Jihočeská univerzita v Českých Budějovicích
Přírodovědecká fakulta

Návrh a implementace parametrické sbírky a generátoru matematických úloh

Bakalářská práce

Jakub Vitoušek

Školitel: Mgr. Jan Eisner, Dr.

České Budějovice 2023

Vitoušek J., 2023: Návrh a implementace parametrické sbírky a generátoru matematických úloh. [Design and implementation of parametric problem collection and generator of mathematical problems. Bc. Thesis, in Czech.] - 58 p., Faculty of Science, University of South Bohemia, České Budějovice, Czech Republic.

Anotace:

This project deals with design and following implementation of parametric collection containing parametrized mathematical problems and problem generator, which derives the structure of individual problems from the mentioned collection. The final outcome is a simple application capable of generating set of tests for chosen number of students based on the configuration of test established by the user. The resulting tests are composed of same type problems, while differing in numerical values and can be exported to PDF format.

Prohlašuji, že jsem autorem této kvalifikační práce a že jsem ji vypracoval pouze s použitím pramenů a literatury uvedených v seznamu použitých zdrojů.

České Budějovice, 12. dubna 2023

. . ..

Poděkování

Rád bych poděkoval svému vedoucímu bakalářské práce Mgr. Janu Eisnerovi, Dr., za jeho věcné připomínky, cenné rady a ochotu. Též bych chtěl poděkovat své rodině a blízkým, že při mně stáli po celou dobu mého studia.

Obsah

Seznam obrázků

Seznam zdrojových kódů

Úvod

Systém výuky matematiky a ostatních přírodovědných oborů se na českých školách z dlouhodobého hlediska příliš nezměnil, ba naopak v některých ohledech částečně degradoval. Dochází k neustálé recyklaci stejných učebnic a materiálů na úkor možností, které nabízí digitalizace dnešní doby. V důsledku toho studenti k výuce mnohdy používají desítky let staré učebnice, kterých se s postupujícími ročníky zbavují a musí kupovat jiné. Stejně tak každý učitel preferuje práci podle jiných materiálů, které se liší obtížnostmi, v určité míře i obsahově. Dojde-li u dané třídy k výměně učitele, může jeho nástupce vyžadovat práci s jinými materiály, což vyvíjí na studenty či jejich rodiče další finanční nátlak, kdy jsou nuceni shánět požadované učebnice a současně nutí studenty si přivykat na jiný systém nové učebnice. Z pohledu učitele nastávají obdobné komplikace jak v hledání vhodných typových příkladů pro dané probírané téma, tak i ve tvorbě testů. V případě typových příkladů je hlavním limitujícím faktorem omezená kapacita učebnice, tudíž tyto příklady jsou snadno vyčerpatelné a nutí učitele i žáky využívat externí zdroje. V případě tvorby testů nastávají další komplikace, mezi něž patří distribuce do skupin a zachování férovosti, výběr vhodných příkladů i časová náročnost přípravy a následné opravy testu.

Zmíněné problémy se snaží reflektovat návrh parametrické sbírky a generátoru matematických úloh, o nichž tato práce pojednává. Hlavní výhodou by byla snadná aktualizovatelnost, jež z logických důvodů u papírových výtisků není možná, a prakticky neomezená možnost dalšího rozšiřování obsahu. V porovnání s běžnými sbírkami a učebnicemi by zde mohlo být vše na jednom místě a snáz dostupné. Samotná parametrická sbírka, jak už název napovídá, by využívala parametrizace, tudíž každá typová úloha může být vygenerována v libovolném počtu hodnot se lišících variant. To umožňuje

v případě testu, místo přípravy běžných dvou skupin, vygenerovat, tolik individuálních testů, kolik je ve třídě studentů. Navíc lze každému testu přiřadit i řešení, aby se ušetřila časová náročnost přípravy i opravy testů. Výsledné testy tedy mohou být složeny ze stejných typových příkladů, avšak individualizovanými pro každého studenta zvlášť.

Jako student oboru matematiky a chemie se zaměřením na vzdělávání pro SŠ jsem se z vlastní iniciativy rozhodl získat základní zkušenosti s učitelstvím skrze doučování jednotlivců a skupin studentů. Během přípravy materiálů a hledání vhodných příkladů jsem začal narážet na některá úskalí, zejména nutnost mnohdy čerpat z více zdrojů, ať už knižních, či webových. Vzhledem k nekompletnosti a omezenému obsahu některých z nich, jsem musel trávit zbytečně více času hledáním a vytvářením nových příkladů. Z vlastní zkušenosti vím, že mnohdy je potřeba k řádnému naučení a pochopení látky spočítat značné množství příkladů. A proto bych se odvážil tvrdit, že čím více příkladů student spočítá, tím lépe látce porozumí. Což mě přivedlo na myšlenku, že by bylo vhodné přijít s jednotným řešením, které by vyřešilo tuto situaci. Dílčí částí tohoto řešení by, dle mého názoru, mohl být obecný generátor úloh. Jeho velkou výhodou dle implementace, kterou zamýšlím, by byla v podstatě neomezená kapacita příkladů. Za předpokladu, že celý projekt splní očekávání, je mou vizí využít ho i na další přírodovědné předměty a nadále ho rozšiřovat.

Celá práce je rozdělena do osmi kapitol, včetně úvodu, cílů práce a závěru. Jednotlivé kapitoly jsou s ohledem k příslušným tématickým celkům dále děleny do jednotlivých sekcí.

Cíle práce

Práce si klade za cíl vytvořit návrh parametrické sbírky, generátoru matematických úloh a následnou implementaci obou částí. V rámci realizace projektu je v plánu nejprve se v Kapitole 1 *Rešerše* seznámit s některými již dostupnými ke studiu zaměřenými programy, weby a studijními materiály.

Následně je záměrem navrhnout a implementovat parametrickou sbírku matematických úloh a generátor příkladů, jež by generoval příklady právě z této sbírky. Obě tyto části by pak měly být implementovány do jednoho celku, který bude na závěr otestován a jeho výstupy kriticky zhodnoceny.

Praktická část práce bude realizována především pomocí programovacího jazyka Python vzhledem k rozsáhlému množství knihoven, jež tento program nabízí. Z hlediska výstupů bude pravděpodobně zapotřebí některého nástroje ke konvertování vygenerovaných příkladů do PDF (zkratka. Portable Document Format). Tato problematika bude více rozebrána v Kapitole 2 *Konceptuální návrh*, kde by byla představena prvotní úvaha a schéma, podle něhož bude projekt uvažován. Jednotlivé použité nástroje budou analyzovány a popsány v kapitole 3 *Analýza dostupných nástrojů a programů*.

Kapitola 4 *Návrh a implementace* by pak měla představit konečné funkční řešení, které by bylo v rámci této kapitoly podrobně popsáno.

Následně pak v kapitole 5 *Praktické testování* bude celkový projekt řádně otestován na základě potenciálních uživatelských požadavků. Některé konkrétní ukázky zde budou prezentovány a okomentovány.

Nakonec budou celý projekt a jím dosažitelné výstupy zhodnoceny a z toho vyvozen závěr a to jak retrospektivně, tak i vzhledem k potenciální budoucnosti tohoto projektu.

Kapitola 1

Rešerše

V této kapitole jsou popsány systémy a studijní materiály používané školami či při osobní výuce. Všechny případy mají souvislost s touto prací vzhledem k přímé podobnosti či z hlediska vize, na jejímž základě práce vznikla.

1.1 Elektronické školní systémy

Elektronické školní systémy poskytují školám technické řešení administrativních záležitostí a pomáhají s celkovou správou školy. Jejich cílem je zjednodušení těchto úkonů pomocí automatizace, čímž lze ušetřit práci nejen vedení školy, ale i studentům a učitelům.

1.1.1 Bakaláři

Software Bakaláři je produktem firmy *Bakaláři software s.r.o*, která ho uvedla na trh jako školní informační systém snažící se školám poskytnout technické řešení administrativních záležitostí a podporuje provoz ve školní síti, cloudově či hybridně. V současné době je údajně využíván více než 60% škol v ČR. Bakaláři nabízejí širokou škálu modulů a to: Společné prostředí, Evidence, Bakalář, Rozvrh, Suplování, Plán akcí školy, Maturity, GDPR, Webová aplikace, Internetová žákovská knížka a další. Modul Webová aplikace obsahuje osobní údaje žáka, pololetní klasifikaci, průběžnou klasifikaci, průběžnou docházku, výchovná opatření, rozvrh, třídní knihu, domácí úkoly, vyučované předměty, přehled výuky, absence, ankety, plán akcí a komunikační systém školy [1].

Moduly systému bakaláři [2]:

1.1.1.1 Společné prostředí

Je základním modulem systému bakaláři, které slouží ke správě společných modulů samotného systému [2].

1.1.1.2 Modul Evidence

Obsahuje osobní data žáků a zaměstnanců, umožňuje napojení na modul Rozvrh a zobrazení rozvrhů daných žáků, či zaměstnanců. Dále též eviduje platby, či klasifikaci studentů a to dle zadaných parametrů (pololetní, čtvrtletní, průběžná) [2].

1.1.1.3 Modul Bakalář

Je obecný systém zpracující jakékoli evidence. Slouží k definování libovolných aplikací u nichž je následně možno modifikovat přístupová práva uživatelů k dispozici jsou i již existující šablony. Zmiňovanými aplikacemi může například být aplikace k přijímacím zkouškám, databáze školní knihovny, rozpočet školy, evidence úrazů, apod. [2].

1.1.1.4 Modul rozvrh

Slouží k tvorbě rozvrhů. Dohlíží také na to, aby nedocházelo ke vzniku kolizí, či aby bylo rezervováno pro žáky i učitele volno na obědy [2].

1.1.1.5 Modul suplování

Navazuje na Rozpis maturit a Plán akcí školy, odkud sbírá data o absencích, na základě něhož dokáže nabídnout všechny volné učitele a všechny třídy, které potřebují suplující učitele. Samotná volba konkrétního učitele už závisí na lidské obsluze, systém pouze nabídne vhodné možnosti [2].

1.1.1.6 Modul plán akcí školy

Umožňuje zadávání školních akcí s příslušnými daty, tzn. která třída s jakým dozorem kam kdy půjde, apod [2].

1.1.1.7 Modul maturity

Pracuje s daty žáků, jejich výběrem maturitních předmětů, umožňuje tvorbu komise, či rozvržení zkoušení podle tříd a žáků do různých dnů na základě parametrů jako je vytíženost zkoušejících a dalších účastníků zkoušky [2].

1.1.1.8 Modul GDPR

Pomáhá řešit povinnosti vyplývající z GDPR, avšak konečná zodpovědnost je na škole [2].

1.1.1.9 Modul webová aplikace, internetová žákovská knížka

Zpřístupňuje studentům a rodičům studentů informace o průběžné a pololetní klasifikaci, průběžné docházce studentů, osobní rozvrh studenta a změny v něm, plán akcí, domácí úkoly, výchovná opatření, ankety a komunikační systém školy [3]. Webová aplikace obsahuje:

- Osobní údaje žáka – dle výběru, rodiče mohou nejen kontrolovat, zda škola eviduje správné údaje (telefony, adresy...), ale systém umožňuje i přímé ohlašování změn přes webové rozhraní.
- Pololetní klasifikace – kompletní pololetní klasifikace ve všech ročnících od počátku studia, přehled zameškaných hodin atd.
- Průběžná klasifikace – přehled všech průběžně zadaných známek; lze volit, zda zobrazovat i témata, váhy známek, poznámky, průměr apod.
- Průběžná docházka – napojení na modul Třídní kniha, evidence zameškaných hodin (zobrazení po dnech, měsících, ale i přímo po vyučovacích hodinách).
- Výchovná opatření – přehled třídních důtek, pochval atd. za aktuální pololetí nebo za celé studium.
- Rozvrh – tabulka rozvrhu, a to buď stálý rozvrh, nebo aktuální rozvrh včetně změn v souvislosti se suplováním.
- Třídní kniha – možnost zápisu hodin do elektronické třídní knihy.
- Domácí úkoly – přehled domácích úkolů (aktivních i ukončených).
- Vyučované předměty – seznam předmětů s uvedením učitelů, kteří daný předmět žáka vyučují.
- Přehled výuky – seznam odučených témat.
- Absence – přehled zameškaných hodin.
- Ankety – modul pro vyplňování anket pro žáky a rodiče.
- Plán akcí – kalendář a seznam plánovaných akcí školy.

