujících oddílech.

#### 2.2.1.1 Nativní aplikace

Způsob, jenž musí napadnout snad každého kdo má s programováním alespoň trochu zkušeností je vyvinout aplikaci, která bude fungovat pouze na jedné platformě. Pro takovou platformu je vždy k dispozici alespoň jeden primární programovací jazyk a ve většině případů je poskytováno i SDK, které obsahuje sadu nástrojů pro ulehčení práce na dané platformě.

Jelikož se jedná o aplikace, jež jsou tvořeny právě pro danou platformu, mají zpravidla i přístup ke všem jejím funkcím (např. sdílené komponenty, senzory, specifické assembly, ...). Tohle je největší výhoda oproti ostatním typům, jelikož díky tomu jsou schopny maximalizovat

---

<sup>1</sup>Systémový software, Utility

výpočetní efektivitu. Jediné, co tedy výkon aplikace může snížit, jsou schopnosti programátora samotného.

Bohužel to, co získáme na výkonu a možnostech využití konkrétního zařízení, ztrácíme na portabilitě. Pro přenos na jinou platformu je tedy nutné, přepsat vysoce specializovaný kód, což se ne vždy vyplatí. Dále mezi nevýhody lze zařadit i nutnost instalace a ne vždy pro uživatele dostupnou nejaktuálnější verzi (nutnost ručně aktualizovat aplikaci).

Nativní aplikace můžeme dále dělit podle konkrétního typu cílové platformy:

### Desktopová aplikace

Je aplikace, jež běží na samostatném stolním počítači či notebooku, na kterém je nainstalována některá z dostupných distribucí OS.[3] Nejčastěji to bývají OS typu Windows, Linux, nebo MacOS. U desktopových aplikací je typická a nutná instalace na každé zařízení zvlášť. Největší výhodou oproti webovým aplikacím je jejich výkonost, neboť nejsou závislé na internetovém připojení a nemusí řešit problémy související se šířkou pásma.[4]

### Mobilní aplikace

Aplikace určená pro mobilní zařízení. Na těchto zařízeních najdeme nejčastěji mobilní OS Android, nebo iOS. Jejich hlavní nevýhodou oproti desktopovým aplikacím jsou často limitované zdroje, nutná konsiderace různých rozlišení obrazovek a časté problémy s nedostatečným připojením k internetu. Naopak přenosnost platformy samotné je jedna z velkých výhod.[5]

### Cloudové aplikace

Jsou poměrně novým členem mezi nativními aplikacemi. I přestože přesná definice cloudové aplikace je pořád nejasná, CNA by měly být vyvíjeny podle *principů CNA*:

- Provozovány na automatizačních platformách
- Využívají softwarizace a automatizace sítě
- Podpora migrace a interoperability napříč různými cloudovými infrastrukturami a platformami

Tyto principy nám umožňují budovat service-based *CNA architektury* (často obsahující self-contained DU) aplikací, které mají požadované *CNA vlastnosti*:

- Horizontální škálovatelnost
- Elasticita
- Odolnost
- Striktně konzistentní, případně alespoň konzistentní

Pro realizaci CNA existují *CNA metody*, jež jsou založeny na již existujících programovacích vzorech.[6]

#### 2.2.1.2 Hybridní aplikace

Hybridní aplikace často také označované termínem "*rich internet aplikace*" (RIA) vznikly kombinací webového a nativního přístupu. Mnohými je tento typ aplikace považován za podtyp webových aplikací.[7] Z mého pohledu si ale zaslouží vlastní kategorii. Svoje rozhodnutí odůvodňuji nesplněním definice webové aplikace uvedené v následující sekci, to jest nejedná se o aplikaci, ke které se přistupuje pomocí webového prohlížeče. Zároveň některé, nebo všechny části aplikace jsou uloženy na platformě klienta.

Tento přístup využívá web view komponenty<sup>2</sup> dané platformy, do které je následně načtena webová aplikace. Tedy aplikace se navenek tváří jako nativní aplikace pro danou platformu<sup>3</sup>, ale

<sup>2</sup>Komponenta zobrazující webové stránky bez nutnosti použití webového prohlížeče

<sup>3</sup>Wrapper



její jádro je stále napsáno pomocí postupů a technologií určených pro webové aplikace (JS, CSS, HTML, C#.NET ...) s využitím nativních API pro danou platformu.

Výhodami tohoto přístupu je portabilita mezi platformami, jelikož jediné, co musíme změnit, je wrapper. Dále nízké časové i finanční náklady na vývoj. Výhodou je samozřejmě i možnost offline přístupu při využití místního úložiště cílové platformy (v případě, že přístup k neaktuálním informacím není problémem).

Tento přístup jako každý, má i svoje nevýhody, a to především v nevyužití plného potenciálu platformy, na kterou je aplikace cílena. Neboli hybridní aplikace nejsou tak výkonné jako čistě nativní aplikace.[8] Mezi nevýhody patří, stejně jako u nativních aplikací, nutnost instalace a ruční aktualizace aplikace.

### 2.2.1.3 Webové aplikace

Webové aplikace jsou aplikace založené na client-server architektuře, která může být rozšířena do komplexnějších vícevrstevných, nebo SOA modelů tak, aby byly splněny požadavky pokročilých uživatelů. K aplikacím se přistupuje pomocí webovém prohlížeči na straně klienta a komunikace probíhá pomocí HTTP. U webových aplikacích se všechny komponenty nenacházejí na straně klienta a jsou staženy při každém spuštění aplikace.

Přínosem toho přístupu je nezávislost na typu platformy. Jediné, co uživateli stačí, je webový prohlížeč. Zároveň jelikož je aplikace je uložena na serveru odpadá nutnost instalace a uživatel má vždy k dispozici aktuální verzi.

Nevýhodami jsou nutnost připojení k internetu. Pomalejší odezva aplikace kvůli work-wait patternu. Zároveň odpadá možnost detailní konfigurace GUI.[8]

## 2.2.2 Rozdělení podle typu licence

Důležité dělení aplikačního softwaru je i podle typu jeho licence. Bez právního pohledu na problematiku aplikačního systému bychom nebyli schopni aplikace účinně chránit před neoprávněnou distribucí a užitím. Potřeba přiřazovat počítačovým programům, tedy i aplikačnímu softwaru, licenci vznikla až v roce 1968, kdy pod tlakem vlády USA začala společnost IBM distribuovat některé své počítačové programy bez nutnosti si zakoupit hardware IBM.[9]

### 2.2.2.1 Proprietární licence

Aplikační software, jenž je distribuován pod proprietární licencí, je software, kde autor upravuje jeho licenci pomocí EULA či jiným způsobem zasahuje do možností jeho používání. K takovému aplikačnímu softwaru zpravidla nebývá volně dostupný zdrojový kód, neboť se ve velké části jedná o komerční počítačový program, jež je vytvořen za účelem zisku. Má-li být rozsah licence upraven, pak je důležité ošetřit tři hlavní aspekty licence — územní, časový a množstevní. Licence na standardní proprietární software je často omezoována na území určitého státu.[10] Časový rozměr zahrnuje na jak dlouho je daný software prodáván. Software můžeme prodávat s neomezenou licencí<sup>4</sup>, nebo licencí časově omezenou, např. na dobu jednoho roku, u které je často nabízena možnost obnovy. Množstevní rozměr zahrnuje počet uživatelů, procesorů či jiných jednotek.[11]

### 2.2.2.2 Svobodné licence

Aplikací pod svobodnou licencí, lze označit takový aplikační software, jež může oprávněný uživatel použít za jakýmkoliv účelem. Studovat, jak pracuje, dále ho upravovat podle svých potřeb, šířit jeho kopie a vylepšovat ho s tím, že tato vylepšení může následně sdílet a dále šířit.[12] U aplikace se svobodnou licencí má tedy nabyvatel právo na přístup ke zdrojovému kódu a provádění jeho úprav, a dále ke spuštění a dalšímu šíření, a to bez dodatečných nákladů. V

<sup>4</sup>Časová doba užívání není nijak omezena

některých případech může být u aplikačního softwaru se svobodnou licencí spojena povinnost sdílet provedené úpravy s širší komunitou uživatelů<sup>5</sup>. [10] Pojem svobodná licence můžeme bez obav zaměnit s pojmem *open source* licence. Rozdíl mezi těmito pojmy není z právního [12] ani běžného hlediska zásadní. Svobodné licence můžeme rozlišovat podle různých kritérií, ale z praktického hlediska je však nejvýznamnějším kritériem dělení přítomnost již zmíněné copyleftové doložky. [10] S přihlédnutím k tomuto kritériu můžeme rozlišovat tři typy licenčních podmínek:

- 1. Silně copyleftové** Požadují, aby původní počítačový program i jakýkoliv jiný program, ve kterém se původní program vyskytuje, byť ve změněné podobě, byl šířen pod licenčními podmínkami původního programu, a současně garantují nabyvateli počítačového programu přístup ke zdrojovému kódu. [13] Cílem těchto podmínek je zachování otevřenosti zdrojového kódu odvozených počítačových programů, proto jsou charakteristickým příkladem *open source* licence.
- 2. Slabě copyleftové** Stejně jako silně copyleftové podmínky vyžadují sdílení odvozených počítačových programů pod stejnými licenčními podmínkami a zpřístupnění jejich zdrojových kódů. Umožňují však šíření vytvořených programů, které jsou propojené a šířené společně s původním programem, aniž by měnily jeho zdrojový kód pod libovolnou licenci. Typicky se jedná o softwarové knihovny a jiné prvky, jež jsou použity ve více počítačových programech. [13]
- 3. Necopyleftové** Jsou licence, jež neobsahují žádnou nebo obsahují velmi omezenou copyleftovou doložku a ukládají pouze minimální omezení ve vztahu k dalšímu nakládání s daným programem. Tudíž programy šířené pod touto licencí lze použít i v rámci vývoje proprietárního softwaru. [13]

## 2.2.3 Příklady aplikačního softwaru

### 2.2.3.1 Správce souborů

Byly vytvořeny za účelem poskytnutí rozhraní pro správu složek a souborů. Mezi nejčastěji podporované operace patří: *vytvoření, smazání, otevření, přejmenování, kopírování, přesunutí*, dále i *změna atributů souboru* a *změna přístupových práv*. V současné době jsou ve velké většině přímo součástí OS, nebo jsou zabudovány přímo do jeho prostředí, ale existují i samostatné na OS nezávislé distribuce. Někteří správci souborů podporují pouze CLI, ale samozřejmě najdeme i ty, jež implementují GUI. Mezi nejpoužívanější správce souborů patří samozřejmě nativní správce souborů OS a také aplikace Total Commander.

### 2.2.3.2 Kancelářské aplikace

Aplikační software používaný, jak již název napovídá, především pro práci v kanceláři. Většinou je součástí většího balíku programů, jež obsahuje *textový editor, tabulkový editor, databázový program, prezentační manager*, atd. Nejčastěji používanými kancelářskými balíky jsou Microsoft Office a OpenOffice.org.

### 2.2.3.3 CAx aplikace

Jsou aplikace využívané během různých fází životního cyklu produktu. Svoje využití nalézají v leteckém průmyslu a kosmonautice, automobilovém průmyslu, medicíně, energetickém průmyslu a ve spoustě další odvětvích. Jelikož se jedná o rozsáhlou oblast aplikací [14], jejíž detailní popis by pravděpodobně mohl být samostatnou prací, uvedu zde jen pár v současnosti existujících typů CAx aplikací.

---

<sup>5</sup>Copyleft licence

### CAD aplikace

Mohou být definovány jako aplikace, které pomáhají uživateli vytvořit přesný konceptuální model výrobku, bez toho, aniž by se prováděly jakékoliv dodatečné výpočty ohledně strukturální integrity daného výrobku. Tedy jedná se o nástroje pro vizualizaci konceptu. CAD aplikací existuje velké množství a podporují 2D i 3D modelování.

### CAE aplikace

CAE aplikace jsou velice podobné CAD aplikacím, ale mají jednu velkou výhodu. Jsou schopny provádět matematicko-fyzikální strukturální analýzu výrobku. Tudiž jsou velmi užitečným nástrojem v praxi, kde je nutno redukovat jakékoliv designové chyby ve struktuře. Dále díky nim lze simulovat změny na výrobku změněním určitých parametrů (např. materiál výrobku, proudění vzduchu, ...) a odpadá nutnost při každé změně či zničení vyrobit nový zkušební vzorek. Tyto aplikace využívají například metodu konečných prvků, dále výpočetní dynamiku tekutin, matematickou optimalizaci a mechaniku soustavy těles.

### CAM aplikace

Je aplikační software určený pro kontrolu obráběcích strojů při výrobě dílů. Tato definice není jediná, ale používá se nejvíce.[15] Využití těchto aplikací zjednodušuje změnu produkovaného výrobku, neboť stačí nahrát 3D model z CAD/CAE aplikace a také snižuje výrobní náklady, protože redukuje množství použitého materiálu na nutné minimum.

## 2.2.3.4 Animační a vizualizační software

Aplikace určené pro generování 2D či 3D modelů a jejich následnou animaci snímek po snímku. Tyto modely jsou dále používány v dalších aplikacích, především v zábavním softwaru, nebo i ve filmovém průmyslu. Aplikace zajišťuje preview modelu a následný rendering.

Většina moderních keyframe<sup>6</sup> animačních aplikací využívá stejného přístupu jako program BBOP[16], tj. využívají hierarchického přístupu ke kostrám objektů, interpolace keyframů v různých kanálech, aby každý kloub měl unikátní klíč přes všechny kanály, výběr interpolačních funkcí, přehrání animace v jakoukoliv dobu bez nutnosti renderingu a editor interpolací.[17]

Příklady animačních a vizualizačních systému jsou například Blender, Cinema 4D, Maya.

## 2.2.3.5 Zábavní software

Je veškerý aplikační software, jež slouží pro pobavení uživatele. Největším zástupcem jsou počítačové hry. Jelikož počítačové hry jsou jedním z největších zábavních průmyslů vůbec, není je třeba detailněji představovat. Jelikož jich existuje nepřehledné množství a druhů, nebudu je v této práci podrobněji popisovat.

## 2.2.3.6 Business management software

Jsou aplikace, jež jsou výkonné analytické a vykazovací nástroje, které umožňují využít firemní data nejen k analýze již proběhlých jevů, ale také k predikcím budoucího vývoje. Dále poskytují informace pro výkonné a vrcholové řízení v potřebném tvaru a určeném čase. Jedná se tedy o software určený pro získávání dat a informací o firmě v přehledné formě, jež může být dále zpracována člověkem.

<sup>6</sup>Jsou důležité animační framy, které obsahují informaci o počátečním/koncovém bodu akce

### 2.2.3.7 Simulační software

Simulační software je nástroj pro zobrazení simulace určitého jevu. Tyto aplikace jsou díky své podstatě velice specifické. Jejich naprogramování vyžaduje expertní znalost studovaného jevu a tudíž i spoustu zdrojů. V současné době se používají v odvětvích průmyslu, jež s sebou nesou vysoká rizika a chyba může mít rozsáhlé následky, ať již finanční, tak i na lidský životech. Proto je simulační software v současnosti nedílnou součástí jaderné energetiky, leteckého průmyslu, chemického průmyslu, akciových trhů a celkově finančního sektoru samotného. Ale průmysl není jedinou oblastí lidské činnosti, kde se simulace a simulační software používají. Svoje uplatnění našly i v molekulové fyzice, astrofyzice, biomechanice, městském plánování, meteorologii, atd. Lze říci, že bez tohoto typu aplikací by rychlost současného technologického a vědeckého pokroku nebyla ani zdaleka na takové úrovni.

## 3.1 Co je to API?

Application Programming Interface (API) poskytuje abstrakci problému a specifikuje, jak by klient měl interagovat se softwarovými komponentami, které implementují řešení pro danou úlohu. Samotné komponenty jsou typicky distribuované jako softwarové knihovny, což umožňuje jejich použití v různých aplikacích. Rozhraní API v podstatě definují znovupoužitelné stavební bloky softwaru, jež umožňují začlenit jednotlivé funkcionality do end-user aplikací. Hlavním konceptem API je, že se jedná o dobře navržený interface, který poskytuje specifickou funkcionalitu ostatním částem aplikace.

Moderní aplikace jsou často postaveny s využitím mnoha API, kde některé mohou být dále závislé na jiných API. Například aplikace pro zobrazování map může využívat API pro načtení mapy a tato samotná API může využívat jiných API pro kompresi a dekompresi potřebných dat.[18]

V současné době se API používají především pro osvobození programátora od nutnosti programování rutinních, často používaných struktur a procesů.[19] Dále nabízejí implementaci složitých konceptů, u kterých může získání dostatečné znalosti pro jejich naprogramování trvat měsíce, či dokonce i roky.[20] Díky tomu urychlují proces vývoje, neboť těchto konceptů potom může využít i programátor, jenž o nich má jen základní povědomí, nebo nemá na jejich implementaci dostatečnou zkušenost s vybraným programovacím jazykem. V neposlední řadě je jejich výhodou i znovupoužitelnost.[21] Programátor může mít dostatečné dovednosti pro naprogramování vybraného konceptu, ale jeho implementace zabere spoustu času. Tudíž se rozhodne využít API, pokrývající tento koncept a opět se urychlí proces vývoje konečné aplikace.

## 3.2 Příklady API

### 3.2.1 Staticky linkované API

Hlavním rysem staticky linkovaných API je jejich zkopírování do cílové aplikace v době kompilace pomocí kompilátoru, linkeru, nebo binderu.[22] Následně jsou tyto API buďto spojeny s ostatními staticky linkovanými objekty do jednoho objektového souboru, jenž vytvoří jeden spustitelný soubor, nebo jsou načteny až během běhu do adresového prostoru aplikace na korespondující statický offset, jež byl rozhodnut v době kompilace.

### 3.2.1.1 API operačního systému

Jedná se o interface mezi aplikací a službami operačního systému.[23] Daná API tedy popisuje správný postup přístupu ke službám OS, nebo k informacím z jiných aplikací běžících na daném systému. Interface pro systémové služby se skládá ze dvou částí:[24]

#### 1. Rozhraní pro vyšší programovací jazyky

- Provádí operace v uživatelské módu
- Implementováno pro akceptaci procedurálního volání
- Následně musí volat kernelovou část rozhraní

#### 2. Kernelová část

- Provádí operace v systémovém módu
- Implementuje systémové služby
- Může způsobit blokování volajícího
- Po ukončení vrací uživateli zprávu o provedení

Mezi API operačního systému patří například:

#### POSIX API

Je IEEE standardizovaná API. Této API využívají Unix, nebo i některé Unix-like operační systémy. [25] Jelikož se jedná o IEEE standard, je tato API velmi dobře dokumentována a neobsahuje žádné skryté části. V době psaní práce je nejnovější verze standardu IEEE Std 1003.1-2017.

#### Windows API

Je API od společnosti Windows, jež si klade za cíl udržovat zpětnou kompatibilitu aplikací spustitelných na Windows OS. Tato API není plně popsána a obsahuje skryté systémové volání a systémové funkcionality.[24] Verze API:

- **Win16** – 16bit verze pro Windows 3.1
- **Win32** – 32 bit verze pro WindowsNt a dále
- **Win32 pro 64-bit Windows** – 64 bitová verze Win32, hlavní změnou je pouze typ paměťových pointerů[24]

### 3.2.1.2 Grafické API

Byly vytvořeny za účelem pomoci při renderování počítačové grafiky. Toto typicky zahrnuje poskytování optimalizovaných verzí algoritmů, které stojí za běžnými vykreslovacími úlohami. Pokud má daný systém přístup k GPU, jsou schopny využít i hardwarové akcelerace.

#### DirectX

Je kolekce několika API pro správu úkonů týkajících se počítačových her, multimediálních aplikací, ale i grafického uživatelského rozhraní na Microsoft platformách. DirectX je zkratkou pro všechny zastřešené API, kde X nahrazuje "zaměření knihovny". Mezi tyto API patří například Direct3D, Direct2D, DirectSound, atd. V současnosti je nejaktuálnější verzí této API *DirectX 12 Ultimate*, jež byla vydána v roce 2020. Tato verze podporuje ray-tracing, variable rate shading<sup>1</sup>, sampler feedback<sup>2</sup> a mesh shading<sup>3</sup>. [26]

<sup>1</sup>Technologie renderování upravující míru detailů v různých částech scény, například při použití ve VR je vykreslováno více detailů tam, kam se oči uživatel právě dívají a méně detailů v okrajových částech obrazu

<sup>2</sup>Zlepšuje výpočetní procesy mezi texturou a hardwarem shaderu, dále lze použít k implementaci prostorového stínování textur nebo k implementaci sofistikovaných algoritmů pro streamování textur

<sup>3</sup>Inteligentní výběr úrovně detailů a teselace různých objektů scény

## OpenGL

Obsahuje několik stovek procedur a funkcí, jež umožňují programátorovi specifikovat objekty a operace spojené s vykreslováním grafických obrázků, specificky barevných obrázků a 3D objektů. Velká část OpenGL požaduje po grafickém hardwaru, aby obsahoval framebuffer<sup>4</sup>. Samotné OpenGL obsahuje volání týkající se vykreslování objektů jako jsou body, linie a polygony, ale ve speciálních případech (např. pokud je povolen antialiasing<sup>5</sup>) je závislé na existenci framebufferu. Kromě toho některé části jsou přímo zaměřeny na manipulaci s framebufferem.

Z pohledu programátora se jedná o soubor příkazů, které umožňují specifikaci geometrických objektů ve 2D nebo ve 3D, společně s možností specifikovat, jak jsou tyto objekty renderovány do framebufferu.

Typický program využívající OpenGL začíná voláním k otevření okna do framebufferu, jež bude program využívat pro vykreslování. Potom se GL kontext alokuje a asociuje s daným oknem. Jakmile je GL kontext alokovan, může programátor začít používat OpenGL příkazy.[27]

## 3.2.2 Webové API

Po vzoru práce autorů Tiaga Espinhaa, Andyho Zaidmana a Hanse-Gerharda Grosse[21] využijí pro definici webových API definici pro webové služby, kterou ve své práci používají také Alonso et al.[28], tj. *”Softwarová aplikace identifikovatelná pomocí URI, jejíž interface a vazby je možné definovat, popsat jako XML artefakty.”*s jednou malou změnou, odpustíme od restrikce na technologii XML a povolíme i její možné alternativy (např. JSON). Tato definice přesto nemusí být jediná správná, neboť různé technologie s sebou nesou různé překážky, s nimiž se musí vývojáři potýkat a tudíž i konečná implementace webové API se pro každý problém liší. V této práci jsem si pro popis vybral tři hlavní implementace popsané v následujících podsekcích.

### 3.2.2.1 SOAP API

SOAP poskytuje distribuovaný procesní model pro výměnu strukturovaných informací v decentralizovaném prostředí s využitím XML, jež předpokládá, že SOAP zpráva vznikla u počátečního odesílatele (*SOAP sender*) a je odeslána konečnému příjemci (*SOAP receiver*) přes neurčený počet prostředníků (*SOAP intermediaries*). Takovýto model může podporovat mnoho SOAP vzorů výměny zpráv. Rámec byl navržen tak, aby byl nezávislý na konkrétním programovacím modelu a jiných implementačních specifikách aplikace, jež využívá SOAP. Předtím než dojde k samotnému volání SOAP API si klient vyžádá WSDL, jež definuje, které metody mohou být použity, a zároveň jaké datové typy webová SOAP API očekává.

## Elementy SOAP architektury

### 1. Protokolové elementy

#### Uzel

Ztělesnění logiky nezbytné pro přenos, příjem, zpracování a/nebo předání SOAP zpráv. Může být inicializována odesílatelem, koncovým příjemcem, a nebo prostředníkem. Uzel musí přijatou zprávu zpracovat dle SOAP modelu zpracování. Uzel je indetifikován pomocí URI. [29]

#### Role

Očekávaná funkce příjemce SOAP při zpracování zprávy. Příjemce může vystupovat pod více rolemi. Důvod existence rolí je identifikace uzlu, nebo skupiny uzlů. Až na

<sup>4</sup>Součástí RAM, jenž obsahuje bitmapu, která řídí vykreslení obrazu

<sup>5</sup>Výhlazování hran

tři hlavní typy rolí (*next*<sup>6</sup>, *none*<sup>7</sup>, *ultimateReceiver*<sup>8</sup>), pro ně neexistuje jmenná konvence, či kritéria, podle kterých daný uzel určuje sadu rolí, v nichž bude jednat s danou zprávou.[29]

### Vazba

Formální soubor pravidel pro přenos zprávy SOAP v rámci jiného protokolu za účelem výměny. Mezi příklady vazeb protokolu SOAP patří přenášení zpráv v rámci těla protokolu HTTP nebo přes TCP. Vazba jako taková neposkytuje procesní model a nepředstavuje uzel. Specifikace vazby musí obsahovat:

- Deklaraci funkcí poskytovaných vazbou
- Popis přenosu infosetu zprávy pomocí vybraného protokolu
- Popis, jak se služby vybraného protokolu používají k dodržení kontraktu vytvořeného funkcemi dané vazby
- Popis zacházení se všemi možnými vyvolanými chybami
- Definuje požadavky na vytvoření vyhovující implementace specifikované vazby

[29]

### Featura

Rozšíření SOAP frameworku pro zasílání zpráv. SOAP neklade žádná omezení na jejich možný rozsah.

Model rozšiřitelnosti SOAP poskytuje dva mechanismy, jejichž prostřednictvím lze vyjádřit rozšíření: procesní model a framework vazeb protokolu SOAP. První popisuje chování uzlu SOAP při zpracování jednotlivých zpráv. Druhý zprostředkovává odesílání a přijímání zpráv uzlem prostřednictvím základního protokolu.

Požadavky na specifikaci featury:

- K pojmenování je použita URI
- Informace o stavu v každém z uzlů
- Informace ke zpracování v každém z uzlů, včetně zpracování chyb
- Informace, jež má být přesunuta mezi uzly

[29]

### Modul

Specifikace, která obsahuje kombinovanou syntaxi a sémantiku bloků záhlaví. Modul realizuje nula a nebo více featur. Specifikace modulu se řídí následujícími pravidly:

- Musí sám sebe identifikovat pomocí URI.
- Musí deklarovat poskytovanou featuru.
- Musí jednoznačně a kompletně popsat obsah a sémantiku head bloků, jež jsou použity pro implementaci cíleného chování, pokud dojde k modifikaci procesního modelu musí uvést i tyto změny.
- Může použít následující konvence:
  - Vlastnosti jsou pojmenovány pomocí URI
  - Všude, tam kde je to vhodné, by vlastnosti měly mít uveden svůj XML typ schématu ve specifikacích, jež danou vlastnost popisují

Pokud jsou tyto konvence dodržovány, specifikace modulu musí jasně popisovat vztah mezi abstraktními vlastnostmi a jejich reprezentacemi v obálce.

- Musí jasně specifikovat všechny známé interakce se SOAP tělem nebo změny jeho interpretace. Dále musí jasně specifikovat všechny známé interakce s ostatními featurami a moduly nebo změny jejich interpretace.

[29]

<sup>6</sup>Každý SOAP prostředník a koncový příjemce musí zastávat tuto roli

<sup>7</sup>SOAP uzli nesmí zastávat tuto roli

<sup>8</sup>koncový příjemce SOAP musí zastávat tuto roli



### Vzor výměny zpráv (MEP)

Šablona pro výměnu zpráv mezi uzly, kterou umožňuje jedna nebo více vazeb na základní SOAP protokol. Specifikace vzoru musí obsahovat:

- Vlastnosti jsou pojmenovány pomocí URI
- Popis životního cyklu výměny zpráv podle vzoru
- Popis případných časových/příčinných vztahů více zpráv vyměňovaných v souladu se vzorem
- Popis normálního a abnormálního ukončení výměny zpráv podle vzoru

Samotný protokol může využívat i více pojmenovaných MEP služeb. Jelikož MEP je SOAP featura, její specifikace musí splňovat stejné požadavky. Zároveň musí navíc obsahovat:

- Případné požadavky na generování dalších zpráv (například odpovědi na požadavky v request/response MEP)
- Pravidla pro doručování nebo jiné nakládání s chybami generovanými během provozu MEP

[29]

### SOAP Aplikace

Subjekt, obvykle software, který vytváří, spotřebovává nebo jinak pracuje se SOAP zprávami způsobem odpovídajícím SOAP modelu zpracování.[29]

## 2. Elementy zapouzdření dat

### Zpráva

Základní jednotka komunikace mezi uzly. Je definována jako XML infoset, jehož informace o komentářích, elementech, attributech, jmenných prostorech a znacích lze serializovat jako XML 1.0. Infoset zprávy se skládá z informačního prvku dokumentu s přesně jedním členem ve vlastnosti [children], což musí být informační element obálka. Tento informační element je zároveň hodnotou vlastnosti [document element]. Vlastnosti [notations] a [unparsed entities] jsou obě prázdné. XML infoset zprávy nesmí obsahovat informační položku s prohlášením o typu dokumentu.

Zpráva poslána prvotním odesílatelem nesmí obsahovat informační položky pokynů ke zpracování. Tyto položky zároveň nesmí být přidány ani prostředníky. Všechny zprávy, jež tuto informační položku obsahují a dorazí k příjemci musí vyhodit chybu.[29]

### Obálka

Nejvrchnější *informační element* zprávy. Součástí obálky je [local name], jež popisuje jméno obálky, [namespace name], což je jméno jmenného prostoru. Dále obsahuje libovolný počet atributů daného jmenného prostoru pod vlastností [attributes]. A také jeden nebo dva informační elementy ve vlastnosti [children], které mají dané pořadí:

1. Nepovinný element *záhlaví*
2. Povinný element *těla*

[29]

### Záhlaví

Kolekce žádných nebo více bloků záhlaví, z nichž jakýkoliv může být použit příjemcem, jež je součástí cesty zprávy. Poskytuje mechanismy pro decentralizované a modulární rozšíření zprávy. Obsahuje vlastnost [local name], která slouží k pojmenování elementu, [namespace name] pro pojmenování jmenného prostoru. Dále obsahuje žádný a nebo více atributů daného jmenného prostoru pod vlastností [attributes] a žádný a nebo více informačních elementů, které jsou uloženy ve vlastnosti[children].[29]

### Blok záhlaví

*Informační element*, jež slouží k ohraničení dat, které tvoří jednu výpočetní jednotku v elementu záhlaví. Jeho typ je popsán pomocí XML. Musí mít neprázdnou vlastnost [namespace name]. Dále může mít jakýkoliv počet informačních položek o charakterech a

informační elementů. Tyto elementy nemusí mít vlastní jmenný prostor. A zároveň může mít nula a více informačních položek ohledně atributů pod vlastností [attributes].[29]

### Tělo

Kolekce žádných nebo více *informačních elementů*, jež jsou použity koncovým příjemcem, jež je součástí cesty zprávy. Obsahuje vlastnosti [local name] popisující jeho jméno, [namespace name] jméno jmenného prostoru, žádný a nebo více atributů daného jmenného prostoru, pod vlastností [attributes] a žádný a nebo více informační elementů ve vlastnosti [children]. Dále může mít jakýkoliv počet informačních položek týkajících se znaků.[29]

### Chyba

*Informační element*, jež obsahuje popis, který je generován uzlem. Opět tento element musí obsahovat vlastnosti [local name] a [namespace name], jež popisují to samé jako u předchozích elementů a dva nebo více child elementů ve vlastnosti [children], kteří musejí mít následující pořadí:

1. Povinný element *kódu*
2. Povinný element *důvodu*
3. Nepovinný element *uzlu*
4. Nepovinný element *role*
5. Nepovinný element *detailu*

Aby zpráva byla rozpoznána jako chybová, musí obsahovat chybový element jako jediného potomka.[29]

## 3. Elementy výměny zpráv

### Odesílatel

Uzel, který odesílá zprávy.

### Příjemce

Uzel, který přijímá zprávy.

### Cesta zprávy

Množina uzlů, které po své cestě SOAP zpráva navštíví. Včetně odesílatele a koncového příjemce.

### Počáteční odesílatel

Odesílatel, stojící na počátku cesty zprávy.

### Prostředník

Jedná se o odesílatele a příjemce, přeposílajícího SOAP zprávy k jejich koncovému příjemci.

### Koncový příjemce

Příjemce, jež je cílovou destinací SOAP zprávy. Je zodpovědný za zpracování těla a bloků záhlaví dané zprávy. V některých případech ale zpráva nemusí dorazit, neboť se může vyskytnout problém na jednom z prostředníků.

## SOAP zprávy

Základní zpracování zpráv a chyb musí být sémanticky ekvivalentní provedení následujících kroků samostatně a v uvedeném pořadí.

1. Určí se sada rolí, pod kterými bude uzel jednat. Při tomto určování může být kontrolován obsah obálky včetně těla a všech bloků záhlaví.
2. Identifikují se všechny povinné bloky záhlaví zaměřené na uzel
3. Pokud uzel nerozumí jednomu nebo více blokům záhlaví identifikovaných v předchozím kroku, vygeneruje chybu. Pokud je taková chyba vygenerována, nesmí se provádět žádné další zpracování. V tomto kroku nesmí být generovány chyby týkající se obsahu těla.

4. Zpracují se všechny povinné bloky záhlaví určené uzlu a v případě konečného příjemce i tělo. Uzel se může rozhodnout zpracovat i nepovinné bloky záhlaví, které jsou na něj zaměřeny.
5. V případě zprostředkovatele a v případě, kde vzor výměny zpráv a výsledky zpracování (např. žádná generovaná chyba) vyžadují, aby byla zpráva SOAP odeslána dále po cestě zprávy SOAP, se zpráva předává dále.

Ve všech situacích, kdy je zpracováno záhlaví, musí uzel záhlaví rozumět a musí zpracování provádět způsobem, který je plně v souladu se specifikací pro daný blok záhlaví.

Uzly mohou při zpracování těla nebo bloku záhlaví odkazovat na jakékoli informace v obálce. Například funkce ukládání do mezipaměti může v případě potřeby uložit do mezipaměti celou zprávu. [29]

Příklad SOAP zprávy najdeme ve výpisu kódu 3.1.

