FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Plánovač alokací na projektech
Student:	Martin Německý
Vedoucí:	Ing. Jan Matoušek
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Ve světě již existuje více aplikací pro řešení alokací (https://www.getharvest.com/forecast, https://www.forecast.app/, https://toggl.com/plan/), ale žádná pořádně nepracuje s transparencí příležitostí (na čem se dá pracovat) a preferencemi zdrojů (kdo chce na čem pracovat). Cílem práce je identifikovat požadavky a následně navrhnout a implementovat jednoduchou webovou aplikaci, která by tento problém řešila. Důležitou součástí systému je plánovací engine hledající řešení problému v souladu s omezeními (kdo na čem chce pracovat, kdo má kapacitu, kdo má kompetence).

Pokyny k vypracování:
1) Analyzujte potřeby plánovací aplikace, která pracuje s transparencí příležitostí.
2) Proveďte rešerši konkurenčních aplikací.
3) Proveďte rešerši plánovacích algoritmů a zvolte vhodný algoritmus pro řešenou aplikaci.
4) Navrhněte architekturu, datové struktury a uživatelské rozhraní aplikace.
5) Implementujte navrženou aplikaci.
6) Aplikaci podrobte vhodným automatickým i uživatelským testům.
7) Dosažené výsledky shrňte a navrhněte vylepšení.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 19. ledna 2023 v Praze.

Bakalářská práce

PLÁNOVAČ ALOKACÍ NA PROJEKTECH

Martin Německý

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jan Matoušek
10. května 2023

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Mé poděkování patří Ing. Janu Matouškovi za vedení mé bakalářské práce. Dále bych rád poděkoval Ing. Filipu Kirschnerovi za trpělivost, ochotu a cenné rady, které mi pomohly tuto práci zkompletovat. V neposlední řadě bych chtěl poděkovat svým rodičům a rodičům své přítelkyně za podporu v průběhu celého mého studia. Poslední mé velké poděkování náleží mé přítelkyni za přísuny neuropeptidu $C_{158}H_{251}N_{39}O_{46}S$, kdykoliv to bylo potřeba.

Prohlášení

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

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

V Praze dne 10. května 2023 .....

Abstrakt

Bakalářská práce se zabývá problémem alokací zdrojů (zaměstnanců) na projekty, kde hlavním kritériem přiřazení jsou preference zaměstnanců na projekty. Je zde využita technika constraint programming, která dovoluje problém alokací řešit deklarativním způsobem pomocí specifikace omezení. Práce dále obsahuje analýzu požadavků, ze které vychází návrh i architektura aplikace. Následná implementace aplikace je otestována pomocí generovaných dat na dobu výpočtu i kvalitu řešení.

Klíčová slova webová aplikace, automatizace rozvrhování, alokace zdrojů, problém splnitelnosti omezujících podmínek, constraint programming, algoritmus simulovaného ochlazování, Optaplanner, Java, React

Abstract

This bachelor thesis focuses on the project resource (employee) allocation problem, where the main assignment criterion is the preferences of employees for projects. The constraint programming technique is used here, which allows the allocation problem to be solved in a declarative way using the constraint specification. The work also contains an analysis of the requirements, from which the design and architecture of the application are based. The subsequent implementation of the application is tested using the generated data for calculation time and solution quality.

Keywords web application, scheduling automation, resource allocation, constraint satisfaction problem, constraint programming, Simulated Annealing Algorithm, Optaplanner, Java, React

Seznam zkratek

API	Application Programming Interface
CSP	Constraint Satisfaction Problem, problém splnitelnosti omezujících podmínek
CSS	Cascading Style Sheets
CSV	Comma-Separated Values
DRY	Don't Repeat Yourself
FTE	Full-Time Equivalent, ekvivalent plného úvazku
GA	Genetic Algorithms, genetické algoritmy
HTTP	Hypertext Transfer Protocol
id	Identity Document
JAR	Java ARchive
JSON	JavaScript Object Notation
KISS	Keep It Simple, Stupid
npm	Node Package Manager
REST	Representational State Transfer
SA	Simulated Annealing, algoritmus simulovaného ochlazování
UC	Use-Case, případ užití
UI	User Interface, uživatelské rozhraní
XML	Extensible Markup Language

Kapitola 1
Úvod

V současné době již existuje mnoho nástrojů pro řešení problému rozvrhování (resp. alokací), ale žádný pořádně nepracuje s transparencí příležitostí (na čem se dá pracovat) a preferencemi zdrojů (kdo chce na čem pracovat).

F. Laloux se ve své knize *Budoucnost organizací* [1] zabývá úvahou nad lidským vývojem a uvažováním v prostoru organizací. V této knize jsou organizace rozděleny do kategorií na základě jejich vnitřního fungování a světa, ve kterém se nacházejí. Nejnovějším typem organizací je taková organizace, která rozpoznává zaměstnance jako lidi, a ne pouze jako zdroje, které produkují ekonomickou hodnotu. Stejně tak uznává, že zaměstnanci jsou emocionální bytosti, které mají své potřeby a touhy. F. Laloux uvádí, že lidi jsou rozumná stvoření, která sama nejlépe vědí, co potřebují, a to i při své práci. Dále říká, že pokud lidé mají svobodu při rozhodování, jsou potom více motivovaní a jejich efektivita práce roste. Konkrétními příklady organizací, které praktikují tuto filozofii, jsou např. FAVI, AES, Buurtzorg, atd.

Cílem této bakalářské práce je vytvořit jednoduchou webovou aplikaci, která bude řešit alokování zdrojů (resp. zaměstnanců) na projekty na základě preferencí zaměstnanců. Tyto alokace je možné pak modelovat pomocí problému splitelnosti omezujících podmínek. Součástí aplikace bude i plánovací engine hledající řešení v souladu se zadanými omezeními. Dílčím cílem je analýza existujících aplikací, která zkoumá komerční i open-source projekty zabývající se zkoumaným problémem. Práce bude také obsahovat rešerši algoritmů a zvolení jednoho konkrétního algoritmu, který bude implementován. Dalším cílem je identifikace požadavků zadavatele a jejich prioritizace. Na základě těchto požadavků bude vytvořen návrh a architektura výsledné aplikace, která bude v dalším kroku implementována a otestována uživatelskými i automatickými testy.

Kapitola 2

Problém splnitelnosti omezujících podmínek (CSP)

CSP lze popsat pomocí množiny proměnných, konečné množiny hodnot, které lze do jednotlivých proměnných přiřadit, a množiny omezení. Množina omezení vyjadřuje sadu podmínek, které rozhodují, jaké hodnoty jsou pro přiřazení v dané situaci přípustné a jaké ne. Řešením (konfigurací) CSP je pak ohodnocení proměnných takovým způsobem, aby byla splněna všechna zadaná omezení.

Některá omezení mohou ovšem hrát větší roli na konečné řešení než jiná, a proto je lze dále dělit na omezení tvrdá (hard) a měkká (soft).

Tvrdá omezení jsou taková omezení, která mají přímý dopad na proveditelnost konečného řešení. Porušením jakéhokoliv tvrdého omezení se stává řešení neproveditelné. Příkladem takového omezení může být omezení na dostupnost učitele, kdy jeden učitel nemůže přednášet ve dvou místnostech ve stejný čas. Často se jedná o fyzické limity reálného světa.

Naopak měkká omezení mají vliv spíše na kvalitu řešení. To znamená, že některá řešení mohou být lepší než jiná bez ohledu na jejich proveditelnost. Mezi měkká omezení může patřit takřka cokoliv. Příkladem mohou být splněné preference zaměstnanců při jejich alokaci na budoucí projekty, kdy více splněných preferencí značí lepší řešení. Naopak lze vytvořit i měkká omezení, která mají negativní dopad. Například při plánování může být překročení uzávěrky projektu nechtěné, přesto naprosto proveditelné řešení.

Žádaným výsledkem je řešení, které neporušuje žádná tvrdá omezení a naopak maximalizuje hodnotu měkkých omezení. Řešení, které je nejlepší, lze nazvat jako optimální. Často ovšem není možné najít optimální řešení v přijatelném čase z důvodu složitosti problému. V takovém případě se lze spokojit i s tzv. suboptimálním řešením, které je dostatečně dobré a zároveň je nalezeno v rozumném časovém horizontu.

Jako CSP lze transformovat různé kombinatorické problémy jako je např. problém batohu nebo problém n dam. Mezi složitější problémy lze uvést problém alokace zdrojů. Výhodou tohoto převodu je přirozenost, se kterou lze tyto problémy modelovat. Modelováním se rozumí specifikace omezení zadaného problému. Přestože se tato činnost může zdát triviální, správný model je klíčovou součástí při hledání řešení. Špatně namodelovaný problém může značně snížit efektivitu hledání řešení.

Paradigma, které řeší různé CSP, se nazývá constraint programming. Tato technika využívá obecného řešitele, který po zadání množiny omezení najde optimální (popř. suboptimální) řešení. Hledání může probíhat systematicky např. metodou backtracking nebo za pomoci různých lokálních prohledávání. Mezi metody lokálního prohledávání patří např. gradientní algoritmus, algoritmus simulovaného ochlazování či dokonce genetické algoritmy.

Tento styl řešení, při kterém je deklarativně specifikován pouze model problému a o zbytek se postará řešitel, je přirozený pro logicky založené jazyky (např. Prolog). Tyto problémy je ovšem možné řešit i v jiných programovacích paradigmatech. [2]

2.1 Problém batohu

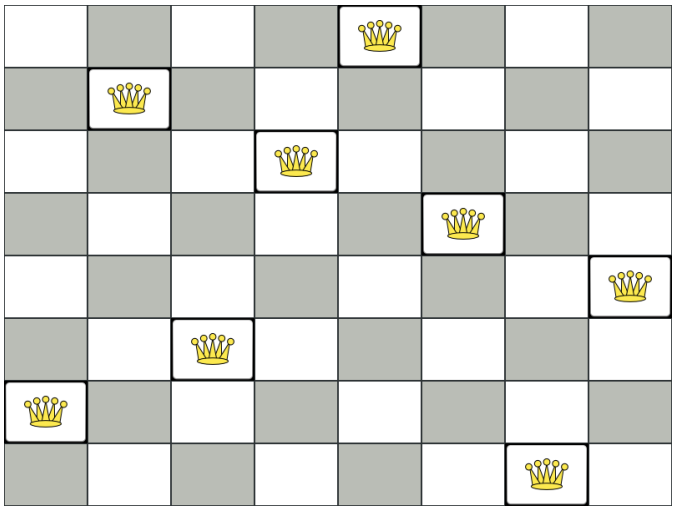
Problém batohu reprezentuje celou řadu praktických problémů, ve kterých je cílem vybrat z množiny prvků takovou podmožinu, která splňuje zadané kritérium. Tyto problémy vznikly na základě analogie stopaře, který si balí různé objekty do batohu takovým způsobem, aby maximalizoval svůj komfort.

Nejčastěji se lze setkat s variantou, kdy každému prvku je přiřazena cena a váha. Požadovaným řešením je pak taková podmožina prvků, která maximalizuje celkovou cenu všech prvků a zároveň není překročena kapacita batohu. Kapacitou batohu se rozumí celková váha všech prvků. [3]

Řešení může být reprezentováno např. permutací nul a jedniček, kde jednička znamená, že prvek je součástí řešení, zatímco nula značí opak. Při konečné množině prvků lze říci, že se jedná o kombinatorický problém, který lze řešit pomocí constraint programming (viz kapitola 2). V takovém případě lze vzít množinu prvků jako množinu vstupních proměnných. Tato varianta problému má 2 omezení a to: tvrdé omezení na celkovou kapacitu a měkké omezení na celkovou cenu. Omezení na kapacitu/cenu lze vyjádřit jako funkci, která sčítá vybrané prvky a vrací jejich celkovou hodnotu, ať už se jedná o celkovou váhu či celkovou cenu. [4] Tento model lze pak předložit obecnému řešiteli, který najde optimální (popř. suboptimální) řešení.

2.2 Problém n dam

Dáma je šachová figurka, která umí útočit na jiné figurky ve vertikálním, horizontálním i diagonálním směru. Problém n dam je takový problém, který má opět několik variant. Jednou z nich je např. konstruktivní varianta, kde je cílem umístit n dam na šachovnici o rozměrech $n \times n$ takovým způsobem, aby žádná dáma nemohla zaútočit na jinou dámu.



Obrázek 2.1 Řešení problému n dam [5]

Tento jednoduchý kombinatorický problém lze opět převést na CSP (viz kapitola 2), kde mezi jediné omezení patří trvdé omezení o neútočení mezi dámami. Vstupními proměnnými je seznam n dam, kde se každé dámě přiřazuje pouze hodnota řádku (1 až n). Lze vypozorovat, že

každá dáma bude mít svůj vlastní sloupec, protože dáma sdílející sloupec s jinou dámou vede okamžitě k porušení podmínky o útočení. Tento model lze opět podat obecnému řešiteli s cílem najít jakékoliv proveditelné řešení. Na obrázku 2.1 je uvedeno řešení pro hodnotu $n = 8$. [2]

2.3 Problém alokace zdrojů

Problém alokace zdrojů se snaží najít nejlepší možné přiřazení vzácných zdrojů (např. lidí či materiálu), přičemž tyto zdroje lze přidělit do různých tříd, regionů, projektů apod. Cílem je najít takové řešení, které maximalizuje profit (resp. minimalizuje náklady) firmy, dosahuje nejlepší kvality produktu atd. Každá instance problému může mít své specifické nároky či omezení, která jasně určují zadaný problém. Takových omezení může být velice mnoho, a proto i výstup pro každý takový problém může být trochu jiný.

Mezi více specifické problémy lze zařadit problém rozvrhování či plánování, kam patří snaha alokovat zdroje k aktivitám v rámci časového harmonogramu. Cílem může být např. splnění preferencí zdrojů, maximalizace vytíženosti zdrojů nebo minimalizace času pro splnění všech aktivit. Celá tato sada problémů, která spadá do operačního výzkumu, se dá velice přirozeně popsat jako CSP a pomocí constraint programming (viz kapitola 2) i vyřešit v rozumném čase. [6, 7, 8]

Kapitola 3
Analýza

V této části jsou zaznamenány všechny informace týkající se budoucí aplikace. Je možné zde nalézt informace o zadavateli, o procesech, které zadavatel využívá, a všechny funkční i nefunkční požadavky na aplikaci.

3.1 Kontext vzniku aplikace

Cílem je vytvořit aplikaci, která umožní automatizovat přiřazení (alokování) zaměstnanců na projekty. Přiřazení zaměstnanců se děje na základě jejich volné kapacity, kompetencí a preferencí na zadané projekty. Hlavním kritériem je uspokojení co nejvíce zaměstnaneckých preferencí. Technologie nejsou specifikovány (výběr je ponechán na vývojáři). Zadavatelem je organizace Applifting s. r. o.

3.2 Současný stav

Tato kapitola popisuje procesy, které se v současné době odehrávají v organizaci Applifting s. r. o. za účelem získávání informací o zaměstnancích a projektech. Dále je zde zmíněno, jakým způsobem probíhá alokování zaměstnanců na projekty.

3.2.1 Proces sbírání informací o zaměstnancích

Jednou za 7 dní se shromáždí informace o každém zaměstnanci. Sbírání informací se provádí manuálním obcházením zaměstnanců a pokládáním otázek. Mezi tyto otázky patří: [9]

- Je zaměstnanec aktuálně volný? (resp. Kdy bude zaměstnanec volný?)
- Jakou volnou kapacitu má zaměstnanec? (kapacita se uvádí v jednotkách FTE = *Full-Time Equivalent*, kde 1 FTE = 160 hodin za měsíc)
- Jaké technologie zaměstnanec ovládá?
- Na jaké úrovni ovládá zaměstnanec každou technologii?
- Jaké projekty by chtěl zaměstnanec dělat v budoucnu?
- Na čem by zaměstnanec rád pracoval před počátkem budoucího projektu?