- Komunikační systém školy – modul pro komunikaci mezi ředitelstvím, učiteli, rodiči, případně žáky (vedle nástěnek obsahuje poznámky vyučujících k hodnocení žáků a omluvenky od rodičů).

1.1.2 Edookit

Jedná se o školní informační systém vyvíjený ve spolupráci s učiteli, který je určený pro všechny střední a nižší typy škol. Systém spojuje administrativní a výukové funkce. Edookit obsahuje různé moduly a to: Elektronickou třídní knihu, Elektronickou žákovskou knihu, Tvorbu rozvrhu, Online matriku a Administrativu školy. Modul Elektronická žákovská knížka umožňuje učitelům snadno provádět třídní administrativu. Při zápisu do třídní knihy učitel v seznamu žáků vybere ty chybějící, jež zapíše, a ostatní jsou automaticky evidováni jako přítomní. Dále modul umožňuje doplnit probírané učivo, domácí úkol, či informace k následující hodině, což je výhodné i pro studenty. V případě, že učitel zapomene provést zápis, systém ho sám upozorní. Modul Elektronická žákovská knížka dává učiteli k dispozici soubor nástrojů ke zjednodušují výuky a administrativy. Modul umožňuje učitelům zapsat známky, slovní hodnocení nebo aktivitu v hodině. Dále umožňuje rodičům přímou komunikaci s učitelem a zobrazení známek všech svých dětí najednou. Modul Tvorba rozvrhu umožňuje rychle zjistit, kdo učí kterou hodinu, a případně kdo za koho supluje. Systém se snaží automaticky k suplování navrhnout učitele s příslušnou aprobací, vhodným vztahem ke třídě a nedostatečným počtem suplovaných hodin, čímž ušetří spoustu práce zástupkyním. Modul Online matrika usnadňuje práci s daty pro MŠMT a umožňuje úpravy dat v tabulkách s povolenými hodnotami. Usnadňuje hledání dostupných údajů o každém zaměstnanci, studentovi nebo rodičů. Modul Administrativa školy umožňuje škole snadno spravovat veškeré databáze a zpracovávat data v systému, např. osobní údaje, GDPR evidenci, či platby [4].

1.1.3 IS/STAG

IS/STAG je komplexní vysokoškolský informační systém, který je vyvíjen Centrem informatizace a výpočetní techniky Západočeské univerzity, a je určený k administraci studijní agendy. Nabízí rozsáhlé funkce pro administrativu školy a evidenci studentů a to prezenční formy studia, kombinované formy studia, studenty celoživotního vzdělaní, či účastníky univerzity třetího věku. Systém řeší vše od přijímacího řízení až po vydání diplomu. Od roku 1993, kdy byl poprvé nasazen na Západočeské univerzitě v Plzni, se rozšířil na více než 14 vysokých škol v ČR. Aplikace využívá

databázi Oracle, jež nabízí spoustu doplňujících modulů a aplikací. Systém umožňuje určitou parametrizaci, aby vyhovoval požadavkům jednotlivých fakult, případně i kateder. Portál nabízí především funkce pro studenty a učitele, a to v českém, či anglickém rozhraní, další jazykové verze prozatím nejsou implementovány.

Základní část informačního systému je rozdělena do bloků:

- Blok Studijní programy, obory, plány, předměty je evidence objektů pro různé typy studia umožňující více verzí studijních plánů a bloků předmětů. V mnoha případech je možné vést i kombinace studijních oborů.

- Blok Student se zasazuje o kompletní evidenci studenta, včetně jeho osobních údajů studijního programu, zápisů na studijní plány, přerušení studia, atd.

- Blok Přijímací řízení umožňuje vypsání přijímacích oborů a jim příslušných požadovaných předmětů z předchozího studia atestů. Díky nastavitelné parametrizaci je pak systém schopen vyhodnotit úspěšnost uchazečů a rozdělit je na úspěšné a neúspěšné.

- Blok Rozvrhy umožňuje vytvářet rozvrhové akce pro jednotlivé předměty a k nim stanovit příslušný čas a místo a současně lze určit, kterých studentů se týká, a to dle fakulty, formy studia, studijního programu, apod.

- Blok Předzápis umožňuje studentům zapsat si další semestr a kontrolu podmínek plnění studia. Student má možnost zapsat si jednotlivé předměty nebo celý kroužek, čímž se mu pak zapíší všechny příslušné rozvrhové akce.

- Blok Zkoušky umožňuje vypisovat zkouškové termíny, na které se pak studenti mohou přihlásit. U zkoušek lze nastavit termín, místo konání a způsob zkoušení.

- Blok Semestrální Práce eviduje vypisování témat semestrálních prací, zapisování studentů, elektronické odevzdávání prací, hodnocení učitelem a kontrolu plagiátorství.

- Blok Mobility studentů asistuje s výjezdy studentů a problémy s tím souvisejícími.

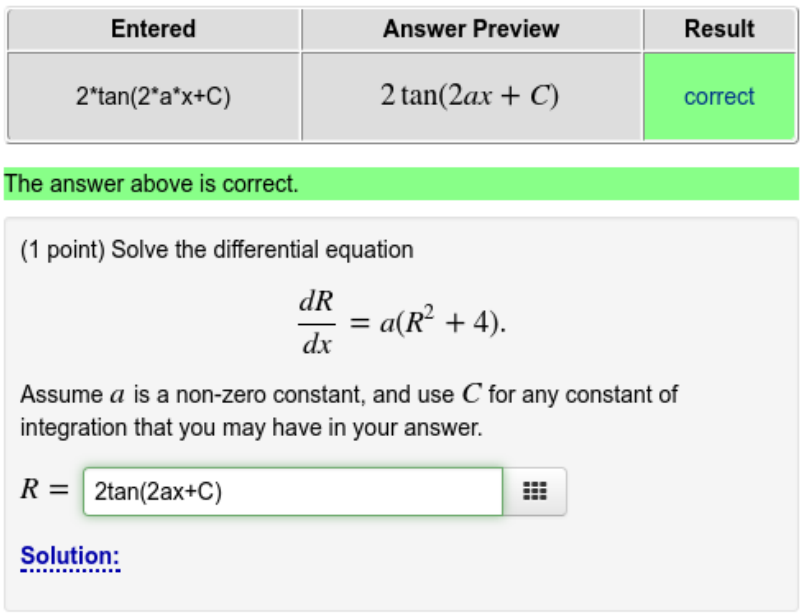
- Blok Evaluace zhodnotí na základě studentské kritiky kvalitu výuky. Zpětná vazba je získávána pomocí povinných dotazníků, z nichž jsou následně získána data a ta jsou dále prezentována.
- Blok Předpisy plateb asistuje s přípravou předpisů plateb, stipendia, poplatky, apod.
- Blok Absolvent eviduje vše týkající se absolventů, od kvalifikačních prací, závěrečných zkoušek až po zápis o závěrečné zkoušce a diplom [5].

1.2 Generátory příkladů a parametrické sbírky

Tato sekce pojednává o generátorech příkladů a jim přidružených parametrických sbírkách dostupných na internetu. Všechny zmíněné systémy generují webové testy, kde uživatel doplňuje své řešení, bez postupu, do předpřipravených políček a po odevzdání je test systémem automaticky vyhodnocen. The WeBWorK Project umožňuje opravený test vygenerovat v PDF formátu, jež si může uživatel stáhnout. Wolfram Problem Generator nabízí stejnou funkci, avšak ta je přístupná pouze v placené verzi.

1.2.1 The WeBWorK Project

WebWork je široce rozšířený, bezplatný, open-source webový systém, využívaný mnoha středními a vysokými školami po celém světě, určený ke generování online úloh na základě předem určených parametrů, a to především z přírodovědných oborů jako je matematika, fyzika, chemie apod. Program nabízí učitelům velmi efektivní a účinný způsob, jak studentům poskytnout online úkoly a testy, po jejichž vyřešení dostanou okamžitou zpětnou vazbu o správnosti jejich řešení. Studenti tak díky WebWorku mohou studované téma snáze pochopit a procvičit si ho. Velkou výhodou je individualita zadání, kdy každý úkol je pro každého studenta mírně pozměněný, například v numerických hodnotách. Software nabízí studentům možnost pokusit se splnit úkol opakovaně, dokud neuspějí, současně mohou i spolupracovat a podílet se vzájemně na řešení těchto úkolů, avšak každý musí splnit ten svůj, tudíž i v případě spolupráce je třeba splnit stejný počet individuálních zadání. Instruktoři jsou současně zpravováni aktuálními statistikami, díky nimž se dozví, jak studenti zvládají jednotlivé typy úloh i tématických celků, díky čemuž mohou upravovat svou výuku, aby lépe

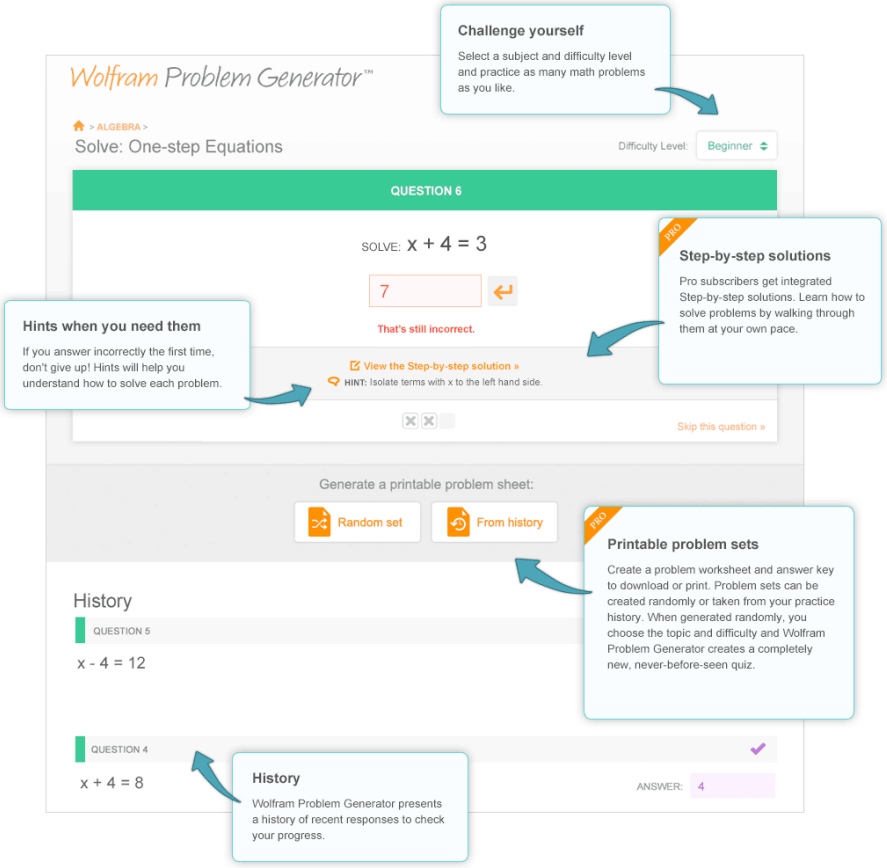


Obrázek 1.1: Náhled příkladu vygenerovaného programem WeBWorK Project [6]

cílila na konkrétní potřeby studentů, což by mělo ve finále vést k jejich zlepšení [7], [8].