### ■ 3.1 Příklad zprávy SOAP, která obsahuje SOAP záhlaví a SOAP tělo

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
      <n:priority>1</n:priority>
      <n:expires>2001-06-22T14:00:00-05:00</n:expires>
    </n:alertcontrol>
  </env:Header>
  <env:Body>
    <m:alert xmlns:m="http://example.org/alert">
      <m:msg>Pick up Mary at school at 2pm</m:msg>
    </m:alert>
  </env:Body>
</env:Envelope>
```

### 3.2.2.2 REST API

Byl poprvé popsán Royem Thomasem Fieldingem v roce 2000[30]. Na rozdíl od distribuovaného objektového stylu, kde jsou všechna data zapouzdřena a skryta uvnitř zpracovávaných komponent, je povaha a stav datových prvků klíčovým aspektem REST. Důvody pro tento návrh lze spatřovat v povaze distribuovaných hypermédií, u kterých, pokud je zavolána adresa, musí být informace přesunuta ze svého úložiště ke klientovi.

REST neomezuje komunikaci na konkrétní protokol, ale omezuje rozhraní mezi komponentami, a tím i rozsah interakce a předpoklady implementace, které by jinak mohly být mezi komponentami učiněny.

#### Elementy REST Architektury

##### 1. Datové elementy

V době prvního popisu konceptu REST měl architekt distribuovaných hypermédií pouze tři základní možnosti, jak nakládat s daty:

1. Vykreslit data tam, kde se nacházejí, a poslat příjemci obraz v pevném formátu. Toto je klasický příklad *client-server* architektury[31], jež umožňuje skrýt pravou podstatu dat v odesílateli, což zabraňuje vytváření mylných předpokladů o struktuře dat a usnadňuje implementaci na straně klienta.
2. Zapouzdřit data pomocí vyskreslovacího enginu a poslat příjemci obojí – data i obraz. Jedná se tedy o příklad *mobilně objektového* stylu[32], poskytující skrytí informace a zároveň umožňující specifické zpracování dat díky existenci rendering enginu. Tento případ ale značně limituje možnosti zpracování na straně příjemce, jelikož je omezen limitacemi použitého enginu, a tím pádem značně zvýšit objem přenášených dat.

3. Poslat příjemci surová data spolu s metadaty, která popisují typ dat. Příjemce si může vybrat vlastní vykreslovací engine. Tento způsob umožňuje udržovat odesílaný formát jednoduchý a dále rozšiřitelný, ale ztrácí schopnost skrýt odesílanou informaci a vyžaduje po odesílatelovi a příjemci rozumět stejnému datovému typu.[30]

Koncept REST je hybridem mezi těmito přístupy, díky zaměření na sdílené porozumění datových typů s metadaty, ale omezuje rozsah informací, jež jsou odhaleny standardizovanému rozhraní. RESTové komponenty mezi sebou komunikují přenášením prostředků<sup>9</sup> ve formátu zvoleném podle možností odesílatele a příjemce a povahy prostředku. Informace o tom, zda je reprezentace dat ve stejném formátu jako jejich původní zdroj, nebo zda je formát ze zdroje pouze odvozen, zůstává skryta. REST se benefitům mobilně objektového stylu se přibližuje díky odeslání reprezentace, jež se skládá z instrukcí, které jsou ve standardním formátu zapouzdřeného vykreslovacího enginu (např. Java).[30] REST tedy poskytuje oddělení zájmů po vzoru client-server architektury bez problému škálovatelnosti serveru, zároveň umožňuje zapouzdření a vývoj poskytovaných služeb.

### Prostředek a Identifikátory prostředku

Hlavní abstrakcí informace v RESTu je *prostředek*. Prostředkem může být jakákoliv informace: dokument, obrázek, kolekce dalších prostředků, atd. Přesněji prostředek  $R$  je v čase se měnící funkce příslušnosti  $M_R(t)$ , která pro čas  $t$  tvoří množinu entit, nebo hodnot, jež jsou ekvivalentní. Hodnoty v této množině jsou nazývány *identifikátor prostředku*, nebo *reprezentace prostředku*.

REST používá identifikátory prostředku k identifikaci konkrétního prostředku při interakci mezi REST komponentami. REST konektory poskytují generické rozhraní pro přístup a manipulaci množiny prostředků bez ohledu na to, jak je funkce příslušnosti definována, nebo jaký software požadavek zpracovává. Autorita, jež pojmenovává identifikátory prostředku, je zodpovědná za udržení sémantické korektnosti mapování v čase.[30]

### Reprezentace

REST komponenty provádějí úkony s prostředkem pomocí reprezentace, jež zachycuje současný nebo plánovaný stav prostředku. Tuto reprezentaci následně posílají dále mezi sebou. Reprezentace je sekce bytů s přiřazenými metadaty pro popis těchto bytů. Občasné i tato metadata mají přiřazena metadata vlastní, obvykle za účelem ověření integrity zprávy. Odpověď může obsahovat metadata reprezentace i metadata prostředku. V tomto případě jsou metadata informacemi o prostředku, které nejsou specifické pro dodanou reprezentaci. Ostatní často užívané ale méně přesné označení pro reprezentaci je: dokument, soubor a entita HTTP zprávy.

Řídící data definují účel zprávy mezi komponentami, například požadovanou akci nebo význam odpovědi. Používají se také k parametrizaci požadavků a k potlačení výchozího chování některých spojovacích elementů. Řídícími daty obsaženými ve zprávě požadavku nebo odpovědi lze například změnit chování mezipaměti.[30]

## 2. Konektory

Konektor představuje abstraktní rozhraní pro komunikaci komponent zvyšující jednoduchost tím, že zajišťuje čisté oddělení jednotlivých problémů a skrývá základní implementaci prostředků a komunikačních mechanismů.

Všechny REST interakce jsou bezstavové. To znamená, že každý požadavek obsahuje všechny potřebné informace pro jeho vykonání konektorem.

REST využívá různých typů konektorů pro zapouzdření aktivit spojených s přístupem a přesunem prostředků. Tyto typy jsou:

- client konektor

<sup>9</sup>V anglickém textu *resource*, v tomto textu vybráno jako překlad slovo prostředek, díky tomu nedochází k záměně se slovem zdroj, anglicky *source*

- server konektor
- cache konektor
- resolver konektor
- tunnel konektor

Primárními typy konektorů jsou *client* a *server*. Zásadním rozdílem mezi nimi je, že *client* započiná komunikaci odesláním žádosti, zatímco *server* naslouchá spojení a odpovídá na požadavky. REST komponenta může obsahovat oba typy konektorů.

Třetím typem je *cache* konektor umístěn na rozhraní u konektoru typu *client*, nebo *server* za účelem uložení odpovědi na aktuální interakce do mezipaměti, aby mohly být znovu použity pro později požadované interakce. Mezipaměť je obvykle implementována v adresním prostoru konektoru, který ji používá. V některých případech jsou *cache* konektory sdílené, což znamená, že jejich odpovědi uložené v mezipaměti mohou být použity v odpovědi pro jiného klienta, než pro kterého byla odpověď původně získána.

Čtvrtý typ konektoru *resolver* převádí částečné nebo úplné identifikátory prostředků na informace o síťové adrese potřebné k navázání spojení. Například většina URI obsahuje DNS jméno jako mechanismus pro identifikaci jmenné autority pro daný prostředek. Pro zahájení požadavku webový prohlížeč získá z URI název hostitele a pomocí resolveru DNS získá adresu internetového protokolu této autority.

Poslední formou typu konektoru je *tunnel*, který jednoduše přenáší komunikaci přes nějakou hranici připojení, například firewall nebo lower-level síťovou bránu. Jediným důvodem, proč je modelován jako součást REST a není abstrahován jako součást síťové infrastruktury, je, že některé aktivní komponenty REST mohou dynamicky přecházet ze svého defaultního chování na chování *tunnel* konektoru.[30]

### 3. Komponenty

Mezi REST komponenty patří:

- origin server komponenta
- gateway komponenta
- proxy komponenta
- user agent komponenta

*User agent* komponenta využívá *client* konektoru pro vyvolání žádosti čím se stane i koncovým příjemcem odpovědi. Nejčastějším případem *user agent* komponenty je webový prohlížeč, který poskytuje přístup k informačním službám a renderuje jejich odpovědi podle požadavků aplikace.

*Origin server* využívá serverového konektoru pro správu jmenného prostoru vyžádaného prostředku. Jedná se o hlavní zdroj pro reprezentaci jeho prostředků a zároveň musí plnit roli konečného příjemce jakoukoliv žádost o změnu prostředků, jež se na origin serveru nacházejí. Každý *origin server* poskytuje rozhraní pro jeho služby jako hierarchii zdrojů.

Zprostředkovací komponenty *gateway* a *proxy* fungují jako klient i server zároveň, díky tomu mohou předávat a překládat procházející žádosti a odpovědi. *Proxy* komponenta je prostředník vybraný klientem, který zajišťuje zapouzdření rozhraní jiných služeb, překlad dat, zvýšení výkonu nebo ochranu zabezpečení. *Gateway* komponenta (tzv. *reverzní proxy*) je prostředník, který je určen síťovým nebo výchozím serverem k zajištění zapouzdření rozhraní jiných služeb, k překladu dat, zvýšení výkonu nebo zajištění bezpečnosti.[30]

### Datový pohled na REST architekturu

Datový pohled na architekturu popisuje stavy REST aplikace při cestě informace komponentami. Vzhledem k tomu, že REST je specificky zaměřen na distribuované informační systémy, nahlíží na aplikace stejně jak je uvedeno v její definici v kapitole č. 2, jen s jednou malou změnou, neboť aplikace má povoleno vykonávat úkony nad zařízením samotným.

Interakce komponent se uskutečňují ve formě velikostně odlišených zpráv. Malé nebo středně velké zprávy se používají pro řídicí sémantiku, ale většina práce aplikace se provádí prostřednictvím velkých zpráv obsahujících kompletní reprezentaci prostředků. Nejčastější formou sémantiky požadavků je načítání reprezentace prostředku (např. metoda "GET" v protokolu HTTP), kterou lze často uložit do mezipaměti pro pozdější opětovné použití.

REST shromažďuje veškerý řídicí stav do reprezentací přijatých v odpovědi na interakce. Cílem je zlepšit škálovatelnost serveru odstraněním nutnosti udržovat povědomí o stavu klienta nad rámec aktuálního požadavku serverem. Stav REST aplikace je tedy definován jejími nevyřízenými požadavky, topologií připojených komponent, aktivními požadavky na konektorech, datovým tokem reprezentací v reakci na požadavky a zpracováním těchto reprezentací, jak je přijímá uživatelský agent.

REST aplikace dosáhne ustáleného stavu vždy, když již nemá žádné požadavky na zpracování, tj. nemá žádné nevyřízené požadavky a všechny odpovědi na aktuální sadu požadavků byly zcela přijaty nebo přijaty do té míry, že je lze považovat za reprezentativní datový tok. U webového prohlížeče tento stav odpovídá načtené webové stránce.

Další řídicí stav REST aplikace je součástí reprezentace prvního požadovaného prostředku. Tudíž získání této reprezentace je prioritou. REST tedy používá protokoly, jež "nejdříve odpovídají a pak přemýšlí". Tedy jedná se o protokoly, které vyžadují několik interakcí pro každou akci klienta.

Stav REST aplikace řídí a ukládá uživatel a může se skládat z reprezentací z více serverů. [30]

# Programovací jazyky

## 4.1 Co je to programovací jazyk?

Programovací jazyk je formální zápis pro specifikaci výpočtu, algoritmu či provedení specifických operací (např. vstup/výstup,...). Tedy jedná se o notaci pro vytváření počítačových programů. Programovací jazyk je většinou definován svojí *syntaxí*<sup>1</sup> a *sémantikou*<sup>2</sup>. Ale může i hodnotit, jak je daný programovací jazyk pragmatický, tj. s jakou mírou úspěšnosti splňuje své cíle z hlediska věrnosti výpočetnímu modelu a zároveň z hlediska užitečnosti pro programátora.[33]

## 4.2 Programovací paradigmatata

V této sekci se zaměřím na popis významu pojmu programovací paradigma a následně představím v současné době nejvíce používané příklady programovacích paradigmat.

K vysvětlení celého pojmu je nejdříve zapotřebí vysvětlit pojem *paradigma*. Paradigma označuje příklad, vzor, v teorii o vědách se jedná o souhrn všech pojetí vědní disciplíny v určitém časovém úseku.[34] Díky této definici můžeme následně zadefinovat *programovací paradigma*, jako soubor programovacích příkladů a vzorů zastřešujících určitý myšlenkový směr psaní softwarového kódu. Pojem programovací paradigma můžeme také nahradit pojmem *programovací styl*.

### 4.2.1 Imperativní a procedurální programování

#### Strukturované programování

Pojem strukturovaného programování poprvé světu představil Edsger W. Dijkstra[35]. Strukturované programování je založeno na následujících principech:[36]

1. Program může být vyjádřen pomocí tří typů dekompozice zřetězení, výběru a opakování, které mají jeden vstup a jeden výstup. Omezení dekompozice programu pouze na tyto tři typy vede k diagramům s omezenou typologií, pokud je porovnáme s diagramy, kde lze vést šipky mezi jakýmkoliv bloky. V porovnání s touto větší volností, omezení se na výše zmíněné klauzule představuje sekvenční disciplínu.
2. Využití top-down principu. Tj. nejprve je úkol rozdělen do několika fází, následně v dalším kroku jsou tyto fáze znovu rozděleny. Toto dělení pokračuje do té doby, než všechny fáze

<sup>1</sup>Soubor pravidel, která definují kombinaci symbolů, které jsou považovány za správně strukturovaný dokument nebo fragment v tomto jazyce

<sup>2</sup>Význam syntakticky platných řetězců v daném programovacím jazyce, včetně jejich výpočtu.

programu nejsou pospány pomocí základních příkazů programovacího jazyka. V každém kroku je provedeno pouze jedno rozhodnutí. Tento přístup umožňuje větší modularitu během vývoje.

3. Důkaz správnosti programu. "Testování programu může být použito k prokázání přítomnosti chyb, ale nikdy ne k prokázání jejich nepřítomnosti." [37] Pro prokázání nepřítomnosti chyb v programu, je nutné dokázat větu, že program zpracovává libovolná vstupní data do očekávaného výstupu. Program by měl být strukturovaný, to znamená, že by měl zahrnovat pouze struktury pořadí, výběru a opakování. Důkaz používá matematické aparáty, včetně matematické indukce.

První dva principy se podařilo prakticky beze změny adoptovat tímto i následujícími programovacími paradigmaty. Třetí princip se zatím nepodařilo prakticky aplikovat, neboť neexistuje efektivní způsob, jak důkaz pro rozsáhlé programy provést, tudíž je zastáván jen v teoretické rovině. V reakci na tuto situaci je zastáváno doporučení, že by se mělo co nejvíce využívat familiárních struktur, u kterých je jejich funkčnost jasná na první pohled a není nutno ji dokazovat.

Z těchto principů vycházejí tedy i omezení strukturovaných jazyků používat pouze sekvence, selekce a cykly. Společně s nutností vytvářet program pomocí postupného skládání.

Pro zkrácení programů a také z důvodu členění kódu byly vytvořeny tzv. *rutiny*. Data se do rutin přenášejí jak prostřednictvím parametrů tak i za pomoci globálních proměnných. Jejich použití není nutné pro dodržení strukturovaného paradigmatu.

### Procedurální programování

Procedurální styl se objevil, když rozdělení programu do rutin, aby se zabránilo duplicitě kódu, přestalo být doporučením a stalo se požadavkem. Procedurální styl je založen na konceptu *procedur*, taky známém jako *rutiny*. Každá procedura může být zavolána v jakémkoliv kroku programu a také zavolána jinou procedurou. Procedurální programování využívá konceptu úrovní abstrakce. [38] Každá úroveň abstrakce má svoji skupinu funkcí. Úroveň abstrakce může nakládat pouze se svými funkcemi, či s funkcemi z nižších úrovní.

Procedurální styl je rozšíření strukturovaného paradigmatu, tedy je jeho podtypem a obohacuje tento styl o následující principy: [35]

1. Funkce jednotlivých úrovní abstrakce jsou popsány pomocí rutiny
2. Každá rutina vykonává jednu funkci ve své vlastní úrovni abstrakce
3. Rutina je řídicí struktura s jedním vstupem a jedním výstupem, přijímající vstupní data prostřednictvím parametrů. Výsledek vrací prostřednictvím parametrů nebo příkazu *return*

Rutiny, nebo také procedury zajišťují modularitu programu, rozdělují program do více úrovní abstrakce a omezují duplicitu kódu.

Mezi příklady procedurálních programovacích jazyků patří:

- FORTRAN
- ALGOL
- COBOL
- BASIC
- Pascal
- C

### Imperativní programování

Imperativní styl programování, je založen na konceptu změny vnitřního stavu aplikace, tento



stav je dán obsahem paměti. Strukturované a procedurální programování jsou typy imperativního programování. V některých knihách (např.[39]) je imperativní programování synonymem pro procedurální programování. Jedná se o dominantní paradigma, které je přirozené pro technologie používající von-Neumanovu architekturu a odpovídá základnímu modelu výpočtu Turingova stroje.[40]

## 4.2.2 Objektově orientované programování

První zmínku o OOP jsou přibližně ze stejné doby, kdy byl světu představen koncept procedurálního programování, ale svoji popularitu OOP získalo především až v 90.letech[35]. OOP je programovací paradigma, které využívá konceptu *objektu*. Každý objekt by měl obsahovat procedury, jež jsou zodpovědné za to, jak objekt reaguje na příchozí podmínky. Přístup je založen na identifikaci objektů daného problému a jejich následné charakterizaci, jakožto nezávislé části programu. Tedy nás plně nezajímá jak objekt vypadá uvnitř, ale jak je charakterizován z vnější, tento posun ve vnímání oproti stylům zmíněných v předchozí podsekcí je esenciální součástí objektově orientovaného přemýšlení.

### Objekt

Objekt můžeme definovat jako reprezentaci konceptu reálného světa pomocí kolekce operací, jež sdílí aplikační stav. Operace určují zprávy, na které objekt může odpovědět, přičemž stav objektu je schován před vnějším světem a je přístupný pouze danému objektu. Proměnné reprezentující vnitřní stav se nazývají *instanční proměnné* a operace objektu se nazývají *metody*. Kolekce instančních proměnných a metod tvoří rozhraní objektu a charakterizují jeho chování.[41] Všechny objekty bez výjimky mezi sebou komunikují pomocí zpráv. Zároveň všechny objekty mají stejný status, tj. "*primitivní*" objekty, jako např. *integer*, jsou objekty stejně jako všechny ostatní koncepty, ať se již jedná o objekty programovacího jazyka (např. *třídy*), nebo objekty vytvořené programátorem.[42]

### Zpráva

Objekty zpracovávají a posílají zprávy, aby splnily požadavky uživatele, nebo i své vlastní. Uživatel žádá objekt o provedení činnosti zprávou. Během provádění požadovaného úkonu, pokud je to nutné, žádá objekt ostatní objekty o informace, nebo provedení výpočtu taky zprávou. Tedy zde je vidět, že zpráva je hlavním komunikačním nástrojem a posílání zpráv je jednou z hlavních funkcí OOP.

Všechny procesy uvnitř programu jsou realizovány pomocí posílání zpráv. Zároveň všechny zprávy mají stejný status, tj. všechny zprávy mají stejnou prioritu zpracování.

Konceptuálně je zpráva textem žádosti. Zpráva může být parametrizována tím, že spolu s textem se odešle jeden nebo více jmen objektů. Jména objektů jsou určena používaným programovacím jazykem. Text zprávy je konstantní, ale parametry se mohou mezi zprávami stejného typu lišit. Objekt reaguje na zprávu odpovědí, která je jménem objektu.

Zpráva slouží k zahájení zpracování nebo vyžádání informací. Text zprávy informuje o požadovaném objektu. Parametry poskytují další potřebné informace nebo parametry potřebné pro vykonání výpočtu.[42]

Objektově orientovaný přístup stojí na následujících čtyřech konceptech. Každý z jazyků tyto koncepty implementuje po svém, ale jsou vždy přítomny.

### Abstrakce

Termín abstrakce označuje rozdíl mezi z vnější dostupnými vlastnostmi objektu a detaily jeho implementace. Abstrakce umožňuje používat objekt, bez nutnosti hlubší znalosti jeho fungování. Dále abstrakce usnadňuje návrh komplexních systémů. V tomto případě potom každá komponenta představuje určitou úroveň abstrakce a odděluje informaci o použití komponenty od její implementace.[39] Abstrakci v OOP například reprezentují *abstraktní třídy objektů*.

## Dědičnost

Dědičnost ulehčuje popis velice podobných objektů, jež se liší pouze v určitých charakteristikách. [39] Můžeme také říci, že dědičnost je mechanismus dovolující derivovat nové *podtřídy* ze starých *nadtříd* přidáním nových atributů a implementací nových metod, případně přepsáním metod původních. Důvodem pro použití dědičnosti je, že objekty, které sdílejí části svých rozhraní, budou často sdílet i některá svá chování a my bychom chtěli tato společná chování implementovat pouze jednou.

## Zapouzdření

Zapouzdření lze definovat jako zabalení atributů, metod objektu do jednoho kompaktního balíčku. V případě OOP je tímto balíčkem *třída*. Díky zapouzdření lze skrýt vnitřní stav objektu před vnějším světem. S objektem mohou tedy přímo manipulovat pouze jeho vnitřní metody. Tento koncept umožňuje rozdělit program do menších snadno kontrolovatelných a čitelných částí.[43]

## Polymorfismus

Obecně můžeme polymorfismus definovat jako "mít či nabývat různé formy". V kontextu OOP označuje schopnost různých tříd objektů reagovat na stejnou situaci různými způsoby.[44] Tedy jinými slovy se jedná o *overloading* metod. Existují ale i jiné typy polymorfismu.[45]

První z těchto typů je tzv. *subtyping*, což je forma typového polymorfismu, kdy *podtyp*<sup>3</sup> je datový typ, který je spojen s jiným datovým typem, tzv. *nadtypem*<sup>4</sup>, pomocí určitého pojmu zastupitelnosti. To znamená, že programové prvky, typicky proměnné či metody, napsané pro práci s prvky nadtypu, mohou pracovat také s prvky podtypu. Neboli dodržují *Liskov Substitution Principle (LSP)*: "Nechť  $\Phi(x)$  je vlastnost dokazatelná o objektech  $x$  typu  $A$ . Pak  $\Phi(y)$  by měla být pravdivá pro objekty  $y$  typu  $B$ , kde  $B$  je podtyp  $A$  ( $B <: A$ )."[46]

Druhým typem je *parametrický polymorfismus*. Pomocí parametrického polymorfismu lze funkci nebo datový typ napsat genericky tak, aby mohla pracovat s různými hodnotami parametrů, bez ohledu na jejich typ. Důležitým rozdílem oproti *overloadingu* je, že implementace daného datového typu či funkce zůstává ve všech případech identická a liší se pouze typem parametrů.[47] Takovéto funkce nazýváme *generické funkce* a datové typy nazýváme *generické datové typy*.

Mezi příklady OOP jazyků patří:

- C++
- Java
- Scala
- C#
- Python
- JS

### 4.2.3 Deklarativní programování

Neformálně můžeme definovat deklarativní programování jako přístup, kdy programování zahrnuje uvedení, čeho chceme dosáhnout, namísto jak toho chceme dosáhnout. Tedy programátor nemá žádnou či jen velice malou kontrolu nad tím, jak bude programovací jazyk požadavek zpracovávat.

<sup>3</sup>A <: B znamená, že A je podtypem B

<sup>4</sup>B >: A znamená, že B je nadtypem A

Tato neformální definice popisuje, jak by deklarativní jazyky měly fungovat, ale nelze z ní odvodit klíčové koncepty těchto jazyků a programů v nich napsaných. Ty jsou tedy následující[48]:

- Program je teorií, ve vhodné vybrané logice
- Výpočet je dedukcí z teorie

Hlavními požadavky na vybranou logiku je existence modelové teorie, důkazní teorie, korektnost (tj. výsledná odpověď by měla být vždy správná) a ideálně by měla splňovat i podmínku úplnosti (tj. správné odpovědi by měly být vždy odvozeny). Tyto podmínky tedy splňuje většina v současnosti používaných logik včetně predikátové logiky prvního řádu či logik vyššího řádu. [48]

Prvotním krokem při programování problému pomocí deklarativního programovacího jazyka je jeho formalizace pomocí vybrané logiky. Interpretace specifikuje příslušné domény problému a význam symbolů jazyka v těchto doménách. V praxi tato interpretace málokdy obsahuje všechny detaily, ale teoreticky by to možné být mělo. Přirozeně ne všechny problémy jdou interpretovat pomocí logiky. Následující text stojí na předpokladu, že řešený problém lze takto interpretovat.

Následně je podle naší interpretace vytvořena logická část programu. Tato logická složka je obvykle vhodně omezena tak, aby umožňovala efektivní postup dokazování teorémů. Tato složka je v každém z typů deklarativního programování zapsána jiným způsobem, ale vždy platí, že se skládá s axiomů, tj. z výroků, jež jsou vždy pravdivé. Bez této podmínky by nebylo možné dojít k pravdivým výsledkům.

V některých případech je programátor nucen napsat tzv. kontrolní část. Tato část vypomáhá programovacímu jazyku s dokazovací částí. Díky této nutnosti můžeme rozdělit deklarativní jazyky na dva typy[48]. Na *slabé* a *silné* jazyky. Slabý programovací styl požaduje po programátorovi dodání dokazovací části pro vytvoření efektivního programu. Naopak silný programovací styl žádné takovéto požadavky na programátora neklade a výsledný program je schopen získat výsledek efektivně bez jakékoliv pomoci.

#### 4.2.3.1 Funkcionální programování

Funkcionální programování je nazýváno funkcionální, neboť celý program se skládá pouze z funkcí. Program samotný je napsán jako funkce, jež na vstupu přijímá argumenty programu a její výstup je zároveň výsledkem programu. Typicky je tato hlavní funkce definována pomocí jiných funkcí, jež jsou dále definovány pomocí jiných funkcí. Koncové prvky jsou tzv. *primitiva* zvoleného programovacího jazyka.

Charakteristické vlastnosti a výhody funkcionálních programovacích jazyků jsou následující.[49] Funkcionální program neobsahuje žádné přiřazovací volání kromě těch počátečních, proměnné tedy za celou dobu běhu programu nemění svoji hodnotu. Obecněji, funkcionální programy neobsahují žádné typy vedlejších efektů. Tedy funkcionální volání neovlivní nic jiného, než výsledek samotný. Díky tomu je pořadí provádění úkonů irelevantní a programátor není nucen programovat samotné zpracování na rozdíl od jazyků, jež jsou založeny na strukturovaném přístupu. Jelikož výrazy mohou být kdykoliv vyhodnoceny, můžeme volně zaměňovat proměnné za jejich hodnoty a vice versa, tj. program je *referenčně transparentní*. [49]

Mezi funkcionální jazyky patří například:

- LISP
- Haskell
- Racket
- F#
- Wolfram Language

### 4.2.3.2 Logické programování

Programovací jazyky logického programování jsou založeny na matematické logice, tedy toto paradigma je založeno na popisu relací pomocí formulí. Pravidla jsou zapisována ve formě klauzulí (axiomů). Příkladem těchto klauzulí mohou být např. Hornovy klauzule. Axiomy mohou mít tvar *faktu*, *pravidla*, nebo *dotazu*. Fakty se skládají pouze z hlavičky, pravidla se skládají jak z hlavičky, tak i z těla a dotaz je pouze klauzulí bez těla. Hlavička a tělo v axiomech se pokládají za *termy*. Za termy jsou považovány *atomy*, *čísla*, *proměnné* a *struktury*. Pro odvozování se většinou používá tzv. *backtracking*, nebo jiné algoritmy prohledávání prostoru.

Výhody programů založených na tomto paradigmatu jsou shodné s výhodami funkcionálního stylu.

Programovací jazyky logického programování:

- Prolog
- SWI-Prolog
- Datalog

# Vývoj aplikačního softwaru

Primárními činnostmi, jež se týkají vývoje systému, jsou:

## Specifikace

Analýza funkčních požadavků aplikace

## Architektura a design

Návrh aplikačního softwaru, neboli z jakých částí se bude aplikace skládat

## Implementace

Vlastní výroba aplikačního softwaru

## Validace

Otestování jednotlivých částí aplikace

Dále existují i činnosti podpůrné mezi, které patří:

## Dokumentace

Detailní popis jednotlivých částí aplikace

## Projektové řízení

Koordinace jednotlivých složek činností při vývoji aplikačního softwaru

## Configuration management

Má za cíl stanovit a udržet konzistenci atributů softwaru s jeho požadavky, designem a provozními informacemi po celou dobu jeho existence

## Release management

Řízení procesu nasazení a distribuce aktualizací aplikačního softwaru

## 5.1 Metodika vývoje aplikačního softwaru

Metodika vývoje softwaru, či jinými slovy *model životního cyklu vývoje softwaru* (SDLC), je množina aktiv nutných k samotnému vzniku softwaru. Přístupy k samotným aktivám se prakticky neliší. To, čím se aktiva mezi sebou liší je jejich souslednost, opakování, vstupy, výstupy a nároky na jejich provedení.

### 5.1.1 Vodopádový model

Je nejznámějším příkladem SDLC. V tomto modelu jsou fáze primitivně naskládány za sebou a jsou striktně odděleny. Proces vývoje začíná analýzou. Následně je celý software navržen. Jakmile je návrh hotov, přechází se do fáze implementace. Opět jakmile je implementace plně hotovo, tak se přechází do další fáze, tj. testovací fáze. Životní cyklus končí nasazením a údržbou softwaru.[50]

Hlavní výhodou přístupu je předem jasně definovaný plán. Díky tomu jsou projekty lehce koordinovatelné a plánovatelné. Díky predikovatelnosti se práce snadno rozděluje na časové úseky a mezi lidmi, kteří se na projektu podílejí.[51]

Naopak problémem tohoto modelu je neschopnost rychlé reakce na změnu požadavků, termínů či jiných proměnných, jež jsou součástí projektu. Toto je způsobeno délkou a uspořádáním cyklu.[50] Pro zareagování na jakoukoliv změnu musíme počkat, než se dokončí současný vývoj a až poté můžeme reagovat na vzniklý problém. S tímto souvisí i další problém a to je rychlost dodání hmatatelného výsledku zákazníkovi.[50] Neboť fáze analýzy a designu jsou velice časově náročné, zákazník může mít pocit, že se projekt nikam neposunul, čímž se dostáváme do horší vyjednávací pozice pro další spolupráci. Problémem je v neposlední řadě i znalost kompletní specifikace programu, již na počátku projektu. Tento požadavek je skoro nereálný, neboť vždy se může najít nová součást, jež je nutnou pro splnění požadavků, ale není zahrnuta v počáteční analýze. Pro vyřešení tohoto problému potřebujeme v týmu mít i několik expertů, kteří mají z daným problémem mnoho zkušeností. Bohužel i tak nemůže vyloučit přehlédnutí implementačně důležité části.[51]

Tento přístup se v současné době používá u projektů, jež potřebují jasně definovaný scope a čas není až tak velkým faktorem, tj. státní zakázky, vesmírné projekty, stavebnictví, atd.

### 5.1.2 Iterativní model

Iterativní přístup, rozdělí systém na několik verzí a ty se následně vypracovávají pomocí vodopádového přístupu. Tedy funkčnosti systému přidáváme postupně. Pokud se vyskytne požadavek mít určitou verzi systému dříve, než bylo požadováno, je takovýto problém často řešen hrubou silou, neboli najmutím více zaměstnanců. Z pohledu zákazníka je tento přístup přijatelnější, než mít k dispozici celý systém hned při spuštění, neboť může reagovat na zájem uživatelů více flexibilněji.[50]

Tento přístup zachovává výhody vodopádového přístupu, tj. plán je předem jasně definován. Projekt je snadno predikovatelný a není náročný na koordinaci. Oproti vodopádovému modelu je i možnost pro zákazníka vidět dopředu, co dostane, neboť má přístup k verzím/prototypům objednaného systému.[50]

Stále zjevná nevýhoda je v potřebě znát kompletní specifikaci dané verze programu.[50] Tudíž i přes provedená vylepšení oproti předešlému přístupu je vývoj pomalý a schopnost rychlé reakce na změnu je stále pomalá.

V současné době se již nejedná o dominantní model, neboť většina firem již přešla, nebo přemýšlí nad přechodem k agilnímu modelu.[50]

### 5.1.3 Agilní model

Agilní SDLC si lze představit jako vylepšenou verzi vodopádového přístupu, kde jednotlivé fáze jdou velice rychle po sobě a jednotlivé iterace jsou velice krátké. U originálního vodopádového přístupu může iterace trvat i měsíce, u agilního přístupu počítáme s délkou iterací v jednotkách týdnů. Aby se bez problému stíhaly jednotlivé verze softwaru vydávat, ne všechny jsou deklarovány jako produkční, tedy odesílají se pouze zákazníkovi, ale nenasazují se do produkce. Jaké úkoly jsou splněny v právě aktuální iteraci se vybírá z listu všech v současnosti objevených problémů seřazených podle nutnosti daný úkol splnit. Jednotlivé iterace nazýváme *sprinty*. Na konci každého sprintu by měl existovat zákazníkovi prezentovatelný výsledek.[51]

Takto strukturovaný vývoj přináší konečně řešení problému pomalé reakce. Díky krátkým iteracím nemusíme znát dopředu celý scope programu a odpadá nutnost analyzovat celý software. Analyzujeme pouze to, o čem v dané chvíli víme, že bude potřeba a další rozhodnutí necháme až do další iterace. Bohužel díky takovému přístupu již nejsme schopni predikovat náročnost na zdroje v takové míře jako u předchozích přístupů a například cenový odhad se během vývoje může značně měnit.[50]

Velkou nevýhodou je nutnost aktivního zapojení všech členů týmu. Takto krátké iterace jsou velice náročné a pokud vypadne jeden člen, může to jednoduše ovlivnit celý výsledek, neboť na každého člena týmu jsou kladeny vysoké nároky na produktivitu. Zároveň je velice náročné takovéto týmy motivovat, tudíž je nutná silná vůdčí osobnost, jež dokáže tyto situace zvládnout. V malých firmách tento problém s koordinací není tak markantní, ale jakmile se podíváme do velkých firem, začíná to být skutečný problém, ale existují techniky na jeho zvládnutí.[50]

## 5.2 Požadavky aplikačního softwaru

Proces získávání požadavků a typy samotných požadavků se zabývá *requirements engineering* (RE). Před popisem RE je ale nutné si zadefinovat pojem *softwarový požadavek*, dále pouze *požadavek*.

### 5.2.1 Požadavky

Požadavky jsou specifikací konceptů, jež mají být implementovány. Jsou popisem toho, jak by se software měl chovat, vlastností softwaru a zároveň popisují i jeho jednotlivé části. Mohou být považovány za *omezení* vývojového procesu.[51]

Každý požadavek by měl obsahovat informace o *Aktorovi*, *Akci* a *Vymezující kritéria*. Aktor je osoba, událost či věc provádějící akci. Akce je sloveso popisující, co aktor dělá. Vymezující kritéria omezují danou akci, lze je rozdělit na kvantitativní a kvalitativní.

V současné době můžeme požadavky rozdělit podle několika různých úhlů pohledů. Požadavky můžeme dělit podle jejich *fokusu*<sup>1</sup>, zaměření požadavku a míry očekávání, tj. jak je požadavek důležitý. Pokud se zaměříme na dělení z pohledu širě záběru, tak můžeme požadavkům přiřadit jednu z následujících kategorií: *business*, *uživatelský*, nebo *systémový*. Klasifikace dle zaměření přiřadí požadavku jeden ze dvou typů, buď *funkční*, a nebo *nefunkční*. Míra očekávání rozlišuje mezi čtyřmi typy požadavků: *samozřejmými*, *běžnými*, *nevídanými* a *zbytečnými*. Toho poslední dělení je velice subjektivní a bude se lišit v závislosti na projektu či na tom kdo dané požadavky sepisuje. [50]

#### Business požadavky

Pokrývají důvody *proč* zákazník implementuje daný software, neboli se jedná o seznam benefitů, jichž chce daný zákazník dosáhnout. Požadavky obvykle pocházejí od sponzora projektu, získávajícího zákazníka, manažera skutečných uživatelů softwaru, marketingového oddělení, produktového vizionáře, nebo jiná v softwaru zainteresované strana. Business požadavky by měly zapadat pod dokument popisující vizi a scope projektu.[51]