3.2.2 Proces získávání projektů

Firma Applifting s. r. o. dostane tzv. lead, což je člověk, který by potřeboval zrealizovat nějaký projekt. Získání leadu je buď aktivní (firma se snaží spojit se zákazníkem) nebo pasivní (zákazník se spojí s firmou). Aktivní získávání se nazývá out-bound lead a pasivní pak in-bound lead. Zaměstanec s rolí business developer má za úkol spojit se se zákazníkem a zjistit informace o projektu. Následně kvalifikuje lead na základě 3 kritérií - rozpočtu, smysluplnosti produktu a technologických vlastnostech produktu. Po zjištění rozsahu projektu se ve firmě Applifting s. r. o. nezávazně vytvoří tým, který si vezme projekt na starost, a vytvoří se harmonogram projektu. Sepíše se smlouva i s uvedeným harmonogramem a předá se zákazníkovi. Pokud zákazník smlouvu podepíše, zaměstnanci firmy Applifting s. r. o. jsou závazně přiděleni k projektu. [9]

3.2.3 Proces ukončení projektu

Rozlišují se dva typy projektů. Jedny končí po fixní době, ty druhé jsou dlouhodobé. V dlouhodobých projektech úkoly přibývají a ubývají podle požadavků zákazníka. Ukončení dlouhodobého projektu záleží na domluvě se zákazníkem. [9]

3.2.4 Proces přiřazení zaměstnanců na projekty

Podle kapacity, schopností a preferencí zaměstnanců se vytváří tým, který bude mít na starosti nový projekt. Tento tým se sejde v jedné místnosti a diskutuje rozdělení povinností mezi členy týmu. Přiřazení úkolů se děje po diskuzi s ostaními členy týmu. [9]

3.2.5 Proces při překročení délky projektu

Pokud by mělo dojít k překročení uzávěrky projektu, potom osoba s rolí business developer se domluví se zákazníkem na dalším postupu. Obvykle existují dvě možnosti: [9]

- nový projekt, na kterém jsou zaměstnanci zamluveni, začne později
- tým se rozdělí na dvě části, první část zůstává u starého projektu a druhá část odchází na nový

3.3 Doménový model

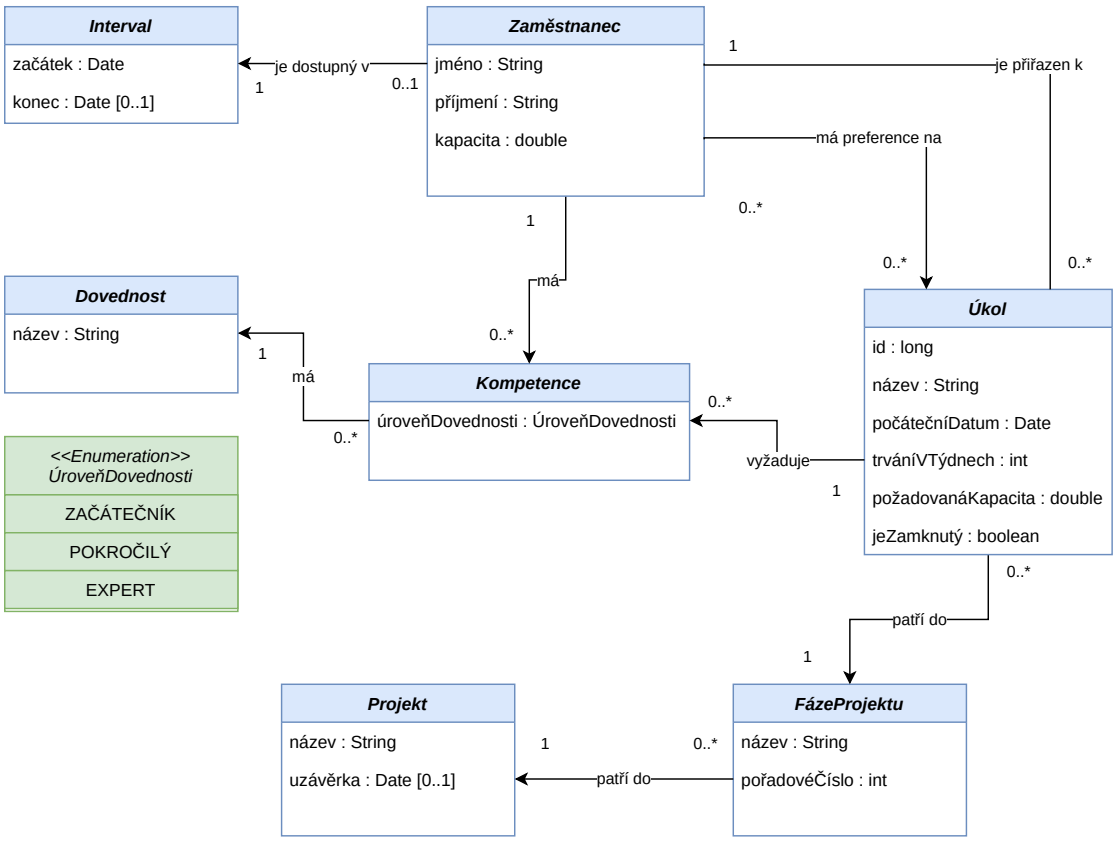
V této části je popsána celá doména řešeného problému. Doména se skládá z jednotlivých entit a vztahů mezi entitami. Mezi klíčové entity patří Zaměstnanec, Projekt a Úkol.

Zaměstnanec je jednoznačně definován pomocí svého jména a příjmení. Dále má kapacitu, seznam kompetencí, seznam preferencí na úkoly a dobu dostupnosti. Kompetence se skládá z typu dovednosnosti a úrovně, které daný zaměstnanec dosahuje. Úroveň má 3 hodnoty – Začátečník, Pokročilý a Expert.

Projekt má svůj název, který ho i jednoznačně definuje. Dále má uvedené datum uzávěrky a seznam fází. Každá fáze má své jméno a pořadové číslo, které určuje kdy se daná fáze může vykonat. Fáze s vyšším pořadovým číslem se může zahájit až po dokončení všech fází s nižším pořadovým číslem.

Úkol je jednoznačně určen svým id. Dále obsahuje datum počátku, dobu trvání (jednotkami jsou týdny), požadovanou kapacitu na vykonání úkolu, požadované kompetence a informaci o tom, zda lze daný úkol plánovat nebo ne. Pokud je úkol uzamčený, nelze jej plánovat. Každý úkol patří do jedné fáze projektu.

Zaměstnance lze přiřazovat k více úkolům, ale každý úkol může mít přiřazeného maximálně jednoho zaměstnance. Celý doménový model je na obr. 3.1. [9]



Obrázek 3.1 Doménový model

3.4 Funkční požadavky

Funkční požadavky jsou požadavky, které souvisejí s funkčností aplikace. Každý funkční požadavek obsahuje obecný popis a prioritu. Priorita nabývá až tří hodnot - vysoká, střední a nízká. Konkrétní hodnota priority byla stanovena po domluvě se zadavatelem.

3.4.1 F01 - Vstupní data (vysoká priorita)

Aplikace musí přijímat vstupní data, která jsou uvedena v procesu sbírání informací (viz kapitola 3.2.1) o zaměstnancích a v procesu získávání projektů (viz kapitola 3.2.2). Konkrétní informace lze vyčíst také z doménového modelu (viz kapitola 3.3). Aplikace by měla umět číst vstupní data ve formátu JSON nebo CSV. [9]

3.4.2 F02 - Plánování (vysoká priorita)

Aplikace musí vytvořit plán ze vstupních dat. Plánem se rozumí alokace zdrojů (zaměstnanců) na úkoly a přiřazení počátečního data k jednotlivým úkolům. Úkoly mají délku trvání v řádu týdnů až měsíců, proto by zrnitost plánu měla být týden.

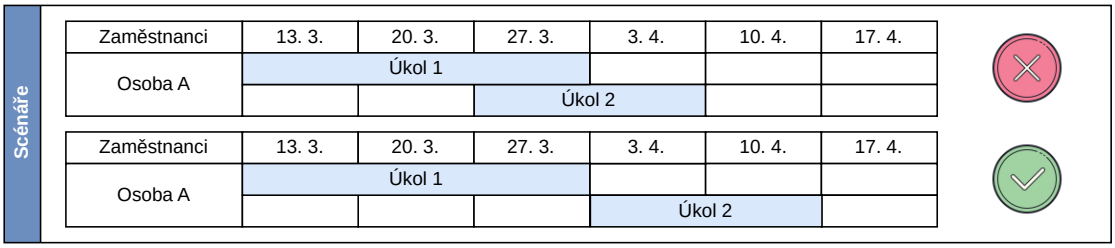
Plán by měl splňovat co nejvíce zadaných omezení. Omezení jsou rozdělena do dvou kategorií, a to měkká a tvrdá. Měkká omezení mohou být porušena za cenu zhoršení kvality plánu, zatímco tvrdá omezení nemohou být porušena nikdy. Každé omezení by mělo mít svoji váhu, která jednoznačně určuje důležitost daného omezení. Díky této váze lze mezi sebou porovnávat různá

omezení. Váha každého omezení by měla být konfigurovatelná a součástí vstupních dat. Zadavatel specifikoval následující omezení, která by měla aplikace podporovat. [9]

3.4.2.1 C01 - Omezení na maximální kapacitu (tvrdé omezení)

Omezení na maximální kapacitu je definováno tak, že zdroj (zaměstnanec) nesmí být v jeden okamžik přiřazen na více úkolů než mu dovoluje jeho vlastní maximální kapacita. [9]

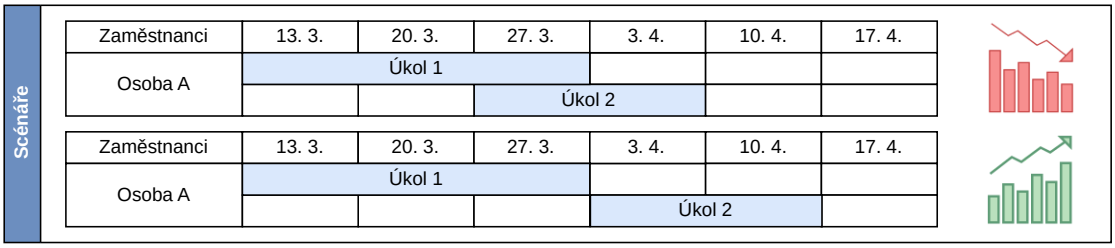
Příklad: Nechť existuje osoba A, která má maximální kapacitu 0,9 FTE. A nechť existuje úkol 1, který požaduje zdroj s kapacitou 0,5 FTE, a úkol 2, který požaduje také zdroj s kapacitou 0,5 FTE. Pokud jsou oba úkoly přiřazeny osobě A a zároveň se doba jejich plnění překrývá, potom je porušeno omezení na maximální kapacitu. Kapacita, která se požaduje v místě překryvu obou úkolů lze jednoduše vypočítat součtem obou požadovaných kapacit (0,5 FTE + 0,5 FTE = 1 FTE). Osoba A má ale definovanou maximální možnou kapacitu pouze 0,9 FTE, a tudíž není schopná plnit nároky obou úkolů současně. Oba úkoly lze přiřadit k osobě A pouze za předpokladu, že se jejich plnění nepřekrývá (viz obr. 3.2).



Obrázek 3.2 Scénáře příkladu na omezení týkající se maximální kapacity [10, 11]

3.4.2.2 C02 - Omezení na základní kapacitu (měkké omezení)

Omezení na základní kapacitu je definováno tak, že zdroj může překročit svou základní kapacitu za cenu snížení kvality plánu. Toto omezení neřeší případy, kdy se překročí maximální kapacita, protože takový případ má na starosti omezení týkající se maximální kapacity (viz kapitola 3.4.2.1). [9]



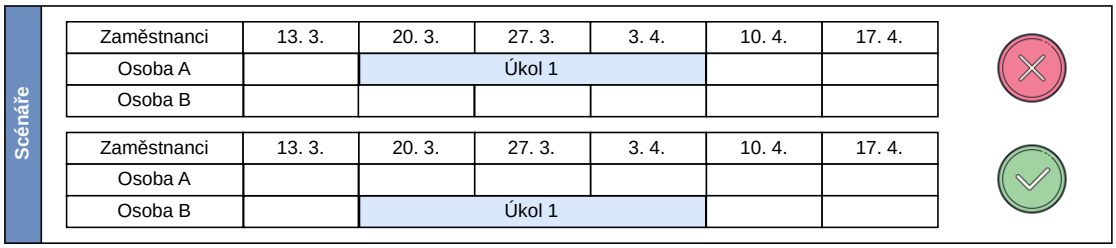
Obrázek 3.3 Scénáře příkladu na omezení týkající se základní kapacity [12, 13]

Příklad: Nechť existuje osoba A, která má základní kapacitu 0,75 FTE a maximální kapacitu 0,9 FTE. A nechť existuje úkol 1, který požaduje zdroj s kapacitou 0,5 FTE, a úkol 2, který požaduje zdroj s kapacitou 0,3 FTE. Pokud jsou oba úkoly přiřazeny osobě A a zároveň se doba jejich plnění překrývá, potom je porušeno omezení na základní kapacitu. Kapacita, která se požaduje v místě překryvu obou úkolů lze jednoduše vypočítat součtem obou požadovaných kapacit (0,5 FTE + 0,3 FTE = 0,8 FTE). Tato situace sice degraduje výsledné řešení, ale stále se jedná o validní plán (viz obr. 3.3).

3.4.2.3 C03 - Omezení na kompetence (tvrdé omezení)

Omezení na kompetence je definováno tak, že zdroj (zaměstnanec) nesmí být přiřazen k úkolu, pokud nemá požadované kompetence. Toto omezení se zabývá pouze typem kompetence, nikoliv úrovní kompetence. [9]

Příklad: Nechť existuje osoba A, která má kompetence pracovat s technologií Node.js a React, a osoba B, která má kompetence pracovat s technologií Java. A nechť existuje úkol 1, který vyžaduje zdroj s kompetencí Java. Pokud je úkol 1 přiřazen k osobě A, dochází k porušení omezení, protože osoba A nesplňuje požadované kompetence úkolu. Naopak Osoba B je plně způsobilá vykonat úkol 1, a proto ji lze alokovat na úkol 1 bez jakékoliv penalizace (viz obr. 3.4).

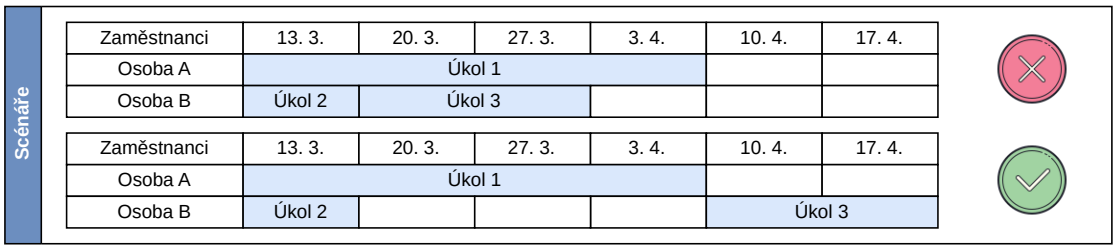


Obrázek 3.4 Scénáře příkladu na omezení týkající se kompetencí [10, 11]

3.4.2.4 C04 - Omezení na fáze projektu (tvrdé omezení)

Omezení na fáze projektu je definováno tak, že úkol nesmí začít dříve než jsou všechny úkoly z předchozí fáze projektu dokončeny. Výjimkou je speciální fáze, u které nezávisí na tom, kdy úkoly proběhnou (v této fázi se nachází především pomocné úkoly, které mohou nastat kdykoliv během projektu). [9]

Příklad: Nechť existuje osoba A a osoba B. Dále nechť existuje úkol 1 a úkol 2, které patří do projektu P1 a do fáze F1. A nechť existuje úkol 3, který patří do projektu P1 a do fáze F2. Úkol 3 musí začít až po době uplynutí Úkolu 1 i Úkolu 2, v opačném případě se jedná o porušení omezení (viz obr. 3.5).

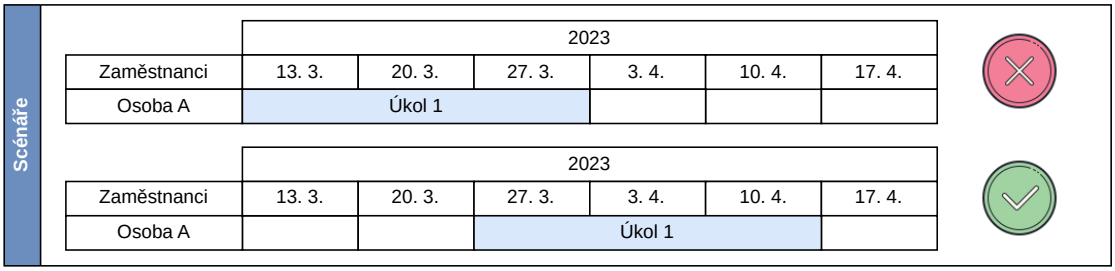


Obrázek 3.5 Scénáře příkladu na omezení týkající se fází projektu [10, 11]

3.4.2.5 C05 - Omezení na dostupnost zdrojů (tvrdé omezení)

Omezení na dostupnost zdrojů je definováno tak, že úkol nesmí být přiřazen ke zdroji v době, kdy není dostupný. Údaje o dostupnosti lze vyčíst ze vstupních dat (viz kapitola 3.4.1). [9]

Příklad: Nechť existuje osoba A, která je dostupná od 22. 3. 2023 až do 22. 3. 2024 a úkol 1. Úkol 1 nelze přiřadit osobě A tak, aby doba plnění úkolu byla mimo interval dostupnosti osoby A (viz obr. 3.6).

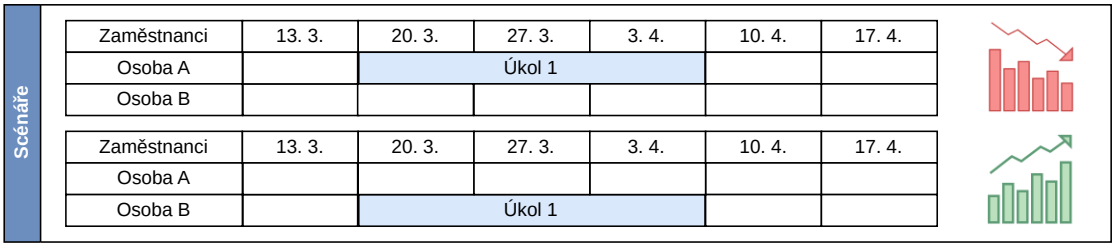


Obrázek 3.6 Scénáře příkladu na omezení týkající se dostupnosti zdrojů [10, 11]

3.4.2.6 C06 - Omezení na úroveň dovednosti (měkké omezení)

Omezení na úroveň dovednosti je definováno tak, že úkol by měl být přiřazen ke zdroji, který má požadovanou úroveň dovedností. Úkol lze přiřadit i zdroji, který má jinou úroveň než takovou, která je vyžadována, ale takový plán je méně kvalitní. [9]

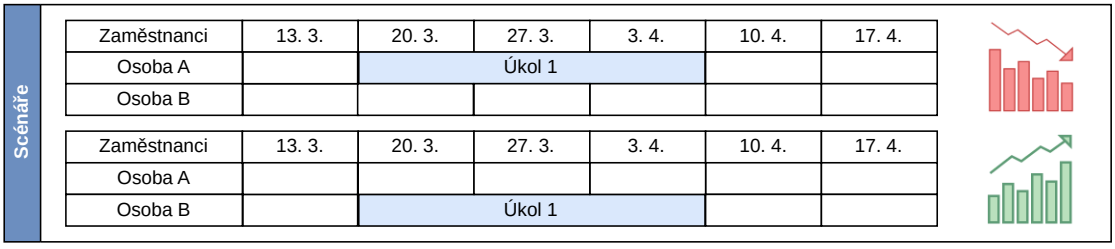
Příklad: Nechť existuje osoba A, která má kompetenci na technologii Java s úrovní dovednosti pokročilý, a osoba B, která má kompetenci na technologii Java s úrovní dovednosti expert. A nechť existuje úkol 1, který vyžaduje osobu s kompetencí Java s úrovní dovednosti expert. Alokace osoby B na úkol 1 je považováno za lepší řešení, než kdyby se na úkol 1 přiřadila osoba A, protože osoba B má požadovanou úroveň dovednosti (viz obr. 3.7).



Obrázek 3.7 Scénáře příkladu na omezení týkající se úrovně dovednosti [12, 13]

3.4.2.7 C07 - Omezení na preference (měkké omezení)

Omezení na preference je definováno tak, že úkol by měl být přiřazen ke zdroji, který má preferenci na daný úkol.