1.2.2 Wolfram Problem Generator

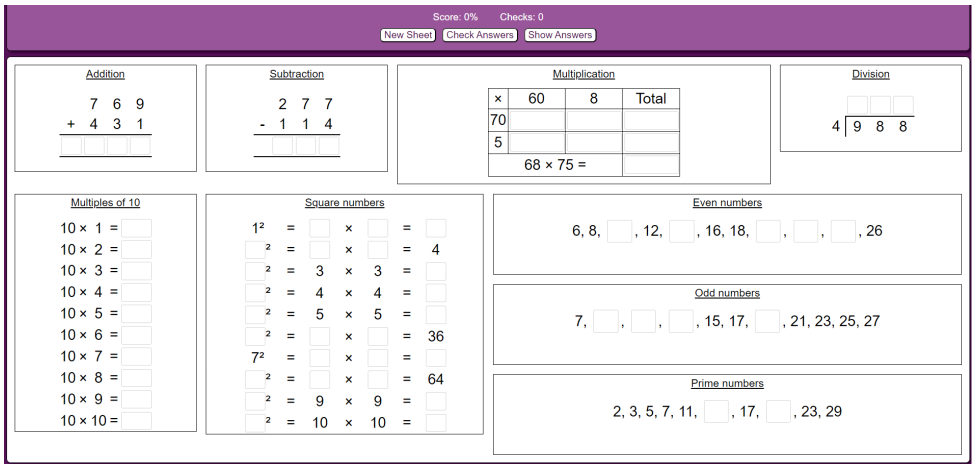
Wolfram Problem Generator je integrovaná část Wolframu Alfa určená ke generování libovolných typů úloh z nabídky a online procvičování. Témata jsou členěna na aritmetiku, číselnou teorii, algebru, kalkulus, lineární algebru a statistiku. Každé z témat je dále vnitřně děleno na dílčí podkapitoly, například v případě aritmetiky na celá čísla, pravidla operací, zlomky a závěrečné shrnutí. Tyto podkapitoly nabízejí široký rozsah typových úloh, z nichž si uživatel může vybrat a nechat vygenerovat konkrétní příklady a k nim i příslušná řešení. V případě zakoupení prémiové verze je možné vytvořit pracovní list uzpůsobený k tisku, či instantně vygenerovat kvíz obsahující žádané úlohy a to buď náhodně generované, či z historie. Je též umožněno zobrazení řešení úloh takzvaně krok po kroku. Dále je k dispozici pro všechny uživatele modifikace obtížnosti a poskytnutí nápovědy. Wolfram Problem Generator však zatím nenabízí slovní úlohy a jím nabízená témata pracují pouze s numerickým a poněkud strohým zadáním [9].



Obrázek 1.2: Ilustrační náhled k porovnání možností bezplatné a prémiové verze Wolfram Problem Generatoru [6]

1.2.3 MathsBot

MathsBot je internetový matematický generátor testů s rozmanitou možností vlastní modifikace. V sekci Question Generator je uživatel dotázán, kolik tématických skupin od 1 do 9 chce vygenerovat a kolik příkladů od 1 do 25 má každá skupina obsahovat. Dále je pro každou skupinu možno vybrat z vláken čísla, algebra, geometrie a měření, poměry a proporce, pravděpodobnost a statistika. Například vlákno algebra nabízí témata jako jsou různé typy rovnic, výrazů a jejich úprav, nerovnice, či posloupnosti. U každé skupiny lze zvolit z obtížností 0 až 10 a pro celý test zvolit časový limit. Poté si může uživatel nechat test vygenerovat. Získané příklady jsou pro dané téma a obtížnost až na numerické hodnoty zcela identické. Například téma Pravděpodobnost při obtížnosti 5 vždy využívá zadání s férovou šestistěn-



Obrázek 1.3: Náhled vygenerovaného testu generátorem MathsBot [10]

nou kostkou, kdy je uživatel tázán vypočítat pravděpodobnost, že padne jedno z vybraných šesti čísel. Tudíž je generátor značně limitován nepříliš velkým rozsahem parametrické sbírky [11].

1.3 Knižní matematické sbírky

Knižní matematické sbírky příkladů jsou kolekce příkladů opatřené klíčem se správným řešením. Jejich hlavní ideou je poskytnou studentům rozsáhlý výběr různorodých příkladů k procvičování, na nichž si mohou studenti upevnit své znalosti.

1.3.1 Učebnice Petáková

Jedná se o knihu obsahující 4500 úloh ze středoškolské matematiky určené k opakování a procvičování učiva k maturitě a přijímacím zkouškám na vysoké školy. Úlohy jsou členěny dle tématických celků do dvaceti kapitol a to: základní poznatky o výrocích a množinách, základní typy rovnic a nerovnic, rovnice s parametrem, funkce, exponenciální a logaritmické rovnice a nerovnice, goniometrické funkce a trigonometrie, goniometrické rovnice a nerovnice, mocninné funkce, lineární lomená funkce, posloupnosti a řady, geometrie - konstrukční úlohy, geometrie - výpočty, stereometrie, vektory, analytická geometrie v rovině, analytická geometrie v prostoru, kuželosečky, komplexní čísla, kombinatorika a binomická věta, diferenciální počet a integrální počet, pravděpodobnost a statistika. Každá kapitola obsahuje různé

typové příklady se stupňující se obtížností, jejichž zvládnutí umožní studentům pochopit probíranou látku. Na závěr sbírka obsahuje výsledky úloh ke kontrole a v některých složitějších případech jsou uvedeny i stručné návody k řešení. Úlohy jsou kromě rozdělení do výše zmíněných kapitol uspořádány tak, aby vyučující mohl snadno pro své studenty vybrat ty, které jsou adekvátní jejich typu třídy a zvolené budoucí vysoké škole. Každá úloha nabízí několik variant, aby měli studenti větší prostor si je procvičit a učitel měl širší výběr příkladů k testu. V závěru knihy jsou uvedena zadání písemných přijímacích zkoušek některých vysokých škol, podle nich pak může učitel žákům doporučit příslušné kapitoly dle vybrané vysoké školy. Vzhledem k široké škále obtížnosti úloh, jež se stupňuje v každé kapitole s přibývajícími příklady, je sbírka mnohdy využívána jako základní materiál pro matematické semináře, na nichž se třídy připravují k maturitě, popřípadě jako nadstavba k semináři základů vyšší matematiky, kde je potřeba, aby studenti dosáhli o něco vyšší úrovně, než stihnou v rámci běžných hodin matematiky [12].

1.3.2 Učebnice Běloun

Jedná se o knižní sbírku úloh z matematiky pro základní školy. Studentům nabízí rozmanitou škálu úloh k procvičování a opakování učiva základní školy a je vhodná k přípravě na přijímací zkoušky na střední školy [13].

1.4 Výukové portály

Výukové portály jsou webové stránky zaměřené na výuku ve formě videí, skript, diskuzního fóra, či kombinovanou formou.

1.4.1 Isibalo

Isibalo je výukový portál zaměřený na výuku základoškolské, středoškolské a vysokoškolské matematiky, chemie, fyziky a biologie pod vedením zakladatele Dominika Chládka. Portál začínal se zaměřením na matematická videa, postupně byl ale rozšiřován o další předměty s příchodem nových spolupracovníků. Z hlediska matematiky web v současné době nabízí rozsáhlou škálu videí, kde je většinou v 5 až 20 minutách rozebráno aktuální téma a spočítáno několik typových příkladů. Videa jsou členěna do tématických celků dle jednotlivých kapitol a seřazena dle školní návaznosti. Aktuálně je z matematiky k dispozici 41 kapitol, jež v součtu nabízejí přes 1000 videí a přes 1400 řešených příkladů, ostatní předměty nabízejí zhruba desetinu

tohoto množství. V sekci *Příklady od Vás* mohou studenti vkládat příklady, se kterými si neví rady. Zároveň mohou pomoci s řešením příkladů ostatních uživatelů. Pro každý předmět nabízí zdarma výuková videa, prémiová verze navíc zpřístupňuje řešené příklady a testy. Isibalo plánuje do budoucna zařadit na svůj portál další předměty, zejména jazyky [14].

1.4.2 Mathematicator

Mathematicator je výukový portál zaměřený na výuku základoškolské, středoškolské a vysokoškolské matematiky pod vedením Marka Valáška, lektora matematiky, fyziky a statistiky. Web nabízí kurzy pro všechny stupně studia, zakoupit lze i pouze jednotlivá témata. Kurz *Základní škola* nabízí především přípravu z matematiky k přijímacím zkouškám na střední školy. *Kurz Střední škola* nabízí jednotlivá témata středoškolské matematiky, přípravu k maturitě a přípravu k přijímacím zkouškám na některé vysoké školy. Kurz *Vysoká škola* pak nabízí jednotlivá témata vysokoškolské matematiky. Bezplatně je pro všechny uživatele k dispozici přes 500 videí týkajících se povětšinou matematiky středních a druhého stupně základních škol. Dále je na webu k dispozici fórum, kde mohou všichni uživatelé nahrávat příklady a problémy, se kterými si neví rady, a kdokoli jim může s řešením pomoci [15].

1.4.3 Online School

OnlineSchool je výukový portál zaměřený na výuku matematiky, fyziky, statiky, termomechaniky, pružnosti a pevnosti. Zde matematika pokrývá témata středoškolské matematiky a základy matematické analýzy. Fyzika nabízí podrobně témata od kinematiky po magnetismus, zbývající předměty jsou shrnuty v kratších celcích. Web nabízí výuku formou mikropřednášek a formou videosbírek. Forma Mikropřednášek poskytuje stručné shrnutí základních principů dané látky ve videích o přibližné délce 15 minut. Forma videosbírek pak poskytuje sbírky řešených úloh, k nimž je k dispozici řešení s postupem na videu. Portál dále nabízí sekci Maturita, kde je uživatelům k dispozici okolo 380 řešených příkladů ve 24 hodinách videí a vysvětlení teorie a principů témat středoškolské matematiky, jež je potřeba znát k úspěšnému zvládnutí maturity z matematiky [16].

1.4.4 Matematické Fórum

Matematické Fórum je rozsáhlá webová komunita dobrovolníků, která aktivně přispívá konzultacemi matematických problémů, o jejichž pomoc s ře-

šením může kterýkoli registrovaný účastník požádat. V současné době web registruje přes 34 000 uživatelů a v součtu přes 610 000 příspěvků. Pro lepší orientaci jsou vytvořeny sekce: Základní škola, Střední škola, Vysoká škola: úvod do studia, Vysoká škola: pokročilá matematika, Didaktika a pedagogika: metodiky výuky a výukové materiály, Ostatní, Projekty a příklady z Diskrétní matematiky, a dalších sekce zaměřené na specifičtější problémy. Ze strany administrátorů webu je kladen požadavek, aby žadatel o radu uvedl kromě zadání i vlastní úvahu, či pokus o postup, a naopak na poskytovatele rady, aby pouze napověděli, ale neposkytovali rovnou celé řešení [17].