#### Uživatelské požadavky

Popisují cíle, nebo úkony splnitelné aktory pomocí popisovaného softwaru. Doména uživatelských požadavků také zahrnuje popis atributů produktu a charakteristik, jež jsou důležité pro spokojenost uživatele. Uživatelské požadavky mohou být popsány pomocí *use case modelů*, "*uživatelských příběhů*"[52], nebo i pomocí event-response tabulek. Tyto požadavky tedy popisují *co* uživatel může dělat se softwarem.[51]

<sup>1</sup>Šíře záběru, rozpětí

### Systémové požadavky

Definují požadavky pro produkt, jež se skládá z více softwarových komponent či podsystémů. Systémem je v tom pojetí myšlen veškerý software i pokud nutno hardware komponent/podsystémů. V některých případech se systémové požadavku mohou týkat i lidí, neboť jsou také součástí dodávaného produktu.[51]

### Funkční požadavky

Specifikují chování, jež bude produkt provádět pod určitými podmínkami. Popisují *co* musí vývojář implementovat ke splnění uživatelských požadavků, a tím tedy i k naplnění business požadavků.[51]

### Nefunkční požadavky

Stanovují podmínky a omezení, jež jsou pro daný požadavek platné. Popisují důležité vlastnosti nebo charakteristiky produktu. Zahrnují dostupnost, použitelnost, bezpečnost, výkonnost, atd. Zároveň se zaměřují i na prostředí, ve kterém daný produkt poběží. Jaké jsou požadavky na licence.[51] Tedy jedná se o seznam všech požadavků netýkajících se přímo funkcí systému.

Další možnosti jak můžeme požadavky dělit je dle kategorizace *FURPS*. Tato kategorizace je vhodná k ověření výsledného systému.

**F (functionality) – funkčnost** Požadavky zaměřuje se na hlavní funkčnosti a schopnosti programu. Tedy jedná se o funkční požadavky z přechodného dělení

**U (usability) – vhodnost k použití** Požadavky, jež lze hodnotit zejména z pohledu koncového uživatele. Tedy jak snadno lze výsledný software použít, či jakým celkovým dojmem působí.

**R (reliability) – spolehlivost** Tyto požadavky hodnotí četnosti a závažnost chyb, přesnosti zpracování vstupů a výstupů

**P (performance) – výkon** Hodnotí celkovou rychlosti odezvy systému a zpracování klíčových aktivit. Zároveň popisují i technické parametry systému, např. vytížení zdrojů OS, zatížení síťového provozu, rozložení zátěže na jednotlivé komponenty systému, atd.

**S (supportability) – schopnost být udržován** Poslední skupina požadavků hodnotí oblast údržby a podpory aplikace, její testovatelnosti. Taktéž hodnotí i přizpůsobitelnost a rozšiřitelnost o nové vlastnosti, společně se schopností zapojení aplikace do již existujících systémů.

## 5.2.2 Requirement engineering

Jak již bylo zmíněno výše RE se zabývá procesem získávání požadavků a typy samotných požadavků. Jeho definice dle IEEE je následující: "Mezioborová disciplína, která se zabývá komunikací mezi odběratelem a dodavatelem, jež si klade za cíl stanovit a udržovat požadavky, které mají být splněny dodávaným systémem, softwarem, nebo službou."[53]

Nejdůležitější součástí vstupu RE je *kontext*. Tedy specifikace očekávání zákazníka, podmínek dodání, omezení,...Tyto specifikace můžeme označit pojmem *5 W's*

- **What & Why** – Popis toho co má vzniknout a proč, jaká je motivace, jaká jsou kritéria úspěchu
- **Who** – Popis zainteresovaných stran a jejich rolí
- **Where & When** – Soubor projektových či jiných omezení, existujících standardů, zvyklostí, atd.



Dalším důležitým vstupem je zadání a vymezení rozsahu, tzv. *scope*. Scope vymezuje, čeho se ho se budoucí specifikace uživatelských požadavků bude týkat. Součástí vstupu by měly být i obchodní požadavky, neboli specifikace požadavků na daný software v jazyce odběratele.

Výstupem RE jsou specifikace uživatelských požadavků pokrývajících všechny FURPS kategorie ve formě jednoho nebo více dokumentů, případně i jiných artefaktů. Vytvořená *dokumentace* by měla obsahovat uživatelsky srozumitelnou *funkční specifikaci* ve formátu typicky daným kontextem. Dokumentace by měla dále obsahovat *katalog* nefunkčních požadavků a mnohdy i obsahuje model budoucího systému, tzv. *mockup* model. Ve výjimečných případech je obsahem dokumentace pouze katalog funkčních požadavků.[50]

### 5.2.2.1 Proces requirement engineeringu

Proces RE si klade za cíl definovaným postupem zjistit uživatelské požadavky, tyto požadavky následně systematicky popsat v očekávaném formátu a v závěru takto popsané požadavky zvalidovat.

Skládá se ze čtyř fází *elicitation*, *analysis*, *specification* a *verification/validation*. Tyto fáze jsou nelineárně propojeny s cílem získat co nejkvalitnější podklady pro následný návrh.

#### Elicitation

V některých kontextech označována názvem "*data gathering*" je proces identifikace potřeb a omezení od jednotlivých zúčastněných stran. Tento proces zároveň zahrnuje zjištění existence a případný zisk existujících podkladů. Jelikož existuje mnoho typů informací, a ve většině případů i mnoho zainteresovaných osob, které preferují odlišné přístupy, existuje mnoho technik získávání informací.

Techniky zahrnují koordinované aktivity, při kterých probíhá komunikace se zúčastněnými stranami a na zákazníkovi nezávislé aktivity, na kterých pracuje vývojář samostatně. Jsou zaměřeny především na získání business a uživatelských požadavků. V této fázi je přímá komunikace s uživateli nezbytná, neboť právě uživatelé budou výsledný systém užívat. Mezi v současnosti používané techniky patří např. *interview*, *workshopy*, *focus groups*, *pozorování*, *dotazníky*, atd.[51]

Při získávání informací se musí klást důraz na zamezení výskytu informací, o kterých ví pouze jedna strana. Tento jev popisuje *Johariho okno*. K zabránění výskytu takovýchto informací je ideální postup ve spirále, neboli vracet se k již probraným informacím.[50]

Výstupem elicitation fáze jsou nestrukturované informace o produktu označované pojmem *stated requirements*. [50]

#### Analysis

Jedná se o proces zpracování state requirements za cílem dát těmto informacím smysl.[50] Cílem je vytvořit systematickou představu o skutečných požadavcích klienta. Tato fáze zahrnuje dekompozici state requirements do vhodných úrovní detailu, vytvoření prototypu, vyhodnocování proveditelnosti a stanovení priorit.

Vhodným přístupem pro některé požadavky je jejich reprezentace více způsoby. Díky tomu je možno nahlédnout na požadavek z více pohledů a získat informace, jež by pouze jedna reprezentace nebyla schopna nabídnout. Zároveň více reprezentací pomáhá všem stranám dojít ke společnému porozumění výsledného produktu.[51]

Mezi osvědčené techniky, jež se v praxi k analýze používají patří např. *profilování uživatelů*, *modelování procesů*, *E-R modelování* a *dekompozice požadavků*. [50] Zvolení použité techniky záleží primárně na typu požadavku, jenž chceme zpracovat.

Výstupem je strukturovaná informace reálných požadavků, tzv. *real requirements*. [50]

## Specification

Neboli česky specifikace uživatelských požadavků je písemné konzistentní, přístupné a přezkoumatelné zdokumentování real requirements tak, jak je v daném kontextu vyžadováno. [50] Jinými slovy jedná se o popis klientské představy o produktu, prostředí softwaru, profilu použití, požadavků na výkon a očekávané efektivity a kvality.[54]

Cílem je vytvoření tzv. *documented requirements*. Typickým výstupem je *funkční specifikace*, což může být strukturovaný dokument doplněný nákresey a diagramy, s možností zahrnutí uživatelských požadavků. Výhodou tohoto zápisu je přístupnost pro uživatele. Handicapem je náročná aktualizace, pokud se jedná o dokument většího rozsahu. Další možností, jak funkční specifikace zapsat, je pomocí nástroje *CASE/UML*, jehož zástupcem je např. Enterprise Architect. Výstupný popis v takovéto formě je exaktní a formalizovaný, ale bývá pro uživatele nesrozumitelný a velice "technický". Zároveň je obtížné zachytit business požadavky a celkové souvislosti<sup>2</sup>. Lze využít i kombinaci obou výše zmíněných přístupů. Součástí *documented requirements* je i *katalog nefunkčních požadavků* a mnohdy také *mockup model*.

Výsledná specifikace dle IEEE standardu by měla mít následující vlastnosti[55]:

1. **Korektnost** Specifikace požadavků softwaru by měla správně popisovat chování systému. Nemí být produktivní mít dokument s požadavky, který popisuje nepravděpodobné nebo nemožné očekávané chování systému nebo cíle uživatele.
2. **Jednoznačnost** Požadavky by měly být napsány tak, aby nemohly být různě interpretovány. Použití specifického a vhodného jazyka může pomoci vyhnout se nejednoznačnému výkladu.
3. **Kompletnost** Dokument obsahující požadavky na software by měl kompletně popisovat očekávané chování systému a soubor funkcí.
4. **Konzistentnost** Požadavky na diskutovaný systém si nesmí vzájemně odporovat.
5. **Ohodnocení** Požadavky na software musí být seřazeny dle důležitosti. Každý požadavek má svou vlastní úroveň důležitosti a kritičnosti a všechny nejsou stejné. Seřazením požadavků zajišťuje vodítko ohledně efektivního stanovení priorit během vývoje.
6. **Ověřitelnost** Pokud nelze ověřit, zda byl požadavek splněn, je napsán špatně. Požadavky musí být testovatelné.
7. **Modifikovatelnost** Požadavky musí být možné snadno upravit nebo změnit.
8. **Trasovatelnost** Požadavky musí být dohledatelné a je nezbytné, aby byly poskytnuty informace o trasovatelnosti, protože dokument s požadavky představuje výchozí bod v hledání specifik požadavků. Je tedy žádoucí znát vazby mezi požadavky.

## Verification/Validation

Poslední fází je ověření věcné správnosti, tedy jestli požadavky jsou opravdu potřeba a jsou úplné, neboli jestli jsou zahrnuty i *samozřejmé* požadavky popisující základní chování, bez kterého aplikace nemůže fungovat. Dále jestli jsou požadavky *proveditelné*, *testovatelné*, *trasovatelné* a formálně správné.

Ověření věcné správnosti se dá provést mnoha způsoby. Jedním z nich je zeptat se kolegů. Jedná se sice o velice primitivní způsob ověření, ale i tak přinese jiný pohled a může potenciálně objevit chybějící požadavky. Dále se ověření dá provést formou workshopu s klienty. Tento workshop moderuje autor samotného dokumentu. Zároveň lze využít i *připomínkování požadavků*, kde zúčastněné strany obdrží zdokumentované požadavky s cílem je prostudovat a napsat připomínky.[50]

Záměrem je minimalizovat pravděpodobnost výskytu překvapujících požadavků, které jsou objeveny až během implementace.

---

<sup>2</sup>Big picture

Výstupem je seznam *verified/valid requirements*. [50] Tento seznam slouží pro finální testování produktu.

## 5.3 Návrh aplikačního softwaru

Návrh se zabývá tím jak vyřešit problémy popsané pomocí valid requirements získané během analytické fáze, a zároveň by návrh měl zajistit dostatečnou kvalitu. Kvalita je popsána pomocí tzv. *kvalitativních atributů*. Během tvoření návrhu musí být dodržena omezení, jež jsou valid requirements. Výstupem návrhové fáze je *návrhová dokumentace*, jež popisuje jak byly dané požadavky splněny. [56]

Funkcionální požadavky lze vyřešit vhodným přiřazením zodpovědností pro jednotlivé návrhové elementy. Přiřazení těchto zodpovědností je fundamentálním rozhodnutím návrhu. Kvalita návrhu je splněna zvolením vhodných struktur, společně se správně namodelovaným chováním elementů obsažených v těchto strukturách. [57]

### 5.3.1 Výběr programovacího jazyka

Prvním krokem návrhu by měl být výběr vhodného programovacího jazyka. Výběr by měl primárně záviset na typu softwaru a jeho velikosti. Procedurální jazyky jsou vhodné pro nízkouúrovňové programování, ale využití při tvorbě rozsáhlých informačních systémů nevyhovují. Při vývoji rozsáhlých IS se v současné době ve většině případů používají OOP jazyky. Proto se dále v tomto textu počítá s objektově orintovaným přístupem.

Při výběru programovacího jazyka je vhodné zohlednit různá kritéria. Prvním, pravděpodobně tím nejdůležitějším, je znalost jazyka samotným vývojářem. Dále je vhodné zohlednit dostupnost potřebných frameworků pro daný IS, neboť pokud by daný programovací jazyk nenabízí příliš velký výběr použitelných frameworků, nejspíše není příliš vhodný pro samotný vývoj. Žádoucí kritériem výběru je i požadavek na přenositelnost daného systému. [56]

### 5.3.2 Způsob uložení dat

Rozhodnutí, jak uložit data, je závislé na několika faktorech. Při výběru musí být zohledněny obecné (nefunkční) požadavky, možný budoucí rozvoj systému, znalost dostupných lidských zdrojů a rozpočet na daný projekt.

Mezi hlavní způsoby, jak uložit data, patří *relační databáze*, *objektové databáze* a *soubory*. Nejrozšířenější způsobem ukládání dat jsou relační databáze. Výhodami jsou především doba jejich existence, tudíž jsou velmi dobře vyzkoušené na ostatních projektech, databáze jsou nabízeny mnoha výrobci a především řeší problém s přístupem k datům, díky čemuž odpadá tato zodpovědnost vývojářům. Nevýhodou této volby je nutnost mapování objektů. Tento problém odpadá při volbě objektové databáze. Bohužel jejich problémem je nízká rozšířenost a tedy i podpora napříč různými systémy. Kompromisem mezi těmito typy jsou *objektově relační databáze*, jež umožňují tvorbu uživatelsky definovaných typů a ukládání kolekcí do jednotlivých sloupců. Ukládání do souborů je vhodné pro menší IS, nebo jednočipové aplikace, neboť všechny problémy, jež řeší databáze, musí naprogramovat samotný vývojář. Tedy musí řešit paralelní přístup, řízení transakcí, atd. [56]

### 5.3.3 Softwarová architektura

Softwarová architektura si klade za cíl vyřešit funkční a *kvalitativní požadavky* na IS. [58]. Architektura je především množinou *struktur*. Struktury jsou množiny *elementů* spojených pomocí *relací*. Software se skládá z mnoha takovýchto struktur, které jsou nazývány *softwarové struktury*.

### Architektonické struktury

Je softwarová struktura podporující uvažování o systému a jeho vlastnostech. Cílem tohoto uvažování měly být atributy, jež jsou důležité pro některou ze zúčastněných stran. Množina architektonických struktur není nijak limitována velikostí, vždy by se mělo jednat o struktury, jež jsou v daném kontextu důležité. Tyto struktury můžeme dělit do tří typů, které odpovídají třem obecným druhům rozhodnutí, které architektonický návrh zahrnuje.

1. *Modulární struktury* rozdělují IS do jednotlivých implementačních jednotek. Tyto jednotky se nazývají *moduly*. Modulům jsou přiřazeny jednotlivé implementační povinnosti a jsou základem pro rozdělení práce mezi týmy. Tyto moduly, pokud jsou dostatečně velké, můžeme dělit do dalších menších částí. Struktura, která popisuje takovouto dekompozici se nazývá *dekompozice modulu*. Další typ modulární struktury vzniká jako výstup objektově orientované analýzy a návrhu tříd. Pokud se výstupní moduly agregují do vrstev, vznikne další užitečná modulární struktura. Modulární struktury jsou statické, jelikož se zaměřují na způsob, jakým se rozdělují funkce systému a jak jsou následně přiřazeny jednotlivým týmům. Ostatní dva typy jsou dynamické, neboli se zaměřují na způsob, jakým se prvky za běhu vzájemně ovlivňují při plnění funkcí systému.[57] Modulární struktury odpovídají na následující otázky:
  - Jaká je hlavní odpovědnost přiřazená každému modulu?
  - Jaké další softwarové prvky smí modul používat?
  - Jaké moduly souvisejí s jinými moduly prostřednictvím vztahů zobecnění nebo specializace (tj. dědičnosti)?
2. Předpokládejme, že systém má být sestaven jako soubor služeb. Služby, infrastruktura, se kterou komunikují, synchronizace a interakční vztahy mezi nimi tvoří další typ struktury, jež se často používá pro popis systému. Tyto služby jsou sestaveny z programů z různých implementačních jednotek. Tyto struktury jsou nazývány *component-and-connector (C&C) struktury*. V tomto případě je komponentem myšlena runtime entita.[57] C&C struktury pomáhají se zodpovězením následujících otázek:
  - Jaké jsou hlavní spouštěcí komponenty a jak se vzájemně ovlivňují během běhu?
  - Jaké jsou hlavní sdílené datové sklady?
  - Jak data procházejí systémem?
  - Může se struktura systému během jeho běhu měnit, a pokud ano, jak?
3. Poslední druh struktury popisuje mapování softwarových struktur na organizační, vývojové, instalační a exekuční prostředí systému. Například moduly jsou přiděleny týmům k vývoji a přiřazeny k místům v souborové struktuře pro implementaci, integraci a testování. Komponenty jsou nasazovány na hardware, aby se mohly vykonávat. Tato přiřazení se nazývají *alokační struktury*. [57] Alokační struktury pomáhají zodpovědět otázky jako jsou například:
  - Na jakém procesoru se spouští jednotlivé softwarové prvky?
  - Do jakých adresářů nebo souborů se ukládají jednotlivé prvky během vývoje, testování a budování systému?

#### 5.3.3.1 Pohledy na softwarovou architekturu

Jedním z nejdůležitějších pojmů spojovaných se softwarovou architekturou a její dokumentací je *pohled*<sup>3</sup>. Softwarová architektura je komplexní entitou, jež nelze jednoduše jedno-dimenzionálně popsat. Například pohled na systém z pohledu vrstev by zobrazoval pouze prvky typu vrstva, tj. zobrazoval by rozklad systému na vrstvy a vztahy mezi těmito vrstvami. Takovýto pohled by však nezobrazoval služby systému, client-server model, datový model ani žádný jiný typ prvku.

<sup>3</sup>Angl. *view*

Princip pohledu nám tedy dovoluje rozdělit vícerozměrnou entitu, jakou je softwarová architektura, na několik rozličných reprezentací systému.[57]

Kontext pohledu vždy závisí na tom, co je zájmem dané zúčastněné strany. Různé pohledy podporují různé cíle a použití. Proto neexistuje žádný konkrétní pohled, nebo souboru pohledů, které by měli být uvedeny, vše závisí na tom, jaké použití dokumentace se očekává. Z tohoto důvodu jsou následující typy pouze obecným popisem struktury, jak by pohled daného typu mohl vypadat.

### Modulární pohledy

Modul může mít podobu třídy, kolekce tříd, vrstvy, aspektu nebo jakékoli jiné dekompozice implementační jednotky. Příklady pohledů na modul jsou dekompoziční pohled, pohled na použití a pohled na vrstvy. Každý modul má přiřazené vlastnosti, jež vyjadřují důležité informace a omezení spojené s modulem. Vztahy, které mezi sebou mohou jednotlivé moduly mít jsou např.: *is part of*, *depends on* a *is a*.

Jelikož moduly rozdělují systém, mělo by být možné z takového pohledu určit, jak jsou funkční požadavky systému podporovány vlastnostmi modulů. Modulární pohledy, které ukazují závislosti mezi moduly nebo vrstvami (což jsou skupiny modulů, které mají specifický model přípustného užití), poskytují dobrý základ pro analýzu dopadů změn.

Modulární pohled lze zároveň použít k vysvětlení funkčnosti systému někomu, kdo s ním není obeznámen. Různé úrovně granularity dekompozice poskytují pohled na vlastnosti systému shora dolů a mohou tudíž být použity k hlubšímu pochopení IS. Toho lze využít u již zavedených systému pro zaučení nových vývojářů pod podmínkou, že pohledy jsou udržovány aktuální.

Na druhou stranu je obtížné používat modulární pohled k předvedení chování softwaru za běhu. Proto se obvykle nepoužívá pro analýzu výkonu, spolehlivosti a mnoha dalších runtime vlastností. Pro tyto účely se spoléháme na C&C pohledy a na alokační pohledy.

Modulární pohledy jsou často připojeny k C&C pohledům, kde jsou jednotlivé moduly namapovány na runtime komponenty. Ve většině případů je takto mapovaný model abstrakcí/obalením daných komponent. Například mnoho komponent může být obsaženo v jedné vrstvě.[57]

### C&C pohledy

C&C pohledy zobrazují elementy, které mají alespoň nějakou přítomnost během běhu programu. Takovýmto elementem může být proces, objekt, klient, server,...Tyto elementy jsou nazývány *komponenty*. Dále tyto pohledy zahrnují i elementy tykající se samotné komunikace jako jsou například protokoly, toky informací a procedury přístupu. Takovéto elementy jsou definovány jako *konektory*. Mezi C&C pohledy můžeme zařadit například SOA a client-server pohledy.

Komponenty mají rozhraní zvané *port*. Port definuje bod potencionální interakce komponenty s prostředím. Port má ve většině případů přiřazen explicitní typ, neboť tento typ definuje typ interakce, jež v daném bodě probíhá. Komponent může mít jeden či více portů bez ohledu na jejich typ.

Konektory mají také svá rozhraní. Tato rozhraní jsou označují pojem *role*. Role definují způsoby, jak mohou jednotlivé komponenty využívat daný konektor. Příkladem konektorů mohou být roury, multicast událostí, asynchronní fronty zpráv, volání služeb, atd. Konektory často představují mnohem složitější formy interakce, například transakčně orientovaný komunikační kanál mezi databázovým serverem a klientem nebo sběrnici podnikových služeb, která zprostředkovává interakce mezi kolekcemi uživatelů služeb a poskytovatelem.

Primární relací v tomto typů pohledů je *vazba*. Vazby udávají, které konektory jsou připojeny ke kterým komponentám, čímž definují systém jako graf komponent a konektorů. Konkrétně je vazba označena přiřazením portu komponenty k roli konektoru. Platná vazba je taková,

ve které jsou porty a role vzájemně kompatibilní, a to v rámci sémantických omezení definovaných pohledem. Prvek (komponenta nebo konektor) zobrazení C&C bude mít různé přidružené vlastnosti. Každý prvek by měl mít název a typ. Další vlastnosti závisí na typu komponenty nebo konektoru.

C&C pohledy se běžně používají k tomu, aby vývojářům a dalším zúčastněným stranám ukázaly, jak systém funguje. Zároveň se také používají k posuzování kvalitativních atributů běhu, jako je výkon a dostupnost. Pro jejich popis je vhodné využít UML sémantiky.[57]

### Alokační pohledy

Alokační pohledy popisují mapování softwarových jednotek (procedury, funkce, třídy) na prvky prostředí, v němž je software vyvíjen nebo v němž je spuštěn. Prostředím může být hardware, operační prostředí, ve kterém software běží, souborové systémy podporující vývoj a nasazení, nebo případně vývojová organizace.

Jediným vztahem v tomto zobrazení je "allocated to". Obvykle se jedná o mapování od softwarových prvků k prvkům prostředí, ačkoli opačné mapování může být také relevantní. Jeden softwarový prvek může být přiřazen více prvkům prostředí a více softwarových prvků může být přiřazeno jednomu prvku prostředí. Pokud se tyto alokace v průběhu času změny, ať už během vývoje nebo běhu systému, říká se o architektuře, že je s ohledem na tuto alokaci dynamická. Například procesy mohou migrovat z jednoho procesoru nebo virtuálního stroje na jiný.

Prvky softwaru a prvky prostředí mají vždy přiřazeny nějaké vlastnosti. Obvyklým cílem alokačního pohledu je porovnat vlastnosti požadované softwarovým prvkem s vlastnostmi poskytovanými prvky prostředí a určit, zda bude alokace úspěšná, či nikoli.[57]

### Kvalitativní pohledy

Modulární, C&C a alokační pohledy jsou všechny strukturálního typu, neboť primárně zobrazují struktury, jež jsou důležité pro splnění funkčních a kvalitativních požadavků.

Tyto pohledy jsou vynikajícím nástrojem pro vedení a usměrňování vývojářů, jejichž hlavním úkolem je tyto struktury implementovat. Avšak v systémech, v nichž jsou určité kvalitativní atributy (nebo třeba jiné druhy zájmů zainteresovaných stran) obzvláště důležité a vsudypřítomné, nemusí být strukturální pohledy nejlepším způsobem, jak prezentovat architektonické řešení těchto potřeb. Jiný druh pohledu, který nazýváme kvalitativní pohledem, může být přizpůsoben konkrétním zúčastněným stranám nebo pro řešení konkrétních problémů. Tyto pohledy na kvalitu se vytvářejí extrakcí příslušných částí strukturálních pohledů a jejich následným spojením.[57]

#### 5.3.3.2 Architektonické vzory

Architektonický vzor je soubor návrhových rozhodnutí, která se v praxi opakovaně vyskytují. Má známé vlastnosti, jež umožňují jeho znovupoužití a popisuje *třídu* architektur. Protože vzory se vyskytují v praxi, člověk je nevymýšlí, ale objevuje. Z tohoto důvodu nikdy nebude existovat úplný seznam vzorů. Vzory spontánně vznikají v reakci na podmínky prostředí a dokud se tyto podmínky mění, objevují se nové. Komplexní systémy mohou využívat více vzorů dohromady.

Architektonický vzor stanovuje vztah mezi *kontextem*, *problémem* a *řešením*. Kontextem je myšlena opakující se situace ve světě, která vede k problému. Problém je popsán vzorem společně s jeho variantami a případně doplňujícími nebo protichůdnými faktory. Popis problému často zahrnuje kvalitativní atributy, jež musí být splněny. Řešení popisuje architektonické struktury zabývající se daným problémem a faktory ovlivňující řešení. Dále popisuje odpovědnosti a statické vztahy mezi prvky pomocí modulárních struktur, nebo popisuje chování runtime prvků a jejich vzájemné působení pomocí C&C, nebo alokačních struktur. Řešení vzoru je určeno a popsáno:

- Sadou typů jednotlivých prvků (například datových úložišť, procesů a objektů)

- Topologickým uspořádáním prvků
- Souborem sémantických omezení zahrnující topologii, chování prvků a interakční mechanismy

Z popisu řešení by mělo být také zřejmé, jaké kvalitativní atributy jsou poskytnuty statickými a runtime konfiguracemi prvků.

### Vícevrstvá architektura

Všechny komplexní systémy dojdou do fáze, kdy je nutné daný systém rozdělit a jednotlivé části vyvíjet odděleně. Z tohoto důvodu je potřeba detailní dokumentace rozdělení problému do jednotlivých modulů, aby bylo možné provést jejich separovaný vývoj a údržbu.

Takováto segmentace ale není bezproblémová. Software musí být segmentován tak, aby moduly mohly být vyvíjeny a rozvíjeny samostatně s malou interakcí mezi jednotlivými částmi, díky čemuž je následně zajištěna přenositelnost, modifikovatelnost a možnost opakovaného použití.

K dosažení takto oddělených zájmů rozděluje vícevrstvý vzor software na jednotky nazývané *vrstvy*. Každá vrstva je seskupením modulů, jež poskytuje ucelenou sadu služeb. Vztahy mezi jednotlivými vrstvami musí být jednosměrné. Takto definované vrstvy rozdělují celý software do jednotlivých částí, jež jsou přístupné skrze veřejné rozhraní. Pokud (A,B) je relací mezi vrstvami, tvrdíme, že implementace vrstvy A může využívat veřejné rozhraní poskytnuté vrstvou B. Takováto relace může vzniknout pouze mezi vrstvami, jež odděluje maximálně jedna další vrstva. Příklad, kdy vrstva využívá nižší přímo nesousedící vrstvu, se nazývá *layer bridging*, což lze do češtiny přeložit jako *přemostění vrstev*. Pokud se takovýchto případů vyskytne příliš mnoho může dojít ke znemožnění splnění požadavků na přenositelnost a modifikovatelnost. Tento styl povoluje užití vrstev pouze směrem shora dolů. Vrstva však může volat nahoru, pokud od nich neočekává odpověď.

Pokud vrstvy nebudou navrženy korektně, může takto špatně navržené dělení způsobit problémy při implementaci zahrnující vyšší úroveň abstrakce, jelikož nebude poskytovat dostatečně kvalitní abstrakci nižších abstrakčních úrovní. Zároveň vrstvení bude vždy přidávat na výpočetní náročnosti, neboť pokud je zavolána funkce z vyšších vrstev, je nutné projít mnoho nižší vrstev předtím, než daný požadavek bude zpracován.[57]

### Broker architektura

Mnoho systémů je složených z kolekce služeb distribuovaných přes více serverů. Implementace těchto systémů je velice komplexní, neboť je nutné vyřešit problém spolupráce, tedy jak se bude jedna služba připojovat k druhé a jak budou mezi sebou vyměňovat informace. Zároveň je nutné vyřešit i dostupnost dílčích služeb.

Tento architektonický styl řeší problém, jak strukturovat distribuovaný systém, aby uživatelé nevěděli skutečný charakter a lokaci poskytovatelů služeb, a aby bylo možné jednoduše měnit vazby mezi uživateli a poskytovateli.

Broker vzor odděluje uživatele služeb (*klienty*) od poskytovatelů služeb (*serverů*) vložením prostředníka, jež se nazývá *broker*. Pokud klient vyžaduje přístup ke službě, dotazuje se *brokera* pomocí rozhraní služby. Broker následně předá klientský požadavek server, který daný požadavek zpracuje. Výsledek je předán ze serveru zpět brokerovi, jež vrátí výsledek či vzniklou výjimku zpátky klientovi. Díky tomu je klient odstíněn od jakékoliv informace o serveru. Pokud dojde k výpadku jednoho ze serverů, díky této separaci je to broker a ne samotný klient, jež najde nový vhodný server.

Nevýhodou tohoto stylu je přidání složitosti (je třeba navrhnout a implementovat zprostředkovatele a případně proxy servery a protokoly pro zaslání zpráv) a nutnost přidání nových přeměrování, které zvyšují latenci mezi klientem a serverem. Debugování brokera může být náročné, jelikož broker pracuje ve velmi dynamickém prostředí a může být těžké chybu replikovat. Zároveň musíme zajistit dostatečnou ochranu tohoto prvku, neboť je očividným terčem potenciálního útoku.[57]

### MVC architektura

Uživatelské rozhraní softwaru (UI) je typicky nejvíce modifikovaná část interaktivní aplikace. Z tohoto důvodu je důležité separovat modifikátory UI od ostatních částí systému. Uživatelé se často chtějí podívat na data z různých úhlů pohledu. Tyto reflexe by tedy měly zastávat současný stav dat.

Problémem tedy je, jak udržet funkcionality samotného softwaru od funkcionalit UI, ale přitom stále reagovat na uživatelské vstupy nebo na změny v základních datech aplikace. A zároveň jak lze vytvořit, udržovat a koordinovat více pohledů na UI, když se změní základní data aplikace.

MVC tento problém řeší separováním aplikačních funkcionalit do tří typů komponent:

- **Model** obsahuje aplikační data
- **View** zobrazuje část podkladových dat a komunikuje s uživatelem
- **Controller** zprostředkovává spojení mezi modelem a pohledem a spravuje oznámení o změnách stavu

MVC není ideální pro každou situaci, neboť návrh a implementace tří různých druhů komponent a jejich různých forem interakce může být nákladná činnost a u relativně jednoduchých uživatelských rozhraní nemusí mít tyto náklady smysl. Dále View a Controller oddělují vstup a výstup, tyto funkce jsou často spojeny do jednotlivých widgetů. To může mít za následek koncepční nesoulad mezi architekturou a sadou nástrojů, jež používá UI.

Komponenty MVC jsou navzájem propojeny prostřednictvím určité formy oznámení jako jsou například události, nebo zpětná volání. Tato oznámení obsahují aktualizace stavu. Změnu v modelu je vždy třeba sdělit view komponentám, aby mohly být aktualizovány. Externí událost, například vstup uživatele, musí být sdělena controlleru, který následně může aktualizovat zobrazení a/nebo model.[57]

### Pipe-and-Filter architektura

Mnoho systémů musí transformovat toky datových položek ze vstupu na výstup. Mnoho typů transformací se v praxi vyskytuje opakovaně, a proto je žádoucí vytvářet je jako nezávislé, opakovaně použitelné části.

Takovéto systémy musí být rozděleny do znovupoužitelných, slabě provázaných komponent s jednoduchým, generickým interakčním mechanismem. Díky tomu mohou následně flexibilně komunikovat mezi sebou. Komponenty, jež jsou slabě provázané, mohou být jednoduše znovu použity. Komponenty, jež jsou nezávislé, lze spouštět paralelně.

Vzor interakce v pipe-and-filter stylu je charakterizován průběžnou transformací toků dat. Data přicházejí na vstupní porty *filteru*, následně jsou transformována a výstupními porty poslána pomocí *pipe* komponentu do dalšího filteru. Každý filter má jeden či více vstupních i výstupních portů.

S tímto architektonickým vzorem je spojeno několik nevýhod. Například není obecně vhodnou volbou pro interaktivní software, neboť zakazuje cykly (cykly jsou důležité pro zpětnou odezvu uživatele). Zároveň velké množství nezávislých filterů může značně přidat na výpočetní náročnosti, neboť každý filter běží v nezávislém vlákne či procesu. Problémem je i použití v systémech, v nichž probíhá dlouhodobý výpočet. Bez použití checkpointů, jež data průběžně ukládají, dojde ke ztrátě všech zpracovaných dat.

Pipy během komunikace ukládají data do bufferu. Díky této vlastnosti filtry mohou pracovat asynchronně a souběžně. Filter navíc obvykle nezná identitu svých předřazených ani následných filterů. Z tohoto důvodu mají pipe-and-filter systémy tu vlastnost, že celkový výpočet lze považovat za funkcionální kompozici výpočtů filterů, což architektovi usnadňuje uvažování o celkovém chování.[57]



### Client-Server architektura

Existují sdílené prostředky a služby, k nimž chce mít přístup velké množství klientů a u nichž chceme řídit přístup nebo kvalitu služeb.

Správou sady sdílených zdrojů a služeb můžeme podpořit modifikovatelnost a opakované použití tím, že vyčleníme sdílené služby a ty následně budeme modifikovat na jednom místě nebo na malém počtu míst. Zároveň chceme zlepšit škálovatelnost a dostupnost systému centralizací řízení těchto prostředků a služeb, současně chceme během distribuce rozdělit samotné prostředky na více fyzických serverů.

Klienti interagují se systémem odesláním požadavků o služby serverům, které následně poskytují jejich dostupnou sadu. Některé komponenty mohou fungovat jako klient i server zároveň. Může existovat jeden centrální server nebo více distribuovaných.

Mezi nevýhody vzoru client-server patří, že se na serveru může vytvořit bottleneck a kvůli tomu se může zpomalit či zastavit zpracování požadavků. Problémová jsou i rozhodnutí o umístění funkcí (v klientovi nebo na serveru), neboť jsou často složitá a jejich změna po vytvoření systému je nákladná.

Výpočet v nativních systémech client-server je asymetrický: klienti zahajují interakce voláním služeb serverů. Klient tedy musí znát identitu služby, aby ji mohl vyvolat. Klienti iniciují všechny interakce. Naproti tomu servery identitu klientů předem neznají a musí na iniciované požadavky klientů reagovat.

V minulosti client-server vzoru bylo volání služeb synchronní: žadatel o službu čekal nebo byl blokován, dokud požadovaná služba nedokončila svou činnost, případně neposkytla zpětný výsledek. Nicméně současné varianty vzoru client-server mohou používat i sofistikovanější protokoly.[57]

### Peer-to-peer (P2P)

Distribuované výpočetní entity, z nichž každá je považována za stejně důležitou z hlediska zahájení interakce a z nichž každá poskytuje své vlastní zdroje, musí spolupracovat a kooperovat, aby poskytly službu distribuované komunitě uživatelů.

Problémem je vzájemné propojení množiny "stejných" distribuovaných výpočetních entit prostřednictvím sdíleného protokolu tak, aby mohly organizovat a sdílet své služby s vysokou dostupností a škálovatelností.