Obrázek 3.8 Scénáře příkladu na omezení týkající se preferencí [12, 13]

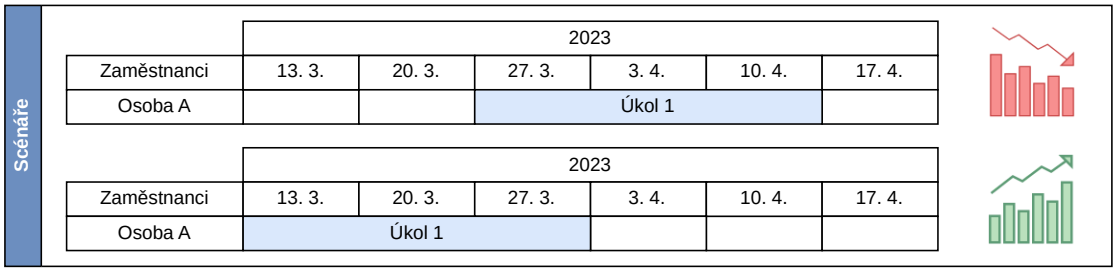
Pokud ani jeden zdroj nemá preferenci na daný úkol, potom nezáleží na tom, komu je úkol přiřazen. Pokud existuje více zdrojů se stejnou preferencí, potom by úkol měl být přiřazen tomu, kdo má preferenci výše na svém seznamu preferencí (preference jsou uspořádány sestupně od nejvíce po nejméně významnou). [9]

Příklad: Nechť existuje nějaký úkol 1. A nechť existuje osoba A, která nemá žádné preference, a osoba B, která má preferenci na úkol 1. Alokace osoby B na úkol 1 je považováno za lepší řešení, než kdyby se na úkol 1 přiřadila osoba A, protože osoba B má splněnou jednu svoji preferenci. Naopak alokace osoby A na úkol 1 neplní nikomu žádnou preferenci (viz obr. 3.8).

3.4.2.8 C08 - Omezení na uzávěrku projektu (měkké omezení)

Omezení na uzávěrku projektu je definováno tak, že všechny úkoly v rámci jednoho projektu by měly skončit před uzávěrkou projektu. Překročení uzávěrky je možné, ale takové řešení by mělo být vysoce penalizováno. [9]

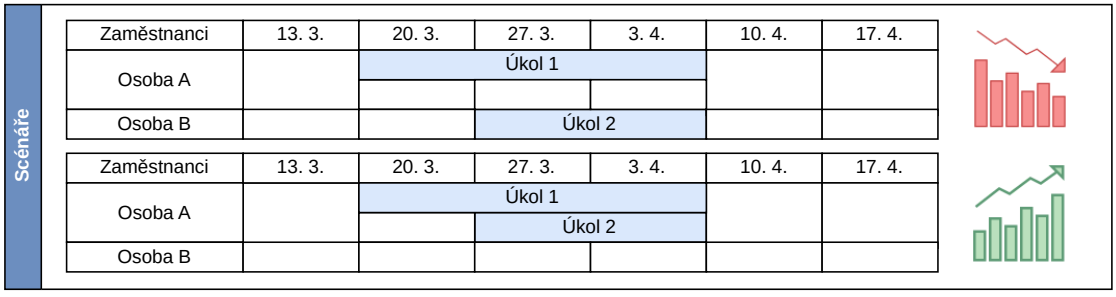
Příklad: Nechť existuje nějaká osoba A a nechť existuje úkol 1, který patří do projektu P1 s uzávěrkou 4. 4. 2023. Pokud úkol 1 skončí až 17. 4. 2023, dojde k porušení omezení, neboť uzávěrka je 4. 4. 2023. Naopak pokud úkol skončí 3. 4. 2023 je vše v pořádku a k žádné penalizaci nedojde (viz obr. 3.9).



Obrázek 3.9 Scénáře příkladu na omezení týkající se uzávěrky projektu [12, 13]

3.4.2.9 C09 - Omezení na volné týdny (měkké omezení)

Omezení na volné týdny je definováno tak, že počet volných týdnů všech zdrojů (zaměstnanců) by měl být co největší. Volný týden znamená, že osoba je zatížena na 0 FTE (resp. celá její kapacita je volná). Díky těmto volným týdnům lze pak získat přehled o zatížení celé organizace (resp. jejích členů). V případě malého zatížení lze přidávat nové projekty, na které je možné přiřadit volné zaměstnance. [9]



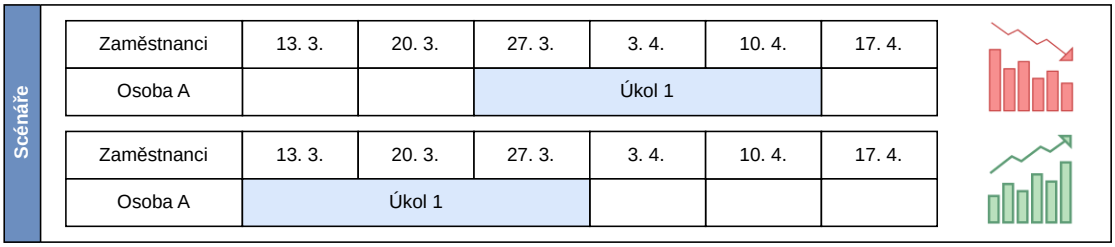
Obrázek 3.10 Scénáře příkladu na omezení týkající se volných týdnů [12, 13]

Příklad: Nechť existuje osoba A, která má kapacitu 1,0 FTE, a osoba B, která má kapacitu také 1,0 FTE. A nechť existuje úkol 1, který vyžaduje kapacitu 0,5 FTE, a Úkol 2, který také vyžaduje kapacitu 0,5 FTE. Pokud úkol 1 a úkol 2 je rozdělen mezi obě osoby, potom celkový počet volných týdnů obou zaměstnanců činí 7. Zatímco pokud oba úkoly jsou přiřazeny pouze např. osobě A tak, aby se překrývala doba jejich plnění, potom celkový počet volných týdnů činí 10. Lze tedy vypozorovat, že druhá varianta je lepší než ta první (viz obr. 3.10).

3.4.2.10 C10 - Omezení na počáteční datum (měkké omezení)

Omezení na počáteční datum je definováno tak, že úkol by neměl být zbytečně odkládán. Lze toho dosáhnout například tím, že za každý týden mezi počátečním datem úkolu a prvním datem plánu je výsledné řešení penalizováno předem definovanou hodnotou. [9]

Příklad: Nechť existuje osoba A a úkol 1 (viz obr. 3.11). Pokud úkol 1 má počáteční datum 27. 3., potom má výsledné řešení hodnotu -2 (za předpokladu, že váha omezení je jedna), protože úkol mohl začít o dva týdny dříve. Na druhou stranu, pokud úkol 1 začne 13. 3., potom má výsledek hodnotu 0 a jedná se tedy o lepší řešení.



Obrázek 3.11 Scénáře příkladu na omezení týkající se počátečního data [12, 13]

3.4.2.11 C11 - Omezení na počet nepřiřazených úkolů (měkké omezení)

Omezení na počet nepřiřazených úkolů je definováno tak, že počet nepřiřazených úkolů by měl být minimální. Může se stát, že do plánu, který má pouze omezený a konečný počet týdnů, se nevejdou všechny zadané úkoly. V takovém případě je požadovaným chováním aplikace minimalizovat tento počet. [9]

3.4.3 F03 - Výstupní data (vysoká priorita)

Aplikace musí vracet výstupní data, která jsou stejná jako vstupní data s tím rozdílem, že úkoly budou přiřazeny k jednotlivým zaměstnancům a budou mít určené datum jejich začátku. Výstupní data musí být ve formátu JSON, CSV nebo jako webová stránka. [9]

3.4.4 F04 - Zobrazení (nízká priorita)

Aplikace by měla zobrazit výstupní data v přehledné tabulce či jako webovou stránku. Z důvodu možné dlouhé doby generování plánu je požadováno zobrazení mezivýsledků. [9]

3.5 Nefunkční požadavky

Nefunkční požadavky jsou požadavky, které specifikují technický stav aplikace. Tyto požadavky silně ovlivňují návrh aplikace.

3.5.1 N01 – Maximální doba výpočtu

Plánování by nemělo trvat déle než 1 hodinu pro vstup obsahující cca 30 zaměstnanců a 60 úkolů. Dále je požadováno, aby parametr maximální doby běhu byl konfigurovatelný. [9]

3.5.2 N02 – Typ aplikace

Je požadován především implementovaný algoritmus, který dokáže vygenerovat plán. S nízkou prioritou lze vytvořit i webovou aplikaci, která bude mít svůj backend a frontend. Backend by měl mít na starosti samotné plánování. Frontend by měl přehledně zobrazit výstup z plánování. [9]

3.5.3 N03 – Rozšiřitelnost

Aplikace by měla být do budoucna rozšiřitelná. Návrh by tedy měl splňovat základní programátorské praktiky (např. DRY, KISS). [9]

3.6 Rešerše existujících aplikací

Na trhu existuje mnoho nástrojů zabývající se přiřazováním zdrojů na projekty. Mnoho z těchto nástrojů je součástí větších softwarových balíčků, které mají za úkol ulehčit práci projektovým manažerům. Tyto balíčky poskytují celou řadu funkcí od vytváření pracovních příležitostí, automatického plánování až po generování rozsáhlých projektových reportů. Níže uvedené nástroje byly vybrány na základě různorodosti funkcí.

3.6.1 monday.com

Monday.com je nástroj určený pro týmy všech velikostí. Aplikace nabízí vytvoření pracovního prostředí pro jednotlivé projekty. Pro každý projekt je možné definovat úkoly, které lze pak přiřadit uživatelům aplikace. Přiřazování se děje manuálním překlikáváním. Kromě přiřazeného uživatele je možné evidovat i časový interval pro splnění úkolu, prioritu úkolu, status úkolu, lokaci a mnoho dalšího. Úkoly lze rozdělit do několika sekcí, kde v jedné sekci mohou být např. rozpracované úkoly a v jiné hotové úkoly. Automatizace podle zadané podmínky usnadňuje přesouvání úkolů mezi jednotlivými sekcemi.

Data lze zobrazit ve formě tabulky, různých grafů, časové osy, mapy či pracovní zátěže. Kromě správy úkolů nástroj usnadňuje komunikaci mezi členy týmu pomocí chatovacích vláken či vytvořením oken pro online schůzky přímo v rámci aplikace. Aplikaci lze integrovat s celou řadou jiných aplikací jako je např. Gmailem, Google kalendářem, Dropboxem atd. Platformy, které monday.com podporuje, jsou webová a mobilní aplikace. Výhody a nevýhody aplikace lze nalézt na obr. 3.12. [14]

Výhody	Nevýhody
přehledné zobrazení dat	zdarma pouze pro projekty s max. 2 členy, ostatní placené
integrace s platformami třetích stran	nepodporuje evidenci preferencí na projekty
automatizace drobných úkonů	nepodporuje automatické alokování uživatelů

Obrázek 3.12 Výhody a nevýhody nástroje monday.com

3.6.2 Parallax

Parallax je komplexní nástroj pro správu členů organizace, financí, projektů a s nimi souvisejících úkolů. U členů lze evidovat dovednosti a pracovní vytížení v určitém časovém horizontu.

Při přiřazování členů na projekty nástroj nabízí takové členy, kteří mají potřebné dovednosti projektové úkoly vykonávat.

Díky předvídání budoucnosti na základě přiřazení jednotlivých členů týmu na projekty lze v reálném čase pozorovat dopad na celkový rozpočet i trvání projektu. Tato předpověď umožňuje projektovým manažerům rychle řešit případné problémy. Sledováním pracovního času členů týmu lze porovnávat odchylku předpovědi se skutečným průběhem projektu a příslušně na ní reagovat. Zobrazení pracovního vytížení členů týmu poskytuje příležitost vyhnout se syndromu vyhoření nebo naopak nízkému pracovnímu nasazení vedoucí k malé produktivitě. I tento nástroj lze integrovat s více aplikacemi třetích stran jako je např. Slack, MS Teams, Jira, Harvest atd. Výhody a nevýhody aplikace lze nalézt na obr. 3.13. [15]

Výhody	Nevýhody
evidence financí	robustní nástroj určený spíše pro větší projekty
předvídání dopadu změn na projekt v reálném čase	placené
	nepodporuje evidenci preferencí na projekty

Obrázek 3.13 Výhody a nevýhody nástroje Parallax

3.6.3 Kantata

Kantata je dalším nástrojem, který zvládá projektové řízení, řízení zdrojů a přiřazování zdrojů na projekty. Navíc poskytuje vhled do financí pomocí generovatelných reportů různých formátů. Funkce předpovídání umožňuje porovnávat teoritický plán s reálnými daty naměřenými pomocí funkce pro vykazování času stráveného prací. Zakládání chatovacích vláken umožňuje udržovat komunikaci na jednom místě. Stejně jako ostatní nástroje podporuje Kantata integraci s jinými systémy jako je např. Jira, Slack, Sage atd. Výhody a nevýhody aplikace lze nalézt na obr. 3.14. [16]

Výhody	Nevýhody
evidence financí	zdarma pouze demo verze, ostatní je placené
funkce pro vykazování času stráveného v práci	nepodporuje evidenci preferencí na projekty
přehledné grafické rozhraní	nepodporuje automatické plánování alokací

Obrázek 3.14 Výhody a nevýhody nástroje Kantata

3.6.4 Paymo

Paymo je nástroj, který kromě klasické správy projektů a zdrojů podporuje automatické alokace na základě požadavků jednotlivých úkolů. Navíc lze mezi jednotlivými úkoly vytvořit závislosti určující koncové a počáteční časy daných úkolů. Nástroj podporuje komunikaci pomocí různých diskuzí nebo komentářů tak, aby byly vždy všechny potřebné informace na jednon místě. Tento nástroj lze integrovat s aplikacemi třetích stran jako je Slack, Google kalendář, Zapier atd. Platforma, kterou lze využít, je desktopová aplikace nebo mobilní aplikace. Výhody a nevýhody aplikace lze nalézt na obr. 3.15. [17]

Výhody	Nevýhody
možnost vytvářet faktury	zdarma pouze pro jednoho uživatele, ostatní je placené
možnost definovat závislosti mezi úkoly	nepodporuje evidenci preferencí na projekty
možnost automatické alokace	

Obrázek 3.15 Výhody a nevýhody nástroje Paymo

3.6.5 Toggl Plan

Velmi jednoduchý nástroj pro alokování členů na projekty. Intuitivní rozhraní umožňuje rychlé osvojení si základních konceptů. Oproti svým konkurentům neposkytuje takové množství funkcí, proto je lepší pro menší projekty. Integrace Toggl Button umožňuje importovat data např. z Trello, Github, Jira a jiných aplikací. Výhody a nevýhody aplikace lze nalézt na obr. 3.16. [18]

Výhody	Nevýhody
intuitivní grafické rozhraní	placené (ale poskytuje čtrnáctidenní trial verzi)
možnost týmové komunikace přímo v aplikaci	nepodporuje mnoho funkcí jako konkurence

Obrázek 3.16 Výhody a nevýhody nástroje Toggl Plan

3.6.6 Forecast

Komplexní nástroj pro projektové řízení, který má intuitivní rozhraní. Oproti konkurenci nezaostává v žádné funkci. Naopak umožňuje svým uživatelům synchronizovat data i s jinými nástroji na projektové řízení. A díky automatické alokaci členů na projekty se řadí mezi jeden z nejlepších nástrojů na tomto seznamu. Nástroj lze integrovat s celou řadou aplikací, jako je např. Asana, Azure Active Directory, Azure DevOps a další. Forecast zpřístuňuje své REST API pro všechny, kdo by ho chtěl využít pro budování nové integrace. Výhody a nevýhody aplikace lze nalézt na obr. 3.17. [19]

Výhody	Nevýhody
intuitivní grafické rozhraní	placené
synchronizace dat s jinými aplikacemi	nepodporuje evidenci preferencí na projekty
automatické plánování	
veřejné REST API	

Obrázek 3.17 Výhody a nevýhody nástroje Forecast

3.6.7 Souhrn základních funkcí

V této části jsou zobrazeny základní funkce, které se vyskytovaly napříč všemi aplikacemi. Funkce, které lze získat integrací s jinými systémy, nejsou součástí výčtu. Důvodem je nepřeberné množství integrací a často nemožnost si zadaný nástroj vyzkoušet zdarma. Z tabulky 3.1 lze vyčíst, že automatickou alokací zdrojů se zabývá menšinová část aplikací. Dále si lze povšimnout, že žádný nástroj nepodporuje nastavení preferencí zdrojů na projekty.

Tabulka 3.1 Souhrn základních funkcí existujících aplikací

	monday.com	Parallax	Kantata	Paymo	Toggl Plan	Forecast
správa projektů	✓	✓	✓	✓	✓	✓
správa úkolů	✓	✓	✓	✓	✓	✓
správa zdrojů	✓	✓	✓	✓	✓	✓
správa financí	✓	✓	✓	✓		✓
manuální alokace	✓	✓	✓	✓	✓	✓
automatická alokace				✓		✓
předpověď pracovního vytížení	✓	✓	✓	✓		✓
předpověď finančních výsledků	✓	✓	✓	✓		✓
nastavení dostupnosti zdrojů	✓	✓	✓	✓	✓	✓
nastavení závislostí mezi úkoly	✓			✓		✓
nastavení preferencí zdrojů na projekty						
generování faktur	✓	✓	✓	✓		✓
generování reportů	✓	✓	✓	✓		✓
podporuje komunikaci členů týmu	✓		✓	✓	✓	✓

Kapitola 4
Rešerše algoritmů

V této části jsou rozebrány konkrétní algoritmy, které lze využít pro hledání optimálního (popř. suboptimálního) řešení CSP. Algoritmy jsou rozděleny do dvou kategorií podle fáze, ve které se nacházejí.