1.4.5 Moodle

Moodle je výuková webová aplikace poskytující široké možnosti dálkového studia a domácí přípravy na výuku. Jedná se o specializovaný content management system. Instalace CMS systémů obsahují funkce pro snadné vytváření a nahrávání obsahu bez znalosti jazyka HTML, CSS stylů a dalších souvisejících technologií. Vyučující v Moodlu proto nemusí rozumět tvorbě webů. Výuka je rozčleněna do jednotlivých kurzů, do nichž vyučující může umisťovat studijní materiály, přičemž lze nastavit i přesný čas jejich zveřejnění. Dále aplikace podporuje nahrávání vyřešených úkolů ze strany studentů, či vytvoření ankety. K širší komunikaci je k dispozici diskuze či chat. Účastníky lze rozdělit do skupin a pro každou skupinu přizpůsobit výuku zvlášť. Dále je možné vytvořit test, u nějž je měřen čas, vyhodnotit odpovědi a případně provedena automatická klasifikace. Samotný kurz podporuje několik rolí s příslušnými právy, to vše je zcela volně modifikovatelné. Hierachie těchto rolí je správce, tvůrce kurzu, učitel, učitel bez práva upravovat, student, host a registrovaný uživatel [18].

Kapitola 2

Konceptuální návrh

V úvodu kapitoly je rozebrána obecná myšlenka celého projektu a jeho cílů. Následně jsou nastíněny jednotlivé části předpokládaného konceptu řešení a jejich příslušná specifika. Dalším bodem je popis očekávaného výstupu. Na závěr kapitoly je představen konceptuální návrh, který bude později využit k tvorbě konečného návrhu a jeho následné implementaci.

2.1 Obecná myšlenka projektu

Cílem celé práce je příprava parametrické sbírky a generátoru matematických úloh pro podporu výuky matematiky, popřípadě i dalších přírodovědných předmětů jako je fyzika, chemie, atd. Sbírka by měla být realizována jako množina parametrických struktur, které reprezentují jednotlivé úlohy. Příklady ze sbírky mají následně sloužit jako obecné šablony pro generátor úloh, který bude umět vybranou strukturu konkretizovat. Konkretizace jednotlivých úloh může mít různé podoby v závislosti na typu úlohy. Společným postupem by mělo být dosazení náhodně generovaných hodnot na místa předepsaných parametrů. Výběr náhodných hodnot by měl být omezen na předem definovaný interval či množinu tak, aby se zamezilo absurdním kombinacím vygenerovaných hodnot.

Konceptuální návrh řešení se tak omezuje na situaci, kdy je vyžadováno vytvoření zkušebního testu pro skupinu studentů. Zkouška tedy bude generována jako soubor jednotlivých testů, kde počet testů bude odpovídat počtu zúčastněných studentů a každý test bude odpovídat nastavení dle zvolené konfigurace úloh. Každé dva testy z vytvořeného souboru budou stejné co do typu a množství příkladů, avšak díky randomizaci parametrů se každá z úloh v rámci statistiky numericky liší. Celý návrh tedy pracuje

s obecným přístupem, kdy máme n studentů, kteří se budou účastnit testu. Dále máme celkový počet k problémů, jež chceme, aby se v každém testu objevily. Jednotlivé typové příklady by mělo být možné v rámci každého testu použít i vícekrát s tím, že každý z příkladů bude odlišný z hlediska numerických hodnot.

2.2 Předpokládané součásti projektu

V rámci realizace projektu se očekává, že bude implementována parametrická sbírka a generátor příkladů. Oba dva moduly jsou popsány v následujících bodech.

2.2.1 Parametrická sbírka

Parametrická sbírka je množina všech typových úloh, které je možné poskytnout generátoru příkladů. Strukturu a parametrizaci každé typové úlohy je nutné nejprve manuálně vytvořit a vložit ji do této sbírky příkladů. Celou sbírku by mělo být možné rozdělit na jednotlivé sekce dle typů problémů a ty pak případně dále dělit na základě specifičnosti daných úloh. Jediným limitujícím faktorem, pro variabilitu celé sbírky, je počet typových úloh, jež budou připraveny. Nicméně toto omezení je naprosto přirozené, neboť se uplatňuje na jakoukoliv existující sbírku úloh, zejména pokud se jedná o knižní provedení.

2.2.2 Generátor příkladů

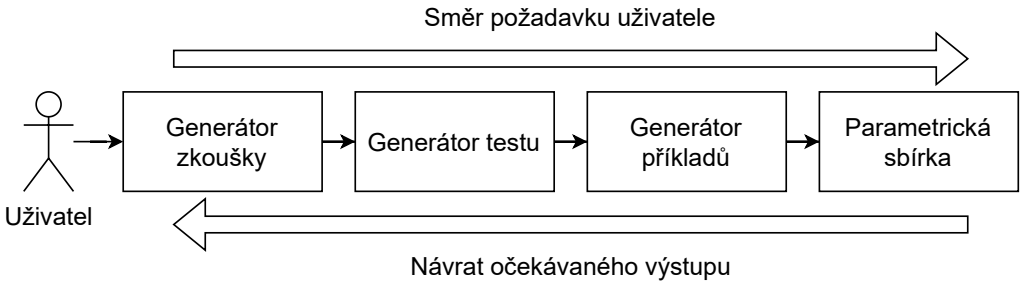
Generátor příkladů má na základě zvolené konfigurace testu vybírat vhodné úlohy z parametrické sbírky. Jednotlivé příklady jsou přitom konkretizovány ve svých parametrech dosazením generovaných hodnot dle příslušných pravidel. Z takto získaných příkladů je následně sestaven test.

Zvolený návrh generátoru by měl umožnit vytvoření libovolného počtu numericky rozdílných úloh ze stejného typového příkladu.

2.3 Konceptuální návrh struktury řešení

Návrh uvažuje strukturu, v níž bude propojen modul parametrické sbírky s generátorem příkladů a která bude navenek využita k vytváření příkladů pro jednotlivé testy připravované zkoušky. Jako koncept řešení se nabízí zřetězení všech těchto částí do jednoho celku. Pro usnadnění interakce mezi jednotlivými vrstvami budou do řešení zahrnuty generátory jednotlivých

částí. Na nejnižší vrstvě bude samotná sbírka, z ní bude generátor příkladů vybírat jednotlivé úlohy, které parametrizuje a poskytne je do vyšší úrovně. Následovat bude test, který bude přijímat výsledky generátoru tak, aby se naplnil požadovaným množstvím příkladů. Jednotlivé testy pak budou vytvářeny pomocí generátoru testů, který bude připojen k modulu reprezentujícímu zkoušku. Samotná zkouška pak bude připojena k uživatelskému rozhraní. Konceptuální návrh popsané struktury je prezentován schématem 2.1.



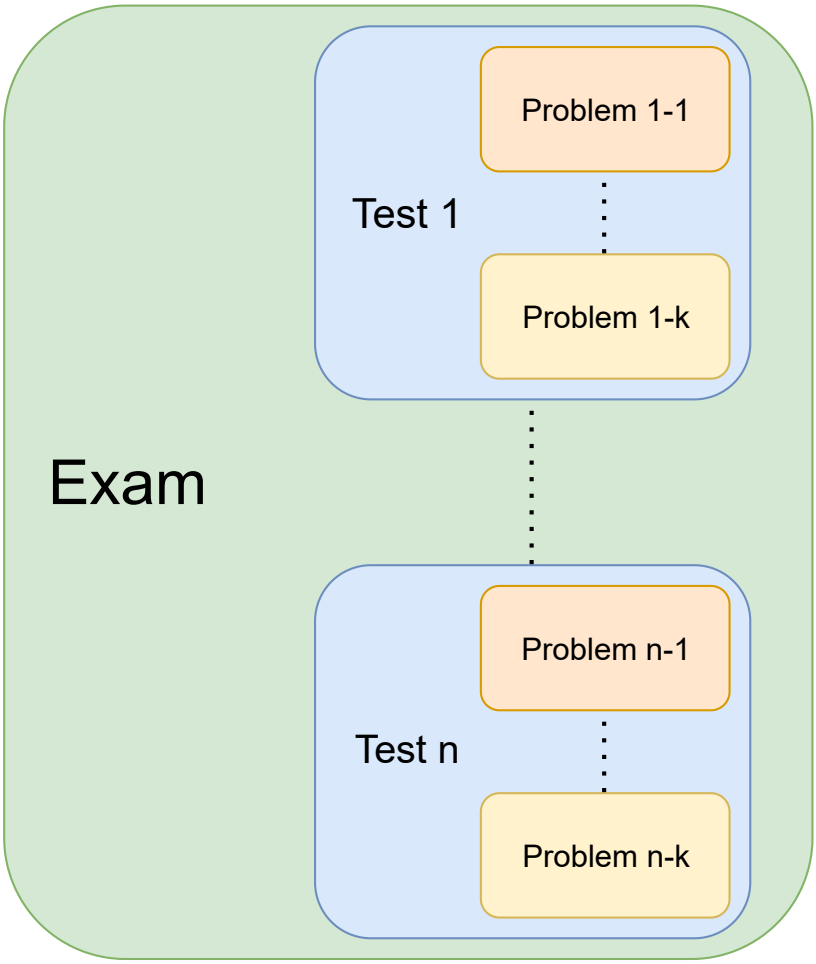
Obrázek 2.1: Konceptuální návrh generování zkoušky.

Výhoda navržené struktury je ta, že na každé úrovni existuje jednoduché a přesně definované rozhraní pro přístup do nižší úrovně. Vše, co je potřeba v dané úrovni vědět, je to, který z generátorů použít. Veškerá zbylá zodpovědnost je pak přenesena do nižších úrovní.

2.4 Očekávaný výstup projektu

Plánovaným výstupem celého projektu by měly být na míru vygenerované zkouškové testy. Obecným výstupem je tedy jedna konkrétní realizace zkoušky s k příklady pro každého z n studentů. Pro zjednodušení návrhu je vhodné si separovat jednotlivé části a provést rozklad celku na menší podproblémy. Na zkoušku je možné nahlížet jako na soubor n testů. Obdobným způsobem lze dekomponovat každý z těchto testů na soubor k příkladů. Obecně pak můžeme považovat jednotlivé příklady za specifické zástupce téže entity vybrané úlohy, přestože se mezi sebou liší. Stejným způsobem je možné přistoupit i k jednotlivým testům, které jsou zástupci obecné entity reprezentující test. Zjednodušeně řečeno, jednotlivé testy budou zadáním typově stejné, nicméně každý z příkladů by měl mít odlišné číselné hodnoty.

Níže popsaná dekompozice očekávaného výstupu, která je reprezentovaná schématem na Obrázku 2.2, výrazně zjednoduší komplexnost návrhu finálního řešení. V implementační části bude díky tomu možné zaměřit větší pozornost na vlastnosti jednotlivých entit bez zbytečně složitého uvažování o vazbě na širší kontext.



Obrázek 2.2: Schéma navrhovaného výstupu, kde jednotlivé problémy jsou opatřeny dvěma indexy. Zatímco první index nabývá hodnot od *1* do n a určuje numerickou variantu dané úlohy dle příslušného testu. Druhý index nabývá hodnot od *1* do k a představuje pořadové číslo dané úlohy.. Konkrétní ukázku dvou testů je možné vidět v Příloze C.

Kapitola 3

Analýza dostupných nástrojů a programů

V této kapitole jsou popsány všechny důležité nástroje a programy, jež byly potřeba k této práci, a to jak z hlediska jejich obecného využití, tak i z hlediska pro ně specifické syntaxe.