Ve vzoru P2P spolu komponenty (*účastníci*<sup>4</sup>) komunikují jako rovnocenné. Všichni účastníci jsou si rovni v tom smyslu, že žádný z nich není kritický pro zdraví systému. P2P komunikace je typicky request/reply interakce bez jakékoliv asymetrie. To znamená, že jakýkoliv účastník může komunikovat s jakýmkoliv jiným účastníkem posláním požadavku o jeho službu. Interakci může započít kdokoli, tudíž v client-server terminologii, každý účastník se chová jako klient i server zároveň. Občas obsahem interakce je pouze přeposlání dat bez nutnosti odpovědi. Každý účastník poskytuje podobné služby a používá stejný protokol jako všichni ostatní.

Účastníci se nejdříve připojí k P2P síti, kde objeví ostatní účastníky, se kterými mohou interagovat, a následně iniciují akce k dosažení svých výpočtů spoluprací s ostatními vrstevníky tím, že si vyžádají jejich služby. Vyhledání jiného účastníka se často šíří od jednoho účastníka k jeho připojeným účastníkům na omezený počet skoků.

Architektura P2P může mít specializované uzly (tzv. *supernodes*), které mají indexovací nebo směrovací schopnosti a umožňují běžnému účastníkovi dosáhnout při vyhledávání dosáhnout většího počtu dalších účastníků.

Účastníky lze přidávat a odebírat z P2P sítě bez výrazného dopadu, což má za následek velkou škálovatelnost celého systému. To poskytuje flexibilitu při nasazení systému na vysoce distribuované platformě.

<sup>4</sup>Angl. *peer*

Obvykle se schopnosti více rovnocenných účastníků překrývají, například poskytují přístup ke stejným datům nebo poskytují ekvivalentní služby. Účastník vystupující jako klient tak může spolupracovat s více účastníky vystupujícími jako servery na dokončení určitého úkolu. Pokud se jeden z těchto účastníků stane nedostupným, ostatní mohou stále poskytovat služby k dokončení úlohy. Výsledkem je lepší celková dostupnost. Výhodou je také výkonnost, neboť zatížení dané komponenty účastníků, jež funguje jako server, se snižuje a povinnosti, které by vyžadovaly větší kapacitu serveru a infrastrukturu, jsou rozděleny. To může snížit potřebu další komunikace pro aktualizaci dat a centrálního serverového úložiště, avšak na úkor lokálního ukládání dat.

Nevýhody P2P modelu jsou silně spojeny s jeho silnými stránkami. Protože P2P jsou decentralizované systémy, správa zabezpečení, konzistence dat, dostupnosti dat, dostupnosti služeb, zálohování, a obnovy jsou složitější. V mnoha případech je obtížné poskytnout jakékoliv záruky pomocí P2P, protože účastníci se průběžně připojují a odpojují. Namísto toho může architekt v nejlepším případě nabídnout pravděpodobnost, že kvalita systému bude zajištěna. Tyto pravděpodobnosti se obvykle zvyšují s velikostí populace rovnocenných účastníků.[57]

## SOA architektura

Poskytovatelé služeb nabízejí (a popisují) řadu služeb, které spotřebitelé využívají.

Otázkou je, jak můžeme podpořit vzájemnou kompatibilitu distribuovaných komponent běžících na různých platformách a napsaných v různých implementačních jazycích, poskytovaných různými organizacemi a distribuovaných po internetu. A zároveň je otázkou jak můžeme lokalizovat služby a kombinovat je do smysluplných uskupení při dosažení přiměřeného výkonu, bezpečnosti a dostupnosti.

SOA přistupuje k tomuto problému pomocí kolekce distribuovaných komponent, jež poskytuje a/nebo využívají služby. V rámci SOA mohou komponenty poskytovatele a komponenty konzumenta používat různé implementační jazyky a platformy. Služby jsou do značné míry samostatné. Poskytovatelé a konzumenti služeb jsou obvykle nasazeni nezávisle na sobě a často patří do různých systémů, nebo dokonce různých organizací. Komponenty mají rozhraní, jež popisují požadované služby od jiných komponent a poskytované služby komponentou samotnou. Kvalitativní atributy služby lze specifikovat a garantovat pomocí dohody o úrovni služeb (SLA). V některých případech jsou tyto dohody právně závazné. Komponenty provádí své výpočty pomocí vzájemného sdílení služeb.

Elementy tohoto vzoru zahrnují poskytovatele a konzumenty služeb, které mohou mít v praxi různé podoby, od JS běžícího ve webovém prohlížeči až po transakce CICS běžící na mainframe. Kromě komponent poskytovatele a konzumenta může aplikace SOA používat specializované komponenty, které fungují jako zprostředkovatelé a poskytují infrastrukturní služby:

- Vyvolání služby může být zprostředkováno pomocí *enterprise service bus* (ESB) komponentou. ESB zprostředkovává zprávy mezi konzumenty a poskytovateli. Zároveň může ESB převádět zprávy z jednoho protokolu nebo technologie na jinou, provádět různé transformace dat (např. změna formátu, změna obsahu, rozdělení, sloučení), provádět bezpečnostní kontroly a spravovat transakce. Použití ESB podporuje vzájemnou součinnost, bezpečnost a modifikovatelnost. Komunikace prostřednictvím ESB samozřejmě zvyšuje režii, čímž snižuje výkonnost, a zavádí další možný bod selhání. Pokud ESB není zavedena, komunikují poskytovatelé a konzumenti služeb mezi sebou způsobem point-to-point.
- Pro zlepšení nezávislosti poskytovatelů služeb lze v architektuře SOA použít *service registry*. Service registry je komponenta, která umožňuje registraci služeb za běhu. To umožňuje dynamické zjišťování služeb, což zvyšuje modifikovatelnost systému tím, že skrývá umístění a identitu poskytovatele. Registr může dokonce umožňovat více živých verzí takovéto služby.
- *Orchestration server* zajišťuje interakci mezi různými konzumenty a poskytovateli služeb v systému SOA. Spouští skripty při výskytu určité události (např. přišel požadavek na

objednávku). Aplikace s dobře definovanými business procesy nebo workflow, jež zahrnují interakce s distribuovanými komponentami nebo systémy, získávají použitím orchestration server komponenty na modifikovatelnosti, vzájemné součinnosti a spolehlivosti.

Základními typy konektorů, jež jsou v SOA použité, jsou následující:

- SOAP
- REST
- Asynchronní zaslání zpráv, což je výměna informací stylem "*fire-and-forget*". Účastníci nemusí čekat na potvrzení o přijetí, protože se předpokládá, že infrastruktura zprávu úspěšně doručila. Konektor pro zaslání zpráv může být typu point-to-point nebo publish-subscribe.

Vzor SOA může být poměrně složitý na návrh a implementaci (kvůli dynamické vazbě a současnému používání metadat). Mezi další potenciální problémy tohoto vzoru patří výkonnostní režie middlewaru, který je umístěn mezi službami a klienty. Mezi problémy se dále řadí i nedostatek záruk výkonu (protože služby jsou sdílené a obecně nejsou pod kontrolou žadatele). Všechny tyto nedostatky jsou společné se vzorem broker, což není překvapivé, protože vzor SOA sdílí mnoho koncepcí návrhu a cílů vzoru broker. Kromě tohoto, může být uživatel nucen snášet vysoké a neplánované náklady na údržbu, neboť obecně nemá kontrolu nad vývojem služeb, jež využívá.

Hlavním přínosem SOA je vzájemná součinnost. Protože poskytovatelé a konzumenti služeb mohou běžet na různých platformách, architektury orientované na služby často integrují různé systémy včetně těch starších. SOA také nabízí nezbytné prvky pro interakci s externími službami dostupnými přes internet. Speciální komponenty SOA, jako je service registry nebo ESB, také umožňují dynamickou rekonfiguraci, což je užitečné, když je třeba vyměnit nebo přidat verze komponent bez přerušení systému.[57]

### Publish-Subscribe Pattern

Existuje řada nezávislých producentů a spotřebitelů dat, kteří musí vzájemně spolupracovat. Přesný počet a povaha producentů a spotřebitelů dat není předem určený ani pevně stanovený, stejně jako data, která mezi sebou sdílejí.

Jak tedy můžeme vytvořit integrační mechanismy, jež podporují schopnost předávat zprávy mezi výrobcí a spotřebiteli takovým způsobem, aby si navzájem nebyli vědomi své identity nebo případně své existence?

Ve vzoru publish-subscribe komponenty komunikují prostřednictvím ohlášených zpráv nebo událostí. Komponenty se mohou přihlásit k odběru sady událostí. Úkolem runtime publish-subscribe infrastruktury je zajistit, aby každá publikovaná událost byla doručena všem *subscriberům* této události. Hlavní formou konektoru v těchto vzorech je *event bus*. *Publisher* komponenty umísťují události do komponenty event bus ohlášením, konektor pak tyto události doručuje subscriber komponentám, které o tyto události projeví zájem. Každá komponenta může být *publisher* i *subscriber*.

Publish-subscribe přidává mezi odesílatele a příjemce vrstvu indirekce. To má negativní vliv na latenci a potenciální škálovatelnost v závislosti na způsobu implementace. Obvykle není vhodné použít publish-subscribe v systému, který by musel dodržovat pevné termíny v reálném čase, protože by to zavádělo nejistotu v časech doručení zpráv. Vzor publish-subscribe má také tu nevýhodu, že poskytuje menší kontrolu nad pořadím zpráv a nezaručuje doručení zpráv (protože odesílatel nemůže vědět, zda příjemce naslouchá). Proto může být vzor publish-subscribe nevhodný pro složité interakce, kde je kritický sdílený stav.

Typickými příklady systémů, které využívají vzor publish-subscribe, jsou:

- GUI, ve kterých jsou vstupní akce uživatele na nízké úrovni považovány za události, jež jsou směrovány příslušným handlerům vstupu.

- Aplikace založené na MVC, v nichž jsou view komponenty upozorňovány na změnu stavu objektu modelu.
- Enterprise resource planning systémy (ERP), které integrují mnoho komponent, z nichž každá se zajímá pouze o podmnožinu systémových událostí.
- Sociální sítě, kde jsou přátelé/followeri informováni o změnách na webových stránkách dané osoby

Vzor publish-subscribe se používá k odesílání událostí a zpráv neznámé množině příjemců. Protože množina příjemců je pro producenta událostí neznámá, správnost producenta nemůže obecně záviset na těchto příjemcích. Proto lze přidávat nové příjemce, aniž by se museli měnit producenti.

Neznalost vzájemné identity komponent vede ke snadné modifikaci systému (přidávání nebo odebrání producentů a konzumentů dat), ale za cenu snížení výkonu za běhu, protože infrastruktura publish-subscribe je určitým druhem indirekce, což zvyšuje latenci. Pokud navíc publish-subscribe konektor zcela selže, selže celý systém.

Vzor publish-subscribe může mít několik podob:

- *List-based-publish-subscribe* je realizací vzoru, kdy každý publisher udržuje seznam subscriberů, kteří projeví zájem o příjem události. Tato verze vzoru je méně oddělena než ostatní dále uvedené typy, a proto neposkytuje tolik možností modifikace, ale může být poměrně efektivní z hlediska režie běhu. Pokud jsou navíc komponenty distribuované, existuje více bodů selhání a ne pouze jeden možný.
- *Broadcast-based-publish-subscribe* se liší od list-based-publish-subscribe tím, že publishers mají menší (nebo žádné) znalosti o subscriberech. Publishers jednoduše publikují události, které jsou následně vysílány. Subscribers (nebo v distribuovaném systému služby, které jednájí jménem subscriberů) zkoumají každou příchozí událost a určují, zda je zveřejněná událost zajímavá. Tato verze může být velmi neefektivní, pokud existuje velké množství zpráv a většina z nich není pro konkrétního odběratele zajímavá.
- *Content-based publish-subscribe* se liší od předchozích dvou variant, které jsou obecně klasifikovány jako *"topic-based"*. *Topics* jsou předem definované události nebo zprávy. Komponenta se přihlašuje ke všem událostem v rámci tématu. Naproti tomu *content* je mnohem obecnější. Každá událost je spojena se sadou atributů a je doručena subscriber komponentě pouze v případě, že tyto atributy odpovídají vzorům definovaným subscriberem.

V praxi je vzor publish-subscribe obvykle realizován nějakou formou middlewaru orientovaného na zprávy, kde je middleware realizován jako broker, který spravuje spojení a informační kanály mezi producenty a konzumenty. Tento middleware je často zodpovědný za transformaci zpráv (nebo protokolů zpráv) kromě směrování a někdy i ukládání zpráv. Vzor publish-subscribe tedy dědí silné a slabé stránky vzoru broker.[57]

### Shared-Data architektura

Různé výpočetní komponenty potřebují sdílet a manipulovat s velkým množstvím dat. Tato data nepatří pouze jedné z těchto komponent.

Shared-data vzor řeší problém, jak mohou systémy ukládat a manipulovat s trvalými daty, ke kterým přistupuje více nezávislých komponent.

Hlavní shared-data interakcí je výměna trvalých dat mezi více *přístupovými prvky* a alespoň jedním *úložištěm* sdílených dat. Výměna může být iniciována přístupovými prvky nebo datovým úložištěm. Typem konektoru je čtení a zápis dat. Obecný výpočetní model spojený se systémy sdílených dat spočívá v tom, že přístupové prvky provádějí operace, které vyžadují data z datového úložiště, a zapisují výsledky do jednoho nebo více datových úložišť. Tato data mohou být zobrazena a mohou s nimi pracovat jiné přístupové prvky. V čistě sdíleném

datovém systému interagují přístupové prvky pouze prostřednictvím jednoho nebo více sdílených datových úložišť. V praxi však systémy sdílených dat umožňují také přímé interakce mezi přístupovými prvky. Součástí shared-data úložišť poskytují sdílený přístup k datům, podporují perzistenci dat, řídí souběžný přístup k datům prostřednictvím správy transakcí, zajišťují odolnost proti chybám, podporují řízení přístupu a zajišťují distribuci a ukládání datových hodnot do mezipaměti.

Specializace shared-data vzoru se liší s ohledem na povahu ukládaných dat. Existující přístupy zahrnují relační, objektové, vrstevnaté a hierarchické struktury.

Přestože sdílení dat je pro většinu velkých a složitých systémů kritickým úkolem, je s tímto vzorem spojena řada potenciálních problémů. Na úložišti sdílených dat se může tvořit bottleneck. Z tohoto důvodu je optimalizace výkonu častým tématem výzkumu databází. Úložiště sdílených dat je také potenciálně jediným místem možného selhání. Producenti a konzumenti sdílených dat mohou být také úzce propojeni díky znalosti struktury sdílených dat.

Shared-data vzor je užitečný vždy, když jsou různé datové položky perzistentní a mají více přístupových prvků. Použití tohoto vzoru má za následek oddělení producenta dat od konzumentů. Tento vzor tedy podporuje modifikovatelnost, protože producenti nemají přímou znalost konzumentů. Konsolidace dat na jednom nebo více místech a přístup k nim společným způsobem usnadňuje ladění výkonu. Analýzy spojené s tímto vzorem se obvykle zaměřují na vlastnosti jako je konzistence dat, výkonnost, bezpečnost, soukromí, dostupnost, škálovatelnost a kompatibilita například s ostatními existujícími úložišti a jejich daty.

Pokud má systém více než jedno datové úložiště, je klíčovým problémem architektury mapování dat a výpočet z daných dat. K použití více úložišť může dojít proto, že data jsou přirozeně nebo historicky rozdělena do oddělitelných úložišť. V jiných případech mohou být data replikována do několika úložišť, aby se zvýšil výkon nebo dostupnost díky redundanci. Takové volby mohou výrazně ovlivnit výše uvedené vlastnosti.[57]

### Map-Reduce architektura

Podniky mají požadavek na rychlou analýzu obrovských objemů dat, které generují nebo ke kterým mají přístup, a to v řádech petabajtů. Příkladem mohou být záznamy interakcí na sociálních sítích, obrovská úložiště dokumentů nebo dat a dvojice webových odkazů <zdroj, cíl> pro vyhledávač. Programy pro analýzu těchto dat by se měly snadno psát, efektivně spouštět a být odolné vůči selhání hardwaru.

Pro mnoho aplikací s velmi rozsáhlými soubory dat stačí data roztrždit a seskupená data poté analyzovat. Problém, který map-reduce vzor řeší, spočívá v efektivním provádění distribuovaného a paralelního třídění velké sady dat a v poskytnutí jednoduchého prostředku pro programátora k určení analýzy, která má být provedena.

Vzor map-reduce vyžaduje tři části. Za prvé vyžaduje, specializovanou infrastrukturu, jež se stará o přidělení softwaru hardwarovým uzlům v prostředí masivně paralelních výpočtů a podle potřeby třídí data. Uzlem může být samostatný procesor nebo jádro ve vícejádrovém čipu. Za druhé a za třetí vyžaduje programátorem vytvořené funkce nazvané *map* a *reduce*.

Funkce *map* přijímá jako vstup klíč (*key1*) a sadu dat. Účelem funkce *map* je filtrovat a třídít soubor dat. Veškerá náročná analýza probíhá ve funkci *reduce*. Vstupní klíč funkce *map* slouží k filtrování dat. O tom, zda má být datový záznam zapojen do dalšího zpracování, rozhoduje funkce *map*. Ve funkci *map* je důležitý také druhý klíč (*key2*). Jedná se o klíč, který se používá pro třídění. Výstup funkce *map* se skládá z dvojice <*key2*, *value*>, kde *key2* je hodnota pro třídění a *value* je hodnota odvozena ze vstupního záznamu.

Třídění se provádí kombinací *map* funkce a infrastruktury. Každý záznam vyvedený z funkce *map* je zahashován pomocí *key2* do nějakého oddílu disku. Infrastruktura udržuje indexový soubor pro *key2* na daném oddílu disku. To umožňuje načítat hodnoty z oddílu v pořadí podle *key2*.

Výkonnost mapovací fáze map-reduce se zvyšuje tím, že existuje více instancí map, z nichž každá zpracovává jinou část souboru. Vstupní soubor je rozdělen na části a pro zpracování každé části je vytvořeno několik instancí funkce map. Map rozděluje svou část dat na několik částí podle programátorem vytvořené logiky.

Funkci reduce jsou poskytnuty všechny množiny dvojic  $\langle \text{key2}, \text{value} \rangle$  vytvořené všemi instancemi map v seřazeném pořadí. Reduce provede určitou programátorem zadanou analýzu a poté zveřejní výsledky této analýzy. Výstupní množina je téměř vždy mnohem menší než vstupní množina, odtud název reduce. Někdy se pro popis výsledné množiny dat používá termín "load".

Předpokládejme, že chceme průběžně analyzovat příspěvky na Twitteru za poslední hodinu a zjistit, jaká témata jsou aktuálně "trendy". To je obdoba počítání výskytů slov v milionech dokumentů. V takovém případě lze každému dokumentu (tweetu) přiřadit vlastní instanci funkce map. (Pokud uživatel nemá po ruce miliony procesorů, může kolekci tweetů rozdělit do skupin, jež odpovídají počtu procesorů v jeho procesorové farmě, a zpracovávat kolekci ve vlnách jednu skupinu za druhou). Nebo můžeme použít slovník, který nám poskytne seznam slov a každé funkci map přiřadí vlastní slovo, které bude hledat ve všech tweetech.

Může existovat i více instancí funkce reduce. Ty jsou obvykle uspořádány tak, že redukce probíhá postupně, přičemž každá fáze zpracovává menší seznam (s menším počtem instancí reduce) než předchozí fáze. Závěrečnou fází zpracovává jediná funkce reduce, která vytváří konečný výstup.

Vzor map-reduce samozřejmě není vhodný ve všech případech. Některé případy, kdy není vhodné tento vzor využít, jsou následující:

- Nejsou-li na vstupu velké soubory dat, protože pak režie map-reduce není opodstatněná
- Pokud nelze rozdělit množinu dat na podobně velké podmnožiny, neboť výhody paralelismu se tímto ztrácejí
- Pokud existují operace, které vyžadují vícenásobnou redukci. Jelikož bude složité je následně organizovat

Map-reduce vzor je základním stavebním prvkem softwaru některých nejznámějších jmen na internetu, včetně společností jako je Google, Facebook a eBay.[57]

### Víceúrovňová architektura

Při distribuovaném nasazení je často potřeba rozdělit infrastrukturu systému do různých podskupin. Důvody mohou být provozní nebo obchodní (například různé části infrastruktury mohou patřit různým organizacím).

Otázkou tedy je, jak můžeme rozdělit systém na několik výpočetně nezávislých exekučních strukturních skupin softwaru a hardwaru propojených komunikačním médiem. Toto se provádí za účelem zajištění specifických serverových prostředí optimalizovaných pro provozní požadavky a využití zdrojů.

Realizační struktury mnoha systémů jsou organizovány jako soubor logicky seskupených komponent. Každé seskupení se označuje jako *úroveň*. Seskupení komponent do úrovně může být založeno na různých kritériích jako je typ komponenty, sdílení stejného běhového prostředí, nebo stejný účel běhu.

Použití úrovně lze použít pro jakoukoli kolekci (nebo vzor) runtime komponent, ačkoli v praxi se nejčastěji používá v kontextu client-server vzorů. Úrovně zavádějí topologická omezení, jež limitují, jaké komponenty mohou komunikovat s jinými komponentami. Konkrétně mohou existovat konektory pouze mezi komponentami ve stejné úrovni nebo sídlícími v sousedních úrovních.

Hlavní slabinou vícevrstvé architektury jsou náklady na hardware a software. Zároveň je problémem i složitost návrhu a implementace. U jednoduchých systémů nemusí výhody vícevrstvé architektury ospravedlnit její počáteční a provozní náklady.

Úrovně nejsou komponenty, ale spíše logická seskupení komponent. Částé bývá zaměnění úrovně a vrstvy. Vrstvy jsou moduly (implementační jednotky), zatímco úrovně se vztahují pouze na runtime entity.

V současné době existuje několik publikovaných stylů tohoto vzoru. Společnost Microsoft proslazuje "*Tiered Distribution*", jež předepisuje konkrétní přiřazení komponent ve víceúrovňové architektuře k hardwaru, na kterém poběží. Podobně příručka "*WebSphere*" společnosti IBM popisuje řadu takzvaných "*topologií*" a spolu s kritérii kvalitativních atributů pro výběr mezi nimi.[57]

### 5.3.4 Návrhové vzory

Návrhové vzory se netýkají prvků jako jsou propojené seznamy a hashovací tabulky, které lze zakódovat do tříd a používat je znovu tak, jak jsou. Nejde ani o složité, doménově specifické návrhy pro celou aplikaci nebo subsystém. Návrhové vzory jsou popisy komunikujících objektů a tříd, jež jsou přizpůsobeny řešení obecného návrhového problému v konkrétním kontextu.

Návrhový vzor pojmenovává, abstrahuje a identifikuje klíčové aspekty společné struktury návrhu, které jsou užitečné pro vytvoření opakovaně použitelného OOP návrhu. Návrhový vzor identifikuje zúčastněné třídy a instance, jejich role, spolupráci a rozdělení odpovědností. Každý návrhový vzor se zaměřuje na konkrétní problém nebo otázku objektově orientovaného návrhu.[59]

Návrhové vzory se liší svou granularitou a úrovní abstrakce. Protože existuje mnoho návrhových vzorů, potřebujeme způsob, jak je uspořádat.

Návrhové vzory klasifikujeme podle dvou kritérií. První kritérium se nazývá *účel*, vyjadřuje, k čemu vzor slouží. Vzory mohou být buď *konstrukční*, *strukturální*, *behaviorální*, nebo *souběžné*. Konstrukční vzory se týkají procesu vytváření objektů. Strukturální vzory se zabývají složením tříd nebo objektů. Behaviorální vzory charakterizují způsoby interakce tříd nebo objektů a rozdělení odpovědností. Vzory souběžnosti se týkají programů, jež využívají více vláken.[59][60]

Druhé kritérium, nazývané *scope*, určuje, zda se vzor vztahuje primárně na třídy, nebo na objekty. Vzory tříd se zabývají vztahy mezi třídami a jejich podtřídami. Tyto vztahy se vytvářejí prostřednictvím dědičnosti, takže jsou staticky fixovány v době kompilace. Objektové vzory se zabývají vztahy mezi objekty, které lze za běhu měnit, a jsou tedy dynamičtější. Téměř všechny vzory používají do určité míry dědičnost. Takže vzory označované jako *vzory tříd* jsou ty, které se zaměřují na vztahy mezi třídami. Vzory, jež se zabývají objekty, jsou nazývány *objektové vzory*. [59]

#### 5.3.4.1 Vzory tříd

##### Factory Method

Znám také pod názvem *Virtual Constructor*. Definuje rozhraní pro vytvoření objektu, ale nechá podtřídy rozhodnout, kterou třídu instanciovat. Factory Method umožňuje třídě přenechat instanciaci na podtřídy.

Frameworky používají abstraktní třídy k definování a udržování vztahů mezi objekty. Framework je často zodpovědný i za vytváření těchto objektů.

Uvážíme-li framework pro aplikace, kde mohou uživatelé zobrazit více dokumentů. Dvěma klíčovými abstrakcemi v tomto rámci jsou třídy *Application* a *Document*. Obě třídy jsou abstraktní a klienti je musí konkretizovat, aby mohli realizovat své implementace specifické pro

danou aplikaci. Chceme-li například vytvořit aplikaci pro kreslení, definujeme třídy `DrawingApplication` a `DrawingDocument`. Třída `Application` je zodpovědná za správu dokumentů a bude je vytvářet podle potřeby.

Vzhledem k tomu, že konkrétní podtřída dokumentu, jež se má instanciovat, je specifická pro danou aplikaci, nemůže třída `Application` předvídat, jakou podtřidu dokumentu má instanciovat. Třída `Application` pouze ví, kdy má být vytvořen nový dokument, nikoliv jaký druh dokumentu má být vytvořen. To vytváří dilema, neboť framework musí instanciovat třídy, ale ví pouze o abstraktních třídách, které instanciovat nemůže.

Řešení nabízí vzor `Factory Method`. Jelikož zapouzdřuje znalosti o tom, kterou podtřidu dokumentu vytvořit, a přesouvá tyto znalosti mimo framework. Podtřídy `Application` předefinují abstraktní operaci `CreateDocument` na `Application` tak, aby vracela příslušnou podtřidu `Document`. Jakmile je podtřída `Aplikace` instanciována, může pak instanciovat dokumenty specifické pro aplikaci, aniž by znala jejich třídu. `CreateDocument` nazýváme factory metodou, protože je zodpovědná za vytvoření objektu.

`Factory Method` by měla být použita vždy pokud třída neví jaký typ objektu má vytvořit, třída chce, aby její podtřídy určovaly objekty, které vytváří a v případě kdy třídy delegují odpovědnost na jednu z několika pomocných podtříd a uživatel chce lokalizovat znalost toho, která pomocná podtřída je delegátem.

Tento vzor bere v potaz následující účastníky:

- **Product** definuje rozhraní objektů, které factory metoda vytváří
- **ConcreteProduct** implementuje rozhraní abstraktní třídy `Product`
- **Creator** deklaruje metodu `factory`, která vrací objekt typu `Product`. `Creator` může také definovat výchozí implementaci tovární metody, která vrací výchozí objekt typu `ConcreteProduct`. Zároveň může zavolat tovární metodu pro vytvoření objektu `Product`
- **ConcreteCreator** přepisuje `factory` metodu tak, aby vracela instanci produktu `ConcreteProduct`

`Creator` spoléhá na to, že jeho podtřídy definují `factory` metodu tak, aby vracela instanci příslušného produktu `ConcreteProduct`.

`Factory` metody odstraňují nutnost vázat do kódu třídy specifické pro danou aplikaci. Kód se zabývá pouze rozhraním `Product`, proto může pracovat s libovolnými uživatelsky definovanými třídami `ConcreteProduct`.

Potenciální nevýhodou vzoru `Factory Method` je to, že klienti mohou být nuceni vytvořit podtřidu třídy `Creator` jen proto, aby vytvořili konkrétní objekt `ConcreteProduct`. Vytvoření podtřídění je v pořádku, pokud klient stejně musí vytvořit podtřidu třídy `Creator`, ale v opačném případě se klient musí vypořádat s dalším bodem vývoje.[59]

## Adapter

Jelikož *Adapter*, známý taktéž jako *Wrapper*, je zároveň názvem vzoru třídy i vzoru objektu a oba z těchto vzorů vycházejí ze stejného základu, popíši je pouze zde a mezi objektovými vzory již objektový typ tohoto vzoru zmiňovat nebudu.

Chceme-li převést rozhraní třídy na jiné rozhraní, které klienti očekávají, využijeme právě vzoru `Adapter`. `Adapter` umožňuje spolupráci tříd, které by jinak nemohly fungovat kvůli nekompatibilním rozhraním. Této adaptace lze docílit dvěma způsoby. Prvním je vytvořit novou třídu, jež bude dědit rozhraní jedné z daných tříd a implementaci té druhé, nebo druhým způsobem je opět vytvořit novou třídu, ale tentokrát tato vytvořená třída obsahuje instanci jedné z nekompatibilních tříd a implementuje interface třídy druhé. Oba z těchto přístupů odpovídají `Adapter` vzoru.



Wrapper by měl být použit v případech, kdy chceme použít již existující třídu, ale její interface není ve vhodném formátu. Dále pokud chceme vytvořit znovupoužitelnou třídu, která spolupracuje s nesouvisejícími, nebo předem neurčenými třídami, tj. třídami, jež nemusí mít kompatibilní rozhraní. Nebo pokud se bavíme pouze o objektové verzi, tak v případě, kdy nastane potřeba použít několik existujících podtříd, ale je nepraktické přizpůsobovat jejich rozhraní upravením každé podtřídy. Objektový Adapter může přizpůsobit rozhraní své nadřazené třídy.

Jednotlivými účastníky při použití tohoto vzoru jsou:

- **Target** definuje doménově specifické rozhraní, jež klient používá
- **Client** spolupracuje s objekty odpovídajícími rozhraní Target
- **Adaptee** definuje stávající rozhraní, které je třeba upravit
- **Adapter** přizpůsobí rozhraní třídy Adaptee na rozhraní třídy Target

Adaptéry se liší množstvím práce, jež odvedou při přizpůsobování Adaptee rozhraní Target. Je možné provést celou škálu konverzí, od jednoduché konverze rozhraní, například změna názvů operací, až po podporu zcela odlišné sady operací. Množství práce, kterou adaptér vykoná, závisí na tom, jak moc je cílové rozhraní podobné rozhraní Adaptee. Třída je lépe znovupoužitelná, pokud jsou minimalizovány předpoklady, které musí jiné třídy učinit, aby ji mohly použít. Zabudování adaptace rozhraní do třídy eliminuje předpoklad, že ostatní třídy vidí stejné rozhraní. Jinak řečeno, adaptace rozhraní umožňuje začlenit třídu do existujících systémů, jež mohou očekávat jiná rozhraní třídy.[59]

## Interpreter

Vzor *Interpreter* lze použít chceme-li definovat reprezentaci jazyka pro jeho gramatiku spolu s interpretem, který tuto reprezentaci používá k interpretaci vět v jazyce.

Pokud se určitý druh problému vyskytuje dostatečně často, je vhodné vyjádřit případy tohoto problému jako věty v jednoduchém jazyce. Pak lze sestavit interpret, který problém řeší interpretací těchto vět. Častým problémem je například hledání řetězců, které odpovídají nějakému vzoru. Regulární výrazy jsou standardním jazykem pro zadávání vzorů řetězců. Namísto vytváření vlastních algoritmů pro porovnávání jednotlivých vzorů s řetězcem by vyhledávací algoritmy mohly interpretovat regulární výraz, který specifikuje množinu řetězců, jimž se má odpovídat.

Vzor *Interpreter* popisuje, jak definovat gramatiku pro jednoduché jazyky, reprezentovat věty v jazyce a interpretovat tyto věty.

Vzor *Interpreter* by měl být použit v případě, že existuje jazyk, který je třeba interpretovat, a příkazy v tomto jazyce lze reprezentovat jako abstraktní syntaktické stromy. Vzor *Interpreter* funguje nejlépe, když je gramatika jednoduchá. U složitých gramatik se hierarchie tříd pro gramatiku stává rozsáhlou a nezvládnutelnou. V takových případech jsou lepší alternativou nástroje jako jsou generátory parserů, jež dokážou interpretovat výrazy bez vytváření abstraktních syntaktických stromů, což může ušetřit místo a případně i čas. Nebo tento vzor lze využít pokud účinnost není rozhodujícím faktorem. Nejefektivnější interprety se obvykle neimplementují přímou interpretací parsovacích stromů, ale nejprve jejich překladem do jiné formy. Například regulární výrazy se často převádějí na stavové stroje. Ale i v takovém případě lze překladač implementovat pomocí vzoru *Interpreter*, takže vzor je stále použitelný.

Zúčastněnými strukturami při použití tohoto vzoru jsou:

- **AbstractExpression** deklaruje abstraktní operaci *Interpret*, jež je společná všem uzlům v abstraktním syntaktickém stromu
- **TerminalExpression** implementuje operaci *Interpret* spojenou s terminálními symboly v gramatice. Instance *TerminalExpression* je vyžadována pro každý koncový symbol ve větě

- **NonterminalExpression** je vyžadována pro každé pravidlo  $R := R_1R_2 \dots R_n$  v gramatice. Zároveň udržuje proměnné instance typu `AbstractExpression` pro každý ze symbolů  $R_1aR_n$ . A implementuje operaci `Interpret` pro neterminální symboly v gramatice. `Interpret` se obvykle volá rekurzivně na proměnné reprezentující  $R_1aR_n$
- **Context** obsahuje informace, které jsou pro interpreta globální
- **Client** vytvoří abstraktní syntaktický strom reprezentující konkrétní větu v jazyce, který gramatika definuje. Abstraktní syntaktický strom je sestaven z instancí tříd `NonterminalExpression` a `TerminalExpression`. `Client` také vyvolává operaci `Interpret`

Díky použití vzoru `Interpreter` lze gramatiku snadno měnit a rozšiřovat. Protože vzor používá třídy pro reprezentaci pravidel gramatiky, může se ke změně nebo rozšíření gramatiky použít dědičnost. Stávající výrazy lze měnit postupně a nové výrazy lze definovat jako variace na staré výrazy. Samotná implementace gramatiky je také snadná. Třídy definující uzly v abstraktním syntaktickém stromu mají podobnou implementaci. Tyto třídy je snadné napsat a často lze jejich generování automatizovat pomocí překladače nebo generátoru parseru.

K nevýhodám vzoru `Interpreter` patří, že definuje alespoň jednu třídu pro každé pravidlo v gramatice (pravidla gramatiky definovaná pomocí Backusova–Naurova formy (BNF) mohou vyžadovat více tříd). Proto mohou být gramatiky obsahující mnoho pravidel náročné na správu a údržbu. Pro zmírnění tohoto problému je vhodné použít jiné návrhové vzory. Pokud je však gramatika velmi složitá, jsou vhodnější jiné techniky, například generátory parserů nebo kompilátorů.[59]

## GRASP

*GRASP* je behaviorální vzor řešící základní principy přiřazení zodpovědností jednotlivým třídám. Zodpovědnost je myšlen úkol, jež má třída řešit. Existuje mnoho způsobů, jak rozdělit zodpovědnosti mezi třídy a neexistuje jediné správné řešení takového rozdělení. *GRASP* je souborem následujících devíti principů:[61]

1. **Informační expert** řeší problém výběru vhodného objektu pro vykonání dané zodpovědnosti. Zodpovědnost by měla být přidělena informačnímu expertovi – prvku, který má informace potřebné pro splnění této zodpovědnosti.
2. **Tvůrce** tvůrce se zabývá otázkou, komu přidělit zodpovědnost za vytváření objektů. Znění principu je následující: Třídě B přiřadte zodpovědnost za vytvoření třídy A, pokud je splněn alespoň z následujících požadavků:
  - B agreguje objekty A
  - B obsahuje objekty A
  - B zaznamenává instance objektů A
  - B úzce spolupracuje s objekty A
  - B je vlastníkem inicializačních dat A, neboli B je informačním expertem pro objekty A
3. **Slabá provázanost** popisuje, jak silně je jeden element provázán s druhým. Element, jenž má malou provázanost není závislý na mnoha třídách, tedy využívá co nejméně tříd pro svoj činnost. Díky tomu je elementům jednodušší porozumět v izolaci. Samotný princip říká, že by zodpovědnost měla být přiřazena tak, aby provázanost zůstala co nejmenší.
4. **Vysoká soudržnost** v kontextu OOP soudružnost, přesněji *funkcionální soudružnost*, je míra toho, jak moc spolu zodpovědnosti jednoho elementu souvisejí. Díky dodržení tohoto principu jsou následné třídy lehké na údržbu, pochopení a znovupoužití. Malá soudržnost většinou značí velkou míru detailu použité abstrakce. Tedy princip požaduje přiřazovat zodpovědnosti tak, aby byla zachována co největší soudružnost.