První fáze se nazývá konstruktivní a má na starosti generování počátečního řešení. Dobré počáteční řešení může významně pomoci při následném hledání optimálního (popř. suboptimálního) řešení. Do této fáze patří konstruktivní algoritmy, mezi které patří např. algoritmus First Fit a First Fit Decreasing. Druhá fáze se nazývá optimalizační a snaží se upravit vygenerované počáteční řešení tak, aby se výsledek co nejvíce podobal optimálnímu řešení. Pro tuto fázi lze využít stochastické optimalizační algoritmy (metaheuristiky), mezi které patří např. gradientní algoritmus (Hill Climbing Algorithm), algoritmus simulovaného ochlazování nebo genetické algoritmy. [5] Pro optimalizaci lze využít i algoritmy vyčerpávajícího prohledávání jako je např. backtracking, nicméně z důvodu jejich časté neefektivity nejsou dále v této práci uvažovány. [6]

4.1 Algoritmus First Fit

Cílem algoritmu First Fit je přiřadit každé proměnné hodnotu, která je nejvíce vyhovující. Po přiřazení se už tato hodnota nikdy nezmění, a tudíž může dojít i k situaci, kdy konečným výstupem z algoritmu bude neproveditelné řešení. V konstruktivní fázi se však proveditelnost řešení neuvažuje, protože každé řešení má šanci být později vylepšeno ve fázi optimalizační.

Jako příklad lze uvést problém n dam (viz kapitola 2.1), kde $n = 4$. Cílem je vygenerovat počáteční řešení pomocí algoritmu First Fit. Hodnoty, které lze přiřadit ke každé dámě jsou 1, 2, 3 a 4, protože šachovnice má 4 řádky. První dámě je přiřazeno první možné číslo řádku, tedy číslo 1. Druhé dámě je přiřazeno číslo řádku 3, protože řádek 1 i 2 jsou kolizní s dámou číslo jedna. Třetí dámě je přiřazeno číslo řádku 2, protože řádek 1 je kolizní s dámou číslo jedna. Poslední dámě je přiřazeno číslo 4, protože řádek 1, 2 i 3 jsou kolizní s ostatními dámami. Výsledkem je neproveditelné řešení, protože dáma číslo 1 a dáma číslo 4 na sebe mohou diagonálně zaútočit (viz příloha A.1). [5]

4.2 Algoritmus First Fit Decreasing

Algoritmus First Fit Decreasing narozdíl od algoritmu First Fit nejdříve pracuje s proměnnými, u kterých je těžší přiřadit hodnotu podle určitého kritéria. Přiřazení hodnot pak funguje stejně jako u algoritmu First Fit.

Jako příklad lze uvést zase problém n dam (viz kapitola 2.2), kde $n = 4$. Těžší je přiřadit hodnotu dámě ve druhém či třetím sloupci, proto se algoritmus snaží nejdříve přiřadit hodnotu řádku právě jim. Dáma ve druhém sloupci dostane první možnou hodnotu řádku, a to je číslo 1. Dámě ve třetím sloupci je přiřazen řádek číslo 3, protože řádek 1 i 2 jsou kolizní s dámou ve druhém sloupci. Nakonec zbývají dvě dámy na obou okrajích šachovnice. Dámě v prvním sloupci je přiřazen řádek 4, protože řádek 1, 2 i 3 jsou kolizní. A nakonec dámě ve čtvrtém sloupci je přiřazen řádek číslo 2, protože se jedná o nejlepší možnou hodnotu v závislosti na ostatních dámách (viz příloha A.2). I zde je ovšem výsledné řešení neproveditelné. [5]

4.3 Gradientní algoritmus (Hill Climbing Algorithm)

Gradientní algoritmus, někdy též zvaný jako horolezecký, se využívá pro hledání řešení optimalizačních problémů. Obvykle začíná s náhodným řešením, které lze vytvořit např. některým z konstruktivních algoritmů. Následně vygeneruje všechny sousedy počátečního řešení a ohodnotí je pomocí zadané objektivní funkce (tzv. fitness funkce). Dalším krokem je výběr souseda, který je lepší než současné řešení. Tento postup se opakuje dokud existuje nějaký lepší soused. Pokud takový soused neexistuje, algoritmus je ukončen a vrací dosud nejlepší nalezené řešení. Pseudokód právě popsaného postupu je ve výpisu kódu 4.1. [20]

Výpis kódu 4.1 Pseudokód gradientního algoritmu

```
current solution = initial solution;

repeat
    for all neighbours of current solution do
        obtain random neighbour;
        if cost of neighbour <= cost of current solution then
            current solution = neighbour;
            break;
        end
    end
until cost of current solution <= cost of all neighbours;
```

Výhodou je jednoduchost, která dovoluje implementovat algoritmus velmi rychle. Naopak nevýhodou tohoto algoritmu je, že nedokáže překonat lokální extrém z důvodu předčasné konvergence. Následkem toho může být konečné řešení horší než u algoritmů, které tento nedostatek nemají. Toto omezení lze částečně obejít tím, že se algoritmus spustí několikrát po sobě s jiným náhodně vygenerovaným řešením. [21]

4.4 Algoritmus simulovaného ochlazování (SA)

Tento algoritmus se také řadí do rodiny algoritmů lokálního prohledávání. Na rozdíl od gradientního algoritmu se snaží překonat lokální extrém tím, že dovoluje za určitých podmínek přijmout i horší řešení na úkor současného řešení.

SA využívá při svém výpočtu tzv. fitness funkci *f(x)*, která převádí zadané řešení na číselnou hodnotu reprezentující kvalitu řešení. Pokaždé, když se vygeneruje nové řešení (soused současného řešení), se vypočte rozdíl *delta* kvality současného a nového řešení. Přijmutí či odmítnutí nového řešení pak závisí právě na této *deltě*, která je předložena tzv. příjimací funkci *g(x)*. Tuto funkci lze obvykle definovat jako:

$$g(x) = e^{-delta/T}$$

Parametr T reprezentuje teplotu převzatou z analogie o tavení materiálu, kdy za vysoké teploty materiál přijímá změny bez velkého odporu, ale postupným chlazením je stále těžší a těžší

Výpis kódu 4.2 Pseudokód algoritmu simulovaného ochlazování

```
select initial solution I from S (set of all solutions);
select an initial temperature T > 0;
set temperature change counter t = 0;

repeat
    set repetition counter n = 0;

    repeat
        generate solution J; // neighbour of  I
        calculate change delta = f(J) - f(I);
        if delta < 0 then
            I = J;
        else if random(0, 1) < exp(- delta / T) then
            I = J;
        end

        n = n + 1;
    until n = N(t);

    t = t + 1;
    T = T(t);
until stopping criterion is true;
```

materiál upravit. V určitém bodě už je teplota tak nízká, že s materiálem nelze nic provést, a tudíž se stane koncovým řešením procesu. Stejně to funguje i u SA, kdy nové řešení má větší šanci na přijmutí, pokud je hodnota *delta* nízká a teplota T vysoká. Postupným snižováním teploty se šance na přijmutí snižuje. Na každé teplotní úrovni lze obvykle zkusit několik kandidátů (sousedních řešení) a vybrat z nich vhodného nástupce současného řešení. Pseudokód je ve výpisu kódu 4.2. [21]

4.5 Genetické algoritmy (GA)

Genetické algoritmy vycházejí z Darwinovy teorie o přirozeném výběru a jsou často využívány při řešení těžkých kombinatorických problémů, mezi které patří např. problém obchodního cestujícího, problém rozvrhování apod. [22]

Prvním a klíčovým krokem GA je reprezentace problému (resp. množiny řešení). Způsob, jakým je řešení reprezentováno, může mít později zásadní dopad na efektivitu algoritmu. Každé možné řešení daného problému označuje jedince (neboli chromozom), který žije v populaci (množině všech řešení). Nejsilnější jedinci mají největší šanci být vybráni pro proces reprodukce a tím zplodit potomky, kteří se jim podobají. Existuje mnoho technik, jak vybrat jedince, kteří se budou podílet na vytváření nové generace. Mezi tyto techniky patří např. ruletová či turnajová selekce. Po výběru jedinců přichází na řadu proces křížení, kdy dochází ke kopírování určitých vlastností rodičů na právě vznikajícího potomka. Křížením jedinců může vzniknout potomek, který je silnější než jeho rodiče. Opět existuje mnoho technik, jak takového křížení dosáhnout, jako je např. jednobodové, dvoubodové či k-bodové křížení. Je nutné zaručit různorodost populace, aby nedošlo k předčasné konvergenci algoritmu. Čím různorodější je populace, tím je větší šance nalezení globálního optima. Diverzitu lze zvýšit pomocí mutace, která dokáže upravit chromozom často nepředvídatelným způsobem. Mutace se využívá pouze s nízkou pravděpodobností u všech jedinců, protože vysoká pravděpodobnost by mohla vést k tomu, že potomci by se nemuseli podobat svým rodičům. Nově vzniklý potomek je poté přidán do nové populace jedinců.

Výpis kódu 4.3 Pseudokód genetického algoritmu

```
Y = initial population of n chromosomes;
MAX = max number of iterations;
set iteration counter t = 0;
evaluate fitness value for each chromosome of Y;

while(t < MAX)
    P = select a pair of chromosomes of Y based on fitness value;
    apply crossover operation on P with crossover probability;
    apply mutation on the offspring with mutation probability;
    replace old population with newly generated population;
    t = t + 1;
end while

return best solution;
```

Tento proces se opakuje tak dlouho, dokud není splněna koncová podmínka, kterou může být např. doba trvání, počet populací apod. GA stejně jako dříve zmíněné optimalizační algoritmy nezaručuje, že výsledné řešení je optimální, nicméně využitím správných postupů lze najít alespoň dostatečně dobré řešení. Pseudokód obecného průběhu GA je ve výpisu kódu 4.3. [23, 22]

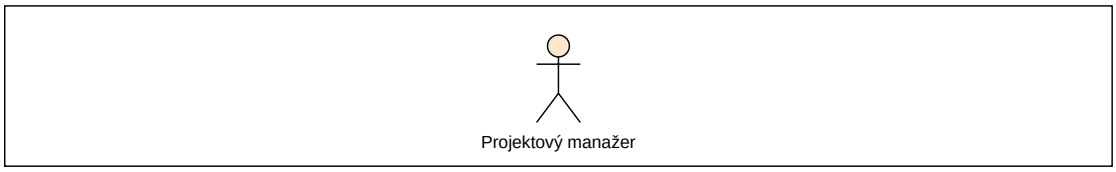
Kapitola 5
Návrh

Návrh se zabývá tím, jak se bude aplikace chovat, a to především z pohledu uživatele. [24] Naopak v této kapitole není možné najít, jak bude systém interně fungovat. Tato kapitola obsahuje identifikaci uživatelů budoucí aplikace, identifikaci jednotlivých případů užití a stručné shrnutí ve formě diagramu případů užití a tabulky pokrytí funkčních požadavků.

5.1 Identifikace aktérů

Aktér reprezentuje roli, kterou uživatel v systémů zastává. Může se stát, že jeden uživatel má přiděleno více rolí. [24]

Systém nevyžaduje žádnou autentizaci ani autorizaci. Jediným aktérem, který bude aplikaci využívat, je role Projektový manažer (viz obr. 5.1). Tato role bude mít přístup ke všem funkcím systému bez jakýchkoliv omezení. Projektový manažer je osoba, která má zkušenosti s generováním vstupních dat ve formátu JSON.



Obrázek 5.1 Aktéři

5.2 Případy užití

Případy užití (nebo také zkráceně UC) jsou akce, které mohou aktéři vykonávat v informačním systému. Každý případ užití by měl reprezentovat právě jednu funkcionalitu systému. Mezi takové funkcionality může patřit např. registrace uživatele, přihlášení uživatele do systému, stáhnutí souboru apod. Případ užití slouží jako černá skříňka, která říká, co je možné se systémem dělat, ale už neříká, jakým způsobem bude systém implementován. [24]

Jednotlivé případy užití vycházejí z funkčních požadavků (viz kapitola 3.4). Obecně platí, že jeden funkční požadavek by měl mít jeden či více případů užití.

5.2.1 UC01 - nahrání vstupních dat

Případ užití začíná tím, že Projektový manažer stiskne tlačítko *Importovat* a vybere možnost *Ze souboru*. Systém zobrazí dialogové okno, které vyzve Projektového manažera k nahrání souboru. Projektový manažer vybere ze svého zařízení soubor, který chce nahrát, a potvrdí svou volbu. Systém nejprve zkontroluje, jestli data v souboru odpovídají vstupním datům definovaným v kapitole 3.4.1. Pokud jsou data validní, systém zobrazí zaměstnance a jejich přiřazené úkoly ve formě tabulky. Dále zobrazí nepřiřazené úkoly včetně informací jako jsou jméno úkolu, projekt a fáze projektu, ve které se úkol nachází, požadovaná kapacita, požadované kompetence a doba trvání úkolu. Pokud data nejsou validní, potom systém zobrazí upozornění i s nápovědou, kde se v souboru nachází chyba.

5.2.2 UC02 - spuštění plánování

Případ užití začíná tím, že Projektový manažer stiskne tlačítko *Spustit plánování* (tlačítko by nemělo být klikatelné, pokud nejsou nahrána vstupní data). Po kliknutí se tlačítko změní na *Zastavit plánování* a systém začne proces řešení problému rozvrhování (viz kapitola 2.3). Systém řeší problém podle zadaných omezení v kapitole 3.4.2 a každých 5 vteřin aktualizuje nejlepší řešení. Pokud dojde ke zlepšení současného řešení, systém automaticky zobrazí Projektovému manažerovi lepší řešení. Systém zároveň ukládá všechny verze do historie pro budoucí procházení. Historie se skládá z čísla verze řešení, kvality řešení (skóre) a celkového počtu splněných preferencí.

5.2.3 UC03 - zastavení plánování

Případ užití začíná tím, že Projektový manažer klikne na tlačítko *Zastavit plánování*. Tlačítko se po stisknutí změní na *Spustit plánování* a systém zastaví proces plánování. Proces plánování může Projektový manažer kdykoliv obnovit kliknutím na tlačítko *Spustit plánování* (viz kapitola 5.2.2).

5.2.4 UC04 - přepínání mezi verzemi plánu

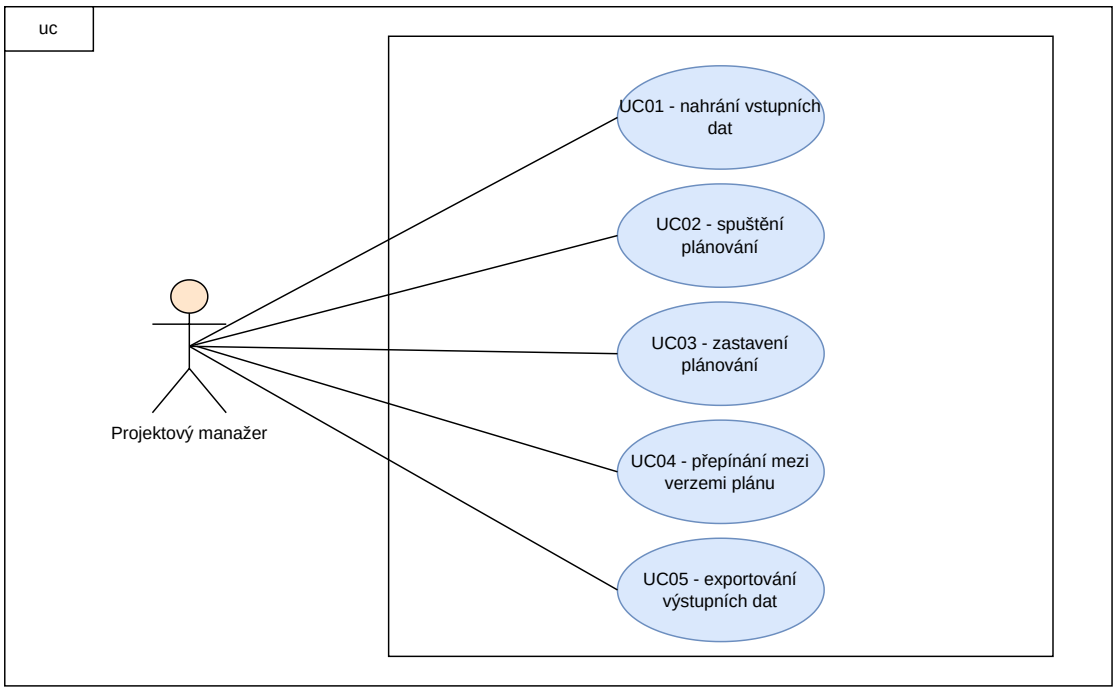
Případ užití začíná tím, že Projektový manažer klikne na jednu z položek v historii, která vznikla v důsledku plánování (viz kapitola 5.2.2). Systém následně překreslí tabulku a nepřiřazené úkoly na základě vybrané položky. Vybraná položka se navíc označí jinou barvou, aby bylo poznat, která verze plánu je vybraná.

5.2.5 UC05 - exportování výstupních dat

Případ užití začíná tím, že Projektový manažer klikne na tlačítko *Exportovat* a vybere možnost *Do souboru*. Systém nabídne Projektovému manažerovi dialogové okno, které požádá o specifikaci jména a umístění souboru. Projektový manažer potom potvrdí svoje volby a soubor se stáhne. Data v souboru se řídí funkčním požadavkem na výstupní data (viz kapitola 3.4.3). Stažený soubor lze poté kdykoliv znovu nahrát (viz kapitola 5.2.1) a zobrazit.

5.3 Diagram případů užití (Use Case Diagram)

Diagram případů užití slouží jako vizuální pomůcka pro rychlé shrnutí aktérů a jejich pravomocí (resp. případů užití). Tento diagram je obvykle využíván při návrhu informačních systémů. [24] Diagram lze nalézt na obr. 5.2.



Obrázek 5.2 Diagram případů užití

5.4 Pokrytí funkčních požadavků

V této části lze nalézt pokrytí funkčních požadavků (viz kapitola 3.4), které je shrnuto v tabulce 5.1. Z této tabulky jednoznačně vyplývá splnění či nesplnění funkčních požadavků a zároveň je možné vypozorovat, že neexistuje zbytečný případ užití, který by nesplňoval žádný funkční požadavek.