3.1 Regulární výrazy

Regulární výrazy (Regular Expressions) jsou sekvencí znaků definující vzor shody ve sledovaném textu. Specifikací regulí, tj. podmínek a vzorů, jimiž musí hledaný řetězec disponovat, lze text následně prohledat a nalézt všechny případy výskytu daného regulárního výrazu. Časté využití nalézá v programování, textových editorech či internetových vyhledávačích.

Regulární výrazy nabízí efektivní způsob pro vyhledávání v textu a manipulaci s ním. K definování vzoru či manipulaci s textem je potřeba použít syntaxi regulárních výrazů, jež využívají různé znaky a operátory. Každý znak v regulárních výrazech je buď metacharacter (metaznak), který má speciální význam, nebo regular character (běžný znak), který má doslovný význam. Jejich kombinací pak lze definovat vzor shody vyžadující naprostou rovnost vzoru a hledaných výrazů, či pouze obecnou podobnost, jejíž parametry lze široce specifikovat [19].

3.1.1 Některé metaznaky a jejich význam

Metaznaky jsou znaky, neboli speciální symboly, jež mají specifický význam a funkci pro různé programy [19].

- Znak `|` odděluje alternativy. Například výraz `example1|example2` splňuje jak řetězec `example1` i `example2`.
- Znak `^` odpovídá počáteční pozici v textu. Například `^example` odpovídá pouze takovému řetězci, kde se `example` se objevuje na jeho začátku.
- Znak `$` odpovídá koncové pozici v textu. Například `example$` odpovídá pouze takovému řetězci, kde se `example` objevuje na jeho konci.
- Znak `.` (tečka) odpovídá libovolnému znaku. Například `exam.le` odpovídá každému řetězci, kde se na místě `.` nachází právě jeden libovolný znak.
- Dvojice závorek `[]` odpovídá právě jednomu znaku, ze znaků, jež jsou v ní uvedeny. Například:
  - a) `[123a-zA-Z]` odpovídá právě jednomu libovolnému malému, či velkému písmenu v rozsahu `a/A` až `z/Z`, nebo některému z čísel: `1, 2, 3`. Znak `-` zde znamená rozsah, určuje ho od a až po Z.
  - b) `[abc]` odpovídá právě jednomu znaku, tedy buď `a`, `b`, nebo `c`.
- Znak `[^ ]` odpovídá jednomu z neuvedených znaků. Například `[^A-Za-z]` odpovídá právě jednomu libovolnému znaku, jež není malým ani velkým písmenem. Tato syntaxe funguje zcela antagonisticky vůči té předchozí.
- Kvantifikátor `?` odpovídá žádnému, nebo jednomu výskytu předchozího znaku. Například `e?xample` odpovídá řetězcům: `xample`, `example`.
- Kvantifikátor `*` odpovídá žádnému či více výskytům předchozího znaku. Například `e*xample` odpovídá řetězcům: `xample`, `example`, `eexample`, atd.
- kvantifikátor `+` odpovídá jednomu či více opakováním předchozího znaku. Například `example+` odpovídá řetězcům: `example`, `examplee`, `eexampleee`, atd.
- Kvantifikátor `{}` určuje počet opakování předchozího výrazu. Například:
  - Výraz `{m}` hledá právě m počtu opakování výskytu předchozího výrazu. Například `example{2}` odpovídá řetězci `exampleexample`.

  - Výraz `{m,n}` hledá *m* až *n* počtu opakování výskytu předchozího výrazu, kde *m* je minimální a *n* maximální počet výskytů. Například `example{1,3}` odpovídá řetězcům: `example`, `exampleexample`, `exampleexampleexample`, atd.
  - Výraz `{m,}` odpovídá *m* a více počtu opakování předchozího výrazu. Například `example{2,}` odpovídá řetězcům: `exampleexample`, `exampleexampleexample`, atd.

- Znak \ umožňuje vyhledat libovolný metaznak. Například `\*` umožňuje nalézt výskyt znaku `*`.

- Závorky `()` specifikují část vzoru na nějž se dané operátory vztahují. Například `(exa)*mple` odpovídá řetězcům: `mple`, `example`, `exaexample`, atd.

3.1.2 Capture groups a non-capturing groups

Groups (česky skupiny) jsou v regulárních výrazech části vzoru shody uzavřené v kulatých závorkách. *Capture groups* pak při prohledávání řetězce nalezené shody ukladá a přiřazuje jim ID od *0* do *n-1*, kde *0* je index první a *n-1* index n-té shody. *Non-capturing groups* stejně jako *groups* hledají shody dle zadaného vzoru, avšak oproti předchozímu případu nalezenou shodu nevracejí. [19].

- Výraz `(?:...)` je non-capturing group, umožňující aplikovat kvantifikátory na část regulárního výrazu, bez toho, aby mu přiřadil ID. Například `text (?:example)` aplikovaný na řetězec `text example` nalezne shodu, nicméně nic nevrátí.

- Výraz `(...)` izoluje část celé shody regulárního výrazu a přiřadí jí ID v poli nalezených skupin. Například `(example)+` aplikovaný na řetězec `example_text_example` nalezne v řetězci slovo `example` dvakrát. Oba výsledky tedy zahrne do stejné skupiny s různými ID shody.

- Výraz `(?#...)` ignoruje jakýkoli text v této groupě. Například `exa(?#mple)` aplikovaný na řetězec `example` vrátí jako shodu `exa`.

- Výraz `(?<name>...)` je capturing group, umožňující odkazovat se na zadané `name`, místo příslušného čísla. Například `(?<name>example)` aplikovaný na řetězec `text_example` vrátí groupu se jménem `name`.

3.1.3 Aplikace regulárních výrazů v programovacím jazyku Python

Regulární výrazy mají velké využití i v programovacím jazyku Python, kde je lze snadno implementovat jako knihovnu *re*, jejímž importováním do skriptu pomocí commandu `import re` je zajištěna podpora regulárních výrazů. Toho pak například využívají metody `result = re.search(pattern, string)` a `result = re.findall(pattern, string)`, *pattern* je vzor shody a *string* je řetězec, na který jednu z metod chceme zavolat. První z metod prohledá řetězec a v případě úspěchu vrátí první nalezenou shodu jako výsledek. Druhá z metod prohledá řetězec a v případě úspěchu vrátí jako výsledek seznam všech nalezených shod. Jak je vidět v kódu (3.1) u konkrétního příkladu, metoda *re.findall* prohledá zadaný řetězec (konkrétně hledá všechny části řetězce složené z jednoho či více číslic). Jako shody jsou nalezeny čísla *12*, *89* a *34*. Ty jsou jako seznam uloženy do proměnné *result*. Ten byl následně vypsán pomocí metody `print()`

Metoda *re.compile* umožňuje kompilovat vzor shody regulárního výrazu z poskytnutého řetězce do objektu *re.pattern*. Ten pak lze využít k hledání shody v dalších cílových řetězcích pomocí zmíněných metod, aniž by bylo potřeba vzor shody přepisovat. Metodu *re.compile* lze zavolat zápisem `re.compile(pattern)` [20] [21].

```
1  # Program to extract numbers from a string
2  import re
3
4  string = 'hello 12 hi 89. Howdy 34'
5  pattern = '\d+'
6
7  result = re.findall(pattern, string)
8  print(result)
9
10 # Output: ['12', '89', '34']
```

Kód 3.1: Příklad použití Python modulu *re* pro hledání čísel v textu pomocí regulárního výrazu [22]

3.1.4 Základní patterny

Základní patterny v regulárních výrazech jsou obecně definované vzory, díky nimž lze zjednodušit a zkrátit zápis požadovaného vzoru [19].

- Znaky `0-9a-zA-Z` patří mezi běžné znaky a představují samy sebe bez dalšího významu.
- Výraz `.` (tečka), tečka představuje každý znak krom nové řádky.
- Výraz `\d` odpovídá libovolné cifře od 0-9.
- Výraz `\w`, takzvaný *word character*, odpovídá právě jednomu ze znaků z nabídky [a-zA-Z0-9_].
- Výraz `\W`, takzvaný *non-word character*, je opakem předchozího případu a odpovídá jakémukoli jinému znaku než je v nabdídce [a-zA-Z0-9_]. Alternativně ho lze tedy zapsat jako [^a-zA-Z0-9_].
- Výraz `\b` je vazba mezi znakem ze skupiny *word character* a znakem ze skupiny *non-word character*. Například `\bEXAMPLE` by mohl splňovat řetězec `text EXAMPLE`, kde mezera patří mezi znaky ze skupiny *non-word character* a `EXAMPLE` je část řetězce složeného ze znaků ze skupiny *word character*. Typické použití je při hledání celých slov.
- Výraz `\s` odpovídá libovolnému bílému znaku. Mezi bílé znaky patří například mezera, nový řádek, return a tab.
- Výraz `\S` odpovídá libovolnému nebílému znaku.
- Výrazy `\t`, `\n` a `\r` odpovídají tabulátoru, novému řádku a returnu.
- Znaky `^` a `$` odpovídají začátku a konci řetězce.

3.2 Objektově orientované programování

Objektově orientované programování je přístup programování organizující návrh softwaru vzhledem k datům či objektům, se kterými má být manipulováno. Zde *objekt* je datové pole s danými atributy a chováním. Výhodou je pak snazší a efektivnější práce a údržba rozsáhlých a komplexních programů. Organizace objektově orientovaného programu dělením projektů do skupin činí metodu vhodnou pro kolaborativní vývoj. Při práci v objektově orientovaného programování je nejprve potřeba definovat potřebné objekty a jejich vzájemnou komunikaci. Známý objekt je následně označen třídou objektů, která definuje jaký druh dat objekt obsahuje, a metody, které s ním mohou manipulovat. Základní datové typy jsou *čísla*, *řetězce*, *seznamy*, *n-tice* a *slovníky*. Logickým sekvencím, popř. funkcím, které patří danému objektu se říká metody. Další výhodou objektově orientovaného programování je krom

snadné modifikace a bezpečnosti kódu například *zapouzdření* (viz Subsekce 3.2.2), pomocí kterého lze lépe nalézt a opravit vzniklé problémy. Dále je možné díky *dědičnosti* (viz Subsekce 3.2.2) snadno opakovaně použít vytvořenou, a již funkční část kódu a předejít tak jeho duplikaci [23], [24], [25].

3.2.1 Struktura objektově orientovaného programování

Základními stavebními bloky objektově orientovaného programování jsou třídy, objekty, metody a atributy [23], [24], [25].

- Třídy jsou šablony kódu k vytváření objektů, atributů a metod.
- Objekty jsou instance třídy se specificky definovanými daty. Objektem je v Pythonu prakticky vše od čísel až po seznamy, slovníky či n-tice.
- Metody jsou funkce definované uvnitř třídy a popisují chování daného objektu. Odkazuje se na ně jménem, jímž je lze zavolat v libovolném bodě programu.
- Atributy jsou proměnné definované uvnitř třídy. Mohou být třídní a instanční. Třídní atributy popisující třídy a jsou sdílené všemi jejími instancemi na rozdíl od atributů instancí, které jsou vlastněny pouze jednou konkrétní instancí třídy.

3.2.2 Hlavní principy objektově orientovaného programování

Hlavními principy objektově orientovaného programování je *zapouzdření*, *abstrakce*, *dědičnost* a *polymorfismus* [23] [24] [25].

- *Zapouzdření* (angl. *encapsulation*) zajišťuje, že všechna důležitá data jsou uchována uvnitř objektu a k expozici dochází jen u vybraných informací. Implementace každého objektu je zcela uzavřena uvnitř třídy a ostatní objekty k nim nemají přístup, čímž je zajištěna větší bezpečnost a přehlednost programu.
- *Abstrakce* (angl. *abstraction*) zajišťuje, že objekty odhalují pouze tu část vnitřních mechanismů, která je podstatná k použití dalšími objekty.