5. **Controller** odpovídá na otázku toho, kdo by měl být zodpovědný za zpracování *vstupních systémových událostí*. Vstupní systémová událost je událost generovaná externím aktérem.

Události jsou asociovány se systémovými operacemi, jež jsou reakcí na systémové události. Dle tohoto principu je přiřazení zodpovědnosti za příjem či zpracování zprávy systémové události jedné z tříd reprezentující následující možnosti:

- Reprezentuje celý systém, zařízení nebo subsystém
  - Představuje scénář případu užití, v jehož rámci dochází k systémové události, často s názvem končícím slovem Handler, Coordinator nebo Session
6. **Polymorfismus** řeší problém nakládání s alternativami na základě jejich typu a jak vytvořit zásuvný softwarový modul. Polymorfismus v tomto kontextu znamená dávat stejný název službám rozdílných objektů. Princip polymorfismu v kontextu GRASP zní následovně: Pokud chování závisí na typu objektu (třídě), přiřadte zodpovědnost za toto chování pomocí polymorfických metod třídě, na které toto chování závisí.
  7. **Indirection** řeší přiřazení zodpovědnosti bez vytvoření přímé vazby. Řešení jež poskytuje je přiřazení zodpovědnosti prostředníkovi, který zprostředkuje propojení mezi prvky a zamezí tak vzniku přímé vazby. Použití prostředníků snižuje provázanost kódu.
  8. **Chráněná změna** se zabývá následující otázkou: Jak navrhnout objekty, subsystémy a systémy, aby výkyvy nebo nestabilita těchto prvků neměly nežádoucí dopad na ostatní prvky? Řešení, jež nabízí je identifikace míst pravděpodobných změn a nestability s následným přiřazením zodpovědnosti stabilním rozhraním, která budou kolem daných míst vytvořena.
  9. **Pure fabrication** je použita v případech kdy by přiřazení odpovědnosti některé třídě představující objekt z problémové domény narušilo soudržnost (cohesion) této třídy, zvýšilo její provázání (coupling) nebo porušilo jiné principy. V takovém případě je vhodné vytvořit novou třídu, která nepředstavuje nic z oblasti domény – třídu vytvořenou pouze pro zlepšení kvality návrhu. Takovéto třídy nazýváme *pure fabrication*. Třídy představující pure fabrication nerepresentují nic, co by existovalo v rámci problémové domény. Tyto třídy tedy nenajdeme v doménovém modelu vzniklém při analýze požadavků. Pure fabrication by mělo být použito pouze pokud ostatní řešení, jež jej nevyžadují, mají zjevné nevýhody.

### 5.3.4.2 Konstrukční objektové vzory

#### Abstract factory

Vzor *Abstract factory* poskytuje rozhraní pro vytváření rodin příbuzných nebo závislých objektů, aniž by bylo nutné specifikovat jejich konkrétní třídy.

Zvažme sadu nástrojů UI, jež podporuje více standardů vzhledu. Různé standardy definují různé vzhledy a chování widgetů UI jako jsou posuvníky, okna a tlačítka. Aby byla aplikace přenositelná napříč standardy vzhledu a prostředí, neměla by své widgety natvrdo kódovat pro určitý vzhled a prostředí. Instancování tříd widgetů specifických pro vzhled a styl v celé aplikaci ztěžuje pozdější změnu vzhledu a stylu.

Tento problém můžeme vyřešit definováním abstraktní třídy *WidgetFactory*, která deklaruje rozhraní pro vytváření každého základního druhu widgetu. Pro každý druh widgetu existuje také abstraktní třída a konkrétní podtřídy implementují widgety pro konkrétní standardy vzhledu a chování. Rozhraní *WidgetFactory* má operaci, která vrací nový objekt widgetu pro každou abstraktní třídu widgetu. Klienti volají tyto operace, aby získali instance widgetů, ale nejsou si vědomi, které konkrétní třídy používají. Klienti tak zůstávají nezávislí na standardu vzhledu a chování.

Pro každý standard vzhledu a chování existuje konkrétní podtřída *WidgetFactory*. Každá podtřída implementuje operace pro vytvoření příslušného widgetu pro daný vzhled a chování. Klienti vytvářejí widgety výhradně prostřednictvím rozhraní *WidgetFactory* a nemají žádné znalosti o třídách, které implementují widgety pro konkrétní standard. Jinými slovy, klienti se musí zavázat pouze k rozhraní definovanému abstraktní třídou, nikoli ke konkrétní třídě.

Vzor Abstract factory by měl být použit v těchto případech:

- Systém by měl být nezávislý na tom, jak jsou jeho produkty vytvářeny, skládány a reprezentovány
- Systém by měl být nakonfigurován s jednou z více skupin produktů
- Skupina souvisejících objektů produktů je navržena tak, aby se používala společně, a je potřeba dodržovat dané omezení
- Kdy je snaha poskytnout knihovnu tříd produktů a je chtěné odhalit pouze jejich rozhraní, nikoli jejich implementaci

Za normálních podmínek je za běhu vytvořena pouze jedna factory daného druhu. Každá factory má svoji abstraktní a konkrétní verzi.

Vzor má následující účastníky při jeho použití:

- **AbstractFactory** deklaruje rozhraní pro operace, které vytvářejí abstraktní objekty produktu
- **ConcreteFactory** implementuje operace pro vytváření konkrétních objektů produktu
- **AbstractProduct** deklaruje rozhraní pro typ objektu produktu
- **ConcreteProduct** definuje objekt produktu a implementuje AbstractProduct rozhraní
- **Client** používá pouze rozhraní deklarované pomocí AbstractFactory a AbstractProduct

Vzor Abstract factory izoluje jednotlivé konkrétní třídy. Díky tomu vypomáhá s kontrolou tříd objektů, jež aplikace vytvoří. Jelikož továrna zapouzdřuje odpovědnost a proces vytváření objektů, izoluje tím klienty od implementačních tříd. Klienti manipulují s instancemi prostřednictvím jejich abstraktních rozhraní. Názvy tříd produktů jsou izolovány v implementaci konkrétní továrny, tedy v kódu klienta se neobjevují. Třída konkrétní továrny se v aplikaci objeví pouze jednou – tedy tam, kde je instanciována. Díky tomu lze snadno změnit konkrétní továrnu, kterou aplikace používá. Může používat různé konfigurace produktů jednoduše tím, že změní konkrétní továrnu. Protože abstraktní továrna vytváří kompletní skupinu produktů, mění se celá skupina produktů najednou.

Nevýhodou je náročnost rozšíření již existující AbstractFactory o nový typ. Je to proto, že rozhraní AbstractFactory určuje množinu typů, které lze vytvořit. Podpora nových druhů vyžaduje rozšíření rozhraní továrny, což zahrnuje změnu třídy AbstractFactory a všech jejích podtříd.[59]

## Builder

Hlavním cílem *Builderu* je oddělit konstrukci složitého objektu od jeho reprezentace, aby stejný konstrukční proces mohl vytvořit různé reprezentace.

Motivací k jeho využití je například převedení příchozího textu do jiného formátu. Počet možných konverzí je neomezený. Mělo by tedy být snadné přidat novou konverzi, bez nutnosti upravovat *reader*. Řešením je vytvořit abstraktní třídu, tzv. *converter*, jež pro každý typ převodu bude mít jinou podtřídu. Každý druh podtřídy přebírá mechanismus pro vytvoření a sestavení složitého objektu a vkládá jej za abstraktní rozhraní. Tímto způsobem je samotná implementace skryta před třídou *reader*.

Vzor builder lze využít, pokud algoritmus pro vytvoření složitého objektu by měl být nezávislý na částech, ze kterých se objekt skládá, a na způsobu jejich sestavení. A zároveň pokud proces konstrukce musí umožňovat různé reprezentace konstruovaného objektu.

Jednotliví účastníci při použití tohoto vzoru jsou:

- **Builder** určuje abstraktní rozhraní pro vytváření částí objektu Product

- **ConcreteBuilder** konstruuje a sestavuje části produktu implementací rozhraní Builder. Dále definuje a sleduje reprezentace, které vytváří a poskytuje rozhraní pro získání produktu
- **Director** zkonstruuje objekt pomocí rozhraní Builder
- **Product** představuje komplexní objekt, který je předmětem konstrukce. ConcreteBuilder vytváří vnitřní reprezentaci produktu a definuje proces, kterým je sestaven. Zároveň obsahuje třídy, jež definují jednotlivé části, včetně rozhraní pro sestavení částí do konečného výsledku

Objekt Builder poskytuje třídě director abstraktní rozhraní pro konstrukci produktu. Rozhraní umožňuje třídě builder skrýt reprezentaci a vnitřní strukturu produktu. Skrývá také způsob, jakým se výrobek sestavuje. Protože je výrobek konstruován prostřednictvím abstraktního rozhraní, stačí ke změně vnitřní reprezentace výrobku definovat nový druh konstruktoru. Vzor také zlepšuje modularitu tím, že zapouzdřuje způsob konstrukce a reprezentace objektu. Klienti nemusí vědět nic o třídách, které definují vnitřní strukturu produktu, takové třídy se v rozhraní Builderu neobjevují. Každý ConcreteBuilder obsahuje veškerý kód pro vytvoření a sestavení určitého druhu produktu. Kód je napsán jednou, různé director třídy jej pak mohou opakovaně použít k sestavení variant produktu ze stejné sady částí.

Na rozdíl od konstrukčních vzorů, které vytvářejí produkty najednou, vzor Builder vytváří produkt krok za krokem pod kontrolou třídy director. Teprve když je výrobek dokončen, director jej od builderu převezme. Proto rozhraní Builder odráží proces konstrukce produktu více než ostatní konstrukční vzory. To umožňuje jemnější kontrolu nad procesem konstrukce a následně i nad vnitřní strukturou výsledného produktu.[59]

## Prototype

Vzor *Prototype* určuje druhy objektů, které se mají vytvořit pomocí prototypové instance, a vytváří nové objekty kopírováním tohoto prototypu.

Vzor Prototype by měl být použit v případech, kdy má být systém nezávislý na tom, jak jsou jeho produkty vytvářeny, skládány a reprezentovány. Zároveň pokud je splněn jeden z následujících faktorů:

- pokud jsou třídy, které se mají instanciovat, určeny za běhu, například dynamickým načítáním
- pokud instance třídy mohou mít jen jednu z několika různých kombinací stavu. Může být výhodnější vytvořit odpovídající počet prototypů a klonovat je, než třídu instanciovat ručně pokaždé s příslušným stavem.

Při použití vzoru je využito následujících struktur:

- **Prototype** deklaruje rozhraní pro klonování sebe sama
- **ConcretePrototype** implementuje operaci pro klonování sebe sama
- **Client** vytvoří nový objekt požádáním prototypu o naklonování

Prototyp má mnoho stejných vlastností jako abstract factory a builder. Tj. skrývá před klientem konkrétní třídy produktů, čímž snižuje počet jmen, která klienti znají. Zároveň tyto vzory umožňují klientovi pracovat s třídami specifickými pro danou aplikaci, aniž by je musel modifikovat. Další výhodou je možnost vytvořit novou konkrétní třídu produktu registrací nové prototypové instance u uživatele. Tím se stává o něco flexibilnější než jiné vzory, protože klient může prototypy přidávat a odstraňovat za běhu programu.

Vysoce dynamické systémy umožňují definovat nové chování prostřednictvím kompozice objektů, například určením hodnot proměnných objektu, a nikoli definováním nových tříd. Nové typy objektů efektivně definuje instanciováním existujících tříd a registrací instancí jakožto

prototypů klientských objektů. Klient může vykazovat nové chování delegováním odpovědnosti na prototyp. Tento druh návrhu umožňuje uživatelům definovat nové třídy bez programování. Klonování prototypu je vlastně podobné instanciaci třídy. Vzor Prototyp tedy může výrazně snížit počet tříd, které systém potřebuje.

Mnoho aplikací vytváří objekty z částí a dílčích částí. Například editory pro návrh obvodů vytvářejí obvody z dílčích obvodů. Takové aplikace často umožňují pro větší pohodlí instanciovat složité, uživatelem definované struktury. Vzor Prototype podporuje i toto. Jednoduše přidáme tento dílčí obvod jako prototyp do palety dostupných prvků obvodu. Pokud objekt složeného obvodu implementuje funkci Clone jako hlubokou kopii, mohou být prototypem obvodů s různými strukturami.

Hlavním nedostatkem vzoru prototype je, že každá podtřída Prototype musí implementovat operaci Clone, což může být obtížné. Například přidání Clone je náročné, pokud uvažované třídy již existují. Implementace Clone může být obtížná, pokud třída obsahuje objekty, jež nepodporují kopírování nebo mají kruhové reference.[59]

### Singleton

*Singleton* zajišťuje, aby třída měla pouze jednu instanci, a poskytuje k ní globální přístupový bod.

Při potřebě zajistit existenci pouze jedné instance v celém programu je vhodné, aby objekt sám byl zodpovědný za sledování své jediné instance. Objekt může zajistit, aby nemohla být vytvořena žádná další instance (tím, že zachytí požadavky na vytvoření nových objektů), a může poskytnout způsob, jak k instanci přistupovat. Takto vypadá objekt využívající vzor Singleton.

Výhodami užití singletonu je řízený přístup k jediné instanci. Neboť Singleton třída zapouzdřuje svou jedinou instanci, může mít přímou kontrolu nad tím, jak a kdy k ní klienti přistupují. Další výhodou je zmenšení jmenného prostoru, neboť namísto vytvoření globální proměnné je vytvořen singletonu.[59]

### 5.3.4.3 Strukturální objektové vzory

#### Bridge

Vzor *Bridge* řeší oddělení abstrakce třídy od její implementace, aby se následně obě třídy mohly měnit nezávisle.

Pokud může mít abstrakce jednu z několika možných implementací, použití dědičnosti je obvyklým způsobem, jak je přizpůsobit. Abstraktní třída definuje rozhraní abstrakce a konkrétní podtřídy ji implementují různými způsoby. Tento přístup však není vždy dostatečně flexibilní. Dědičnost trvale váže implementaci k abstrakci, což ztěžuje nezávislé úpravy, rozšiřování a opětovné použití abstrakcí a implementací.

Vzor Bridge tyto problémy řeší tím, že abstrakci třídy a její implementaci umísťuje do oddělených hierarchií tříd. Tudíž existuje jedna hierarchie tříd pro rozhraní a druhá samostatná hierarchie pro implementaci.

Vzor Bridge je vhodné použít v situacích, kdy se chceme vyhnout trvalé vazbě mezi abstrakcí a její implementací. Tak tomu může být například v případě, kdy je třeba vybrat nebo přepnout implementaci za běhu. Dále pokud by měly abstrakce i jejich implementace být rozšiřitelné pomocí podtříd. V tomto případě vzor Bridge umožňuje kombinovat různé abstrakce a implementace a rozšiřovat je nezávisle na sobě.

Struktury, jež jsou přítomny při použití tohoto vzoru jsou:

- **Abstraction** definuje rozhraní abstrakce. A zároveň udržuje referenci na objekt typu `Implementor`

- **RefmedAbstraction** rozšiřuje rozhraní definované funkcí **Abstraction**
- **Implementor** definuje rozhraní pro implementační třídy. Toto rozhraní nemusí přesně odpovídat rozhraní abstrakce. Ve skutečnosti se obě rozhraní mohou značně lišit. Typicky rozhraní **Implementor** poskytuje pouze primitivní operace a **Abstraction** definuje operace vyšších úrovní založené na těchto primitivech
- **ConcretImplementor** realizuje rozhraní **Implementor** a definuje jeho konkrétní implementaci

Díky použití tohoto vzoru není realizace trvale vázána na rozhraní. Uskutečnění abstrakce lze nakonfigurovat za běhu. Oddělení tříd **Abstraction** a **Implementor** také eliminuje závislosti na implementaci v době kompilace. Změna třídy implementace nevyžaduje překompilování třídy **Abstraction** a jejích klientů. Tato vlastnost je zásadní, pokud je třeba zajistit vzájemnou kompatibilitu mezi různými verzemi knihovny tříd. Toto oddělení navíc podporuje vrstvení, které může vést k lepší strukturovanosti systému. Vysokoúrovňová část systému musí vědět pouze o třídách **Abstraction** a **Implementor**. Zároveň vylepšuje rozšiřitelnost. Hierarchie tříd **Abstraction** a **Implementor** lze rozšiřovat nezávisle. Dále také odstiňuje klienta od implementačních detailů.[59]

## Composite

*Composite* je vzor složení objektů do stromových struktur, které reprezentují hierarchie částí a celků. *Composite* umožňuje klientům zacházet s jednotlivými objekty a kompozicemi objektů jednotně.

Grafické aplikace, jako jsou editory výkresů a systémy pro zachycení schémat, umožňují uživatelům vytvářet složitá schémata z jednoduchých komponent. Uživatel může seskupovat komponenty do větších celků, které lze zase seskupovat do ještě větších celků. Jednoduchá implementace by mohla definovat třídy pro grafické primitivy. Těmito třídami mohou být například **Text** a **Lines** a samozřejmě i další, které by fungovaly jako kontejnery pro další primitiva.

Tento přístup má však jeden problém. Kód, který tyto třídy používá, musí s primitivními a kontejnerovými objekty zacházet odlišně, i když s nimi uživatel většinou zachází stejně. Vzor *Composite* popisuje, jak používat rekurzivní kompozici, aby klienti nemuseli toto rozlišování provádět.

Klíčem ke *Composite* vzoru je abstraktní třída, která reprezentuje primitiva i jejich kontejnery. Pro výše zmíněný grafický systém je touto třídou **Graphic**. **Graphic** deklaruje operace jako **Draw**, které jsou specifické pro grafické objekty. Deklaruje také operace, které jsou společné všem kompozitním objektům jako jsou metody pro přístup a správu jejich potomků.

Podtřídy **Lines** a **Text** definují primitivní grafické objekty. Tyto třídy implementují funkci **Draw** pro kreslení čar a textu. Protože primitivní grafika nemá podřízené grafické třídy, žádná z těchto podtříd neimplementuje operace související s podřízenými třídami.

Nyní mějme třídu **Picture**, jež definuje souhrn objektů **Graphic**. Třída **Picture** implementuje funkci **Draw** pro volání funkce **Draw** na svých potomcích a podle toho implementuje operace související s potomky. Vzhledem k tomu, že rozhraní **Picture** odpovídá rozhraní **Graphic**, mohou objekty **Picture** rekurzivně skládat další objekty **Picture**. Díky tomu dostáváme rekurzivní kompozici a je splněn vzor *Composite*.

Vzor *Composite* lze využít, pokud chceme reprezentovat hierarchie částí a celků objektů. Nebo pokud chceme, aby **Client** třídy mohli ignorovat rozdíl mezi složenými objekty a jednotlivými objekty. Třída **Client** bude ke všem objektům ve složené struktuře přistupovat jednotně.

Struktury, jež jsou účastníky při použití tohoto vzoru jsou:

- **Component** deklaruje rozhraní pro objekty kompozice. Dále implementuje výchozí chování rozhraní společné pro všechny třídy, pokud je to vhodné a deklaruje rozhraní pro přístup ke svým podřízeným komponentám a jejich správu

- **Leaf** představuje objekty listů v kompozici. List nemá žádné potomky. Definuje chování primitivních objektů kompozice
- **Composite** definuje chování komponent, které mají potomky. Ukládá podřízené třídy. Implementuje operace související s potomky v rozhraní Component
- **Client** manipuluje s objekty kompozice prostřednictvím rozhraní Component

Client používá rozhraní třídy Component k interakci s objekty ve složené struktuře. Pokud je příjemcem Leaf, pak je požadavek zpracován přímo. Pokud je příjemcem Composite, pak obvykle předává požadavky svým podřízeným komponentám, případně provádí další operace před a/nebo po předání.

Vzor Composite definuje hierarchie tříd složené z primitivních a složených objektů. Primitivní objekty mohou být složeny do složitějších objektů, které zase mohou být složeny a tak dále. Kdekoli Client očekává primitivní objekt, může také přijmout složený objekt. Dále zjednodušuje klienta. Client může zacházet s kompozitními strukturami a jednotlivými objekty jednotně. Client obvykle neví, zda má co do činění s Leaf nebo Composite komponentou. To zjednodušuje klientský kód, protože odpadá nutnost psát funkce ve stylu tag-and-case-statement nad třídami, které definují kompozici. Zároveň tento vzor usnadňuje přidávání nových druhů komponent. Nově definované podtřídy Composite nebo Leaf automaticky pracují s existujícími strukturami a klientským kódem. Client třídy není třeba měnit pro nové třídy složek.

Na druhou stranu vzor Composite může způsobit přílišnou obecnost návrhu. Nevýhodou snadného přidávání nových komponent je, že ztěžuje omezení komponent kompozice. V některých případech je žádané, aby kompozice obsahovala pouze určité komponenty. S Composite se nelze spoléhat na systém pro kontrolu typových omezení. Místo toho je nutné použít runtime kontroly.[59]

## Decorator

Chceme-li dynamicky připojit dalších odpovědností k objektu, *decorators* poskytují flexibilní alternativu k rozšiřování funkcí pomocí podtříd.

Někdy chceme přidat odpovědnost k jednotlivým objektům, nikoli k celé třídě. Sada nástrojů pro GUI by například měla umožnit přidat vlastnosti nebo chování do libovolné komponenty uživatelského rozhraní.

Jedním ze způsobů, jak přidávat odpovědnosti, je dědičnost. Zdědění rámečku z jiné třídy vytvoří rámeček kolem každé instance podtřídy. Tento přístup je nevhodný, neboť volba rámečku se provádí staticky. Klient nemůže ovlivnit, jak a kdy má komponentu dekorovat rámečkem.

Flexibilnějším přístupem je obalit komponentu jiným objektem, jenž přidá daný rámeček. Objekt obalující komponentu se nazývá Decorator. Decorator odpovídá rozhraní komponenty, kterou zdobí, tudíž klient o jeho existenci neví. Decorator předává požadavky komponentě a může provádět další akce (například vykreslení rámečku) před nebo po předání. Transparentnost umožňuje rekurzivně vnořovat dekorátory, čímž umožňuje neomezený počet přidávaných odpovědností.

Decorator použijemem chceme-li přidávat odpovědnosti jednotlivým objektům dynamicky a transparentně, tj. bez vlivu na ostatní objekty. Decorator lze použít i pro odpovědnosti obejektu, jež lze odejmout, nebo pokud je rozšíření pomocí podtříd nepraktické. Někdy je možné velké množství nezávislých rozšíření, které by vedlo k explozi podtříd pro podporu každé kombinace.

Při použití vzoru Decorator vznikají následující objekty účastníků:

- **Component** definuje rozhraní pro objekty, ke kterým lze dynamicky přidávat odpovědnosti



- **ConcreteComponent** definuje objekt, ke kterému lze připojit další povinnosti
- **Decorator** udržuje odkaz na objekt Component a definuje rozhraní, které je umožňuje vytvořit v souladu s rozhraním Component
- **ConcreteDecorator** přidává odpovědnosti do komponenty

Decorator poskytuje flexibilnější způsob přidávání odpovědností k objektům, než jakého lze docílit při statické (vícenásobné) dědičnosti. Pomocí dekorátorů lze odpovědnosti přidávat a odebrat za běhu jednoduše jejich připojováním a odpojováním. Naproti tomu dědičnost vyžaduje vytvoření nové třídy pro každou další odpovědnost. Tím vzniká mnoho tříd a zvyšuje se složitost systému. Poskytování různých tříd Decorator pro určitou třídu Component navíc umožňuje kombinovat a přizpůsobovat odpovědnosti. Zároveň se tento vzor vyhýbá třídám s vysokým počtem funkcí v hierarchii. Decorator nabízí přístup k přidávaným odpovědnostem podle toho, jak jsou přidávány. Namísto snahy podporovat všechny předvídatelné funkce ve složité, přizpůsobitelné třídě lze definovat jednoduchou třídu a přidávat funkce postupně pomocí objektů Decorator. Funkcionalitu jde skládat z jednoduchých částí. Výsledkem je, že aplikace nemusí počítat i s funkcemi, jež nepoužívá. Je také snadné definovat nové druhy Decoratorů nezávisle na třídách objektů, které rozšiřují, a to i pro nepředvídaná rozšíření. Rozšíření složité třídy má tendenci odhalovat detaily nesouvisející s povinnostmi, jež jsou přidávány.

Nevýhodou vzoru je, že Decorator a jeho komponenta nejsou totožné. Decorator funguje jako transparentní zaobalení objektu. Z hlediska identity objektu však není dekorovaná komponenta totožná se samotnou komponentou. Proto by se při používání dekorátorů nemělo spoléhat na identitu objektu. Zároveň návrh, jenž používá Decorator, často vede k systémům složených z mnoha malých objektů, které vypadají stejně. Objekty se liší pouze ve způsobu jejich propojení, nikoliv v jejich třídě nebo hodnotě proměnných. Ačkoli tyto systémy mohou snadno upravovat ti, kteří jim rozumí, může být obtížné se je naučit a odladit.[59]

## Facade

Je vhodné použít při poskytování jednotného rozhraní pro sadu rozhraní v subsystému. *Facade* definuje rozhraní vyšší úrovně, které usnadňuje používání subsystému.

Strukturování systému do subsystémů pomáhá snižovat složitost. Běžným cílem návrhu je minimalizovat komunikaci a závislosti mezi těmito subsystémy. Jedním ze způsobů, jak tohoto cíle dosáhnout, je zavést *facade* objekt, který poskytuje jediné zjednodušené rozhraní k obecnějším zařízením subsystému.

Vzor Facade je vhodné využít při tvorbě jednoduchého rozhraní pro složitý subsystém. Subsystémy se často postupně stávají složitějšími během jejich vývoje. Většina vzorů vede ke vzniku více menších tříd. Díky tomu je subsystém opakovaně použitelný a snadněji se přizpůsobuje, ale také se stává hůře použitelným pro klienty, jež jej nepotřebují přizpůsobovat. Facade může poskytnout jednoduchý výchozí pohled na subsystém, který je pro většinu klientů dostatečně dobrý. Pouze klienti, kteří potřebují větší možnosti přizpůsobení, musí využít jiné možnosti přístupu k subsystému.

Facade vzor je vhodný i v případech, kdy existuje mnoho závislostí mezi klienty a implementačními třídami abstrakce. Zavedení fasády oddělí subsystémy od klientů a jiných subsystémů, čímž se podpoří nezávislost a přenositelnost subsystému.

Tento vzor by měl být zároveň používán, pokud je součástí vývoje vrstvení podsystémů. Pomocí fasády se definuje vstupní bod do každé úrovně subsystému. Pokud jsou subsystémy na sobě závislé, lze závislosti mezi nimi zjednodušit tím, že spolu budou komunikovat výhradně prostřednictvím svých fasád.

Struktury, jež se jsou participanty na Facade vzoru:

- **Facade** ví, jaké třídy subsystému jsou zodpovědné za požadavek, a deleguje klientské požadavky na příslušné objekty subsystému

- **Třídy subsystému** implementují funkce subsystému, dále zpracovávají práci přidělenou objektem Facade a přitom na ni neuchovávají žádné odkazy

Benefitem Facade vzoru je odstínění klientů od součástí subsystému, čímž se snižuje počet objektů, s nimiž klienti pracují, a usnadňuje používání subsystému. Zároveň podporuje slabou vazbu mezi subsystémem a jeho klienty. Často jsou součástí subsystému silně provázané. Slabá vazba umožňuje měnit komponenty subsystému, aniž by to ovlivnilo jeho klienty. Facades pomáhají vrstvit systém a závislosti mezi objekty. Mohou eliminovat složité nebo kruhové závislosti, čehož je využíváno, pokud jsou klient a subsystém implementovány nezávisle. Facade také nebrání aplikacím v používání subsystémových tříd.[59]

## Proxy

*Proxy* je používána v případě, kdy je nutné poskytnout náhradní nebo zástupní objekt pro řízení přístupu k jinému objektu.

Jedním z důvodů, proč řídit přístup k objektu, je možnost odložení veškerých nákladů na jeho vytvoření a inicializaci až do doby, kdy jej bude skutečně potřeba použít. Uvažujme editor dokumentu, který může do dokumentu vkládat grafické objekty. Vytvoření některých grafických objektů, například velkých rastrových obrázků, může být nákladné. Otevření dokumentu by však mělo být rychlé, proto bychom se měli vyhnout vytvoření všech nákladných objektů najednou při otevření dokumentu. To stejně není nutné, protože ne všechny objekty budou v dokumentu viditelné najednou.

Tato omezení by naznačovala vytvoření každého výpočetně náročného objektu na vyžádání, k čemuž v tomto případě dojde, když se obrázek stane zobrazitelným. Co však vložíme do dokumentu místo obrázku? A jak můžeme skrýt skutečnost, že se obrázek vytváří na vyžádání, abychom nekomplikovali implementaci editoru? Zároveň by tato optimalizace neměla ovlivnit například rendering či formátovací kód.

Řešením je použití objektu zvaného *proxy*, jenž funguje jako náhrada skutečného obrázku. Proxy se chová stejně jako obrázek a stará se o jeho instanci.

Proxy vytvoří skutečný obrázek pouze tehdy, pokud jej editor dokumentu požádá o zobrazení vyvoláním operace Draw. Následující požadavky předává proxy přímo obrázku, a proto musí po jeho vytvoření zachovat odkaz na obrázek.

Editor dokumentu přistupuje k vloženým obrázkům prostřednictvím rozhraní definovaného abstraktní třídou Graphic. ImageProxy je třída pro obrázky, které jsou vytvářeny na vyžádání. ImageProxy udržuje název souboru jako odkaz na obrázek na disku. Název souboru se předává jako argument konstruktoru ImageProxy.

ImageProxy také uchovává ohraničující rámeček obrázku a odkaz na skutečnou instanci Image. Tento odkaz bude platný až po instanciaci skutečného obrázku pomocí proxy. Operace Draw se ujistí, že je obrázek instanciován, než mu předá požadavek. GetExtent předá požadavek na obrázek pouze v případě, že je instanciován, v opačném případě ImageProxy vrátí ukládaný obsah.

Proxy se používá vždy, když je potřeba univerzálnější nebo sofistikovanější odkaz na objekt než pouhý ukazatel.

Pokud je využíván vzor Proxy, jsou důležité následující objekty:

- **Proxy** udržuje odkaz, jenž umožňuje přístup ke skutečnému subjektu. Proxy může odkazovat na subjekt, pokud jsou rozhraní RealSubject a Subject stejná. Poskytuje rozhraní shodné s rozhraním Subject, takže skutečný subjekt jím lze nahradit. Kontroluje přístup ke skutečnému subjektu a může být zodpovědná za jeho vytváření a mazání
- **Subject** definuje společné rozhraní pro RealSubject a Proxy, takže Proxy lze použít všude, kde se očekává RealSubject

- **RealSubject** definuje skutečný objekt, který proxy zastupuje

Vzor Proxy zavádí při přístupu k objektu úroveň nepřímé vazby. Dodatečná nepřímá vazba má mnohostranné využití v závislosti na druhu proxy:

1. *Remote proxy* může skrýt skutečnost, že se objekt nachází v jiném adresním prostoru
2. *Virtual proxy* může provádět optimalizace, jako je vytvoření objektu na vyžádání
3. *Protection proxy* a inteligentní odkazy umožňují provádět další běžné úkony při přístupu k objektu

Vzor Proxy může před klientem skrýt ještě jednu optimalizaci. Nazývá se *copy-on-write* a souvisí s vytvářením objektu na vyžádání. Kopírování velkého a složitého objektu může být nákladnou operací. Pokud se kopie nikdy nemění, není třeba tyto náklady vynakládat. Použitím proxy k odložení procesu kopírování zajistíme, že cenu za kopírování objektu zaplatíme pouze v případě, že je objekt modifikován.[59]

### 5.3.4.4 Behaviorální objektové vzory

#### Chain of Responsibility

*Chain of responsibility* se použije v případech, kdy je žádané vyhnout se propojení odesílatele požadavku s jeho příjemcem. Toho je docíleno řetězovým posíláním zodpovědnosti na zpracování požadavku mezi objekty, dokud jej některý z nich nezpracuje.

Zvažme kontextovou nápovědu pro GUI. Uživatel může získat informace o nápovědě k libovolné části rozhraní pouhým kliknutím na ni. Poskytovaná nápověda závisí na vybrané části rozhraní a jejím kontextu, např. widget tlačítka v dialogovém okně může mít jiné informace nápovědy než podobné tlačítko v hlavním okně. Pokud pro danou část rozhraní neexistují žádné specifické informace nápovědy, měl by systém nápovědy zobrazit obecnější nápovědu týkající se bezprostředního kontextu, např. dialogového okna jako celku.

Je přirozené uspořádat pomocné informace podle jejich obecnosti – od nejkonkrétnějších po nejobecnější. Dále je zřejmé, že požadavek na nápovědu je zpracován jedním z několika objektů uživatelského rozhraní. Který z nich to bude, závisí na kontextu a na tom, jak konkrétní je dostupná nápověda.

Problém spočívá v tom, že objekt, jenž nakonec nápovědu poskytne, není explicitně znám objektu (např. tlačítku), který požadavek na nápovědu iniciuje. Potřebujeme způsob, jak oddělit tlačítko iniciující požadavek na nápovědu od objektů, jež mohou poskytnout informace o nápovědě. Vzor Chain of responsibility tento proces definuje.

Myšlenkou tohoto vzoru je oddělit odesílatele a příjemce darováním více možností objektům jak zpracovat požadavek. Požadavek se předává v řetězci objektů, dokud jej jeden z nich nevyřídí.

První objekt v řetězci přijme požadavek a buď jej vyřídí, nebo jej předá dalšímu kandidátovi v řetězci, který učiní totéž. Objekt, který požadavek zadal, nemá explicitní znalost o tom, kdo jej vyřídí – říkáme, že požadavek má implicitního příjemce.

Aby bylo možné předávat požadavky v řetězci a zajistit, že příjemci zůstanou implicitní, sdílí všechny objekty v řetězci společné rozhraní pro zpracování požadavků a pro přístup ke svému následníkovi v řetězci.

Vzor Chain of responsibility by měl být použit v případech, kdy požadavek může zpracovávat více než jeden objekt a zpracovatel není předem znám. Zpracovatel by měl být zjišťován automaticky. Dále pokud chceme zadat požadavek jednomu z několika objektů, aniž by byl výslovně uveden příjemce. Nebo v případech kdy by měla být množina objektů zpracovávající požadavek určena dynamicky.

Struktury podílející se na fungování tohoto vzoru jsou:

- **Handler** definuje rozhraní pro zpracování požadavků a případně implementuje odkaz na další objekt v řetězci
- **ConcreteHandler** vyřizuje žádosti, za které odpovídá. Může přistupovat ke svému následníkovi. Pokud ConcreteHandler může požadavek zpracovat, zpracuje jej. V opačném případě předá požadavek svému následníkovi
- **Client** iniciuje požadavek na objekt ConcreteHandler v řetězci

Tento vzor snižuje provázanost, jelikož daný objekt nemá znalost o tom, který jiný objekt požadavek vyřizuje. Objekt musí pouze vědět, že požadavek bude zpracován "vhodným způsobem". Příjemce i odesílatel o sobě nemají žádné explicitní znalosti a objekt v řetězci nemusí vědět o struktuře řetězce. V důsledku toho může Chain of responsibility zjednodušit vzájemné propojení objektů. Namísto toho, aby si objekty udržovaly odkazy na všechny kandidáty na příjemce, udržují jediný odkaz na svého následníka. Zároveň poskytuje flexibilitu při přiřazování odpovědností objektům. Chain of responsibility poskytuje větší flexibilitu při rozdělování odpovědností mezi objekty, neboť lze přidávat nebo měnit odpovědnosti za zpracování požadavku přidáním nebo jinou změnou řetězce za běhu programu. Tuto funkci lze kombinovat s podtřídami a zároveň staticky specializovat handlers.

Nevýhodou je, že zpracování požadavku není zaručeno. Protože požadavek nemá explicitního příjemce, není zaručeno, že bude zpracován. Požadavek může vypadnout z konce řetězce, aniž by byl zpracován. Požadavek také může zůstat nevyřízený, pokud není řetězec správně nakonfigurován.[59]

## Command

*Command* znám také jako *Action*, nebo *Transaction*, je vzor určený pro zapouzdření požadavku do objektu, což umožní parametrizovat klienty s různými požadavky, řadit požadavky do fronty nebo je zaznamenávat a podporovat operace, jež lze zrušit.