Tabulka 5.1 Pokrytí funkčních požadavků

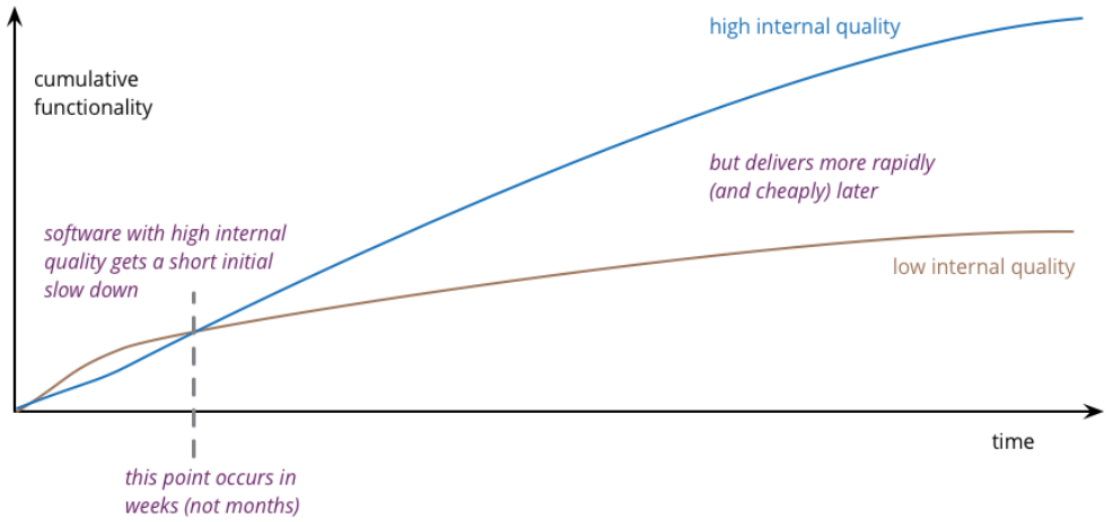
	F01	F02	F03	F04
UC01	✓			
UC02		✓		
UC03		✓		
UC04				✓
UC05			✓	

Kapitola 6

Architektura

Softwarová architektura je považována za stále poměrně nový termín a neexistuje přesná definice, která by ho dokázala přesně popsat. Nicméně obecně přijatá myšlenka je, že architektura se zabývá organizací a interakcí různých softwarových komponent. Tato organizace má velký dopad na vývoj aplikace z dlouhodobého hlediska. Špatně navržený systém má malou interní kvalitu a s postupem času a přibývajícími funkcionalitami se vývoj zpomaluje. Výhodou takového přístupu je relativně rychlý počáteční progres, neboť není potřeba dlouho analyzovat budoucí potřeby aplikace. Naopak dobře navržený systém má sice relativně pomalý rozjezd, ale z dlouhodobého hlediska je přidávání nových funkcionalit jednodušší a rychlejší (viz obr. 6.1). [25, 26]

Tato práce využívá síťovou architekturu klient-server, která umožňuje rozdělit funkcionality mezi klienta a server. Výhodou takového přístupu je, že každá část má jinou zodpovědnost a díky tomu je i možné rozdělit vývoj na dvě části. Každou část lze navíc optimalizovat nezávisle na té druhé. [27] V této kapitole je rozebrána zvlášť klientská a serverová část.



Obrázek 6.1 Dopad špatné a dobré architektury na přidávání funkcionalit aplikace v čase [26]

6.1 Server

Serverová část aplikace slouží pro samotný proces plánování, ukládání výsledných rozvrhů a poskytování těchto dat klientské části.

Samotný server využívá dvouvrstvou architekturu, která je častá u API serverů. Tyto servery přijímají, zpracovávají a následně vracejí surová data. Obvykle se předpokládá, že server komunikuje s jinou aplikací, která tyto data uživateli zobrazí, a proto není nutné mít na serveru prezentační vrstvu. [28]

6.1.1 Výběr technologií

K implementaci je vybrán programovací jazyk Java. Jedná se o objektový jazyk, který vznikl v roce 1995 ve firmě Sun Microsystems. Výhodou tohoto jazyka je především jeho jednoduchost, bezpečnost, platformová nezávislost a široká komunita vývojářů využívajích tuto technologii. [29]

Java je v dnešním světě stále hojně využívaným jazykem, o čemž svědčí i její umístění na 3. místě v žebříčku poptávaných jazyků. Nejvyužívanějšími oblastmi jsou např. webové aplikace, Android aplikace či vestavěné systémy. [30] Díky široké komunitě vývojářů lze nalézt i mnoho knihoven a frameworků řešících nejrůznější problémy. Mezi frameworky, které jsou vybrány pro tuto bakalářskou práci, patří Spring Boot a Optaplanner. Pro sestavení serveru a správu závislostí je zvolen automatizační nástroj Gradle.

6.1.1.1 Spring Boot

Spring Boot je populární Java framework, který výrazně usnadňuje vývoj a monitorování stavu webových aplikací tím, že odstraňuje potřebu řešit mnoho věcí manuálně jako např. tvorbu REST API. [31]

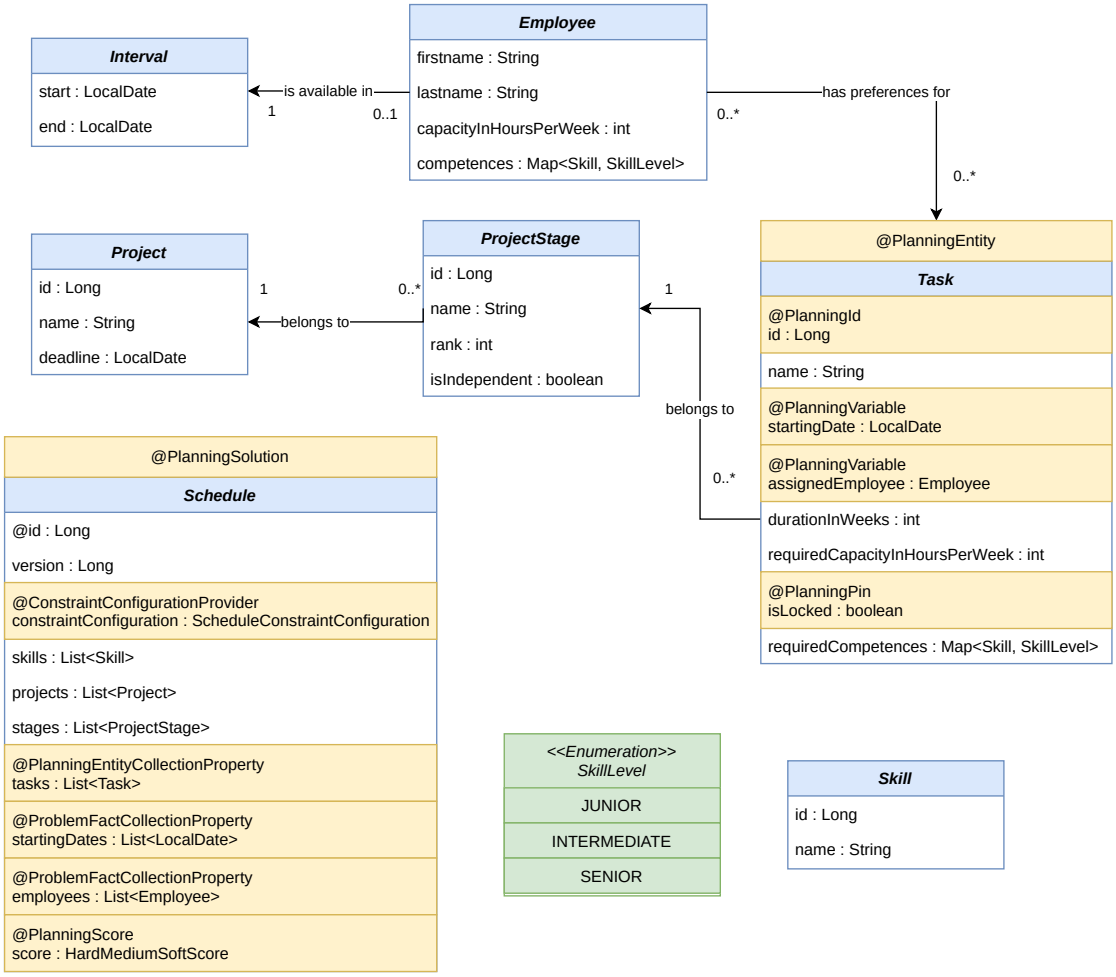
6.1.1.2 Optaplanner

Optaplanner je engine, který dokáže řešit různé optimalizační úlohy pomocí metody constraint programming (viz kapitola 2). Tento nástroj umožňuje definovat problém deklarativním způsobem (především pomocí anotací v kódu) a následně ho vyřešit pomocí svého řešitele. Řešitele je možné použít s výchozím nastavením nebo s vlastní konfigurací, které lze určit např. algoritmus, který se při řešení použije.

Efektivita i kvalita řešení silně závisí na správné reprezentaci problému. Optaplanner umožňuje definovat každý problém pomocí plánovacích faktů, plánovacích entit a plánovacího řešení.

Plánovací fakt (planning fact) je taková entita, která se v průběhu plánování nemění. Naopak plánovací entita (planning entity) je taková entita, která má definované plánovací proměnné (planning variable), které se během výpočtu mění. Plánovací řešení je pak entita, která sdružuje všechny plánovací fakta, plánovací entity, plánovací skóre (planning score) a konfiguraci omezení (constraint configuration), která definuje důležitost jednotlivých omezení. [5]

V této práci mezi plánovací fakta patří konečné množiny datumů a zaměstnanců. Naopak mezi plánovací entity patří konečná množina úkolů. Úkoly mají plánovací proměnné ve formě atributů a patří mezi ně počáteční datum (*startingDate*) a přiřazený zaměstnanec (*assignedEmploeee*). Některé úkoly mohou být navíc vynechány z procesu plánování (např. z důvodu fixního přiřazení ze strany zaměstnavatele) pomocí plánovacího pinu (planning pin), který dokáže uzamknout entitu. Ostatní atributy vycházejí z doménového modelu (viz kapitola 3.3). Analytický model je na obr. 6.2.

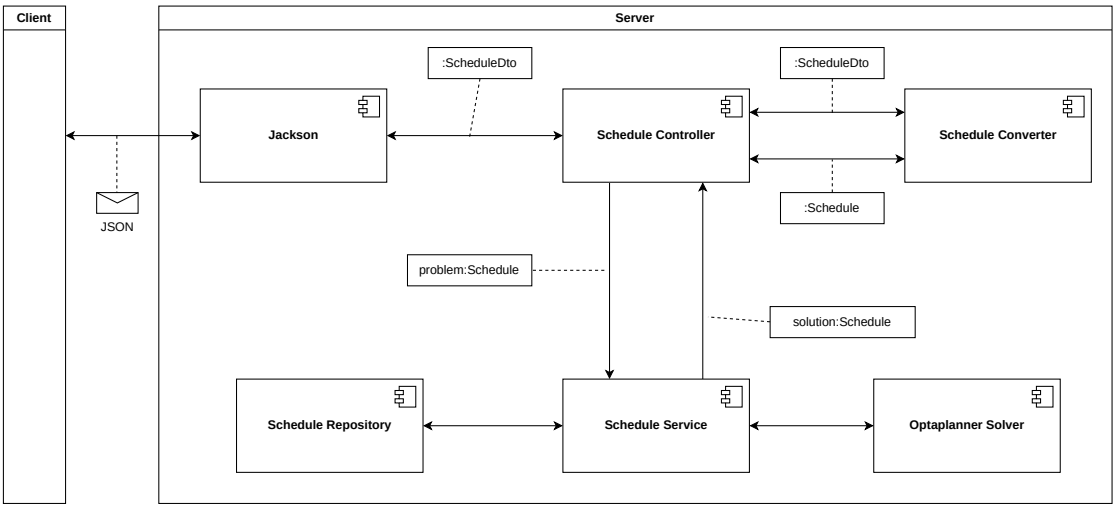


Obrázek 6.2 Analytický model

6.1.2 Architektura serveru

Cílem architektury serveru je nízká provázanost a vysoká soudržnost komponent. Účelem těchto programovacích praktit je rozdělit kód na menší části, které jsou lépe udržitelné a rozšiřitelné. V této kapitole je popsáno rozdělení komponent a jejich účel. Celkový pohled na architekturu serveru je na obr. 6.3.

Jackson je nástroj, který je součástí balíčku Spring Boot, a slouží ke konvertování vstupních dat, které jsou ve formátu JSON, na objekt typu ScheduleDto. Tento objekt je dále pomocí komponenty Schedule Controller a Schedule Converter konvertován na objekt typu Schedule, který reprezentuje zadaný problém. Proces řešení problému je ponechán na obecném řešiteli, který je součástí balíčku Optaplanner. Tento řešitel navíc umožňuje získávat mezivýsledky plánování, které lze uchovávat. Odpovědnost za ukládání mezivýsledků má komponenta Schedule Repository, zatímco komponenta Schedule Service slouží jako prostředník mezi plánovačem, úložištěm a kontrolerem. Výhodou tohoto přístupu je rozvolnění vazeb mezi komponentami.



Obrázek 6.3 Architektura serveru

6.1.3 REST API

Server bude s klientem komunikovat za pomoci REST API a HTTP. Jednolivé koncové body REST API jsou popsány v tabulce 6.1.

Tabulka 6.1 Popsané koncové body

Metoda HTTP	Koncový bod	Popis
POST	/schedule	spustí plánování rozvrhu
PUT	/schedule/:id	zastaví plánování rozvrhu
GET	/schedule/:id/previews/:previewId	vrátí konkrétní náhled rozvrhu
GET	/schedule/:id/previews/last	vrátí náhled poslední verze rozvrhu
GET	/schedule/:id/versions/:versionId	vrátí konkrétní verzi rozvrhu
GET	/schedule/:id/versions/last	vrátí poslední verzi rozvrhu
GET	/schedule/:id/versions	vrátí všechny verze rozvrhu

6.2 Klient

Klientská část aplikace slouží pro přeposílání dat serveru a k zobrazení výsledků ve formě webové stránky.

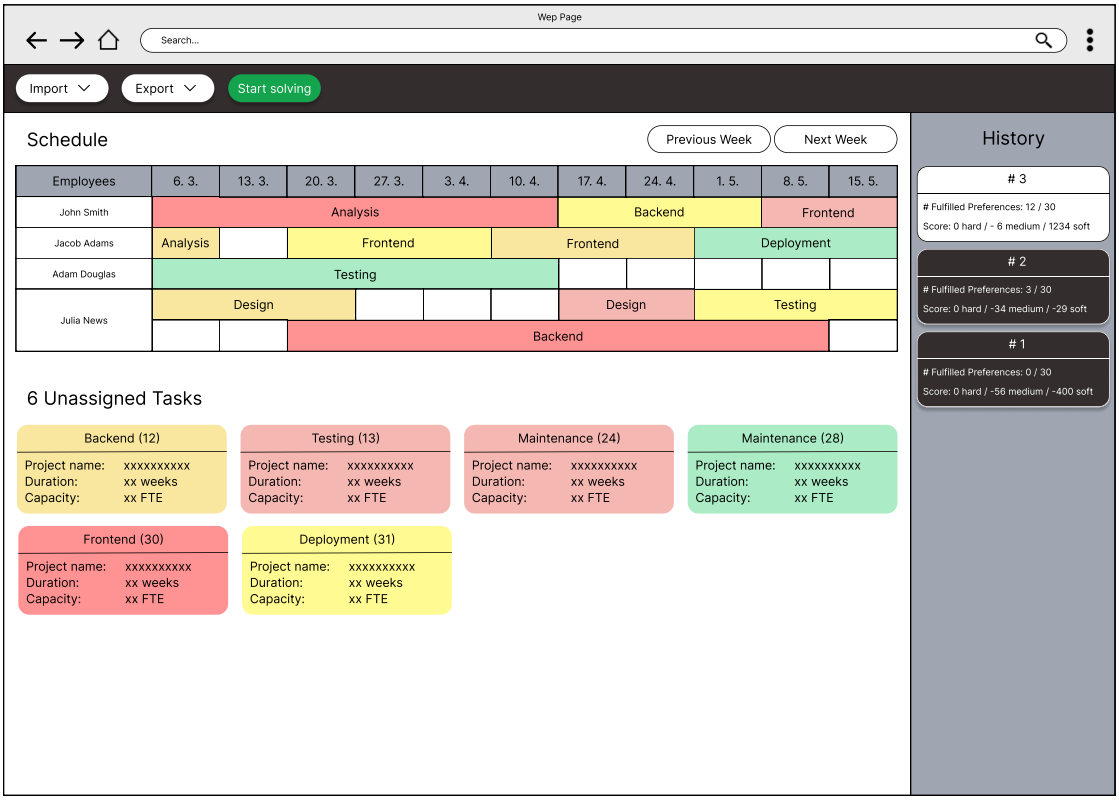
6.2.1 Výběr technologií

Pro implementaci klienta je zvolen programovací jazyk TypeScript a knihovna React, která byla vytvořena společností Meta (resp. Facebook) v roce 2013. Tato technologie patří mezi jednu z nejoblíbenějších frontendových frameworků/knihoven současnosti. Klíčovým konceptem Reactu jsou komponenty, které postupným skládáním mohou vytvořit komplexní UI. [32, 33]

Pro stylování je zvolen CSS framework Bootstrap, který umožňuje využít předdefinované CSS třídy. Pro sestavení klienta a správu závislostí je zvolen správce balíčků npm (výchozí správce balíčků).

6.2.2 Návrh uživatelského rozhraní

Na obr. 6.4 lze nalézt návrh uživatelského rozhraní. Návrh a uspořádání komponent je v souladu s případy užití (viz kapitola 5.2).



Obrázek 6.4 Návrh uživatelského rozhraní

6.3 Distribuce prototypu

Pro distribuci prototypu je zvolena technologie Docker. Docker je nástroj, který dokáže vytvořit tzv. image, který obaluje všechen zdrojový kód společně se všemi jeho závislostmi. Jedná se samostatnou jednotku, která má k dispozici všechno potřebné pro spuštění aplikace. Z těchto jednotlivých balíčků lze následně vytvořit tzv. docker container, který lze spustit nezávisle na okolní infrastruktuře. [34]

Kapitola 7

Implementace

V této kapitole jsou vyzdviženy všechny důležité části systému z pohledu samotné implementace aplikace. Součástí popisu jsou i útržky zdrojových kódů. Implementace využívá technologie definované v kapitole 6.