- *Dědičnost* (angl. *inheritance*) zajišťuje dědičnost částí kódu, tam kde by jinak bylo nezbytné duplikovat kód. Třída, která pak dědí z jiné třídy, používá stejné základy, ale definuje si rozdílné výstupy.

- *Polymorfismus* (angl. *polymorphism*) zajišťuje, aby objekty mohly mít více než jednu formu. Program na základě stanovených parametrů určí, jaké využití je potřebné ke spuštění daného objektu z nadřazené třídy, jejíž funkčnost je rozšířená podřízenými třídami.

3.2.3 Objektově orientované programování v Pythonu

Syntaxe programovacího jazyka Python je soubor pravidel definující, jakým způsobem může být program napsán a následně interpretován. V porovnání s většinou ostatních známých programovacích jazyků Python využívá relativně jednoduchou a čitelnou syntax v podobě anglických klíčových slov tam, kde by jiné jazyky použily symboly. Cílem je, aby byl návrh syntaxe co nejjednodušší a nejkonzistentnější (viz Kód 3.2). Objektově orientované programování nachází v programovacím jazyce Python široké uplatnění. I zde se využívá všech dříve specifikovaných stavebních bloků (viz Subsekce 3.2.1) a stejně tak i všech popsaných principů (viz Subsekce 3.2.2). Třídu je vhodné použít, očekává-li se práce s více objekty stejného chování. Lze je vytvořit zápisem `class ClassName:`, kde *ClassName* je libovolný název třídy. V momentě vytvoření třídy je vytvořen i objekt třídy nesoucí všechny atributy dané třídy. Následně lze definovat dodatečné objekty třídy, zvané instance. Každý objekt třídy je sestaven z atributů, které určují jeho stav, z metod, jež definují jeho chování vůči jiným objektům, a z identity či jména. Chceme-li přiřadit proměnným a instancím dané třídy konkrétní hodnoty (nejen čísla) v momentě, kdy je objekt třídy vytvořen, můžeme zadefinovat metodu `__init__`, která se spustí ve chvíli, kdy je vytvářena instance objektu třídy [26], [27].

Kód 3.3 prezentuje základní syntaxi třídy s konstruktorem. Nejprve byla vytvořena třída *Person*, metoda `__init__` následně umožní přiřadit proměnné *name* v argumentu libovolný řetězec, není-li tento argument uživatelem stanoven, je použit řetězec *Mr. Incognito*. Proměnná `self.name = name` pak přiřadí v konstruktoru daný řetězec vzniklému objektu. Dále je definována metoda třídy *introduce()*, která při svém zavolání vytiskne formátovaný řetězec, kde je za proměnnou *self.name* dosazen řetězec uložený v konstruktoru. Při spuštění programu tedy příkaz `Person().introduce()` vytvoří objekt třídy Person, který má jako hodnotu proměnné *name* řetězec

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better~ than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way
↪ to do it.
Although that way may not be obvious at first unless you're
↪ Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good
↪ idea.
Namespaces are one honking great idea -- let's do more of
↪ those!

```

Kód 3.2: Filozofie Pythonu

Mr. Incognito, to je uloženo a při zavolání metody *introduce()* se vytiskne řetězec *Hello my name is Mr. Incognito.*

3.2.3.1 Shrnutí základní syntaxe tříd v Pythonu

Ideou programovacího jazyka Python je cílit na jednoduchost. Toho lze dosáhnout například použitím anglických výrazů pro různé odpovídající objekty, čímž je práce s nimi přirozenější [27], [28].

- `class ClassName:` je syntax k vytvoření třídy se jménem `ClassName`.

```
1  class Person:
2
3      def __init__(self, name="Mr. Incognito"):
4          self.name = name
5
6      def introduce(self):
7          print(f"Hello my name is {self.name}")
8
9
10 person = Person()
11 person.introduce()
12
13 # Output: Hello my name is Mr. Incognito
14
```

Kód 3.3: Ukázka jednoduché třídy s metodou `__init__`

- `class Child(Parent):` je syntax k nastavení dědičnosti ze super třídy *Parent* do třídy *Child*. Třída *Child* tak může používat všechny metody a vlastnosti třídy *Parent*, dá se tedy říci, že je podědila. Stejně jako v biologii je dědičností ovlivněno potomstvo, i v Pythonu dědičnost ovlivňuje pouze třídu, jež dědí a tudíž logicky na supertřídu *Parent* nemá vliv.
- `__init__` je metoda reprezentující konstruktor třídy. Zavoláním *ClassName()*, je vytvořen objekt, který je přijat jako první parametr metody `__init__`, další argumenty pak lze v rámci dané třídy doplnit dle jejího navržení.
- `def __init__(self):` je syntax, jímž je vytvořen konstruktor třídy.
  - Běžný konstruktor nepřijímá žádné argumenty, kromě *self*.
  - Parametrizovaný konstruktor přijímá v argumentech nejprve *self*, poté další libovolné argumenty. Ty je pak potřeba svázat s daným objektem zápisem `self.some_object = value`, kde *some_object* je jméno argumentu a *value* jemu přiřazená hodnota. Ta nemusí být numerická, klidně se může jednat o řetězec, či jiný datový typ.
- `self` je proměnná představující instanci třídy a umožňuje přístup k metodám a atributům třídy.

3.3 JSON

JSON nebo-li JavaScript Object Notation (česky: JavaScriptový Objektový Zápis), je textový formát pro výměnu dat. Jeho velkou výhodou je to, že je relativně dobře čitelný pro člověka. JSON je nezávislý na programovacím jazyce, ale současně využívá syntaxi podobnou mnohým z nich, včetně Pythonu. Dále umožňuje ukládat textové řetězce, čísla, boolovské hodnoty, reprezentaci hodnoty *null*, pole a objekty. Ostatní datové typy je potřeba převést na jeden z výše zmíněných. Data JSONu mohou být reprezentována následujícími způsoby [29]:

- Kolekce párů ve tvaru `key: value` syntaxí podobná slovníku v Pythonu, kde *key* je klíč a *value* jemu přiřazená hodnota. Je-li v kolekci více párů, jsou odděleny čárkou. Celá kolekce párů je definována mezi závorkami `{` a `}`. Kolekce o více párech pak může vypadat následovně `{key1: value1, key2: value2, key3: value3,....}` pro libovolný počet párů `key: value`.
- Uspořádaný seznam hodnot oddělených čárkou, který je syntaxí podobný seznamu v Pythonu. Seznam více hodnot pak může vypadat následovně `[value1, value2, value3,....]` pro libovolný počet hodnot.

```
1 {
2     "student_id": 42,
3     "name": "Alice",
4     "points": 9
5 }
```

Kód 3.4: Ukázka dat v JSON formátu

3.4 Jinja2

Jinja2, inspirovaný šablonovacím systémem Djanga, je rychlý a flexibilní nástroj pro vytváření šablon pro programovací jazyk Python, jemuž podobnou syntaxi využívá. Jinja2 je využíván k tvorbě formátů HTML a XML. Šablona je tvořena proměnnými a výrazy, za něž jsou v momentě vykreslení šablony dosazeny konkrétní hodnoty, což je vhodné k práci s dynamickými daty. Jako vstupní data jsou přijímány seznamy, n-tice a slovníky. Důležitá

je též podpora dědičnosti šablon, čímž lze ušetřit práci podobně jako u dědičnosti tříd v Pythonu. Chceme-li například u webových dokumentů použít stejné rozložení pro různé stránky, dědičnost šablon přijde vhod. V případě potřeby lze poděděnou šablonu pro každý z případů upravit, aniž by bylo potřeba ji zcela tvořit od základů. Jako šablonu lze načíst libovolný soubor, nehledě na příponu [30].

3.4.1 Oddělovače

Oddělovače jsou znaky či skupiny znaků umožňující označit začátek a konec textového řetězce [30] [31].

- Oddělovač `{%....%}` je výraz používán k zápisu tvrzení. Například *for* smyčka nebo podmínkový blok.
- Oddělovač `{{....}}` je výraz používán k výpisům údajů do výstupu šablony. Například výpis proměnné.
- Oddělovač `{#....#}` je výraz používán k zápisům komentářů, ty mohou sloužit k lepší orientaci uživatele v šabloně. Na výstupu šablony se však nevyskytují.

3.4.2 Proměnné

Proměnné v šabloně jsou definovány importovaným slovníkem a jsou-li vloženy aplikací, je možno s nimi do značné míry manipulovat. Proměnné mohou nést atributy, jež jsou dány aplikací, která je šabloně poskytuje. Konkrétní atributy proměnných lze získat syntaxí `variable_name.attribute_name`, či `variable_name["attribute_name"]`. V obou případech *variable_name* představuje jméno dané proměnné a *attribute_name* jméno atributu. Chceme-li výsledek vytisknout do výstupu šablony, je potřeba celou syntaxi uzavřít do dvojitých složených závorek (viz Subsekce 3.4.1) [30].

3.4.3 Bloky

Funkce v Jinja2 mohou být pro lepší orientaci organizovány v blocích kódu. Začátek bloku je deklarován zápisem `{% blocktype %}` a ukončen zápisem `{% endblocktype %}`, kde *blocktype* je typ funkce, např *for* či *if*. Slovo *end* na konci bloku ve spojení s příslušnou funkcí ukončí daný blok [30] [32].

3.4.3.1 Blok *for* smyčky

For bloky, neboli *for* smyčky se v Jinja2 používají stejně jako v Pythonu k iteraci skrze iterovatelný objekt. Tím mohou být již zmíněné seznamy, *n*-tice, či slovníky. *For* smyčce předáme jednu či více podmínek a v momentě, kdy ji spustíme na objektu, skrze který si přejeme iterovat, projde všechny přítomné předměty a otestuje každý, zda splňuje danou podmínku. Smyčku bychom pak vytvořili zápisem bloku ve tvaru `{% for item in item_group if condition%}`, kde *index* je proměnná představující index předmětu, který je zrovna iterován, *item_group* je poskytnutý objekt k iteraci a `if condition` volá podmínku jménem *condition*. Ukázka viz Kód 3.5 [30], [32].

```
1 {% for item in items if item.prize < 5 %}
2     {{ item.name }}
3 {% endfor %}
```

Kód 3.5: Ukázka využití bloku *for* - vypsání jmen všech předmětů s cenou nižší než pět jednotek

3.5 LaTeX

LaTeX je bezplatný systém k přípravě kvalitní sazby dokumentů, běžně využívaný i pro vědecké či technické publikace. Cílem *LaTeXu* je, aby se uživatel nemusel příliš starat o správné formátování a úpravu dokumentu a místo toho se mohl soustředit na obsahovou stránku své práce. V mnohých známých textových editorech uživatel musí řešit velikosti písma v nových titulcích i textu, odsazování, umístění obrázků a popisků, a současně dohlížet, zda nové úpravy nezměnily nějakým nechtěným způsobem formátování dokumentu. Naproti tomu *LaTeX* všechny tyto starosti obstarává striktně dle šablony, jež mu byla uživatelem poskytnuta. Chce-li uživatel ve formátování cokoli editovat, *LaTeX* neklade prakticky žádná omezení a limity. Mezi další výhody taktéž patří podpora sazby komplexních matematických vzorců a rovnic, tabulek technického obsahu, podpora pro poznámky pod čarou, křížové odkazy a správa bibliografií a mnoho dalších. Důležitá připomínka je, že *LaTeX* využívá funkce některých speciálních symbolů, tudíž pokud některý z nich chceme zapsat v textu, je potřeba použít speciální syntaxi. Například znak _ (podtržítko) je potřeba zapsat jako

`\textunderscore`, popř. znak \ (zpětné lomítko) je potřeba zapsat jako `$backslash$` [33] [34].

3.6 Vkládání dat do Jinja2-LaTeXových šablon pomocí Pythonu

Při využívání Jinja2 šablon pro data z Pythonu k jejich exportování do *LaTeXu* je potřeba dbát na specifický syntax každého z těchto nástrojů. Z tohoto důvodu je potřeba definovat, jak má Jinja2 přistupovat k syntaxi, kterou používá *LaTeX* (viz. Kód 3.6), aby výstupy Jinjy byly s *LaTeXem* kompatibilní a mohlo být dosaženo požadovaných výsledků při exportování šablony do cílového formátu PDF [35].