Někdy je nutné zadávat požadavky objektům, aniž bychom věděli cokoli o požadované operaci nebo o příjemci požadavku. Například sady nástrojů UI obsahují objekty jako jsou tlačítka a nabídky, které provedou požadavek v reakci na vstup uživatele. Sada nástrojů však nemůže požadavek explicitně implementovat do tlačítka nebo nabídky, protože pouze aplikace, jež sadu nástrojů používají, vědí, co se má na kterém objektu provést. Pokud se jedná o návrh sady nástrojů, neexistuje možnost znát příjemce požadavku ani operace, které jej provedou.

Vzor Command umožňuje objektům sady nástrojů zadávat požadavky na nespécifikované objekty aplikace tak, že samotný požadavek promění v objekt. Tento objekt lze ukládat a předávat jako jiné objekty. Hlavním prvkem tohoto vzoru je abstraktní třída Command, která deklaruje rozhraní pro provádění operací. V nejjednodušší podobě toto rozhraní obsahuje abstraktní operaci Execute. Konkrétní podtřídy Command specifikují dvojici receiver-action uložením receiveru jako proměnné instance a implementací Execute pro vyvolání požadavku. Příjemce má znalosti potřebné k provedení požadavku.

Vzor odděluje objekt, jenž operaci vyvolává od objektu, který má znalosti k jejímu provedení. To nabízí velkou flexibilitu při návrhu UI. Aplikace může poskytovat interface pro nabídku i tlačítko funkce jen tím, že nabídka a tlačítko budou sdílet instanci stejné konkrétní podtřídy Command. Příkazy lze dynamicky nahrazovat, což je užitečné pro implementaci kontextových nabídek. Vzor zároveň nabízí podporu skriptování příkazů skládáním příkazů do větších příkazů. To vše je možné, jelikož objekt, který zadává požadavek, musí pouze vědět, jak jej zadat. Nemusí vědět, jak bude daný požadavek proveden.

Použití vzoru Command je vhodné pokud požadujeme parametrizaci objektů pomocí akce, jež mají provést. Takovou parametrizaci lze v procedurálním jazyce vyjádřit pomocí funkce zpětného volání, tj. funkce, která je někde zaregistrována, aby byla později zavolána. Commands jsou objektově orientovanou náhradou zpětných volání. Dále je vhodné využít tohoto

vzoru, když program zadává, řadí do fronty a provádí požadavky v různých časech. Command objekt může mít životnost nezávislou na původním požadavku. Pokud lze příjemce požadavku reprezentovat způsobem nezávislým na adresním prostoru, tak je možné objekt daného požadavku přenést do jiného procesu a požadavek splnit tam. Vzoru lze využít i pokud je vznesen požadavek strukturalizaci systému kolem operací na vysoké úrovni postavených na primitivních operacích. Taková struktura je běžná v IS, jež podporují transakce. Transakce zapouzdřuje soubor změn dat. Vzor Command nabízí způsob, jak transakce modelovat. Příkazy mají společné rozhraní, které umožňuje vyvolat všechny transakce stejným způsobem. Vzor také usnadňuje rozšíření systému o nové transakce.

Sktruktury participující v tomto vzoru:

- **Command** deklaruje rozhraní pro provádění operace
- **ConcreteCommand** definuje vazbu mezi objektem Receiver a akcí a implementuje funkci Execute vyvoláním příslušné operace na Receiveru
- **Client** vytvoří objekt ConcreteCommand a nastaví jeho Receiver
- **Invoker** žádá o provedení požadavku příkazem
- **Receiver** umí provádět operace spojené s provedením požadavku. Jako Receiver může sloužit jakákoli třída

Command vzor odděluje objekt, který operaci vyvolává od objektu, který ví, jak ji provést. Příkazy jsou first-class objekty. Lze tedy s nimi manipulovat a rozšiřovat je jako jakékoliv jiné objekty. Díky použití tohoto vzoru je jednoduché přidávání nových příkazů, neboť není nutné měnit stávající třídy.[59]

## Iterator

*Iterator* poskytuje způsob, jak postupně přistupovat k prvkům agregovaného objektu, aniž by byla odhalena jeho základní reprezentace.

Agregovaný objekt, jako je seznam, by měl umožňovat přístup k jeho prvkům, aniž by byla odhalena jeho vnitřní struktura. Navíc může existovat požadavek na procházení seznamu různými způsoby v závislosti na tom, čeho chce uživatel dosáhnout. Pravděpodobně však není vhodné rozhraní seznamu zahrnovat operacemi pro různé způsoby procházení, i kdyby bylo možné předvídat ty, které bude třeba. Zároveň také může být potřeba, aby na stejném seznamu probíhalo více než jedno procházení. Právě v takovémto případě je vhodné využít tento vzor.

Iterator, jeho funkce a pomocné struktury jsou následující:

- **Iterator** definuje rozhraní pro přístup a procházení prvků
- **ConcreteIterator** implementuje rozhraní Iterator a uděluje informaci o aktuální pozici při procházení agregátu
- **Aggregate** definuje rozhraní pro vytvoření objektu Iterator
- **ConcreteAggregate** implementuje rozhraní pro vytváření třídy Iterator a vrací instanci správného ConcreteIteratoru

Iterator podporuje různé varianty procházení agregátu. Složité agregáty lze procházet mnoha způsoby. Například generování kódu a sémantická kontrola zahrnují procházení parsovacích stromů. Generování kódu může procházet parsovací strom inorder nebo preorder. Iterátory usnadňují změnu algoritmu procházení. Stačí nahradit instanci iterátoru jinou instancí. Lze také definovat podtřídy iterátorů, které podporují nové procházení. Iterátory také zjednodušují rozhraní Aggregate. Rozhraní procházení Iteratoru odstraňuje potřebu podobného rozhraní v Aggregate, čímž zjednodušuje dané rozhraní. Díky vzoru Iterator může jeden agregát být v daný čas vícenásobně procházen, jelikož každá instance Iterator udržuje svoji vlastní současnou pozici.[59]

## Mediator

Vzor použijeme, chceme-li definovat objekt, jenž zapouzdřuje způsob interakce sady objektů. *Mediator* podporuje slabou provázanost tím, že objekty na sebe explicitně neodkazují, a umožňují nezávisle měnit jejich interakci.

Objektově orientovaný návrh podporuje rozdělení chování mezi objekty. Takové rozdělení může vést k objektové struktuře s mnoha vazbami mezi objekty. Objektově orientovaný návrh podporuje rozdělení chování mezi objekty. Takové rozdělení může vést k objektové struktuře s mnoha vazbami mezi objekty; v nejhorším případě každý objekt nakonec ví o každém jiném.

Ačkoli rozdělení systému na mnoho objektů obecně zvyšuje znovupoužitelnost, množící se propojení mají tendenci ji opět snižovat. Množství vzájemných propojení snižuje pravděpodobnost, že objekt může fungovat bez podpory ostatních, tedy systém se chová, jako by byl monolitický. Navíc může být obtížné změnit chování systému nějakým významným způsobem, protože chování je rozděleno mezi mnoho objektů. V důsledku toho můžeme být nuceni definovat mnoho podtříd, abychom přizpůsobili chování systému.

Jako příklad lze uvést implementaci dialogových oken v GUI. Dialogové okno využívá okno k prezentaci souboru widgetů jako jsou tlačítka, nabídky a vstupní pole.

Mezi widgety v takovém dialogovém okně často existují závislosti. Například tlačítko se zakáže, když je určité vstupní pole prázdné. Výběr položky v seznamu možností zvaném list box může změnit obsah vstupního pole. Naopak zadání textu do vstupního pole může automaticky vybrat jednu nebo více odpovídajících položek v list boxu. Jakmile se v zadávacím poli objeví text, mohou se aktivovat další tlačítka, která uživateli umožní s textem něco provést, například změnit nebo odstranit věc, na kterou se text dokazuje.

Různá dialogová okna budou mít různé závislosti mezi widgety. Protože tedy dialogová okna zobrazují stejné typy widgetů, nemohou jednoduše znovu použít standardní třídy widgetů, a musí tedy být přizpůsobeny tak, aby odrážely závislosti specifické pro dialogová okna. Jejich individuální přizpůsobení pomocí podtříd je zdlouhavé, neboť se jedná o mnoho tříd.

Těmto problémům se lze vyhnout zapouzdřením kolektivního chování do samostatného objektu, jež používá vzor Mediator, který se nazývá zprostředkovatel. Zprostředkovatel je zodpovědný za řízení a koordinaci interakcí skupiny objektů. Mediator slouží jako prostředník, který zabraňuje tomu, aby se objekty ve skupině na sebe navzájem explicitně odkazovaly. Objekty znají pouze prostředníka, čímž se snižuje počet vzájemných vazeb.

Mediator by měl být použit vždy, sada objektů komunikuje přesně definovanými, ale složitými způsoby. Výsledné vzájemné závislosti jsou nestrukturované a obtížně srozumitelné. Dále by měl být aplikován pokud opakované použití objektu je obtížné, protože odkazuje na mnoho jiných objektů a komunikuje s nimi. A také v případech, kdy chování, jež je distribuováno mezi několika třídami, by mělo být přizpůsobitelné bez velkého množství podtříd.

Struktury, jež vznikají při použití tohoto vzoru jsou:

- **Mediator** definuje rozhraní pro komunikaci s *Colleague* třídami
- **ConcreteMediator** implementuje kooperativní chování pomocí koordinace tříd *Colleague*. Zároveň zná a udržuje tyto *Colleague* třídy
- **Colleague třídy** každá třída *Colleague* zná svůj *Mediator* objekt. Každá třída *Colleague* komunikuje s ní přiřazenou třídou *Mediator* a nikoliv s ostatními třídami typu *Colleague*

Použití vzoru Mediator omezuje subclassing. Mediator združuje chování, jež by jinak bylo rozděleno mezi několika objekty. Změna tohoto chování vyžaduje pouze podtřídu Mediator. Třídy *Colleague* lze používat opakovaně tak, jak jsou. Dále Mediator podporuje slabou provázanost mezi *Colleague* třídami. Třídy *Colleague* a *Mediator* lze měnit a znovu používat nezávisle na sobě. Vytvoření vazby mediation, jakožto samostatného konceptu a jeho zapouzdření do

objektu umožňuje soustředit se na to, jak spolu objekty interagují, a ne na jejich individuální chování. To může pomoci upřesnit, jak objekty v systému spolupracují.

Nevýhodou je, že vzor Mediator vyměňuje složitost interakce za složitost Mediator třídy. Protože Mediator zapouzdřuje protokoly, může být složitější než kterákoliv jednotlivá Colleague třída. To může ze samotné Mediator třídy udělat monolit, který se obtížně udržuje.[59]

### Token

Chceme-li bez porušení zapouzdření zachytit a zvenčí uložit vnitřní stav objektu tak, aby bylo možné tento stav později obnovit, je vhodné použít právě vzor *Token*.

Token je objekt, který uchovává snímek vnitřního stavu jiného objektu – původce tokenu. Mechanismus zrušení si vyžádá token od původce, když potřebuje zkontrolovat stav původce. Původce inicializuje token informacemi, jež charakterizují jeho aktuální stav. Pouze původce může ukládat a získávat informace z tokenu. Informace uvnitř tokenu jsou pro ostatní objekty skryty.

Tento vzor je vhodné použít v případech, kdy musí být uložen snapshot stavu objektu či jeho části, aby bylo možné tento stav později obnovit. Avšak při použití přímo rozhraní objektu pro získání stavu by došlo k odhalení implementačních detailů a porušilo by se zapouzdření objektu.

Struktury účastníků, při použití Token vzoru jsou následující:

- **Token** ukládá vnitřní stav objektu Originator. Token může ukládat tolik stavů, kolik je třeba podle uvážení objektu Originator. Zároveň chrání před přístupem jiných objektů než Originator. Token má v podstatě dvě rozhraní. Caretaker vidí pouze základní rozhraní k tokenu. Může token předat pouze jiným objektům. Originator naproti tomu vidí rozšířené rozhraní, jež mu umožňuje přístup ke všem datům potřebným k obnovení předchozího stavu. V ideálním případě by přístup k vnitřnímu stavu tokenu měl mít pouze Originator, který token vytvořil.
- **Originator** vytvoří token obsahující snímek jeho aktuálního vnitřního stavu. Daný token používá k obnovení svého vnitřního stavu.
- **Caretaker** zodpovídá za úschovu tokenu. Nikdy nepracuje s tokenem ani nezkoumá jeho obsah.

Výhodou Tokenu je, že se vyhýbá vystavování informací, jež by měl spravovat pouze Originator, ale které přesto musí být uloženy mimo objekt Originator. Vzor chrání ostatní objekty před potenciálně složitými vnitřními prvky struktury Originator, čímž zachovává hranice zapouzdření. Dále zjednodušuje objekt Originator. V jiných provedeních zachovávajících zapouzdření Originator uchovává různé verze vnitřního stavu. To přenáší veškerou zátěž správy úložiště na strukturu Originator. To, že klienti spravují stav, který si vyžádali, zjednodušuje Originator a zabraňuje klientům, aby museli informovat Originator, když skončí.

Nevýhodou tohoto vzoru je výpočetní nákladnost Tokenů. Tokeny mohou být spojeny se značnou režii, pokud musí Originator kopírovat velké množství informací pro uložení do tokenu nebo pokud klienti vytvářejí a vracejí tokeny zpět objektu Originator dostatečně často. Zároveň kvůli existenci dvou typů rozhraní, může být v některých jazycích obtížné tuto funkčnost naprogramovat.[59]

### State

*State* umožňuje objektu změnit své chování, když se změní jeho vnitřní stav. Objekt bude vypadat, že změnil svou třídu.

Uvažujme třídu *TCPConnection*, jež představuje síťové připojení. Objekt *TCPConnection* může být v jednom z několika různých stavů: *Established*, *Listening*, *Closed*. Když objekt *TCPConnection* přijímá požadavky od jiných objektů, reaguje různě v závislosti na svém

aktuálním stavu. Například účinek požadavku Open závisí na tom, zda je spojení ve stavu Closed, nebo Listening. Vzor State popisuje, jak se může TCPConnection v jednotlivých stavech chovat různě.

Klíčovou myšlenkou tohoto vzoru je zavedení abstraktní třídy TCPState, která reprezentuje stavy síťového připojení. Třída TCPState deklaruje rozhraní společné pro všechny třídy, jež reprezentují různé provozní stavy. Podtřídy třídy TCPState implementují chování specifické pro daný stav. Například třídy TCPEstablished a TCPClosed zavádí konkrétní chování pro stav TCPConnection Established a Closed.

Připojení deleguje všechny požadavky specifické pro daný stav na tento stavový objekt. TCPConnection používá svou instanci podtřídy TCPState k provádění operací specifických pro stav spojení. Kdykoli se změní stav připojení, objekt TCPConnection změní stavový objekt, který používá. Když například spojení přejde ze stavu Established na stav Closed, TCPConnection nahradí svou instanci TCPEstablished instancí TCPClosed.

Vzor State je vhodné použít v jednom z následujících případů. Pokud chování objektu závisí na jeho stavu a objekt musí za běhu změnit své chování v závislosti na tomto stavu. Nebo pokud operace mají rozsáhlé, vícedílné podmíněné příkazy, jež závisí na stavu objektu. Tento stav je obvykle reprezentován jednou nebo více konstantami. Často bude několik operací obsahovat stejnou podmíněnou strukturu. Vzor State umísťuje každou větev podmíněného příkazu do samostatné třídy. To umožňuje zacházet se stavem objektu jako se samostatným objektem, který se může měnit nezávisle na ostatních objektech.

Při použití tohoto vzoru vznikají následující struktury:

- **Context** definuje klientské rozhraní a udržuje instanci podtřídy ConcreteState, jež definuje aktuální stav
- **State** definuje rozhraní pro zapouzdření chování spojeného s určitým stavem objektu Context
- **ConcreteState podtřídy** každá podtřída implementuje chování spojené se stavem Context objektu

Vzor State lokalizuje chování specifické pro daný stav a rozděljuje chování pro různé stavy. Tento vzor umísťuje veškeré chování spojené s určitým stavem do jednoho objektu. Vzhledem k tomu, že veškerý kód specifický pro jednotlivé stavy se nachází v podtřídě State, lze snadno přidávat nové stavy a přechody definováním nových podtříd. Dále vzor umožňuje explicitně vyjádřit přechody mezi stavy. Pokud objekt definuje svůj aktuální stav pouze pomocí hodnot vnitřních dat, nemají jeho stavové přechody žádnou explicitní reprezentaci, tedy projevují se pouze jako přiřazení k některým proměnným. Zavedením samostatných objektů pro různé stavy jsou přechody explicitnější. State objekty také mohou chránit Context objekty před nekonzistentními vnitřními stavy, protože stavové přechody jsou z pohledu Context objektu atomické, tj. probíhají reinstanciací jedné proměnné, nikoli několika.[59]

## Strategy

Vzor *Strategy* použijeme chceme-li definovat rodinu algoritmů, zapouzdřit každý z nich a zajistit jejich zaměnitelnost. Strategy umožňuje, aby se algoritmus lišil nezávisle na klientech, kteří ho používají. Vzor Strategy je znám i pod označením *Policy*.

Případem, kdy použít tento vzor, může být například situace, kdy potřeba použít různé varianty algoritmu. Lze například definovat algoritmy, které odrážejí různé časoprostorové kompromisy. Strategy lze aplikovat, pokud jsou tyto varianty implementovány jako hierarchie tříd algoritmů. Dalším příkladem, kdy tento vzor použít, je používá-li algoritmus data, o kterých by klient neměl vědět. Nebo pokud třída definuje mnoho chování, jež se v jejích operacích objevuje jako několik podmíněných příkazů. Místo mnoha podmíněných příkazů lze přesunout související podmíněné větve do vlastní třídy Strategy.

Mezi struktury, jež vznikají při použití tohoto vzoru patří:



- **Strategy** deklaruje rozhraní společné pro všechny podporované algoritmy. Context používá toto rozhraní k volání algoritmu definovaného objektem ConcreteStrategy.
- **ConcreteStrategy** implementuje algoritmus pomocí rozhraní Strategy
- **Context** je nakonfigurován pomocí objektu ConcreteStrategy. Udržuje odkaz na objekt Strategy a může definovat rozhraní, jež umožňuje objektu Strategy přistupovat k jejím datům

Strategy může být alternativou k podtřídě. Dědičnost nabízí další způsob podpory různých algoritmů nebo chování. Lze přímo vytvořit podtřídu třídy Context a dát jí různá chování. Tím se však chování natvrdo zafixuje do třídy Context. Mísí se tak implementace algoritmu s implementací třídy Context, což znesnadňuje pochopení, údržbu a rozšiřování Context třídy. A zároveň nelze algoritmus dynamicky měnit. Tímto se vytvoří mnoho příbuzných tříd, jejichž jediným rozdílem je algoritmus nebo chování, jež používají. Zapouzdření algoritmu v samostatných třídách Strategy umožňuje měnit algoritmus nezávisle na jeho kontextu, což usnadňuje přepínání, pochopení a rozšiřování. Dále Strategy může poskytovat různé implementace stejného chování. Klient si tedy může vybrat mezi strategiemi s různými časovými a prostorovými nároky.

Tento vzor má potenciální nevýhodu v tom, že klient musí pochopit, jak se strategie liší, než si může vybrat, jakou chce použít. Klienti mohou být vystaveni problémům s implementací. Proto by se měl vzor Strategy používat pouze tehdy, když je rozdílnost chování pro klienty důležitá. Zároveň tento vzor zvyšuje počet objektů. Někdy lze tento nedostatek řešit implementací Strategy tříd jakožto bezstavové objekty, jež mohou kontexty sdílet. Případně zbytkový stav udržuje Context, který jej v každém požadavku předává objektu Strategy.[59]

## 5.4 Implementace aplikačního softwaru

Implementací můžeme nazývat proces, jenž převádí návrh na hotový software (program). Tento proces se skládá ze souboru malých rozhodnutí, která jsou řízena v některých případech vzory, v jiných případech samotnou intuicí programátora. Tyto rozhodnutí by měla být vždy v souladu s daným návrhem. Výsledkem tohoto procesu by měl být kvalitní kód splňující dané požadavky. Co jen kvalitní kód, je popsáno v následující podsekcí. Popis implementačních vzorů jednotlivých programovacích struktur je v této práci záměrně vynechán, ale je možné najít jejich popis např. v knize *Kenta Becka: Implementation Patterns*[62].

### 5.4.1 Kvalitní kód

Kvalitní kód je kód psán s přihlédnutím k určitým hodnotám a principům, jež v případě dodržení činí kód lépe čitelným, rozšiřitelným a znovupoužitelným. Kvalita kódu je důležitým faktorem, neboť následná práce s nekvalitním kódem je velmi náročná a pokud se dostane do rukou někoho jiného, než programátora, jenž ho vytvořil, budou náklady na jeho úpravu mnohem vyšší, než u kvalitně napsaného programu.

#### Hodnoty

Tři hodnoty, které jsou v souladu s kvalitním kódem, jsou *komunikace*, *jednoduchost* a *flextibilita*. Tyto tři hodnoty jsou někdy v rozporu, častěji se však vzájemně doplňují. Nejlepší programy nabízejí mnoho možností budoucího rozšíření, neobsahují žádné cizí prvky a jsou snadno čitelné a srozumitelné.

#### 1. Komunikace

Kód dobře komunikuje, pokud mu čtenář rozumí a může ho bezproblémově upravovat nebo používat.[62] Komunikace je důležitým faktorem, neboť programátor stráví více času

čtením daného kódu, než jeho samotným psaním. Snaha psát čitelný a srozumitelný kód pomáhá i samotným řešením problému. Jelikož píšeme-li kód pomocí srozumitelných struktur, dekomponujeme i samotný problém na menší části a jsme schopni ho řešit mnohem efektivněji.

## 2. Jednoduchost

Eliminace nadměrné složitosti umožňuje těm, kteří chtějí daný program číst, užívat či upravovat jej rychleji pochopit. Částečná složitost programu je nezbytná a přímo odráží složitost řešeného problému. Ale ve většině případů je složitost vložena programátorem při snaze daný program udělat spustitelným. Právě tato nadměrná složitost ubírá softwaru na hodnotě tím, že snižuje pravděpodobnost správného běhu programu a zároveň tím, že je obtížnější program v budoucnu úspěšně změnit. Jednoduchosti by mělo být docíleno na všech úrovních a každá část kódu by neměla být smazatelná bez nějakých důsledků na samotnou funkčnost softwaru.[62]

## 3. Flexibilita

Flexibilita obvykle odkazuje na schopnost programu přizpůsobit se možným budoucím změnám v jeho požadavcích.[63] Bohužel ne vždy je flexibilita ideální faktorem, na který se při vývoji zaměřit, neboť ve většině případů, čím flexibilnějším se program stává, tím se stává i komplexnějším a složitějším na pochopení budoucími uživateli.

## Principy

Principy jsou další úrovní obecných myšlenek, jež jsou pro programování specifitější než hodnoty a které rovněž tvoří základ vzorů. Zkoumání principů je cenné z několika důvodů. Jasně principy mohou vést k novým vzorům, zároveň mohou poskytnout vysvětlení motivace, která za určitým vzorem stojí a která je spojena spíše s obecnými než s konkrétními myšlenkami. Tedy rozhodnutí o zvolení konkrétního vzoru je lepší diskutovat spíše z hlediska principů než specifik daného vzoru. Následující uvedené principy stojí za implementacemi programovacích vzorů.

### Lokální důsledky

Kód by měl být strukturován tak, aby změny v jedné třídě nemohly způsobit problém v jiné. Kód, ve kterém mají změny převážně pouze lokální důsledky, komunikuje efektivněji. Hlavním důvodem pro dodržení tohoto principu je, že při jeho dodržení jsou ceny za změny minimální. Princip lokálních důsledků by se dal nazývat klíčovým, neboť se od něj odvíjejí další programovací principy.[62]

### Minimalizace opakování

Přispívá k zachování principu lokálních důsledků, neboť pokud dojde k opakování kódu je nutné měnit jej na více místech a provedené změny již nejsou lokální. Čím více je kód opakován, tím vyšší jsou náklady na jeho úpravy.

Opakování kódu je jednou z forem repetice. Paralelní hierarchie tříd jsou taktéž považovány za repetici a porušují princip lokálních důsledků. V takovémto případě je nutná restrukturalizace kódu.

Problém duplicity se projevuje vždy až po jejím vytvoření a někdy dokonce až po nějaké době. I přestože duplikace kódu není doporučována, není nutné se jí vyhýbat ve všech případech, ale je nutné mýti na paměti, že zvyšuje náklady na provádění změn.

Jedním ze způsobů, jak odstranit duplicity, je rozdělit programy na mnoho malých částí. Velké kusy logiky mají tendenci duplikovat části jiných velkých logických struktur.[62]

### Logika a data dohromady

Dalším principem, jenž je odvozen z prvního zmíněného principu je princip uchovávání logiky a dat dohromady. To znamená uchovávat dat nejlépe v těle samotné metody, následně objektu a pokud je to opravdu nutné, balíčku. Důvodem k vzniku tohoto principu je velká pravděpodobnost, že při změně logiky dojde i ke změně dat, jež s ní souvisejí.[62]

**Symetrie**

Je principem týkající se přemýšlení nad pojmenováním a vytvářením metod. Pokud vytváříme určité metody měli bychom přitom dodržovat symetrii, tj. máme-li metodu `incrementCount()`, měla by v taném objektu existovat třída `decrementCount()`. Symetrie často souvisí s bilaterálními, rotačními a jinými relacemi, kde se nacházejí prvky označující opak či "stupně"stavu.[62]

**Deklarativní kód**

Deklarativní kód, nebo také často "samodokumentující"či "samodokumentační"kód. Je principem, jež se zaměřuje na pojmenovávání entit kódu. [62] Podle tohoto principu by měl název entit vypovídat o jejich povaze a zachytit tuto povahu co nejefektivněji. Zároveň pokud je to možné, měl by být celý kód pojmenován stejným stylem, což zajistí větší přehlednost kódu samotného.

**Úroveň změny**

Posledním principem je návod jak určit místo rozdělí logiky od dat. Data by měla souviset s logikou právě tehdy, když logika mění či čte data pouze na dané úrovni. Tj. data jsou využita v oblasti kódu daného bloku, metody, funkce, třídy, ...a v žádných jiných případech.[62]

# Výsledný aplikační software

## 6.1 Aplikační požadavky

Získávání požadavků bylo prováděno formou konzultací s vedoucím bakalářské práce. Výsledkem těchto konzultací je následující seznam požadavků.

### 6.1.1 Funkční požadavky

#### Vytvoření a zobrazení trasy

Aplikace by měla podporovat tvorbu vlastní trasy výběrem bodů na zobrazované mapě či ručním zadáním souřadnic a průjezdných bodů.

#### Načtení trasy z JSON souboru

Trasu bude možné vytvořit i pomocí načtení jejích bodů ze souboru napsaného ve formátu JSON.

#### Přidání nových bodů trasy

Aplikace umožní přidání nových bodů k již existující trase.

#### Úprava vytvořených úseků

Každý z úseků trasy půjde upravit, tj. upravit pozici jakéhokoliv bodu trasy.

#### Přidání vlastních proměnných pro úsek trasy

Pro všechny vytvořené úseky bude aplikace umožňovat přidání vlastních proměnných.

#### Načtení dat jízdního profilu

Aplikace bude umožňovat načtení dat jízdního profilu pro jednotlivé body trasy.

#### Načtení historických dat o provozu pro daný časový úsek

Pro vytvořenou trasu bude možné definovat časový úsek získávaných dat o provozu. Bude například možné získat informace pouze pro zadané dny týdne. V důsledku změny licenčních práv k jedné ze zvažovaných APIs došlo ke změně požadavku. Nové znění požadavku je načtení předpovědi dat o provozu pro daný časový úsek, daná předpověď by měla z historických dat vycházet.

#### Uložení trasy ve formátu GPX, nebo JSON

Bude podporováno uložení trasy ve formátu GPX, či JSON.

## 6.1.2 Nefunkční požadavky

### Přenositelnost mezi různými operačními systémy

Podmínkou aplikace je její fungování napříč různými operačními systémy.

### Grafické uživatelské rozhraní

Všechny funkčnosti budou podporovány ve formě GUI.

### Možnost využití aplikace z příkazového řádku

Aplikace bude spustitelná z příkazového řádku se zadanými parametry.

### Přehledné rozhraní

Obě uživatelem použitelná rozhraní budou přehledná a pro uživatele snadno pochopitelná.

### Využití OOP stylu

Aplikace bude naprogramována s využitím technik souvisejících s OOP.

## 6.2 Návrh výsledné aplikace

Návrh aplikace vychází ze sesbíraných aplikačních požadavků a snaží se zaručit jejich splnění s co nejmenšími náklady na výslednou implementaci. Samotný návrh softwaru vznikal postupně společně s implementací.

### 6.2.1 Typ aplikace a použitý programovací jazyk

Prvním krokem návrhu byl výběr typu aplikace a vhodného programovacího jazyka pro její implementaci. Rozhodnutí bylo učiněno tak, aby všechny požadavky mohly být co nejjednodušeji splněny s přihlédnutím k mým schopnostem jakožto programátora.

Výsledná aplikace je hybridního typu, neboť pro zajištění vykreslování a úpravy tras využívá web view komponenty a část aplikační logiky je uložena na běhové platformě. Zároveň tento přístup společně s vhodným výběrem programovacího jazyka zajišťuje bezproblémovou přenositelnost mezi různými operačními systémy. Jelikož většina výpočetně náročných operací neprobíhá na straně klienta, ale jsou provedeny pomocí webových REST APIs, není nutné brát v potaz nevyužití plného výkonnostního potenciálu platformy, na které je aplikace spuštěna.

Pro vývoj aplikace byl vybrán jako hlavní programovací jazyk Java, konkrétně verze Java 11. Tato volba umožňuje objektově orientovaný návrh aplikace a zajišťuje běh nezávislý na platformě, neboť aplikace pro její běh využívá *Java Virtual Machine* (JVM). Pro vytvoření grafického rozhraní je využíváno knihovny JavaFX, která podporuje i práci s web view komponentou.

### 6.2.2 Výběr API

Pro splnění aplikačních požadavků bylo nutné využít webových APIs. V následujících podkapitolách jsou popsány všechny použité a zamýšlené APIs společně s kritérii pro jejich výběr.

#### 6.2.2.1 Kritéria pro výběr

APIs, které lze pro splnění aplikačních požadavků použít, je nepřeborné množství. Z tohoto důvodu byly stanoveny následující čtyři kritéria pro jejich výběr:

1. **Cena** Cena za použití API byla stanovena hlavním kritériem výběru. Jelikož se jedná o akademický projekt, služby, které nenabízejí verzi zdarma, byly z výběru vyřazeny.

- 2. Kvalita nabízených služeb** Dalším faktorem, jenž byl při výběru zvažován, je kvalita a objem nabízených služeb, tj. pokud API jednoho poskytovatele nabízela více služeb, nebo data poskytované služby byla s největší pravděpodobností kvalitnější, neboť byla (pravděpodobně) získána od více uživatelů, než API druhého poskytovatele, byla preferována.
- 3. Kompatibilita s ostatními použitými API** Jedním z kritérií při výběru byla také kompatibilita s ostatními API. Neboť kombinace různých API od stejného poskytovatele může nabídnout funkcionality, jež by při kombinaci API od různých poskytovatelů nebyly dostupné, nebo by je bylo velice složité naprogramovat.
- 4. Náročnost na implementaci** Posledním z faktorů výběru byla náročnost na implementaci API. Pokud rozhraní jedné z API bylo jednodušší a lépe pochopitelné, než ostatních, byla následně tato API preferována, i přestože nabízela méně kvalitní služby, pokud rozdíl v kvalitě nebyl příliš markantní.

### 6.2.2.2 APIs pro zobrazení mapy a jejích prvků

#### Mapbox GL JS

Mnou dlouho zvažovaná API. Jedná se o API s jedním z nejvýhodnějších platebních modelů. Vytvoření dynamické mapy a následná manipulace s mapou je velice jednoduchá. Důvodem, proč tato API nebyla použita ve výsledné aplikaci, je nekompatibilita s ostatními klíčovými API, jichž aplikace využívá a nižší pokrytí.

#### OpenStreetMap API

Cenově nejvýhodnější ze zmíněných API, neboť se jedná o opensource projekt. Bohužel její dokumentace je velice zmatečná a nepřehledná. Zároveň je tato API velice náročná na samotnou implementaci a neexistuje žádná reálná podpora pro případné problémy. Jedná o nejhorší ze zvažovaných API.

#### Google Maps JS API

API použita ve výsledné aplikaci. V porovnání s Mapbox GL JS vychází cenově hůře, neboť je jednou z nejdražších API pro zobrazování map na trhu. I přesto je tou nejpoužívanější, jelikož poskytuje velmi dobré pokrytí světových cest, velice kvalitní podporu a jednoduché použití. O jejím použití v aplikaci ale v závěru rozhodla především kompatibilita s ostatními použitými API. Jelikož pro hledání trasy a informací o úsecích je využívána API, kterou poskytuje Google, je nutné pro jejich zobrazení použít tuto API, neboť by jinak mohlo docházet k nesrovnalostem mezi nalezenou a zobrazovanou trasou. Kdyby byla použita Mapbox GL JS API, musela by aplikace zajistit kompatibilitu mezi nalezenými body tras, což vede k velice neefektivnímu porovnávání všech průjezdových souřadnic, kterých můžou dané API generovat až tisíce v jednom volání.

### 6.2.2.3 APIs pro hledání informací o trase

Při rozhodování mezi API, jež použít pro hledání informací o dané trase jsem se rozhodoval mezi API poskytovanými společností Google a společností MapBox. Obě společnosti nabízejí API ve stejném rozsahu. Cenově vychází lépe MapBox API, ale v této aplikaci jsem se rozhodl použít API od Googlu, neboť jím poskytovaná data jsou získána od mnohem více uživatelů, tudíž je lze považovat za mnohem přesnější odraz reality, zvláště při získávání informací o provozu na dané trase v daný časový interval.

**Google Directions API** Ve výsledné aplikaci použita pro získání dat jízdního profilu, optimalizaci průjezdových bodů a k přichycení zadaných souřadnic k existujícím silnicím na zadané trase.

**Google Elevation API** Výslednou aplikací využívána pro získání informací o převýšení zadaných souřadnic.

**TomTom Traffic Stats APIs** Tento soubor APIs měl být použit k získávání informací o historických datech o provozu na dané trase. Bohužel v průběhu tvorby aplikace došlo ke změně licence této API a ta již není volně dostupná. Kvůli tomu došlo ke změně tohoto funkčního požadavku.

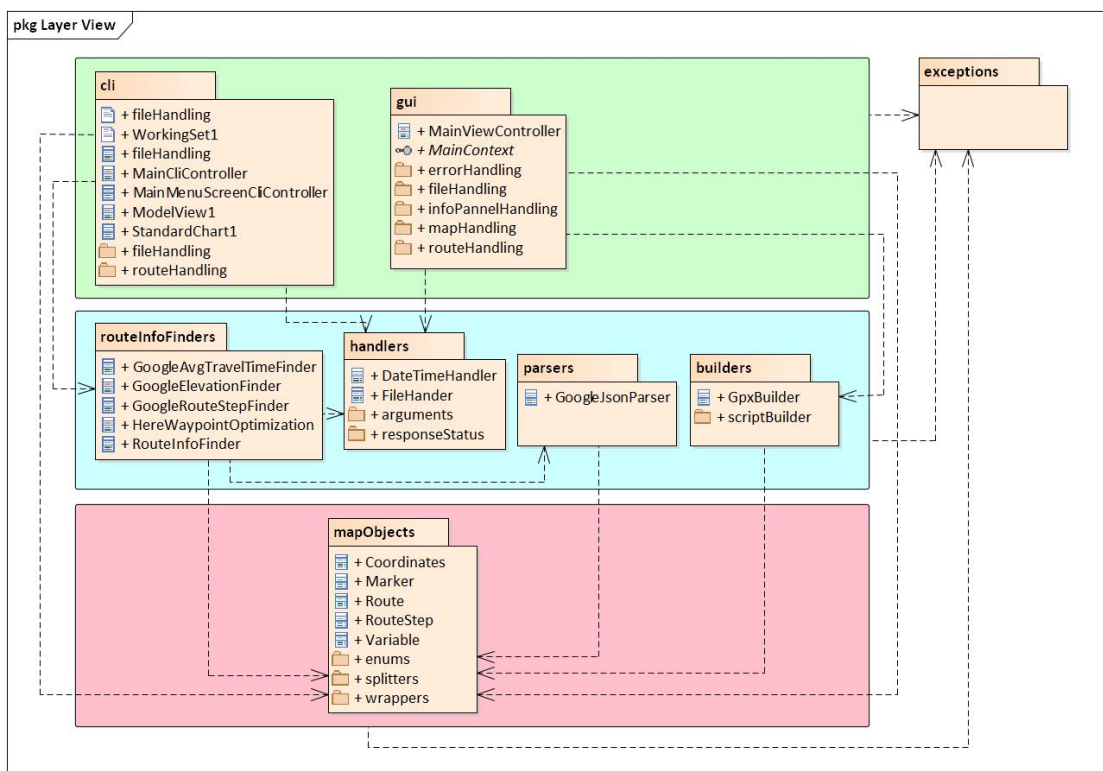
**Google Distance Matrix API** Je využita ke splnění funkčního požadavku o získání predikce dat o provozu v daný časový úsek, jež vychází z historických dat. Výsledkem je matice mezi jednotlivými body trasy, obsahující informace o cestovním čase s přihlédnutím k provozu v zadaný čas jízdy.

**HERE Waypoints Sequence API** Aplikací využívána k předpracování bodů trasy před jejich optimalizací pomocí Google Directions API v případě, kdy hrozí překročení limitu zmíněné API. Dané body uspořádá do nejefektivnějšího pořadí, tedy řeší tzv. problém obchodního cestujícího.

### 6.2.3 Použité architektonické vzory

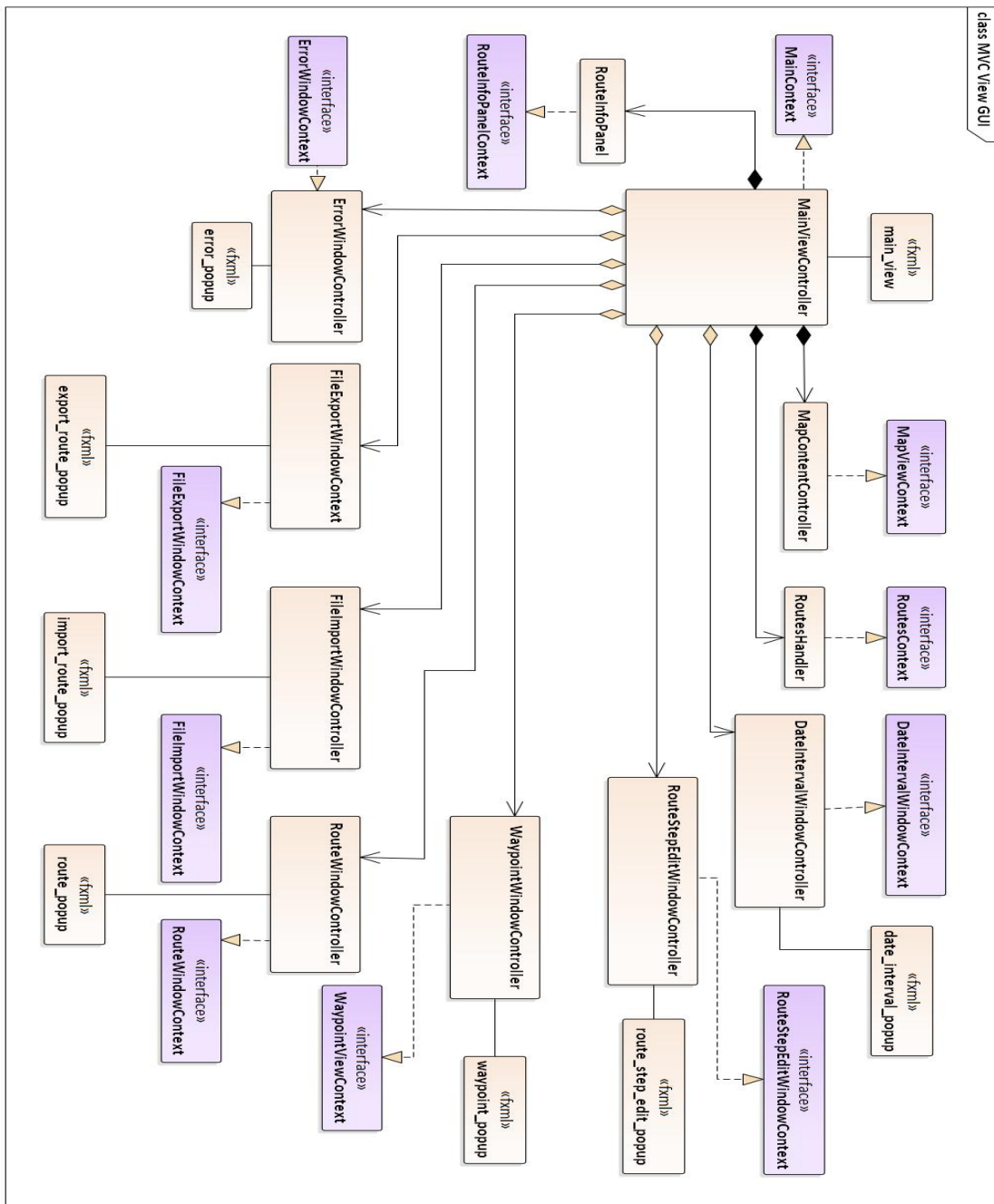
Aplikace využívá vícevrstvé architektury, tedy při návrhu aplikace proběhlo její rozdělení do jednotlivých částí, neboli vrstev. Výsledný aplikační software využívá klasické třívrstvé rozdělení, tj. je rozdělen na prezentační, logickou a datovou vrstvu. Rozdělení aplikace je znázorněno na následujícím obrázku 6.1.

■ **Obrázek 6.1** Vrstvy aplikace



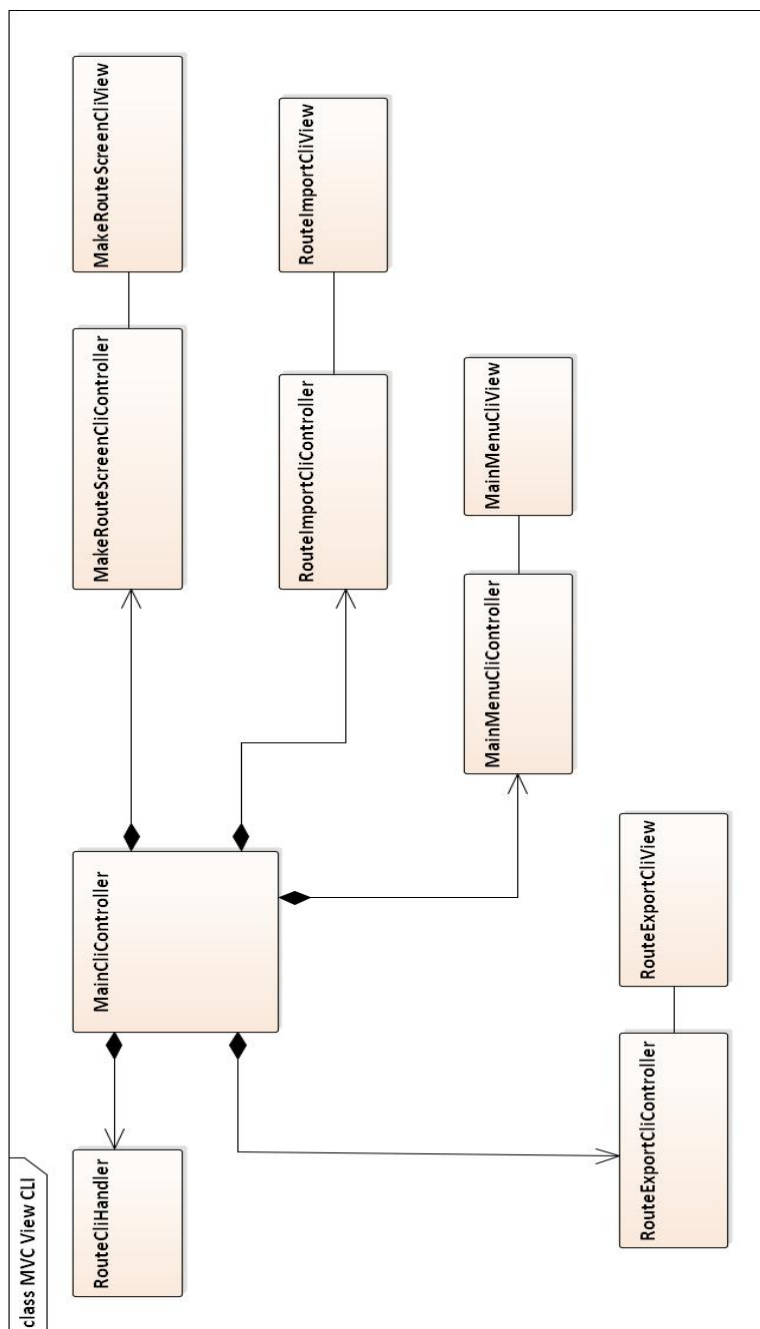
Při návrhu struktury GUI a CLI výsledné aplikace byl použit vzor MVC. V případě GUI zastává View roli fxml soubor, Controller role jsou rozděleny vždy dle kontextu a zastávají je třídy s postfixem Controller a Model roli zastávají třídy využívající rozhraní RouteContext a RouteInfoPanelContext. Rozložení tříd pro GUI je zobrazeno na obrázku 6.2 a rozložení pro CLI na obrázku 6.3.

■ Obrázek 6.2 Struktura GUI





■ Obrázek 6.3 Struktura CLI



### 6.2.4 Způsoby importu a exportu trasy

Pro splnění požadavků o importu a export tras bylo nutné navrhnout, jak budou vstupní a výstupní trasy zapsány do souboru daných formátu, aby je bylo možné pokládat za validní.

### 6.2.4.1 JSON import

JSON soubor, jenž zajistí úspěšné načtení trasy musí mít strukturu popsanou v této podsekcí.

Pole	Vyžadováno	Typ	Popis
origin	Ano	Number	Počátek trasy
destination	Ano	Number	Cíl trasy
waypoints	Ano	Array<Coordinates>	Pole bodů trasy

■ **Tabulka 6.1** Prvky JSON import souřadnic