7.1 Doména

Doména je implementována podle kapitoly 6.1.1.2. Zvláště důležitá je definice plánovací entity Task, která má dvě plánovací proměnné, a to *startingDate* a *assignedEmployee* (viz výpis kódu 7.1).

Výpis kódu 7.1 Implementace třídy Task

```
@PlanningEntity
public class Task  {

    @PlanningVariable(nullable = true)
    private LocalDate startingDate;

    @PlanningVariable(nullable = true)
    private Employee assignedEmployee;

    ...
}
```

Obě plánovací proměnné mají anotaci *@PlanningVariable*, kde nastavení parametru nullable na true označuje, že tyto atributy mohou nabývat i hodnoty null (výchozí nastavení je false). Tento parametr je důležitý, protože může nastat i situace, kdy všechny úkoly nelze alokovat do omezeného plánu (jedná se o tzv. overconstrained planning). V takovém případě je žádaným výsledkem řešení, které má přiřazeno co nejvíce úkolů. Toho lze docílit definováním atributu *score* (v entitě Schedule) jako datový typ HardMediumSoftScore (viz výpis kódu 7.2). Skóre je pak definováno třemi částmi, kde část Hard počítá hodnotu tvrdých omezení, část Soft počítá hodnotu měkkých omezení a část Medium počítá počet nepřiřazených úkolů. Entita Schedule také přidává ke kolekcím *startingDateList* a *employeeList* anotaci *@ValueRangeProvider*, která určuje, že tyto kolekce jsou zdrojem dat pro přiřazování hodnot do plánovacích proměnných třídy Task. [5]

Výpis kódu 7.2 Implementace třídy Schedule

```
@PlanningSolution
public class Schedule {

    @ValueRangeProvider
    @ProblemFactCollectionProperty
    private List<LocalDate> startingDateList;

    @ValueRangeProvider
    @ProblemFactCollectionProperty
    private List<Employee> employeeList;

    @PlanningScore
    private HardMediumSoftScore score;

    ...
}
```

7.2 Dočasné a persistentní úložiště

V době běhu aplikace je potřeba uchovávat jednotlivé verze rozvrhu, proto je úložiště na serveru implementováno jako kolekce, která existuje pouze v době runtime aplikace. Úložiště je definováno pomocí třídy ScheduleInMemoryRepository, která má anotaci *@Repository* sloužící pro Spring Boot jako označení, že se jedná o úložiště, a zároveň pro vytvoření instance této třídy, kterou lze později vložit do jiné třídy jako závislost (viz výpis kódu 7.3).

Výpis kódu 7.3 Implementace třídy ScheduleInMemoryRepository

```
@Repository(value = "scheduleInMemoryRepository")
public class ScheduleInMemoryRepository {
    private List<Schedule> db = new LinkedList<>();
    ...
}
```

Klientské části je nicméně umožněno exportovat výsledné řešení do souboru, který lze označit jako persistentní médium, které udrží data i po ukončení aplikace. Tento soubor lze při znovuspuštění aplikace opět načíst jako vstupní data a pokračovat v plánování.

7.3 Optaplanner řešitel a jeho konfigurace

7.3.1 Konfigurace

Konfigurace řešitele se provádí v XML souboru, který je nazván *ScheduleSolverConfig.xml* (viz výpis kódu 7.4). V tomto souboru lze definovat konstrukční i optimalizační heuristiky, které budou využity při plánování. Algoritmy jsou implementovány knihovnou Optaplanner, a proto je možné v tomto souboru pouze identifikovat, které se mají použít. Pro konstrukci je zvolen algoritmus First Fit (viz kapitola 4.1), zatímco pro optimalizaci je vybrán algoritmus simulovaného ochlazování (viz kapitola 4.4). Simulované ochlazování lze konfigurovat pomocí dvou vlastností, a to *simulatedAnnealingStartingTemperature*, která definuje maximální možný rozdíl hodnot způsobený jedním pohybem mezi sousedními řešeními, a *acceptedCountLimit*, který by měl být co nejmenší. Konkrétní hodnoty jsou nastaveny na výchozí, nicméně je lze kdykoliv upravit.

Výpis kódu 7.4 Konfigurace řešitele

```
<?xml version="1.0" encoding="UTF-8"?>
<solver xmlns="https://www.optaplanner.org/xsd/solver"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://www.optaplanner.org/xsd/solver
        https://www.optaplanner.org/xsd/solver/solver.xsd">

    ...

    <constructionHeuristic>
        <constructionHeuristicType>
            FIRST_FIT
        </constructionHeuristicType>
    </constructionHeuristic>

    <localSearch>
        <acceptor>
            <simulatedAnnealingStartingTemperature>
                2hard/5medium/100soft
            </simulatedAnnealingStartingTemperature>
        </acceptor>
        <forager>
            <acceptedCountLimit>4</acceptedCountLimit>
        </forager>
    </localSearch>
</solver>
```

7.3.2 Spuštění plánování

Spuštění plánování probíhá ve třídě ScheduleService, která slouží jako řídící jednotka mezi úložištěm, řešitelem a kontrolerem, který přijímá vstupní data (viz výpis kódu 7.5). Voláním metody *solveLive()* s parametry problému a maximální délkou výpočtu se nejdříve vytvoří instance třídy SolverManager (třída definovaná knihovnou Optaplanner), na které se následně volá metoda *solveAndListen()* se třemi parametry. První parametr definuje id problému, druhý definuje metodu, která se zavolá jednou pro vyhledání samotného problému, a třetí definuje metodu, která se zavolá po každém nalezeném lepším řešení. [5] Implementovaná metoda *save()* slouží pro uložení nalezeného řešení do úložiště. Díky tomuto postupu lze zobrazovat průběžné mezivýsledky, neboť jsou neustále dostupné z úložiště.

Výpis kódu 7.5 Spuštění plánování pomocí třídy ScheduleService

```
@Service
public class ScheduleService {
    ...

    public void solveLive(Schedule schedule,
                          Long terminationTimeInMinutes) {
        ...
        solverManager = builder
            .withTermination(terminationTimeInMinutes).build();
        solverManager.solveAndListen(
            schedule.getId(), this::findById, this::save
        );
    }
}
```

7.3.3 Zastavení plánování

Zastavení plánování může proběhnout dvěma způsoby. Buď se řešitel ukončí sám po určité době, která je definována v konfiguraci, nebo je řešitel ukončen předčasně. Předčasné ukončení je zajištěno pomocí volání metody *stopSolving()* s jedním argumentem reprezentujícím id problému (viz výpis kódu 7.6).

Výpis kódu 7.6 Zastavení plánování pomocí třídy ScheduleService

```
@Service
public class ScheduleService {
    ...

    public void stopSolving(Long scheduleId) {
        ...
        solverManager.terminateEarly(scheduleId);
    }
}
```

7.3.4 Omezení

Jednotivá omezení jsou definována ve třídě ScheduleConstraintProvider, která implementuje rozhraní ConstraintProvider dodané knihovnou Optaplanner. Tato třída má na starosti implementaci všech omezení bez použití externích závislostí z důvodu efektivity výpočtu. Každé omezení si přepočítává svoji hodnotu (penalizaci či odměnu), která následně ovlivňuje finální skóre daného řešení. Pro implementaci je zvolena metoda inkrementálního výpočtu skóre, kdy jsou přepočítávány pouze položky, které se změnily, zatímco ostatní využívají dřívější hodnotu.

Výpis kódu 7.7 Implementace omezení na preference ve třídě ScheduleConstraintProvider

```
public class ScheduleConstraintProvider
    implements ConstraintProvider {
...
    protected Constraint preferenceConflict(
        ConstraintFactory constraintFactory) {

        return constraintFactory
            .forEach(Task.class).ifExists(
                Employee.class,
                Joiners.filtering((task, employee) ->
                   task.getAssignedEmployee().equals(employee) &&
                   employee.getPreferredTasks().contains(task)
                )
            ).join(ScheduleConstraintConfiguration.class)
            .reward(HardMediumSoftScore.ONE_SOFT,
                    (task, config) -> {
                        int index = task.getAssignedEmployee()
                            .getPreferredTasks().indexOf(task);
                        int pref = config.getPreferenceConflict();
                        int reward =
                            (int) (pref * (1.0 / (index + 1)));
                        return Math.max(reward, 1);
                    }
            ).asConstraint("Preference␣conflict");
    }
}
```

Např. omezení na preference je implementováno ve výpisu kódu 7.7. Metoda *preferenceConflict()* vrací instanci třídy Constraint, která dokáže napočítat hodnotu odměny za každou splněnou preferenci. Výpočet probíhá tak, že se vytvoří tzv. stream ze všech instancí třídy Task a vyfiltrují se všechny úkoly, které splňují preferenci některému ze zaměstanců. Každý takový úkol, který zůstal ve streamu je odměněn hodnotou jednoho měkkého bodu vynásobenou hodnotou váhy omezení, která je definována ve třídě ScheduleConstraintConfiguration. Tato třída definuje váhy všech omezení a zároveň je umožňuje dynamicky konfigurovat (viz výpis kódu 7.8).

Výpis kódu 7.8 Implementace váhy omezení na preference ve třídě ScheduleConstraintConfiguration

```
public class ScheduleConstraintConfiguration {

    ...

    @ConstraintWeight("Preference conflict")
    private HardMediumSoftScore preferenceConflict
        = HardMediumSoftScore.ofSoft(40);

}
```

7.4 Server API

Framework Spring Boot umožňuje definovat jednotlivé koncové body REST API jako metody s příslušnou anotací. Např. metoda *solve()* sloužící pro spuštění plánování je implementována ve výpisu kódu 7.9. Tato metoda přijímá instanci problému, který má server vyřešit, deleguje proces řešení na ScheduleService a vrací náhled přijatého problému.

Výpis kódu 7.9 Implementace koncových bodů REST API pomocí Spring Boot

```
@RestController
@RequestMapping("/schedule")
public class ScheduleController {

    ...

    @PostMapping
    public PreviewDto solve(@RequestBody ScheduleDto scheduleDto) {
        Schedule schedule = scheduleConverter.fromDto(scheduleDto);
        Long terminationTime = (long) scheduleDto
            .getConfigurationParameters()
            .getTerminationTimeInMinutes();

        scheduleService.solveLive(schedule, terminationTime);

        PreviewDto previewDto = new PreviewDto(
                schedule.getId(),
                schedule.getVersion(),
                scheduleService.countPreferences(schedule),
                scheduleConverter.getScoreDto(schedule)
        );

        return previewDto;
    }
}
```

7.5 Klient API

Klient využívá knihovnu Axios pro získávání a odesílání dat. Např. voláním metody *solve()* se vytvoří požadavek na spuštění plánování, který se následně posílá na server (viz výpis kódu 7.10). Metoda vrací objekt typu Promise, který v případě úspěšného dotazu obaluje návratovou hodnotu serveru. Dalším příkladem je metoda *getLastScheduleVersion()*, která se dotazuje na nejlepší (poslední) verzi právě řešeného problému. Návratovou hodnotou je opět objekt typu Promise.

Výpis kódu 7.10 Implementace vytváření požadavků pomocí knihovny Axios

```
const url = "http://localhost:8080/schedule";

export function solve(schedule : ScheduleDto) {
    return axios.post<PreviewDto>(url, schedule);
}

export function getLastScheduleVersion(id : number) {
    const path = url + "/" + id + "/versions/last";
    return axios.get<ScheduleDto>(path);
}
```

7.6 Klientský datový management

React definuje tzv. hooks, které umožňují uživateli využívat vlastnosti Reactu bez psaní nového kódu. V této bakalářské práci je využíván *useContext Hook*, který slouží pro globální správu dat. Jinými slovy je možné organizovat data na jednom místě a potom je zpřístupnit ostatním komponentám. Např. ve výpisu kódu 7.11 je komponenta App obalena pomocí komponenty ScheduleProvider, která má na starosti správu objektu typu ScheduleDto reprezentující zobrazované řešení.

Výpis kódu 7.11 Zpřístupnění dat o rozvrhu

```
const root = ReactDOM.createRoot(
    document.getElementById('root') as HTMLElement
);

root.render(
    <React.StrictMode>
        <ScheduleProvider>
            <App />
        </ScheduleProvider>
    </React.StrictMode>
);
```

Klient se po spuštění plánování dotazuje serveru na nejnovější náhled řešení každých 5 vteřin, a pokud má vrácený náhled lepší skóre než je skóre současného řešení, potom je náhled uložen do historie. Náhled slouží pouze jako meta informace o řešení a zobrazuje se jako položka v historii. Klient si udržuje informace pouze o jednom celém řešení (současné vybrané položce v historii) a seznamu náhledů, kde každý náhled lze využít jako identifikátor pro získání celých řešení. Důvodem tohoto opatření je úspora potřebných dat na straně klienta. Celé uživatelské rozhraní klienta je pak zobrazeno na obr. B.1 v příloze.

7.7 Distribuce aplikace pomocí nástroje Docker

U distribuce se předpokládá, že zákazník má na svém zařízení nainstalován nástroj Docker (resp. docker compose), který je schopen najednou vytvořit tzv. docker images a organizovat jejich spuštění.

Prvním krokem je konfigurace proměnných prostředí. Tyto proměnné jsou definovány v souboru *project-resource-allocation/.env* (viz výpis kódu 7.12).

Výpis kódu 7.12 Ukázka kódu v souboru *project-resource-allocation/.env*

```
SERVER_PORT=8081
CLIENT_PORT=3001
```

Druhým krokem je vytvoření souboru *Dockerfile* v domovském adresáři serveru i klienta. Tento soubor slouží pro vytváření tzv. docker image.

Dockerfile serveru definuje prostředí programovacího jazyka Java 17 a globální proměnné, které jsou převzaty z dříve definovaného souboru *.env*. Dále se předpokládá, že existuje soubor *project-resource-allocation/backend/RELEASES/version_1.jar*, který se využívá pro spuštění serveru (viz výpis kódu 7.13).

Výpis kódu 7.13 Ukázka kódu v souboru *Dockerfile* u serveru

```
ARG SERVER_PORT=${SERVER_PORT}

FROM eclipse-temurin:17-jdk-alpine

ARG SERVER_PORT
ENV SERVER_PORT=${SERVER_PORT}

VOLUME /tmp
COPY RELEASES/version_1.0.jar app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Dockerfile klienta definuje prostředí node 17 a globální proměnné, které jsou opět převzaty z dříve definovaného souboru *.env*. Dále slouží pro stažení všech potřebných závislostí projektu a vystavení kontejneru klienta na portu 3001 (viz výpis kódu 7.14).

Výpis kódu 7.14 Ukázka kódu v souboru *Dockerfile* u klienta

```
ARG SERVER_URL=${SERVER_URL}
ARG CLIENT_PORT=${CLIENT_PORT}

FROM node:17-alpine

ARG SERVER_URL
ARG CLIENT_PORT
ENV SERVER_URL=${SERVER_URL}
ENV CLIENT_PORT=${CLIENT_PORT}

WORKDIR /app
COPY package.json .
COPY package-lock.json .
RUN npm install
COPY . .
EXPOSE ${CLIENT_PORT}
CMD ["npm", "start"]
```

Spuštění a organizace obou kontejnerů je pak definována v souboru *project-resource-allocation/docker-compose.yaml*. Tento soubor lze využít pro spuštění obou částí aplikace. Soubor definuje dvě služby, mezi které patří server (backend) a klient (frontend). Konfigurace souboru

pak určuje, že pro spuštění serveru se použije *Dockerfile* serveru a zároveň se kontejner vystaví na portu 8081. Naopak pro klienta se použije *Dockerfile* klienta a port 3001 (viz výpis kódu 7.15).

Výpis kódu 7.15 Ukázka kódu v souboru *docker-compose.yaml*

```
version: "3.8"
services:
    backend:
        build: ./backend
        container_name: backend_c
        ports:
            - "${SERVER_PORT}:${SERVER_PORT}"
        environment:
            - SERVER_PORT=${SERVER_PORT}
    frontend:
        build: ./frontend
        container_name: frontend_c
        ports:
            - "${CLIENT_PORT}:${CLIENT_PORT}"
        environment:
            - SERVER_URL=http://localhost:${SERVER_PORT}
            - CLIENT_PORT=${CLIENT_PORT}
        stdin_open: true
        tty: true
```

Kapitola 8
Testování

V této kapitole jsou popsané všechny důležité postupy a implementace týkající se testování. Lze zde nalézt, jak se testují jednotlivá omezení, testy rychlosti výpočtu řešení a uživatelské testy.

8.1 Testy jednotlivých omezení

Základem celého plánování jsou správně naimplementovaná omezení (viz kapitola 7.3.4), proto je každé omezení otestováno samostatně (nezávisle na ostatních omezeních). Průběh testu spočívá v tom, že je neprve definováno řešení, které je následně ohodnoceno pomocí zadaného omezení. Návratová hodnota je potom porovnána s očekávaným výsledkem.

Např. implementace testování omezení na preference je ve výpisu kódu 8.1. Vstupními hodnotami jsou dva úkoly a jeden zaměstanec s dvěma preferencemi. Lze vypozorovat, že očekávaným výsledkem po ohodnocení omezení na preference je číslo 60 (za předpokladu, že váha omezení na preference je 40), protože zaměstnanec má přiřazeny dva úkoly, které patří do jeho preferencí. První úkol je odměněn hodnotou 40, zatímco druhý úkol hodnotou 20. Důvodem je klesající důležitost druhé preference.

Výpis kódu 8.1 Testování omezení na preference

```
@Test
public void preferenceConflict() {
    Task task1 = new Task(...);
    Task task2 = new Task(...);
    // employee with preferences for task1 and task2
    Employee employee
        = new Employee(..., List.of(task1, task2), ...);

    task1.setAssignedEmployee(employee);
    task1.setStartingDate(...);
    task2.setAssignedEmployee(employee);
    task2.setStartingDate(...);