```
1  import sqlite3
2  import jinja2
3  import os
4  from jinja2 import Template
5
6  latex_jinja_env = jinja2.Environment(
7      block_start_string = 'BLOCK{',
8      block_end_string = '}',
9      variable_start_string = 'VAR{',
10     variable_end_string = '}',
11     comment_start_string = '#{',
12     comment_end_string = '}',
13     line_statement_prefix = '%-',
14     line_comment_prefix = '%#',
15     trim_blocks = True,
16     autoescape = False,
17     loader = jinja2.FileSystemLoader(os.path.abspath('.'))
18 )
19 # Specification of the template
20 template = latex_jinja_env.get_template('test1.tex')
```

Kód 3.6: Nastavení prostředí Jinja2 pro práci s LaTeXem [35]

3.7 Click

Click je knihovna pro vytváření rozhraní příkazového řádku s využitím programovacího jazyka Python. Tato knihovna umožňuje rozsáhlou konfigurovatelnost a současně přichází s vyváženým výchozím nastavením. Ideou modulu *Click* je maximálně zjednodušit proces volání požadovaných nástrojů v příkazovém řádku a dosažení výsledků za použití minimálního množství kódu. Mezi některé klíčové funkce patří [36]:

- Podpora vnoření libovolných příkazů.
- Líné načítání vedlejších příkazů za běhu programu umožňuje ke zlepšení výkonu systému opozdit načítání či spouštění některých objektů do chvíle, než jich bude zapotřebí.
- Automatické generování stránky nápovědy.
- Podpora načítání hodnot proměnných prostředí (tj. dynamicky pojmenovaných hodnot, ovlivňujících chování spuštěných procesů) i hned po spuštění.
- Podpora vyžádání vlastních hodnot.
- Podpora práce se soubory ihned po vybalení.

Narozdíl od knihovny *Argparse*, jež je implementována ve standardní knihovně, *Click* nabízí plno výhod, jež se ukázaly jako výhodnější k použití pro tuto práci. Jedním z důvodů je, že oproti Argparse, umožňuje odpovídající vnoření dle návrhu a neomezuje práci s rozhraními příkazového řádku. Další komplikace nastává kvůli vestavěnému chování *Argparse*, jež se snaží určit, zda daný objekt je argument či možnost, což způsobuje rozsáhlé problémy v případě, není-li plně znám příkazový řádek [36].

3.7.1 Parametry

Click umí přijímat parametry jako možnosti. Možnostem lze jako vnitřní argumenty stanovit výchozí hodnotu, nechce-li uživatel určit vlastní. Toho lze dosáhnout zápisem `default=basic_input`, kde *basic_input* je výchozí hodnota. Určí-li uživatel vlastní, *default* je pro danou instanci přepsaná na zadanou hodnotu. Dalším vnitřním argumentem může být *prompt*, jemuž lze přiřadit textový řetězec, který uživatele vyzve k zadání daného argumentu. Toho lze dosáhnout zápisem `prompt="instructions"`, kde *"instructions"* je daná výzva pro uživatele. Dále je pak k dispozici ještě možnost pomocí

vnitřního argumentu `help="argument_introduction"` vygenerovat na žádost uživatele stránku nápovědy, kde se zobrazí *"argument_introduction"*, jakožto vysvětlení daného argumentu. Prvním argumentem možnosti je pak parametr ve tvaru `--parameter_name`, kde *parameter_name* je jméno parametru. Typ tohoto argumentu lze dále specifikovat například definicí funkce zápisem `def foo(parameter_name: type)`, kde `def foo()` vytváří funkci jménem *foo* a *type* v argumentu může být *str* jakožto textový řetězec, *int* jakožto celé číslo, *float* jakožto reálné číslo, či *bool* jakožto boolovská hodnota [36].

```
1  import click
2
3  @click.command()
4  @click.option('--count', default=1, help='Number of
   ↪  greetings.')
5  @click.option('--name', prompt='Your name',
6                help='The person to greet.')
7  def hello(count, name):
8      """Simple program that greets NAME for a total of COUNT
       ↪  times."""
9      for x in range(count):
10         click.echo(f"Hello {name}!")
11
12 if __name__ == '__main__':
13     hello()
```

Kód 3.7: Syntax možností a definice typu vstupního parametru

3.8 Modul pathlib

Pythonový modul *pathlib* nabízí třídami tvořený systém cest k souborům. Nejčastěji užívaná je třída Path, která poskytuje snadné rozhraní pro práci s cestami v souborovém systému. Velkou výhodou objektů třídy Path je to, že umožňují snadnou práci s cestami nehledě na to, zda-li se jedná o platformu Windows, či některý z Unixových systémů. Zde je řeč zejména o užívání dopředných a zpětných lomítek (/ a \), jenž jsou jedním z hlavních rozdílů přístupu k souborovým systémům výše zmíněných platforem. Třída Path sama rozpoznává, na které platformě se nachází, a v případě vyžádání poskytne odpovídající textovou reprezentaci dané cesty [37] [38].

3.8.1 Použití třídy Path

Původně byly cesty souborů v Pythonu interpretovány textovými řetězci. Vzhledem k tomu, že samotné cesty však textovými řetězci nejsou, byla práce s nimi značně komplikovaná a vyžadovala i k základním úkonům mnohdy hned několik knihoven. Z tohoto důvodu byla později vytvořena knihovna pathlib, která shromažďuje všechny důležité funkce a zpřístupňuje je pomocí metod a vlastností objektu Path. Cesty lze vytvořit následujícími způsoby [37] [38]:

- Příkazem `pathlib.Path(r"path_to_file")`, kde `r"path_to_file"` je textová reprezentace dané cesty ve formě raw stringu (raw string v Pythonu je řetězec, který ignoruje funkce speciálních symbolů a vnímá je jen jako text). Ukázka viz Kód 3.8.

- Z lokálního adresáře příkazem `pathlib.Path.cwd()`, kde cwd znamená current working directory (aktuální adresář).

- Spojením částí cesty pomocí znaku `/` jímž jsou odděleny jednotlivé adresáře, až po cílový soubor. Syntaxi lze zapsat jako `pathlib.Path "directory1"/ "directory1.1"/ "file_name"`, kde *"directory1"* je název výchozího adresáře, *"directory1.1"* je název vnitřního adresáře a *"file_name"* je jméno souboru, jehož cestu jsme hledali.

  - Stejného výsledku lze dosáhnout příkazem `pathlib.Path .joinpath("directory1", "directory1.1", "file_name")`.

```
1 >>> pathlib.Path(r'C:\Users\gahjelle\realpython\file.txt')
2 WindowsPath('C:/Users/gahjelle/realpython/file.txt')
```

Kód 3.8: Vytvoření cesty s využitím její textové reprezentace [38]

3.8.1.1 Metoda mkdir

Metoda mkdir, zkratka *make directotory* (vytvořit adresář), vytvoří nový adresář v místě kam odkazuje objekt třídy Path. Zavolat ji lze příkazem `Path("path_to_file").mkdir()`, kde *path_to_file* je cesta k cílovému souboru [37] [38].

3.8.1.2 Metoda unlink

Metoda unlink odstraní cílový soubor cesty, či symbolický odkaz k němu. Zavolat ji lze příkazem `Path("path_to_file").unlink()` [37] [38].

3.8.1.3 Metoda rename

Metoda rename přejmenuje aktuální adresář či soubor na jiný název a vrátí příslušnou cestu k tomuto přejmenovanému cílovému souboru či adresáři. Zavolat ji lze příkazem `Path("path_to_file").rename(new_name)`, kde *new_name* je nový název. Pokud již existuje soubor (ne adresář) s tímto názvem, v případě Unixu je nahrazen. V případě Windows program vyhodí výjimku `FileExistsError` [37] [38].

3.8.1.4 Metoda replace

Metoda replace přejmenuje aktuální adresář či soubor na jiný název a vrátí příslušnou cestu k tomuto přejmenovanému cílovému souboru či adresáři. Zavolat ji lze příkazem `Path("path_to_file").replace(new_name)`, kde *new_name* je nový název. Pokud již existuje soubor, či prázdný adresář s tímto názvem, dojde k jeho bezpodmínečnému nahrazení [37] [38].

3.9 PyInstaller

PyInstaller svazuje Python a všechny jeho závislosti v jeden balíček. Jeho funkcí je čtení a analýza kódu, jež mu poskytne uživatel, a vyhledání všech potřebných modulů a knihoven, na nichž je daný skript závislý. Kopie těchto souborů pak shromáždí do jedné složky příkazem *pyinstaller program_name.py*, kde `program_name.py` je jméno programu. Alternativně jde kopie souborů kompilovat do jednoho spustitelného souboru s koncovkou *.exe* pomocí příkazu `pyinstaller -F program_name.py` [39].

3.9.1 Zahrnutí dodatečného souboru

Spuštěním příkazu `pyinstaller some_options program_name.py`, kde *some_options* jsou libovolné možnosti, je vytvořen specifický soubor *program_name.spec*, který (není-li uživatelem určeno jinak) je automaticky uložen do aktuálního adresáře. Je možné přidat dodatečné datové soubory do balíčku pomocí příkazu `--add-data`, nebo vytvořením instance třídy `Analysis()` s argumentem `datas=[]`. V druhém ze zmíněných způsobů musí poskytovaný seznam obsahovat páry dvou hodnot (viz Kód 3.9),

kde obě jsou textové řetězce. První hodnota specifikuje stav souboru či souborů, v němž se nacházejí v rámci systému. Druhá hodnota specifikuje název adresáře, ve kterém se nacházejí. V případě, že chceme zahrnout další soubor do výsledného balíčku, či jinak upravovat specifikační soubor, je nutné následně spustit PyInstaller, kde bude jako cílový soubor použit*program_name.spec*. Upravený příkaz pak bude `pyinstaller some_options program_name.spec` [39].