```
{
  "origin": Coordinates,
  "destination": Coordinates,
  "waypoints": Array<Coordinates>
}
```

■ **Výpis kódu 6.1** Struktura JSON import souboru

#### Coordinates

Pole	Vyžadováno	Typ	Popis
lat	Ano	Number	Informace o zeměpisné šířce
lng	Ano	Number	Informace o zeměpisné délce

■ **Tabulka 6.2** Prvky JSON import souřadnic

```
{
  "lng": Number,
  "lat": Number
}
```

■ **Výpis kódu 6.2** Struktura JSON import souřadnic

```
{
  "origin": {
    "lat": 51.970390,
    "lng": 0.979328
  },
  "destination": {
    "lat": 52.094427,
    "lng": 0.668408
  },
  "waypoints": [{
    "lat": 52.038683,
    "lng": 0.730226
  }, {
    "lat": 51.976413,
    "lng": 1.053112
  }
]
}
```

■ **Výpis kódu 6.3** Příklad JSON import souboru

#### 6.2.4.2 JSON export

Exportovaný soubor ve formátu JSON bude mít následující strukturu.

Pole	Typ	Popis
duration	Number	Doba trvání trasy v sekundách
origin	Number	Počátek trasy
destination	Number	Cíl trasy
waypoints	Number	Seznam jednotlivých bodů trasy
steps	Number	Seznam jednotlivých kroků trasy

■ **Tabulka 6.3** Prvky JSON export souboru

```
{
  "duration": Double
  "origin": Coordinates,
  "destination": Coordinates,
  "waypoints": Array<Coordinates>
  "steps": Array<RouteStep>
}
```

■ **Výpis kódu 6.4** Struktura JSON export souboru

## RouteStep

Pole	Typ	Popis
duration	Int	Doba trvání kroku v sekundách
variables	Double	Pole uživatelem definovaných proměnných
distance	Int	Informace o délce kroku v metrech
origin	Coordinates	Počátek trasy
destination	Coordinates	Cíl trasy
averageSpeed	Double	Průměrná rychlost na trase v km/h
stepNumber	Int	Pořadové číslo kroku trasy

■ **Tabulka 6.4** Prvky JSON export kroku trasy

```
{
  "duration": Int,
  "variables": Array<"String":String>,
  "distance": Int,
  "origin": Coordinates,
  "destination": Coordinates,
  "averageSpeed": Double,
  "stepNumber": Int
}
```

■ **Výpis kódu 6.5** Struktura JSON export kroku trasy

## Coordinates

Pole	Typ	Popis
lat	Number	Informace o zeměpisné šířce
lng	Number	Informace o zeměpisné délce
ele	Double	Informace o převýšení

■ **Tabulka 6.5** Prvky JSON export souřadnic

```
{
  "lng": Number,
  "lat": Number,
  "ele": Double
}
```

■ **Výpis kódu 6.6** Struktura JSON export souřadnic

```
{
  "duration": 276,
  "origin": {
    "lng": 0.9736072,
    "lat": 52.0381429,
    "ele": 54
  },
  "destination": {
    "lng": 0.9645744,
    "lat": 52.0333085,
    "ele": 30.05892562866211
  },
  "waypoints": [],
  "steps": [{
    "duration": 276,
    "distance": 2593,
    "origin": {
      "lng": 0.9736072,
      "lat": 52.0381429,
      "ele": 54
    },
    "destination": {
      "lng": 0.9645744,
      "lat": 52.0333085,
      "ele": 30.05892562866211
    },
    "averageSpeed": 33.82,
    "stepNumber": 0
  }]
}
```

■ **Výpis kódu 6.7** Příklad JSON export souboru

### 6.2.4.3 GPX export

Exportovaný GPX soubor dodržuje veškeré konvence dané tímto formátem. Příklad výstupu je uveden ve výpisu kódu 6.8.

```
{
<gpx version="1.1" creator="jileklu2" xmlns="http://www.topografix.com/GPX/1/1">
<trk>
<trkseg>
<trkpt lat=49.99444236350178 lon=14.649442722753902>
<ele>329.0602416992188</ele>
</trkpt>
<trkpt lat=49.9962837 lon=14.6488779>
<ele>325.7085266113281</ele>
</trkpt>
</trkseg>
</trk>
</gpx>
}
```

■ **Výpis kódu 6.8** Příklad GPX export souboru

## 6.3 Implementace výsledné aplikace

### 6.3.1 Jednotlivé součásti aplikace

Pro efektivní splnění funkčních požadavků byla implementace rozdělena na několik od sebe oddělených fází, kde vstupem do každé z fází byly funkční požadavky, jež bylo nutné v dané fázi naimplementovat. Výstupem byla vždy fungující aplikace podporující funkční požadavky všech proběhlých fází a vstupní funkční požadavky právě ukončené fáze. Každý výstup musel zároveň splňovat i nefunkční požadavky na výslednou aplikaci.

Jméno každé z následujících sekcí odpovídá funkčním požadavkům splněných během ní popisované fázi.

Výsledná aplikace podporuje všechny sesbírané požadavky. Kompletní vzhled grafického rozhraní a všech jeho částí je uveden v dodatku A.

#### 6.3.1.1 Zobrazení mapy

Prvotní fáze vývoje byla zaměřena na pouhé zobrazení mapy pomocí `WebView` komponenty. Tohoto bylo docíleno využitím Google Maps JS API.

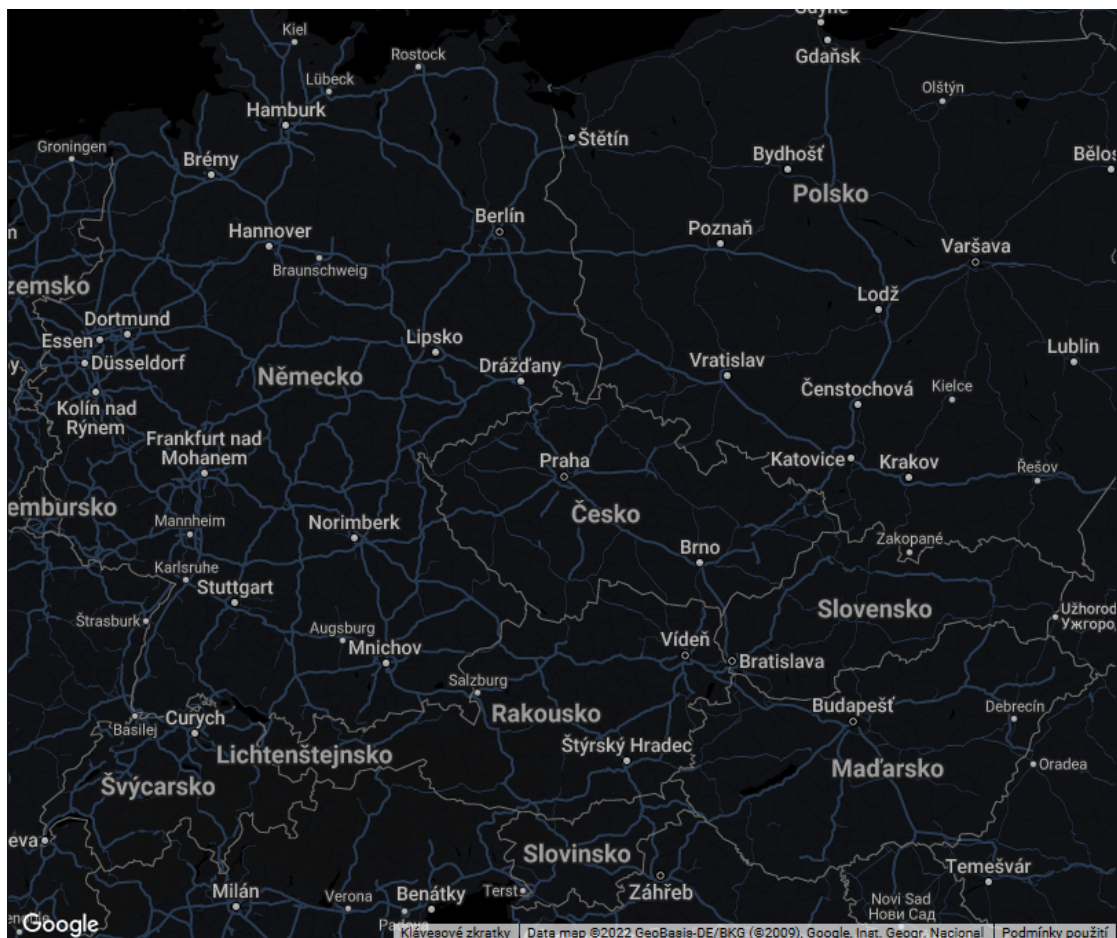
Pro využití Google Maps JS API je nutné se zaregistrovat na *Google Cloud Platform* a vytvořit API key, jež je následně používám v odesílaných požadavcích.

Mapa je vytvořena přímo při inicializaci GUI. Při inicializaci je načítán HTML dokument *google\_map.html*, který využívá JS souboru *script\_google\_map.js* a stylového souboru *style\_google\_map.css*. Mapa je vykreslena pomocí load funkce `initMap()`, která je zobrazena ve výpisu kódu 6.9. Výsledkem toho kódu je mapa zobrazena na obrázku 6.4.

```
function initMap() {  
  const prague = { lat: 50.073658, lng: 14.418540 };  
  map = new google.maps.Map(document.getElementById("map"), {  
    mapId: "9ccba9a851ff1929",  
    zoom: 6,  
    center: prague,  
    disableDefaultUI: true,  
    backgroundColor: '#0f1114'  
  });  
}
```

■ Výpis kódu 6.9 initMap() funkce

■ Obrázek 6.4 Mapa po inicializaci



### 6.3.1.2 Vytvoření, zobrazení a zisk informací o trase

Tato fáze se zaměřuje na splnění funkčního požadavku na vytvoření a zobrazení mapy, zároveň byl v této fázi splněn i požadavek na načtení trasy z JSON souboru. Splnění požadavků bylo dosaženo za pomoci Google Directions API, Google Elevation API a Here Waypoints Sequence API.

Google Directions API je v době programování této fáze využita pro hledání informací o trase a to pouze v čase vytvoření trasy. Tato zodpovědnost se v následujících fázích přenáší na jinou API, která zároveň řeší problém s časovým úsekem. Vlastnosti pro kterou je API využívána i ve výsledné aplikaci je optimalizace pořadí průjezdních bodů a přesun těchto bodů k nim nejbližší silnici.

Google Elevation API ve výsledné aplikaci řeší funkční požadavek na vyhledávání převýšení jednotlivých bodů trasy.

K částečné optimalizaci pořadí průjezdních bodů trasy, která přesahuje délku 25 bodů slouží Here Waypoints Sequence API. Optimalizace je pouze částečná, výsledné pořadí je i tak řešeno pomocí Google Directions API. V tomto případě je však již finální pořadí těchto bodů určeno jen pro úseky po maximálně 25 bodech a pořadí na přelomu zůstává.

Vytvořená trasa musí mít následující vlastnosti:

- Souřadnice počátku a cíle trasy se liší
- Všechny vložené souřadnice musí jsou ve správném formátu
- Počet celkových bodů trasy včetně počátku a cíle je maximálně sto

Ve výsledné aplikaci je možné vytvořit trasu třemi rozdílnými způsoby:

1. Ručním zadáním trasy
2. Výběrem jednotlivých bodů trasy na mapě
3. Načtením trasy z JSON souboru

Ruční zadání bodů a načtení tras z JSON souboru je možné v obou verzích uživatelského rozhraní. Z logických důvodů je výběr jednotlivých bodů na mapě možný pouze při používání aplikace v grafickém režimu.

V CLI režimu je zadání bodů trasy velice přímočaré. Aplikace se spustí, uživatel vybere možnost vytvořit trasu a postupně zadá všechny požadované body trasy. Tento postup je zobrazen na obrázcích 6.5 a 6.6.

```
Select Action:
[1] Make Route
[2] Print Route JSON
[3] Import Route
[4] Export Route
[0] Exit
1
```

(a) Výběr možností z menu

```
Please enter origin:
lat:
51.970390
lng:
0.979328
Please enter destination:
lat:
52.094427
lng:
0.668408
```

(b) Zadání souřadnic

■ **Obrázek 6.5** CLI tvorba trasy (1)



```

Do you want to add route waypoint?
[1] Yes
[2] No
1
lat:
52.038683
lng:
0.730226

```

(a) Přidání bodů trasy

```

Do you want to add route waypoint?
[1] Yes
[2] No
2
Select Action:
[1] Make Route
[2] Print Route JSON
[3] Import Route
[4] Export Route
[0] Exit

```

(b) Ukončení procesu tvorby

■ **Obrázek 6.6** CLI tvorba trasy (2)

Načtení trasy ze souboru probíhá v konzolovém rozhraní zadáním cesty k souboru přímo do argumentu při spuštění aplikace, nebo vybráním odpovídající možnosti v menu a následným vložením cesty. Tento proces je zachycen na obrázku 6.7.