    Schedule solution = new Schedule(
        ..., List.of(task1, task2), ..., List.of(employee), ...
    );
    constraintVerifier.verifyThat(
            ScheduleConstraintProvider::preferenceConflict
        ).givenSolution(solution).rewardsWith(60);
}
```

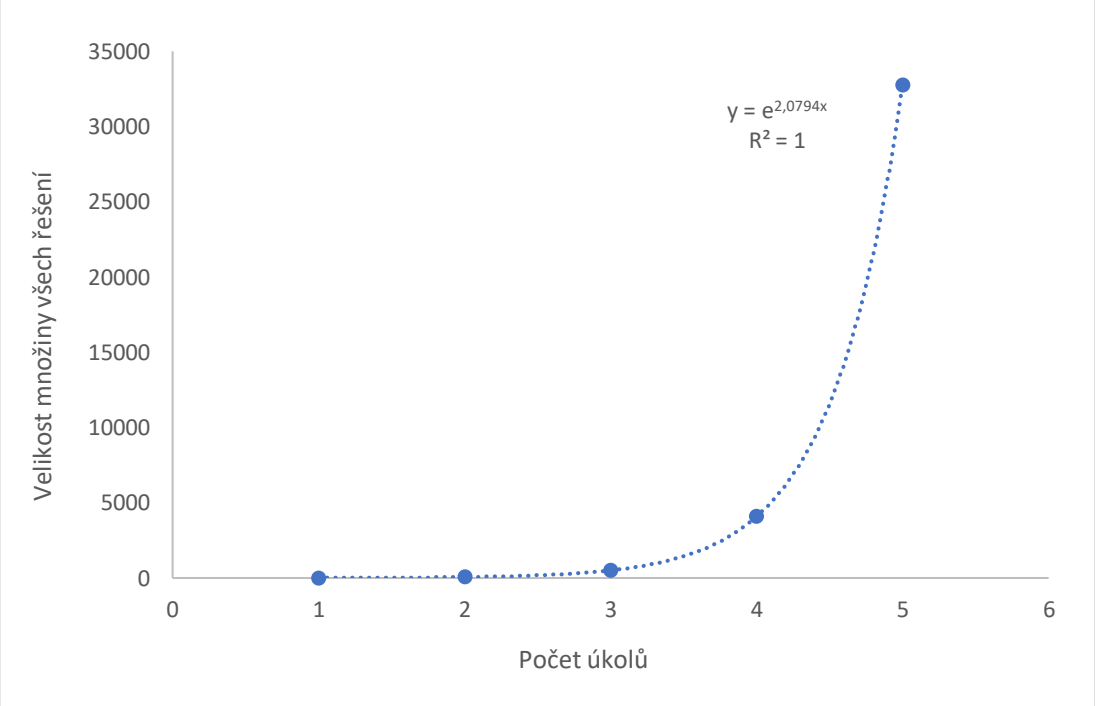
8.2 Testy doby výpočtu řešení

Testy doby výpočtu jsou zaměřené na efektivitu, ale i kvalitu generovaných řešení. Základním předpokladem pro testování jsou dobře navržená a implementovaná omezení.

V ideálním případě by testování probíhalo tak, že se naměří doba, za kterou bylo nalezeno globální optimum. Nicméně takové optimum není většinou předem známé a jeho nalezení by v nejhorším případě vyžadovalo projít celou množinu všech řešení, která se řídí funkcí:

$$h(x) = (pocetTydnuPlanu * pocetZamestnancu)^{pocetUkolu}$$

Z funkce je patrné, že s přibývajícími úkoly roste prohledávaný prostor exponenciálně. Tento fakt lze ilustrovat na problému alokace, kde existují dva zaměstnanci a plán je tvořen na čtyři týdny (viz obr. 8.1). Při řešení plánu, který má třicet zaměstnanců, šedesát úkolů a je tvořen na půl roku, což je problém specifikovaný zadavatelem, by prostor všech řešení zaujímal velikost cca $3,4 \times 10^{173}$ a při prohledávání takového prostoru zařízením s výkonem 2 mld. projitých řešení za vteřinu by nalezení globálního optima trvalo v nejhorším případě $5,3 \times 10^{156}$ let.



Obrázek 8.1 Graf znázorňující velikost množiny všech řešení v závislosti na počtu úkolů

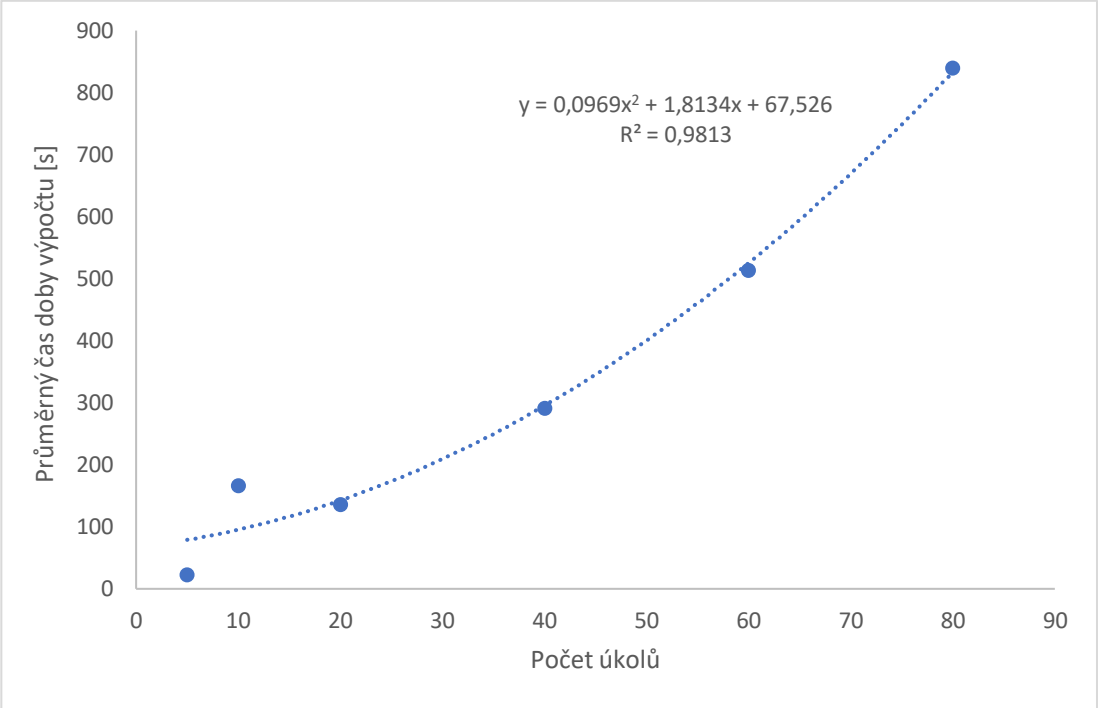
Proto byla pro testování zvolena jiná ukončující podmínka, a to uplynutí pětiminutového časového intervalu od posledního vygenerovaného řešení. Takto zvolená podmínka umožňuje porovnat rychlost uváznutí a naměřené skóre jednotlivých běhů aplikace.

Testování probíhalo na zařízení Dell Inc. Inspiron 7570 s RAM pamětí 8.0 GiB a procesorem Intel® Core™ i5-8250U o frekvenci 1.60 GHz.

Všechna testovací data, která byla na aplikaci testována, jsou v přiloženém médiu v adresáři *testing-data*. Tato data byla automaticky vygenerována, a proto i případy se špatným výsledným skóre mohou být uspokojujícím řešením, neboť jiné řešení nemusí existovat. Vyhodnocení dat bylo rozděleno do třech skupin podle počtu úkolů. První skupina obsahovala problémy o velikosti 5 a 10 úkolů, druhá skupina problémy o velikosti 20 a 40 úkolů a třetí skupina problémy o velikosti

60 a 80 úkolů. Každý test byl proveden třikrát, protože implementovaný algoritmus využívá náhodu při generování sousedních řešení, a tudíž nemusí vždy skončit se stejným výsledkem.

Jednotlivá vyhodnocení jsou v tabulkách B.1, B.2 a B.3 v příloze. Doba výpočtu označuje čas posledního vygenerovaného řešení. Tento čas lze pak využít i pro porovnávání různých testovacích dat mezi sebou. Naopak skóre slouží pouze pro porovnání průběhu jedné testovací sady, protože každá testovací sada obsahuje jiný problém a jiný očekávaný výsledek. Závislost počtu úkolů na průměrném času doby výpočtu je zobrazen na obr. 8.2.



Obrázek 8.2 Graf znázorňující průměrnou dobu výpočtu v závislosti na počtu úkolů

Z naměřených dat vyplývá, že s rostoucím počtem úkolů roste i průměrný čas doby výpočtu aplikace. To znamená, že program s rostoucí velikostí množiny všech řešení má tendenci zlepšovat aktuální řešení déle.

Z tabulky B.1 v příloze je patrné, že v testovací skupině s pěti úkoly, se v každé dílčí sadě řešitel vždy dostal na stejnou hodnotu skóre. Největší rozdíl v době hledání řešení nastal v sadě 1 a byl 4 s. V testovací skupině s deseti úkoly došlo k navýšení celkové doby výpočtu oproti testovací skupině s pěti úkoly, ale trend zůstal stejný. V každé dílčí sadě se řešitel dostal na stejné skóre. Výjimkou je třetí test v sadě 9, kde řešení s hodnotou soft skóre 58, je o 8 bodů horší než skóre u zbylých dvou testů ve stejné sadě. Naproti tomu bylo toto řešení nalezeno o 4 min a 16 s rychleji.

V testovací skupině s dvaceti a čtyřiceti úkoly se řešitel vždy dostal na stejnou hodnotu medium skóre a docházelo pouze k drobným odchylkám u soft skóre. Největší rozdíl soft skóre ve skupině s dvaceti úkoly byl v sadě 11 a to o 13 bodů, kdy toto řešení mělo opět i největší rozdíl v celkové době běhu a to o 5 min a 27 s oproti nejdéle trvajícímu testu ve stejné sadě. Ve skupině se čtyřiceti úkoly byl největší rozdíl soft skóre a opět i v době hledání řešení v sadě 19, kde rozdíl mezi nejlepším a nejhorším řešením byl 176 bodů a 13 min a 3 s (viz tabulka B.2 v příloze).

V testovací skupině s šedesáti a osmdesáti úkoly docházelo ke změnám v hodnotě medium i soft skóre. Největší rozdíl celkového skóre (součet medium a soft skóre) nastal ve skupině

s šedesáti úkoly v sadě 25, kde řešitel nedokázal ve dvou případech přiřadit všechny úkoly do plánu tak, aby nebyla porušena uzávěrka projektu, a tudíž byl penalizován -1000000 body za každý překročený týden uzávěrky projektu. Při jednom ze tří pokusů však ale dokázal najít řešení neporušující uzávěrku s celkovým skóre -1921 bodů v čase 21 min a 3 s. Ve skupině s osmdesáti úkoly byly opět největší rozdíly v celkovém skóre v sadě 28 a 30 z důvodu nepřiřazení úkolů do plánu tak, aby nebyla porušena uzávěrka. V obou dvou sadách však bylo nalezeno alespoň jedno řešení, které uzávěrku neporušuje a nebylo tak penalizováno. Nejdéle hledaným řešením v průběhu celého testování byl třetí test v sadě 29 s osmdesáti úkoly, který trval 54 min a 43 s a zároveň se jedná o nejlepší řešení v rámci této sady, které má soft skóre -1910 a je o 676 bodů lepší než nejhorší výsledek ve stejné sadě, který byl vygenerován za nejkratší čas 1 min a 6 s (viz tabulka B.3 v příloze). Dále z příloh vyplývá, že ani v jednom případě nebylo porušeno tvrdé omezení (viz tabulky B.1, B.2 a B.3 v příloze).

Vzhledem k tomu, že váhy jednotlivých omezení jsou konfigurovatelné, a že uživatel má možnost měnit parametry plánování, tak zodpovědnost za zhodnocení výsledného řešení náleží samotnému uživateli aplikace.

8.3 Uživatelské testy

Uživatelské testy slouží pro ohodnocení aplikace z uživatelské perspektivy. Běžný průběh těchto testů spočívá v návrhu testovacích scénářů, které jsou následně předloženy skupině uživatelů, kteří reprezentují všechny potenciální uživatele aplikace. Je vhodné mít tuto skupinu věkově i zkušenostně rozmanitou. Výsledky testování mohou pomoci odhalit různé chyby či nedostatky aplikace.

8.3.1 Testovací scénáře

V této kapitole jsou definovány scénáře, které mohou nastat při práci s aplikací. Tyto scénáře vycházejí z případů užití (viz kapitola 5.2) a slouží pro testování použitelnosti aplikace z pohledu uživatele.

8.3.1.1 S01 - Vytvoření souboru se vstupními daty

Uživateli je předložena uživatelská příručka, která popisuje strukturu vstupních dat. Uživatel je poté požádán, aby vytvořil soubor, který obsahuje jednu dovednost, jeden projekt, jednu projektovou fázi, jeden úkol, jednoho zaměstnance a konfiguraci, ve které by měla být definována délka plánu 20 týdnů.

8.3.1.2 S02 - Nahrání vstupních dat a spuštění plánování

Uživatel je posazen k počítači se spuštěnou aplikací, která nemá nahraná vstupní data, ani nemá spuštěné plánování. Uživatel je poté požádán, aby nahrál vstupní data a spustil plánování.

8.3.1.3 S03 - Zastavení plánování a exportování řešení

Uživatel je posazen k počítači se spuštěnou aplikací, která už má nahraná vstupní data a má spuštěné plánování. Poté je uživatel požádán, aby zastavil rozběhnuté plánování a uložil řešení pomocí exportování dat do souboru.

8.3.1.4 S04 - Exportování předposledního řešení

Uživatel je posazen k počítači se spuštěnou aplikací, ve které je ukončené plánování, které vygenerovalo alespoň dvě řešení. Uživatel je poté požádán, aby uložil předposlední vygenerované řešení pomocí exportování dat do souboru.

8.3.2 Vyhodnocení testovacích scénářů

Testovací scénáře z kapitoly 8.3.1 byly předloženy pěti uživatelům. U každého uživatele bylo testováno splnění či nesplnění testovacího scénáře. Výsledky jsou zobrazeny v tabulce 8.1.

Tabulka 8.1 Vyhodnocení testovacích scénářů

	S01	S02	S03	S04
Uživatel 1	✓	✓	✓	✓
Uživatel 2	✗	✓	✓	✓
Uživatel 3	✓	✓	✓	✓
Uživatel 4	✓	✓	✓	✓
Uživatel 5	✓	✓	✓	✓

Uživatelé byli schopni vykonat testovací scénáře bez větších potíží. Výjimkou byl Uživatel 2, který neměl velké zkušenosti s výpočetní technikou, a proto ani nebyl schopný vytvořit soubor se vstupními daty.

Mezi časté návrhy na vylepšení aplikace patřilo např. vytvoření grafických formulářů pro zadávání vstupních dat, perzistentní úložiště, které nevyžaduje ukládání do souboru, či načtení dat z jiných formátů (např. CSV).

Kapitola 9
Závěr

Tato bakalářská práce se zabývala problémem alokací zdrojů (zaměstnanců) na projekty a řešila ho pomocí techniky constraint programming. Primárním zaměřením bylo zohlednit preference zaměstnanců na projekty a požadavky projektů na zaměstnance.

Nejdříve byly sesbírány všechny dostupné informace týkající se alokování (plánování) od firmy Applifting s. r. o., která byla též zadavatelem bakalářské práce. Následně byly vytvořeny funkční a nefunkční požadavky na aplikaci, na jejichž základě byla provedena rešerše existujících aplikací, ve které nebyl nalezen žádný nástroj zohledňující preference zaměstnanců. Dále byla provedena rešerše algoritmů zabývajících se hledáním řešení v optimalizačních úlohách jako je problém alokací.

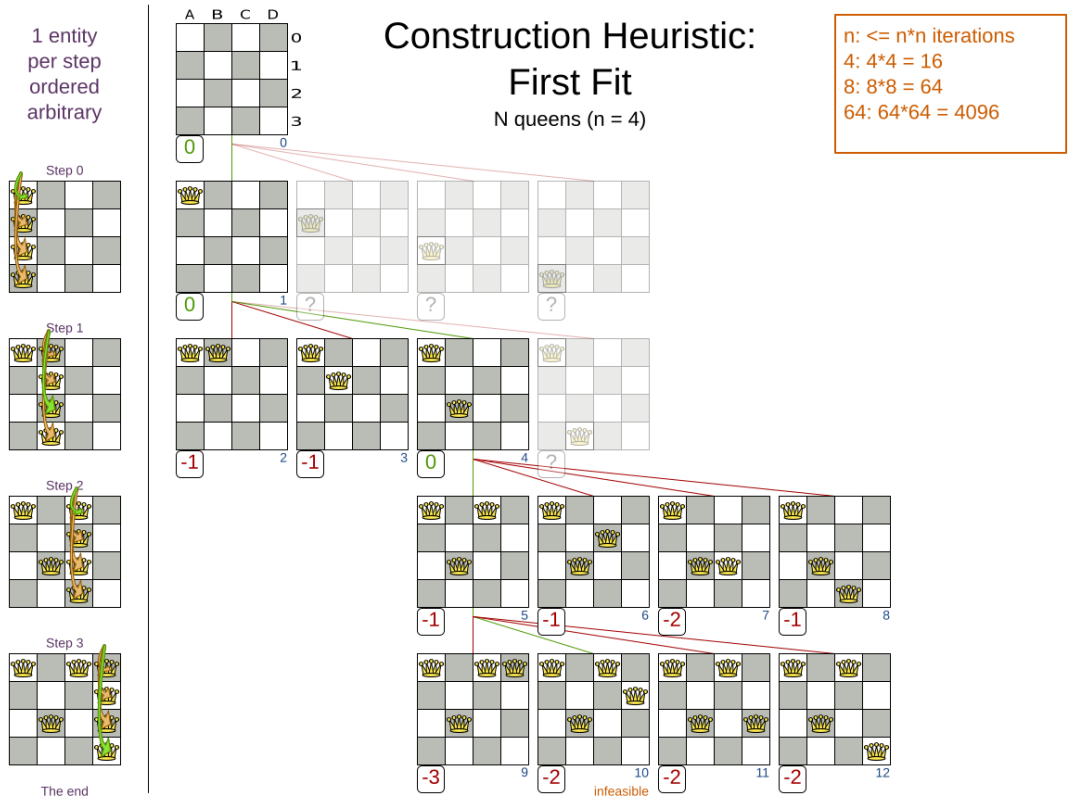
Při tvorbě návrhu byly vytvořeny případy užití vycházející z funkčních požadavků aplikace. Naopak architektura se zabývala návrhem systému s ohledem na nefunkční požadavky aplikace. Byla zde zvolena architektura client-server a konkrétní technologie, pomocí kterých byly obě části později naprogramovány. Pro klientskou část byl vybrán programovací jazyk TypeScript s knihovnou React a pro serverovou část programovací jazyk Java s frameworkem Spring Boot a knihovnou Optaplanner, která sloužila jako plánovací engine celé aplikace.