```

Select Action:
[1] Make Route
[2] Print Route JSON
[3] Import Route
[4] Export Route
[0] Exit
3

```

(a) Zvolení možnosti

```

Please enter file path:
src/main/resources/in_02.json
Select Action:
[1] Make Route
[2] Print Route JSON
[3] Import Route
[4] Export Route
[0] Exit

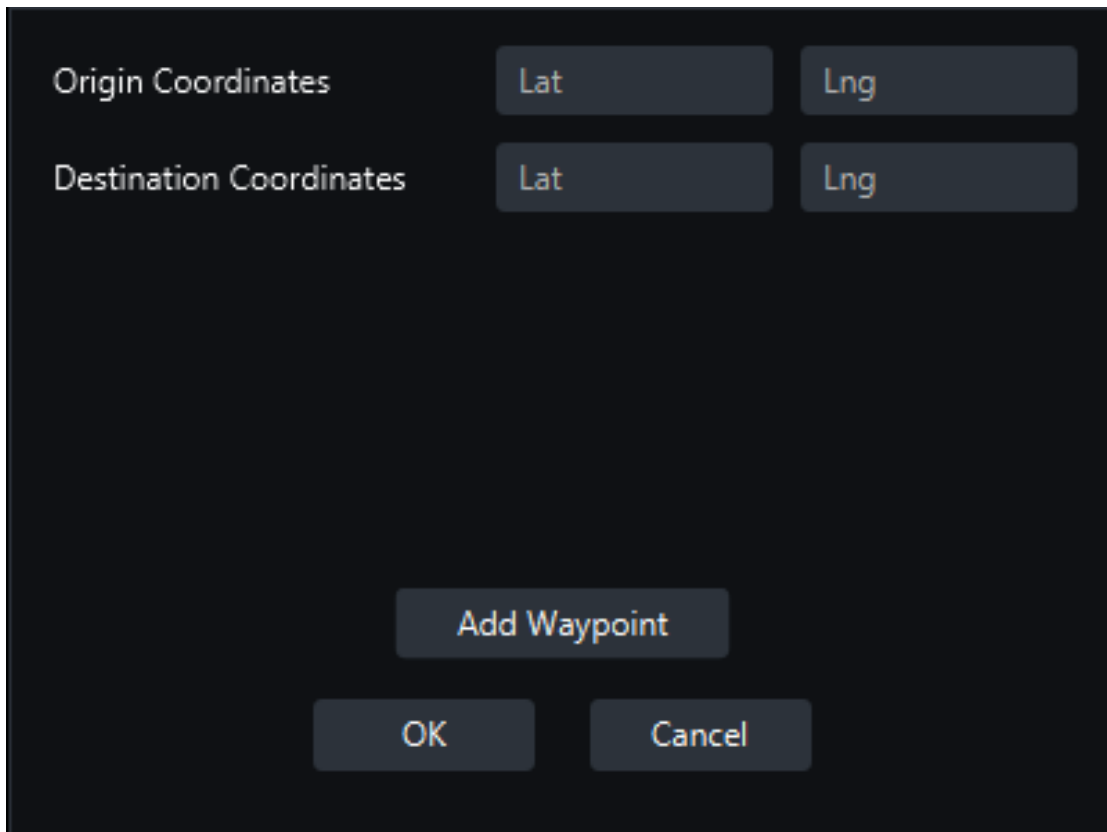
```

(b) Úspěšné načtení trasy

■ **Obrázek 6.7** CLI import trasy

GUI nabízí tvorbu trasy zadáním jednotlivých souřadnic pomocí vyskakovacího okna, jež se zobrazí po kliknutí na tlačítko *Make route*. Po zadání souřadnic počátku, cíle a případně jednotlivých bodů trasy stačí uživateli kliknout na tlačítko *OK*. To způsobí nahrání trasy do aplikace a její zobrazení. Při tvorbě trasy pomocí vyskakovacího okna musí být vyplněna alespoň pole odpovídající souřadnicím počátku a cíle trasy. Okno pro tvorbu trasy je zobrazeno na obrázku 6.8.

■ **Obrázek 6.8** GUI Okno pro tvorbu trasy



Chce-li uživatel nahrát trasu ze souboru, klikne na tlačítko *Import route* a vloží cestu k souboru do nového vyskakovacího okna. Zadání cesty potvrdí tlačítkem *OK*. Okno pro načtení trasy je zobrazeno na obrázku 6.9.

■ **Obrázek 6.9** GUI Okno pro import trasy



Tvorba trasy výběrem na mapě probíhá postupným přidáváním bodů pomocí kliknutí do místa kde se má daný bod nacházet. Svou volbu trasy uživatel potvrdí dvojklikem při výběru posledního bodu.

Při použití výše zmíněných API dochází k volání jejich služeb. Tato volání vrací následující stavové kódy, které je nutné následně zpracovat:

## Google Directions API

Kód	Název	Popis
200	OK	Odpověď obsahuje validní výsledek
204	ZERO_RESULTS	Zadaná trasa nebyla nalezena
400	INVALID_REQUEST	Špatně zadaná žádost
402	OVER_DAILY_LIMIT	API kód nebyl uveden, nebyl nastaven způsob platby, byl překročen limit užití, nebo metoda platby již není validní
403	REQUEST_DENIED	Požadavek byl zamítnut
404	NOT_FOUND	Alespoň jeden ze zadaných bodů nebyl nalezen
429	OVER_QUERY_LIMIT	Bylo zasláno příliš mnoho požadavků za určitý čas
520	UNKNOWN_ERROR	Neznámý problém
4130	MAX_WAYPOINTS_EXCEEDED	Trasa překročila limit 25 bodů
4131	MAX_ROUTE_LENGTH_EXCEEDED	Trasa je příliš komplexní na zpracování

■ **Tabulka 6.6** Statusy odpovědi Google Directions API

## Google Elevation API

Kód	Název	Popis
200	OK	Odpověď obsahuje validní výsledek
400	INVALID_REQUEST	Špatně zadaná žádost
402	OVER_DAILY_LIMIT	API kód nebyl uveden, nebyl nastaven způsob platby, byl překročen limit užití, nebo metoda platby již není validní
403	REQUEST_DENIED	Požadavek byl zamítnut
404	DATA_NOT_AVAILABLE	Pro zadané body neexistují data
429	OVER_QUERY_LIMIT	Bylo zasláno příliš mnoho požadavků za určitý čas
520	UNKNOWN_ERROR	Neznámý problém

■ **Tabulka 6.7** Statusy odpovědi Google Elevation API

## Here Waypoints Sequence API

Kód	Název	Popis
200	OK	Odpověď obsahuje validní výsledek
201	CREATED	Odpověď obsahuje validní výsledek, výskyt pouze v případě použití PUT
401	UNAUTHORIZED	Neautorizovaný přístup k API službám
403	FORBIDDEN	Požadavek byl zamítnut
404	NOT_FOUND	Pro zadané body neexistují data

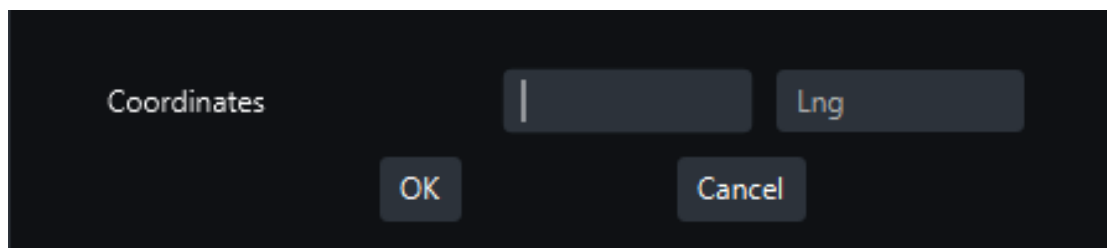
■ **Tabulka 6.8** Statusy odpovědi Here Waypoints Sequence API

### 6.3.1.3 Přidání nového bodu trasy

V této fázi byla vyřešen pouze jeden z funkčních požadavků na aplikaci. Tímto požadavkem je přidání nového bodu trasy. Při přidání nového bodu je vytvořena nová trasa, a tedy všechny vytvořené vlastní proměnné pro úseky jsou smazány, společně s jejich nastavenými středními rychlostmi. Přidání nového bodu trasy CLI nepodporuje.

Používá-li uživatel GUI, nový bod vytvoří kliknutím na tlačítko *Add waypoint* a ve vyskakovacím okně zadá jeho souřadnice. Okno pro přidání souřadnic je zobrazeno na obrázku 6.10.

■ **Obrázek 6.10** GUI Okno pro přidání nového bodu



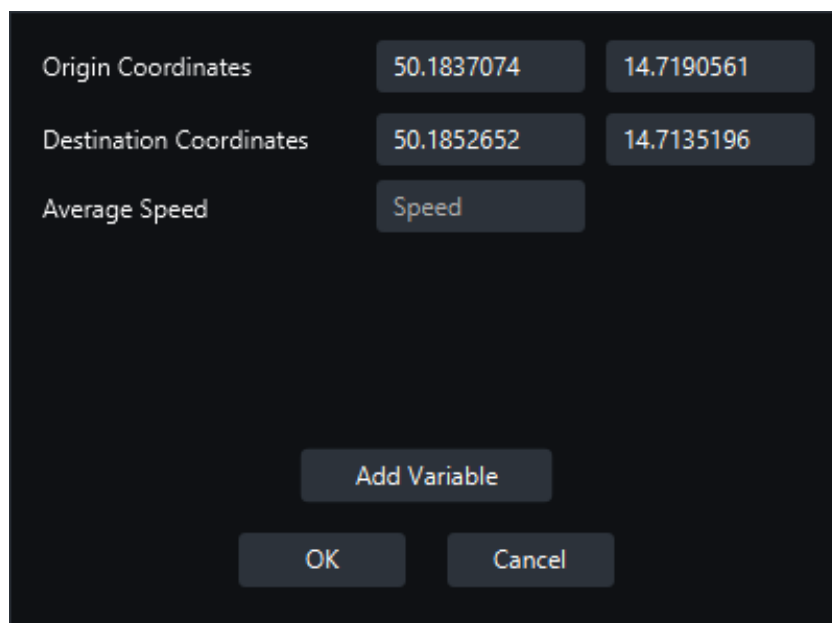
### 6.3.1.4 Úprava vlastností jednotlivých úseků

Požadavek na úpravu jednotlivých bodů zvoleného úseku a přidání nových vlastních proměnných byl vyřešen během této vývojové fáze.

Úprava úseků je možná pouze v GUI verzi finální aplikace.

Pro úpravu souřadnic a proměnných při používání GUI je používáno vyskakovací okno, jež se otevírá dvojklikem na výpis úseku. Souřadnice lze upravit přepsáním jejich hodnoty a nová proměnná se přidává pomocí tlačítka *Add variable*. Všechny změny jsou potvrzeny tlačítkem *OK*. Jednotlivé souřadnice úseků lze taktéž upravovat jejich přetažením na žádané místo na mapě. Vyskakovací okno úpravu úseku je zobrazeno na obrázku 6.11.

■ **Obrázek 6.11** GUI Okno pro úpravu úseku trasy

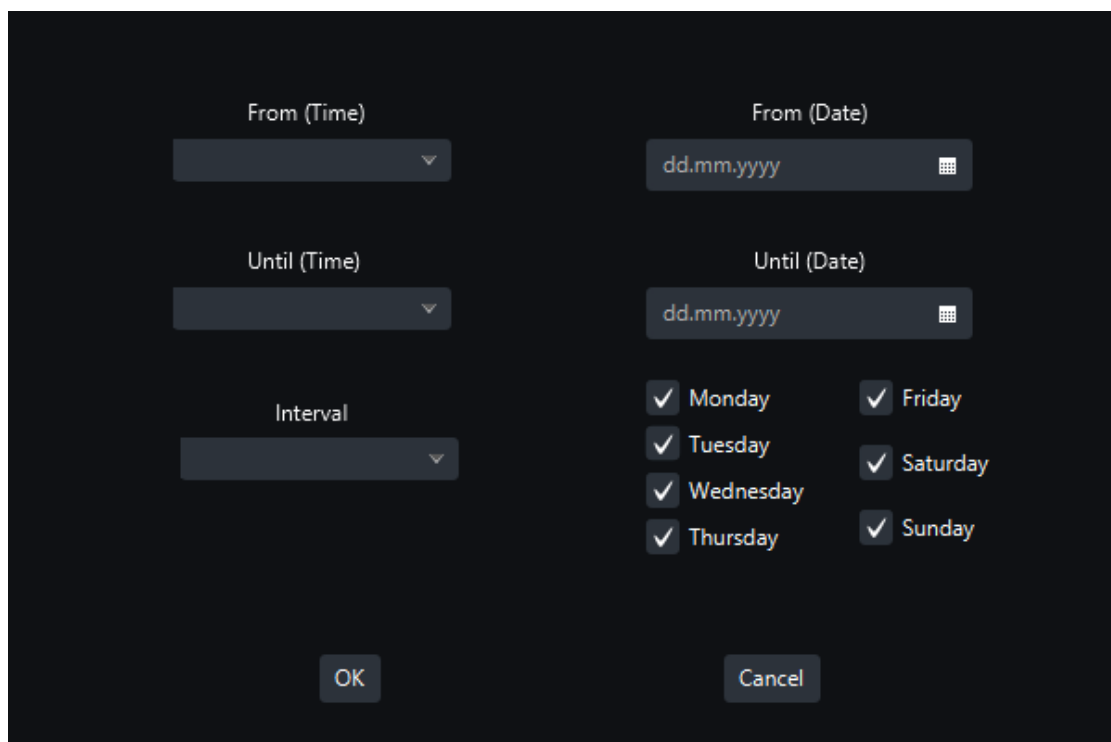


### 6.3.1.5 Získání informací o trase a jejích částech v zadaný čas

Nejproblematictější fází vývoje byla implementace zisku informací trasy pro zadaný časový úsek. Nejprve se tento funkční požadavek týkal historický dat. K jejich zisku měla být použita TomTom Traffic Stats API. Bohužel během vývoje došlo ke změně jejích licenčních podmínek a již nebylo možné tuto API použít. Po následné rešerši možných alternativních řešeních a konzultaci s vedoucím práce došlo ke změně funkčního požadavku na zisk dat, která se týkají budoucnosti, ale vycházejí z historických pozorování. Ke splnění tohoto požadavku je využívána Google Distance Matrix API. Bohužel tato API svůj účel neplní zcela dle představ, neboť data o provozu nejsou získávána zvláště, ale promítají se do doby jízdy. Zároveň bylo nutné naimplementovat agregaci těchto dat přes všechny datумы v časovém úseku s následným výpočtem jejich průměru. Tedy výsledkem je průměrná predikce času průjezdu dané trasy. Tato funkce je podporována pouze v GUI režimu, protože byl vznesen požadavek na zahrnutí složitější forem kombinací času, např. pouze určité dny v týdny.

Výběr časového úseku probíhá ve vyskakovacím okně, jež se otevře po kliknutí na tlačítko *Set interval*. A je následně potvrzen stiskem tlačítka OK. Toto vyskakovací okna je zobrazeno na obrázku 6.12.

■ **Obrázek 6.12** GUI Okno pro nastavení časového intervalu



The image shows a dark-themed GUI dialog box for setting a time interval. It contains the following elements:

- From (Time):** A dropdown menu.
- From (Date):** A text field with the placeholder "dd.mm.yyyy" and a calendar icon.
- Until (Time):** A dropdown menu.
- Until (Date):** A text field with the placeholder "dd.mm.yyyy" and a calendar icon.
- Interval:** A dropdown menu.
- Days of the Week:** A list of days (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday) with checked checkboxes.
- Buttons:** "OK" and "Cancel" buttons at the bottom.

Jelikož je využívána webová API, je nutné zpracovat následující stavové kódy:

## Google Distance Matrix API

Kód	Název	Popis
200	OK	Odpověď obsahuje validní výsledek
400	INVALID_REQUEST	Špatně zadaná žádost
402	OVER_DAILY_LIMIT	API kód nebyl uveden, nebyl nastaven způsob platby, byl překročen limit užití, nebo metoda platby již není validní
403	REQUEST_DENIED	Požadavek byl zamítnut
429	OVER_QUERY_LIMIT	Bylo zasláno příliš mnoho požadavků za určitý čas
520	UNKNOWN_ERROR	Neznámý problém
4130	MAX_ELEMENTS_EXCEEDED	V matici je vytvořeno příliš mnoho prvků
4131	MAX_ROUTE_DIMENSIONS_EXCEEDED	Do matice vstupuje příliš mnoho prvků

■ **Tabulka 6.9** Statusy odpovědi Google Distance Matrix API

### 6.3.1.6 Export ve formátu GPX a JSON

V poslední fázi byl naprogramován export trasy do formátu GPX a JSON.

Při používání CLI aplikace může být výstupní soubor zadán přímo mezi argumenty při spuštění, nebo vyplněn po výběru možnosti *Export route*. Proces exportu je zachycen na obrázku 6.13.

```
Select Action:
[1] Make Route
[2] Print Route JSON
[3] Import Route
[4] Export Route
[0] Exit
4
```

(a) Zvolení možnosti

```
Please enter file path:
src/main/resources/out_01.gpx
Export successful.
Select Action:
[1] Make Route
[2] Print Route JSON
[3] Import Route
[4] Export Route
[0] Exit
```

(b) Úspěšný export trasy

■ **Obrázek 6.13** CLI export trasy

Pokud uživatel používá GUI verzi, tak je export proveden pomocí tlačítka *Export route* a následného vložení cesty do pole ve vyskakovacím okně. Export je potvrzen stisknutím tlačítka *OK*. Okno exportu je zobrazeno na obrázku 6.14

■ **Obrázek 6.14** GUI Okno export trasy



## 6.4 Dokumentace výsledné aplikace

Dokumentace kódu je vygenerována pomocí nástroje Javadoc ve formátu HTML a popisuje jednotlivé struktury společně s jejich funkcí. Dokumentace je psána v anglickém jazyce, neboť je přímou součástí kódu, který je také napsán s využitím anglického jazyka pro pojmenování elementů.

Dokumentaci kódu lze nalézt na přiloženém médiu, nebo se její výtisk nachází v příloze B této práce.

## 6.5 Testování výsledné aplikace

Struktury aplikace jsou jednotlivě testovány unit testy s využitím knihovny JUnit. Pokrytí aplikace není stoprocentní, neboť metody, jež spoléhají na webové APIs, nelze spolehlivě otestovat z důvodu jejich nedeterministického chování pro určité vstupy a těžké replikaci podmínek konkrétních situací.



## Kapitola 7

# Závěr

Výsledkem práce je desktopová aplikace pro generování tras pro použití v simulačních modelech vozidla. Aplikace splňuje veškeré požadované funkcionality, jež jsou v této práci zmíněny. Aplikace tedy podporuje načtení či vytvoření trasy, načtení informací o dané trase, přidání či odebrání jednotlivých bodů trasy, úpravu parametrů úseků trasy a umožňuje export do GPX formátu. Požadavek na získání informace ohledně doby trvání trasy nebyl splněn podle původních představ zadavatele, neboť došlo ke změně licence klíčové API a informace, jež tato API poskytovala, již nejsou volně dostupné. Ke splnění tohoto požadavku došlo použitím alternativního, ale ne již tak přesného přístupu, neboť byla použita jiná API, jež nebyla pro tento účel přímo určena. Zároveň aplikace využívá vzorů a zásad objektově orientovaného programování. Do budoucna by tedy neměl být problém tuto aplikaci rozšířit či upravit. Aplikace byla otestována pomocí knihovny JUnit a dokumentována s použitím nástroje Javadoc.



# Bibliografie

1. ISO/IEC/IEEE 24765:2017(en), Systems and software engineering — Vocabulary. *ISO - International Organization for Standardization* [online]. 2017 [cit. 2022-04-04]. Dostupné z: <https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:24765:ed-2:v1:en>.
2. POTHOS, Emmanuel M.; WILLS, Andy J. *Formal Approaches in Categorization*. Cambridge University Press, 2011. ISBN 978-1-139-49397-0.
3. PCMAG. What does desktop application actually mean? *PCMAG* [online]. 2022 [cit. 2022-04-04]. Dostupné z: <https://www.pcmag.com/encyclopedia/term/desktop-application>.
4. POP, Paul. *Comparing Web Applications with Desktop Applications: An Empirical Study*. 2002.
5. NAYEBI, Fatih; DESHARNAIS, Jean-Marc; ABRAN, Alain. The State of the Art of Mobile Application Usability Evaluation. *Canadian Conference on Electrical and Computer Engineering*. 2012. Dostupné z DOI: 10.1109/CCECE.2012.6334930.
6. KRATZKE, Nane; QUINT, Peter-Christian. Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. *Elsevier*. 2017. Dostupné z DOI: 10.1016/j.jss.2017.01.001.
7. ALLAIRE, Jeremy. Macromedia Flash MX—A next-generation rich client. 2002, s. 14.
8. DISSANAYAKE, Nalaka; DIAS, Kapila. Web-based Applications: Extending the General Perspective of the Service of Web. In: 2017.
9. CERUZZI, Paul E. *A History of Modern Computing*. MIT Press, 2003.
10. POLČÁK, Radim. *Právo informačních technologií*. Praha: Wolters Kluwer, 2018. ISBN 978-80-7598-045-8.
11. JANSÁ, Lukáš. *Softwarové právo*. Brno: Computer Press, 2014. ISBN 978-80-251-4201-1.
12. MYŠKA, Matěj. *Veřejné licence v České republice : verze 2.0*. Brno: Masarykova univerzita, 2014. ISBN 978-80-210-7192-6.
13. SHEMTOV, Noam. *Free and open source software : policy, law, and practice*. Oxford, United Kingdom: Oxford University Press, 2013. ISBN 9780199680498.
14. WERNER DANKWORT, C.; WEIDLICH, Roland; GUENTHER, Birgit; BLAUROCK, Joerg E. Engineers' CAX education—it's not only CAD. *Computer-Aided Design*. 2004, roč. 36, č. 14, s. 1439–1450. ISSN 0010-4485. Dostupné z DOI: 10.1016/j.cad.2004.02.011.
15. UNITED STATES, O.T.A.C. *Computerized manufacturing automation : employment, education, and the workplace*. Congress, Office Of Technology Assessment, 1984. Č. sv. 2. ISBN 9781428923645. Dostupné také z: <https://books.google.cz/books?id=LYdF5akRL6sC>.

16. STERN, Garland. Bbop - a system for 3D keyframe figure animation. 1983.
17. WOOD, Aylish. *Software, Animation and the Moving Image*. Palgrave Macmillan UK, 2015. ISBN 978-1-349-49660-0. Dostupné z DOI: 10.1057/9781137448859.
18. REDDY, Martin. *API design for C++*. Morgan Kaufmann, 2011. ISBN 978-0-12-385003-4.
19. QIU, Dong; LI, Bixin; LEUNG, Hareton. Understanding the API usage in Java. *Information and Software Technology*. 2016, roč. 73, s. 81–100. ISSN 09505849. Dostupné z DOI: 10.1016/j.infsof.2016.01.011.
20. OFOEDA, Joshua; BOATENG, Richard; EFFAH, John. Application Programming Interface (API) Research: A Review of the Past to Inform the Future. *International Journal of Enterprise Information Systems*. 2019, roč. 15, s. 76–95. Dostupné z DOI: 10.4018/IJEIS.2019070105.
21. ESPINHA, Tiago; ZAIDMAN, Andy; GROSS, Hans-Gerhard. Web API growing pains: Loosely coupled yet strongly tied. *Journal of Systems and Software*. 2015, roč. 100, s. 27–43. ISSN 0164-1212. Dostupné z DOI: <https://doi.org/10.1016/j.jss.2014.10.014>.
22. WHEELER, David A. Static Libraries. *Program Library HOWTO, 2. Static Libraries* [online]. 2003 [cit. 2022-04-09]. Dostupné z: <https://tldp.org/HOWTO/Program-Library-HOWTO/static-libraries.html>.
23. LEWINE, Donald A. *POSIX programmer's guide: writing portable UNIX programs with the POSIX.1 standard*. 1st ed. O'Reilly & Associates, 1991. ISBN 978-0-937175-73-6.
24. SILBERSCHATZ, Galvin; GAGNE. Lecture 2: System Calls & API Standards[přednáška] [online]. 2005 [cit. 2022-04-11]. Dostupné z: [https://cw.fel.cvut.cz/old/\\_media/courses/ae3b33osd/osd-lec2-14.pdf](https://cw.fel.cvut.cz/old/_media/courses/ae3b33osd/osd-lec2-14.pdf).
25. WEST, Joel; DEDRICK, Jason. Open Source Standardization: The Rise of Linux in the Network Era. 2001, s. 25.
26. HARGREAVES, Shawn. Announcing DirectX 12 Ultimate. *DirectX Developer Blog* [online]. 2020 [cit. 2022-04-13]. Dostupné z: <https://devblogs.microsoft.com/directx/announcing-directx-12-ultimate/>.
27. SEGAL, Mark; AKELEY, Kurt. The OpenGL Graphics System: A Specification (OpenGL 4.0 (Core Profile) - March 11, 2010) [online]. 2010, s. 489 [cit. 2022-04-12]. Dostupné z: <https://www.khronos.org/registry/OpenGL/specs/gl/glspec40.core.pdf>.
28. ALONSO, Gustavo; CASATI, Fabio; KUNO, Harumi; MACHIRAJU, Vijay. Web Services. In: *Web Services: Concepts, Architectures and Applications*. Springer Berlin Heidelberg, 2004, s. 123–149. ISBN 978-3-662-10876-5. Dostupné z DOI: 10.1007/978-3-662-10876-5\_5.
29. GUDGIN, Martin; HADLEY, Marc; MENDELSON, Noah; MOREAU, Jean-Jacques; NIELSEN, Henrik F.; KARMARKAR, Anish; LAFON, Yves. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). *W3C Recommendation* [online]. 2007 [cit. 2022-04-13]. Dostupné z: <https://www.w3.org/TR/soap12-part1/#intro>.
30. FIELDING, Roy Thoman. Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST). *Architectural Styles and the Design of Network-based Software Architectures* [online]. 2000 [cit. 2022-04-13]. Dostupné z: [https://www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm).
31. CHIN, Roger S.; SAMUEL; CHANSON, T. Distributed object-based programming systems. *ACM Computing Surveys*. 1991, roč. 23, s. 91–124.
32. FUGGETTA, A.; PICCO, G.P.; VIGNA, G. Understanding code mobility. *IEEE Transactions on Software Engineering*. 1998, roč. 24, č. 5, s. 342–361. ISSN 1939-3520. Dostupné z DOI: 10.1109/32.685258.

33. AABY, Anthony A. Introduction [online]. 2012 [cit. 2022-04-14]. Dostupné z: <https://web.archive.org/web/20121108043216/http://www.emu.edu.tr/aelci/Courses/D-318/D-318-Files/plbook/intro.htm>.
34. *Internetová jazyková příručka - Paradigma* [online]. Praha: Ústav pro jazyk český AV ČR, v.v.i., 2021 [cit. 2022-04-14]. Dostupné z: <https://prirucka.ujc.cas.cz/?slovo=paradigma>.
35. AVACHEVA, T; PRUTZKOW, Alexander. The Evolution of Imperative Programming Paradigms as a Search for New Ways to Reduce Code Duplication. *IOP Conference Series Materials Science and Engineering*. 2020, roč. 714, s. 012001. Dostupné z DOI: 10.1088/1757-899X/714/1/012001.
36. DAHL, Ole-Johan; DIJKSTRA, Edsger Wybe; HOARE, Charles Antony Richard. *Structured programming*. Academic Press Ltd., 1972.
37. RANDELL, Brian; BUXTON, JN. Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27th-31st October 1969. 1970.
38. DIJKSTRA, Edsger W. Letters to the Editor: Go to Statement Considered Harmful. *Commun. ACM*. 1968, roč. 11, č. 3, s. 147–148. ISSN 0001-0782. Dostupné z DOI: 10.1145/362929.362947.
39. BROOKSHEAR, J. Glenn; SMITH, David T.; BRYLOW, Dennis. *Computer science: an overview*. 11th ed. Boston: Addison-Wesley, 2012. ISBN 978-0-13-256903-3.
40. JANOUŠEK, Jan. *Programovací paradigmat, Úvod, Přehled hlavních programovacích paradigmat, Imperativní programování* [online]. Prague: Czech Technical University in Prague, 2019 [cit. 2022-04-16]. Dostupné z: [https://courses.fit.cvut.cz/BI-PPA/lectures/files/ppa2019-01.pdf?fbclid=IwAR0FNZlTsye\\_aPvmeT43oWciJRbpMchrW7CUw8w3MeLSwncItZJZoKHAki](https://courses.fit.cvut.cz/BI-PPA/lectures/files/ppa2019-01.pdf?fbclid=IwAR0FNZlTsye_aPvmeT43oWciJRbpMchrW7CUw8w3MeLSwncItZJZoKHAki).
41. WEGNER, Peter. Concepts and Paradigms of Object-Oriented Programming. 1990, roč. 1, č. 1, s. 7–87. ISSN 1055-6400. Dostupné z DOI: 10.1145/382192.383004.
42. RENTSCH, Tim. Object oriented programming. *ACM Sigplan Notices*. 1982, roč. 17, č. 9, s. 51–57.
43. KŘÍKAVA, Filip. *BI-OOP 2019 - Lecture 1: Introduction* [online]. Prague: Czech Technical University in Prague, 2019 [cit. 2022-04-16]. Dostupné z: [https://docs.google.com/presentation/d/1IEFqGU\\_OrXRc2fK1\\_PHM9-LarjWYRt80xlQmclvARaM](https://docs.google.com/presentation/d/1IEFqGU_OrXRc2fK1_PHM9-LarjWYRt80xlQmclvARaM).
44. STEFIK, Mark; BOBROW, Daniel G. Object-oriented programming: Themes and variations. *AI magazine*. 1985, roč. 6, č. 4, s. 40–40.
45. KŘÍKAVA, Filip. *BI-OOP 2021 - Lecture 2: Polymorphism* [online]. Prague: Czech Technical University in Prague, 2021 [cit. 2022-04-16]. Dostupné z: <https://docs.google.com/presentation/d/1Pqvoxu2MZq1cT2Wm0dwIKiCbNDSXE7P7ABHCyBgiFhA>.
46. LISKOV, Barbara H.; WING, Jeannette M. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.* 1994, roč. 16, č. 6, s. 1811–1841. ISSN 0164-0925. Dostupné z DOI: 10.1145/197320.197383.
47. PIERCE, Benjamin C. *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002. ISBN 978-0-262-16209-8.
48. LLOYD, JW. Practical advantages of declarative programming. In: 1994, s. 3–17. Conference Proceedings/Title of Journal: Joint Conference on Declarative Programming.
49. HUGHES, J. Why Functional Programming Matters. *The Computer Journal*. 1989, roč. 32, č. 2, s. 98–107. ISSN 0010-4620. Dostupné z DOI: 10.1093/comjnl/32.2.98.
50. *Requirements Engineerin: Sběr a analýza požadavků* [online]. PROFINIT, 2021-10 [cit. 2022-04-23]. Dostupné z: <https://web.microsoftstream.com/video/60bc2cbb-f855-442f-8dc4-79a33dfc3d23>.

## Bibliografie

51. WIEGERS, K.; BEATTY, J. *Software Requirements*. Pearson Education, 2013. Developer Best Practices. ISBN 978-0-7356-7962-7. Dostupné také z: <https://books.google.cz/books?id=nbpCAwAAQBAJ>.
52. COHN, M. *User Stories Applied: For Agile Software Development*. Addison-Wesley, 2004. Addison-Wesley signature series. ISBN 978-0-321-20568-1. Dostupné také z: <https://books.google.cz/books?id=SvIwuX4SVigC>.
53. *ISO/IEC/IEEE 29148:2018(en), Systems and software engineering — Life cycle processes — Requirements engineering* [online]. 2018 [cit. 2022-04-22]. Dostupné z: <https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:29148:ed-2:v1:en>.
54. IEEE. IEEE Guide for Developing System Requirements Specifications. *IEEE Std 1233-1996*. 1996, s. 1–30. Dostupné z DOI: 10.1109/IEEESTD.1996.81000.
55. IEEE. IEEE Recommended Practice for Software Requirements Specifications. *IEEE Std 830-1998*. 1998, s. 1–40. Dostupné z DOI: 10.1109/IEEESTD.1998.88286.
56. MLEJNEK, Jiří. *Návrh softwarových systémů* [online]. Czech Technical University in Prague, 2021 [cit. 2022-04-25]. Dostupné z: [https://moodle-vyuka.cvut.cz/pluginfile.php/506252/mod\\_resource/content/6/05.prednaska.pdf](https://moodle-vyuka.cvut.cz/pluginfile.php/506252/mod_resource/content/6/05.prednaska.pdf).
57. BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice: Software Architect Practice*. Pearson Education, 2012. SEI Series in Software Engineering. ISBN 978-0-13-294278-2. Dostupné také z: <https://books.google.cz/books?id=-II73rBDXCYC>.
58. CLEMENTS, P.; BACHMANN, F.; BASS, L.; GARLAN, D.; IVERS, J.; LITTLE, R.; MERSON, P.; NORD, R.; STAFFORD, J. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2010. SEI Series in Software Engineering. ISBN 978-0-13-248859-4. Dostupné také z: <https://books.google.cz/books?id=UTZbsrA4qAsC>.
59. GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994. Addison-Wesley Professional Computing Series. ISBN 978-0-321-70069-8. Dostupné také z: <https://books.google.cz/books?id=6oHuKQe3TjQC>.
60. BUSCHMANN, F.; SCHMIDT, D.C.; MEUNIER, R.; ROHNERT, H.; KIRCHER, M.; STAL, M.; SOMMERLAD, P.; JAIN, P. *Pattern-Oriented Software Architecture, A System of Patterns*. Wiley, 1996. EBL-Schweitzer. ISBN 978-0-471-95869-7. Dostupné také z: <https://books.google.cz/books?id=gJjgAAAAMAAJ>.
61. LARMAN, Craig. *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development*. Prentice Hall PTR, 2005. ISBN 978-0-13-148906-6. Google-Books-ID: tuxQAAAAMAAJ.
62. BECK, K. *Implementation Patterns*. Pearson Education, 2007. Addison-Wesley Signature Series (Beck). ISBN 9780132702553. Dostupné také z: <https://books.google.cz/books?id=xLyXPCxBhQUC>.
63. HANSON, C.; SUSSMAN, G.J. *Software Design for Flexibility: How to Avoid Programming Yourself into a Corner*. MIT Press, 2021. ISBN 9780262045490. Dostupné také z: <https://books.google.cz/books?id=AOUbEAAAQBAJ>.