Následně byly v této práci podrobněji rozebrány všechny důležité části implementace, včetně konfigurace vybraného algoritmu simulovaného ochlazování. Výsledná aplikace byla poté otestována pomocí vygenerovaných dat na rychlost výpočtu a kvalitu řešení, která vycházela ze zadaných omezeních. Dále byla otestována uživatelská přívětivost aplikace pomocí uživatelských testů.

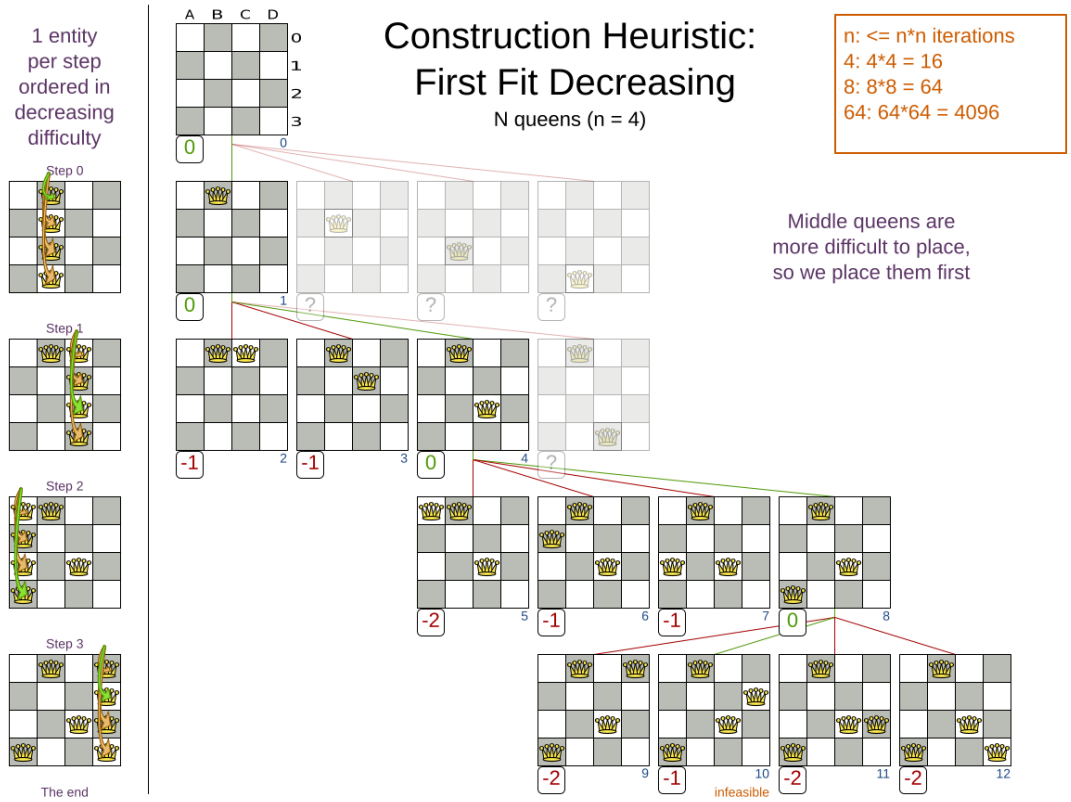
Bakalářská práce splnila všechny body zadání, včetně všech funkčních i nefunkčních požadavků aplikace, přesto je možné tuto práci dále rozvíjet. Lze např. vytvořit přívětivější uživatelské rozhraní nebo dále optimalizovat a testovat výpočetní algoritmus.

Příloha A

Problém n dam



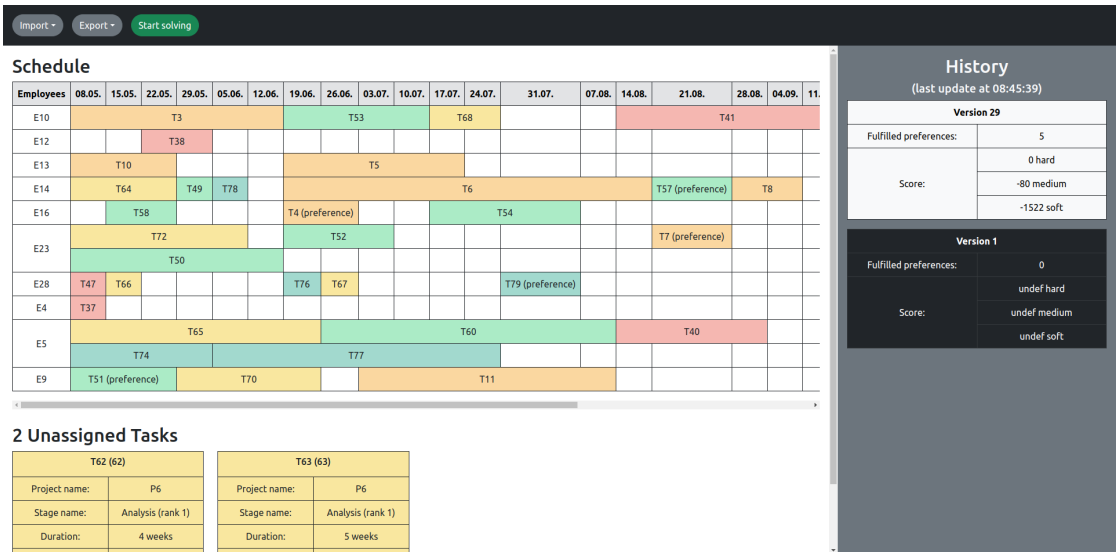
Obrázek A.1 Vizualizace procesu řešení problému n dam pomocí algoritmu First Fit [5]



Obrázek A.2 Vizualizace procesu řešení problému n dam pomocí algoritmu First Fit Decreasing [5]

Příloha B

Náhled a testovací data



Obrázek B.1 Ukázka implementované aplikace

Tabulka B.1 Testování malé sady úkolů vzhledem k době výpočtu a dosaženého skóre

testovací sada	počet úkolů	doba výpočtu	hard skóre	medium skóre	soft skóre
sada 1	5	0 min 5 s	0	-120	71
sada 1	5	0 min 1 s	0	-120	71
sada 1	5	0 min 1 s	0	-120	71
sada 2	5	0 min 43 s	0	-40	-9
sada 2	5	0 min 41 s	0	-40	-9
sada 2	5	0 min 42 s	0	-40	-9
sada 3	5	0 min 1 s	0	-160	264
sada 3	5	0 min 1 s	0	-160	264
sada 3	5	0 min 1 s	0	-160	264
sada 4	5	0 min 50 s	0	-80	136
sada 4	5	0 min 51 s	0	-80	136
sada 4	5	0 min 50 s	0	-80	136
sada 5	5	0 min 19 s	0	-80	108
sada 5	5	0 min 19 s	0	-80	108
sada 5	5	0 min 19 s	0	-80	108
sada 6	10	0 min 18 s	0	-160	177
sada 6	10	0 min 18 s	0	-160	177
sada 6	10	0 min 19 s	0	-160	177
sada 7	10	5 min 45 s	0	-200	268
sada 7	10	5 min 49 s	0	-200	268
sada 7	10	5 min 46 s	0	-200	268
sada 8	10	1 min 42 s	0	-240	5
sada 8	10	1 min 42 s	0	-240	5
sada 8	10	3 min 2 s	0	-240	5
sada 9	10	5 min 45 s	0	-160	64
sada 9	10	5 min 48 s	0	-160	64
sada 9	10	1 min 32 s	0	-160	58
sada 10	10	1 min 21 s	0	0	-419
sada 10	10	1 min 21 s	0	0	-419
sada 10	10	1 min 13 s	0	0	-419

Tabulka B.2 Testování střední sady úkolů vzhledem k době výpočtu a dosaženého skóre

testovací sada	počet úkolů	doba výpočtu	hard skóre	medium skóre	soft skóre
sada 11	20	6 min 44 s	0	-40	-659
sada 11	20	3 min 27 s	0	-40	-670
sada 11	20	1 min 17 s	0	-40	-672
sada 12	20	0 min 20 s	0	-160	-765
sada 12	20	0 min 20 s	0	-160	-765
sada 12	20	0 min 21 s	0	-160	-765
sada 13	20	4 min 35 s	0	-80	-501
sada 13	20	2 min 44 s	0	-80	-500
sada 13	20	6 min 16 s	0	-80	-500
sada 14	20	0 min 48 s	0	-240	-301
sada 14	20	0 min 47 s	0	-240	-301
sada 14	20	0 min 48 s	0	-240	-301
sada 15	20	0 min 30 s	0	-240	13
sada 15	20	4 min 39 s	0	-240	11
sada 15	20	0 min 29 s	0	-240	13
sada 16	40	1 min 6 s	0	-80	-1287
sada 16	40	4 min 48 s	0	-80	-1299
sada 16	40	1 min 57 s	0	-80	-1292
sada 17	40	0 min 53 s	0	0	-1883
sada 17	40	1 min 4 s	0	0	-1794
sada 17	40	1 min 22 s	0	0	-1770
sada 18	40	3 min 21 s	0	-40	-1480
sada 18	40	6 min 46 s	0	-40	-1481
sada 18	40	5 min 12 s	0	-40	-1476
sada 19	40	13 min 56 s	0	0	-1506
sada 19	40	0 min 53 s	0	0	-1682
sada 19	40	9 min 55 s	0	0	-1569
sada 20	40	8 min 26 s	0	-440	-899
sada 20	40	11 min 24 s	0	-440	-953
sada 20	40	1 min 32 s	0	-440	-960

Tabulka B.3 Testování velké sady úkolů vzhledem k době výpočtu a dosaženého skóre

testovací sada	počet úkolů	doba výpočtu	hard skóre	medium skóre	soft skóre
sada 21	60	1 min 30 s	0	-40	-2719
sada 21	60	7 min 20 s	0	-40	-2487
sada 21	60	3 min 51 s	0	0	-2579
sada 22	60	9 min 26 s	0	-40	-1610
sada 22	60	14 min 57 s	0	-40	-1614
sada 22	60	1 min 20 s	0	-80	-1688
sada 23	60	7 min 15 s	0	0	-1569
sada 23	60	9 min 36 s	0	0	-1521
sada 23	60	15 min 10 s	0	0	-1523
sada 24	60	3 min 11 s	0	0	-1448
sada 24	60	11 min 26 s	0	0	-1345
sada 24	60	9 min 36 s	0	0	-1360
sada 25	60	4 min 5 s	0	0	-1002262
sada 25	60	21 min 3 s	0	0	-1921
sada 25	60	8 min 32 s	0	0	-1002055
sada 26	80	3 min 42 s	0	-40	-2825
sada 26	80	5 min 50 s	0	-80	-2580
sada 26	80	2 min 35 s	0	-80	-2751
sada 27	80	34 min 48 s	0	0	-1957
sada 27	80	14 min 43 s	0	0	-2135
sada 27	80	53 min 1 s	0	0	-1829
sada 28	80	9 min 0 s	0	0	-2483
sada 28	80	0 min 56 s	0	0	-6003271
sada 28	80	4 min 28 s	0	0	-3049
sada 29	80	1 min 6 s	0	0	-2586
sada 29	80	2 min 53 s	0	0	-2324
sada 29	80	54 min 43 s	0	0	-1910
sada 30	80	0 min 8 s	0	0	-1003049
sada 30	80	7 min 29 s	0	0	-2702
sada 30	80	14 min 41 s	0	0	-1002525

Bibliografie

1. LALOUX, Frederic; JUREK, Viktor. *Budoucnost organizací*. PeopleComm, 2016. ISBN 978-80-87917-68-8.
2. ROSSI, Francesca; BEEK, Peter van; WALSH, Toby (ed.). *Handbook of Constraint Programming*. Sv. 2. Elsevier, 2006. Foundations of Artificial Intelligence. ISBN 978-0-444-52726-4. Dostupné také z: `https://www.sciencedirect.com/science/bookseries/15746526/2`.
3. MARTELLO, Silvano; TOTH, Paolo. Algorithms for Knapsack Problems. In: MARTELLO, Silvano; LAPORTE, Gilbert; MINOUX, Michel; RIBEIRO, Celso (ed.). *Surveys in Combinatorial Optimization*. North-Holland, 1987, sv. 132, s. 213–257. North-Holland Mathematics Studies. ISSN 0304-0208. Dostupné z DOI: `https://doi.org/10.1016/S0304-0208(08)73237-7`.
4. CHU, P. C.; BEASLEY, John E. A Genetic Algorithm for the Multidimensional Knapsack Problem. *Journal of Heuristics*. 1998, roč. 4, s. 63–86.
5. THE OPTAPLANNER TEAM. *OptaPlanner User Guide* [online]. 2023. [cit. 2023-04-17]. Dostupné z: `https://docs.optaplanner.org/latest/optaplanner-docs/html_single/index.html`.
6. BRAILSFORD, Sally C.; POTTS, Chris N.; SMITH, Barbara M. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*. 1999, roč. 119, č. 3, s. 557–581. ISSN 0377-2217. Dostupné z DOI: `https://doi.org/10.1016/S0377-2217(98)00364-6`.
7. BOUAJAJA, Sana; DRIDI, Najoua. A survey on human resource allocation problem and its applications. *Operational Research*. 2017, roč. 17, s. 339–369.
8. OSMAN, MS; ABO-SINNA, Mahmoud A; MOUSA, AA. An effective genetic algorithm approach to multiobjective resource allocation problems (MORAPs). *Applied Mathematics and Computation*. 2005, roč. 163, č. 2, s. 755–768.
9. KIRSCHNER, Filip. 2023. zadavatel, spoluzakladatel firmy Applifting s. r. o. [ústní sdělení]. Praha, 15.2.2023.
10. ICONS8 LLC. *Cross mark Icons – Download for Free in PNG and SVG* [online]. 2023. [cit. 2023-05-07]. Dostupné z: `https://icons8.com/icon/97743/cancel`.
11. ICONS8 LLC. *Tick Icons – Download for Free in PNG and SVG* [online]. 2023. [cit. 2023-05-07]. Dostupné z: `https://icons8.com/icon/101666/checkmark`.
12. ICONS8 LLC. *Decrease Icons – Download for Free in PNG and SVG* [online]. 2023. [cit. 2023-05-07]. Dostupné z: `https://icons8.com/icon/qSSdYz6igIn1/decrease`.
13. ICONS8 LLC. *Increase Icons – Download for Free in PNG and SVG* [online]. 2023. [cit. 2023-05-07]. Dostupné z: `https://icons8.com/icon/L-SUKXwXg0RN/increase`.

14. MONDAY.COM. *A new way of working* [online]. 2021. [cit. 2023-04-19]. Dostupné z: https://monday.com/.
15. PARALLAX. *Resource And Capacity Planning Software* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://www.getparallax.com/.
16. KANTATA. *Realize the potential of your services business* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://www.kantata.com/.
17. PAYMO LLC. *Paymo™ - Project Management, Time Tracking, and Invoicing Software for Teams* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://www.paymoapp.com/.
18. TOGGL PLAN. *Time Tracking Software, Project Planning And Hiring Tools* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://toggl.com/.
19. FORECAST. *AI Project And Resource Management Software For Professional Services* [online]. 2023. [cit. 2023-04-19]. Dostupné z: https://www.forecast.app/.
20. EMAMBOCUS, Bibi Aamirah Shafaa; JASSER, Muhammed Basheer; AMPHAWAN, Angela. An optimized continuous dragonfly algorithm using Hill climbing local search to tackle the low exploitation problem. *IEEE Access*. 2022, roč. 10, s. 95030–95045.
21. EGLESE, Richard W. Simulated annealing: a tool for operational research. *European journal of operational research*. 1990, roč. 46, č. 3, s. 271–281.
22. ROCKE, DM. Genetic Algorithms+ Data Structures= Evolution programs (3rd. *Journal of the American Statistical Association*. 2000, roč. 95, č. 449, s. 347.
23. KATOCH, Sourabh; CHAUHAN, Sumit Singh; KUMAR, Vijay. A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*. 2021, roč. 80, s. 8091–8126.
24. ČÁPKA, David. *Lekce 2 - UML - Use Case Diagram* [online]. 2023. [cit. 2023-04-20]. Dostupné z: https://www.itnetwork.cz/navrh/uml/uml-use-case-diagram.
25. GARLAN, David. Software architecture: a roadmap. In: *Proceedings of the Conference on the Future of Software Engineering*. 2000, s. 91–101.
26. FOWLER, Martin. Is High Quality Software Worth the Cost? *Martin Fowler*. 2019.
27. KAMBALYAL, Channu. 3-tier architecture. *Retrieved On*. 2010, roč. 2, č. 34, s. 2010.
28. ITNETWORK.CZ. *Lekce 2 - Monolitická a dvouvrstvá architektura* [online]. 2023. [cit. 2023-05-07]. Dostupné z: https://www.itnetwork.cz/navrh/architektury-a-dependency-injection/monoliticka-a-douvrstva-architektura.
29. ORACLE. *What is Java and why do I need it?* [online]. 2023. [cit. 2023-05-07]. Dostupné z: https://www.java.com/en/download/help/whatis_java.html.
30. AXON. *Is Java still relevant in 2023 — Axon* [online]. 2023. [cit. 2023-05-07]. Dostupné z: https://www.axon.dev/blog/is-java-still-relevant-in-2022.
31. VMWARE, INC. *Spring — Home* [online]. 2023. [cit. 2023-05-07]. Dostupné z: https://spring.io.
32. STACK DIARY. *The Most Popular Front-end Frameworks in 2023* [online]. 2023. [cit. 2023-05-07]. Dostupné z: https://stackdiary.com/front-end-frameworks/.
33. GREAT LEARNING. *History of ReactJS - Great Learning* [online]. 2023. [cit. 2023-05-07]. Dostupné z: https://www.mygreatlearning.com/react-js/tutorials/history-of-reactjs.
34. DOCKER INC. *What is a container?* [online]. 2023. [cit. 2023-04-29]. Dostupné z: https://www.docker.com/resources/what-container/.

Obsah přiloženého média