```
a = Analysis(
    ...
    datas=[ ('src/README.txt', '.') ],
    ...
)
```

Kód 3.9: Syntax zahrnutí dodatečného souboru skrze soubor *spec* [39]

Kapitola 4

Návrh a implementace

Současná implementace pracuje se dvěma typy problémů, jež byly zvoleny k demonstraci celého nápadu, a to *FractionProblem* a *WordProblem*, kde první zmíněný je problém definovaný pro zlomky a druhý pro slovní úlohy. Uživateli stačí k tvorbě zkoušky (viz Sekce 4.1.1) zadat jako vstupní parametry počet studentů n a konfiguraci testu ve formě slovníku problémů. Výsledkem je zkouška složená z n testů.

Zjednodušené shrnutí a popis implementace

V rámci této sekce jsou zjednodušeně popsány základní principy, na nichž stojí výsledná implementace.

Uživatel využívající tuto aplikaci si může vybrat z dostupných příkladů na zlomky a slovní úlohy. Příklady se zlomky konkrétně podporují operace sčítání, odčítání, dělení a násobení, vždy právě pro dva zlomky. Slovní úlohy jsou čerpány z parametrické sbírky 4.4.2, která obsahuje tři úlohy na konstantní rychlost, tři na Ohmův zákon a tři na Newtonův druhý zákon.

Předpokládejme, že si uživatel přeje vygenerovat test pro 10 studentů o vlastním zadání. Úlohy, které by chtěl, aby byly v zadání jsou dva příklady na sčítání zlomků, jeden na násobení zlomků, druhou a třetí slovní úlohu na konstantní rychlost. V momentě spuštění aplikace je dotázán na dva parametry potřebné k vygenerování testu. První z nich je počet studentů (očekáváme, že počet studentů odpovídá počtu testů). Druhým parametrem je zadání testu, ve formě konfiguračního souboru testu (viz Kódy: 5.1 a 5.4. Konfigurační soubor testu je textový soubor formátu *JSON*, v němž lze zadat typové úlohy, ze kterých chceme, aby se test skládal.

Zadáním zmíněných parametrů se následně vygeneruje 10 testů, z nichž každý obsahuje všech pět typových příkladů zvolených v konfiguračním

souboru. Vygenerované testy jsou vůči sobě identické zadáním, liší se ale numerickými hodnotami.

4.1 Proces generování zkoušky

Proces generování zkoušek (viz Obrázek 2.2) vycházel z představy konceptuálního návrhu (viz Kapitola 2). Jak již bylo specifikováno, cílem bylo vytvořit program, který na základě počtu studentů a požadovaných úloh vytvoří zkoušku, jež bude souborem testů o stejném zadání. Jak je vidět na ukázce dvou testů (viz Obrázek C), jednotlivé testy se budou lišit numerickými hodnotami vzájemně si odpovídajících příkladů.

Konstruktor třídy *Exam* přijímá jako vstupní parametry počet studentů a slovník problémů. Následně je vytvořen objekt třídy *TestGenerator*, kterému je jako argument předán slovník problémů. Je-li na tento objekt zavolána metoda *get_test*, vytvoří se objekt třídy *Test*, který po zavolání metody *prepare_test* prohledá slovník klíč po klíči a u každého klíče identifikuje. o jakou úlohu (neboli třídu typu *BaseProblem*) se jedná. Následně zavolá příslušný generátor této typové úlohy, jež je parametrizována a proběhne proces, během kterého je vygenerována úloha tohoto typu s konkrétními hodnotami dosazenými za příslušné parametry. Nově vzniklá úloha je pak uložena a identický proces probíhá pro každý klíč slovníku. Vytvořené numerizované hodnoty jsou pak vráceny jako seznam objektu třídy *Exam*. Celý postup je opakován, tolikrát kolik byl zadaný počet studentů. Výstupem je pak seznam testů.

4.1.1 Třída Exam

Třída *Exam* zajišťuje generování zkoušky, která je souborem k testů se stejnými typovými problémy. Její konstruktor přijímá dva parametry: počet studentů a slovník problémů. Následně je vytvořen počet testů odpovídající počtu studentů, kde každý test obsahuje stejné typové úlohy, definované ve slovníku problémů. Jednotlivé testy jsou uloženy v seznamu jménem *tests*, jenž je atributem třídy *Exam*. Na objekt třídy *Exam* je pak možno použít metodu *to_tex()*, která v rámci své implementace, zavolá na každý test tohoto objektu a konvertuje jeho problémy do podoby, jež je kompatibilní s *LaTeXem*. Následně metoda *export_to_latex()* exportuje jednotlivé testy do šablon ve formátu *.tex* a uloží je do připravené složky. Ty pak stačí zkompilovat pomocí vhodného *LaTeXového* kompilátoru, jehož výstupem bude například soubor PDF. Během testování funkčnosti bylo použito webové

aplikace Overleaf, kde je možné vyplněnou *LaTeXovou* šablonu zkompilovat.

```
class Exam:

    def __init__(self, cnt_students, problem_dictionary):
        self.cnt_students = cnt_students # počet studentů
        self.test_generator =
        ↪  TestGenerator(problem_dictionary)
        self.tests = [] # seznam vygenerovaných testů
        self._prepare_exam() # volá metodu která seznam
        ↪  testů, naplní jednotlivými testy

    def to_tex(self): # metoda, jež konvertuje testy do
    ↪  LaTeXové syntaxe
        tests_out = []
        for test in self.tests:
            tests_out.append(test.to_tex())
        return tests_out

    def export_to_latex(self, output_folder): # metoda, jež
    ↪  exportuje testy do šablony připravené pro LaTeX
        latex_exporter = LatexExporter()
        for i, test in enumerate(self.to_tex()):
            latex_exporter.export(output_folder /
            ↪  f"./test{i}.tex", test)
```

Kód 4.1: Náhled do kódu třídy Exam

4.1.2 Třída TestGenerator

Třída *TestGenerator* zajišťuje generování testů. Její konstruktor přijímá od objektu třídy *Exam* jediný vstupní parametr a to slovník problémů. Výstupem metody *get_test()* je pak vrácen objekt třídy *Test*.

4.1.3 Třída Test

Třída *Test* zajišťuje generování matematických problémů dle zadaných požadavků. Konstruktor této třídy je používán v rámci objektu třídy *TestGenerator* a přijímá od něj požadavky pro daný test ve formě slovníku

problémů. Ten je následně metodou *prepare_test()* prohledán a u každého klíče slovníku je určeno, o jaký typový problém se jedná. Na základě toho jsou typové problémy rozděleny a s každým z nich se pracuje dle specificky definovaných kroků. Typové úlohy jednotlivých klíčů jsou získány pomocí příslušných generátorů úloh, které je již vrací v konkrétní podobě s náhodně dosazenými hodnotami v místech parametrů. Vygenerované problémy jednotlivých zadání úloh jsou pak uloženy do seznamu, jenž je atributem třídy *Test*.

4.1.4 Abstraktní třídy BaseProblem a BaseProblemGenerator

Třída *BaseProblem* je abstraktní třída, která obecně představuje libovolný matematický problém z naší sbírky. Předepisuje nutné metody, které každá z konkrétních problémových tříd musí splňovat, aby bylo možné daný problém integrovat do celého systému. Mezi tyto metody patří metoda k vyhodnocení problému *evaluate()*, metoda pro vytvoření reprezentace v *LaTeXovém* formátu a klasická metoda k textové reprezentaci. V rámci bakalářské práce byly implementovány jako samostatné problémy třídy *FractionProblem* a *WordProblem*, které jsou rozvedeny dále v Sekcích: 4.2 a 4.3.

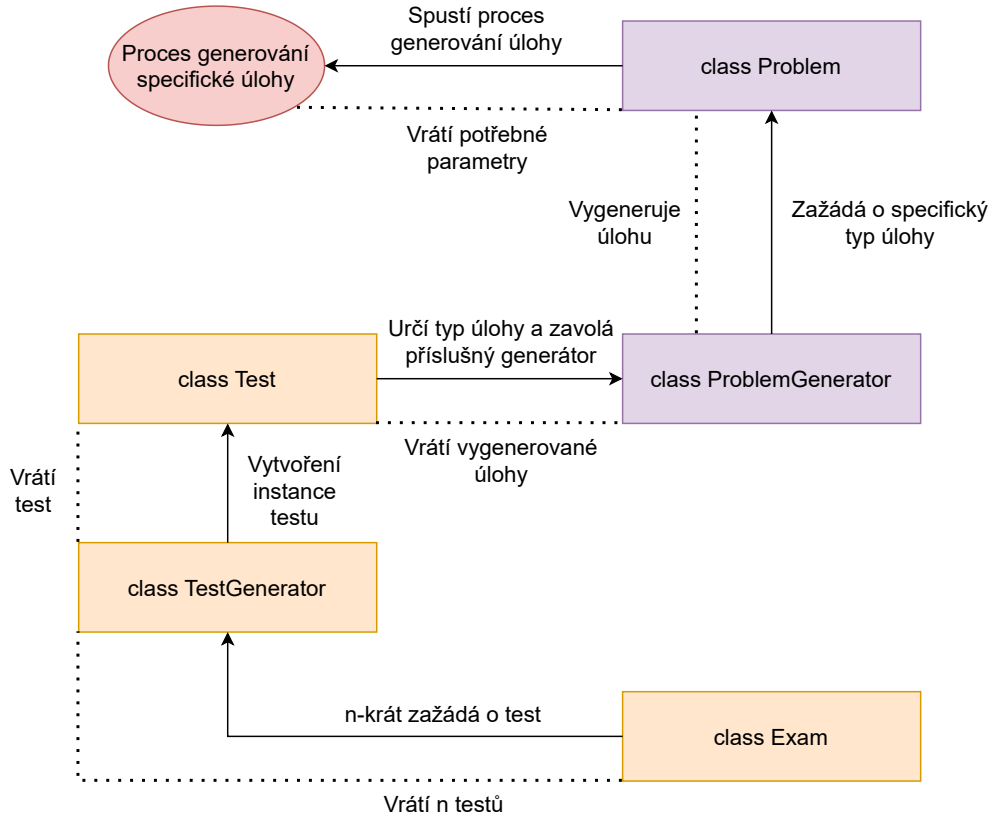
Třída *BaseProblemGenerator* je stejně jako *BaseProblem* abstraktní třídou. Obecně představuje libovolný generátor matematického problému s vlastní metodou *get_problem*, která generuje konkrétní instance příslušné třídy problému. V rámci práce byly implementovány jako generátory samostatných problémů třídy *FractionProblemGenerator* a *WordProblemGenerator*, jež odpovídají výše zmíněným třídám problémů, a budou taktéž dále rozvedeny v Subsekcích: 4.2.1 a 4.3.1.

4.1.5 Proces generování specifické úlohy

Proces generování specifické úlohy je zobecnění procesu, který je individuální pro každou třídu typu *BaseProblem*. Jedná se o soubor tříd a metod, jež daný problém vyžaduje ke generování instancí sebe sama.

4.2 Proces generování úloh třídy FractionProblem

Tato sekce popisuje funkci a komunikaci jednotlivých tříd, které zajišťují generování příkladů se zlomky.



Obrázek 4.1: Schéma generování zkoušky

4.2.1 Třída FractionProblemGenerator

Třída *FractionProblemGenerator* zajišťuje generování matematických úloh se zlomky. Konstruktor třídy přijímá jediný parametr a to *operator_string*, což je operátor v textové formě, konkrétně `"+"`, `"-"`, `"*"`, `"/"`. Na objekt této třídy pak lze zavolat její metodu *get_problem*, který vytvoří instanci třídy *FractionProblem* a vrátí vygenerovaný problém s příslušným operátorem.

4.2.2 Třída FractionProblem

Třída *FractionProblem* definuje strukturu matematických úloh se zlomky. Konstruktor třídy přijímá od objektu třídy *FractionProblemGenerator* jako argument operátor v textové formě. Ten je v rámci implementovaného slovníku operátorů rozpoznán a objektu této třídy je přiřazena funkce da-

ného operátoru. V rámci třídy je pak implementována abstraktní metoda *to_tex()*, která umožňuje vygenerované úlohy se zlomky převést do podoby kompatibilní s LaTeXem.

```
1 def to_tex(self):
2     text = rf"Vyřešte:
      ↪  $\frac{{{self.frac1.num}}}{{{self.frac1.den}}}
      ↪  {self.operator_string}
      ↪  \frac{{{self.frac2.num}}}{{{self.frac2.den}}} =$"
3     return text
```

Kód 4.2: Metoda `to_tex()`

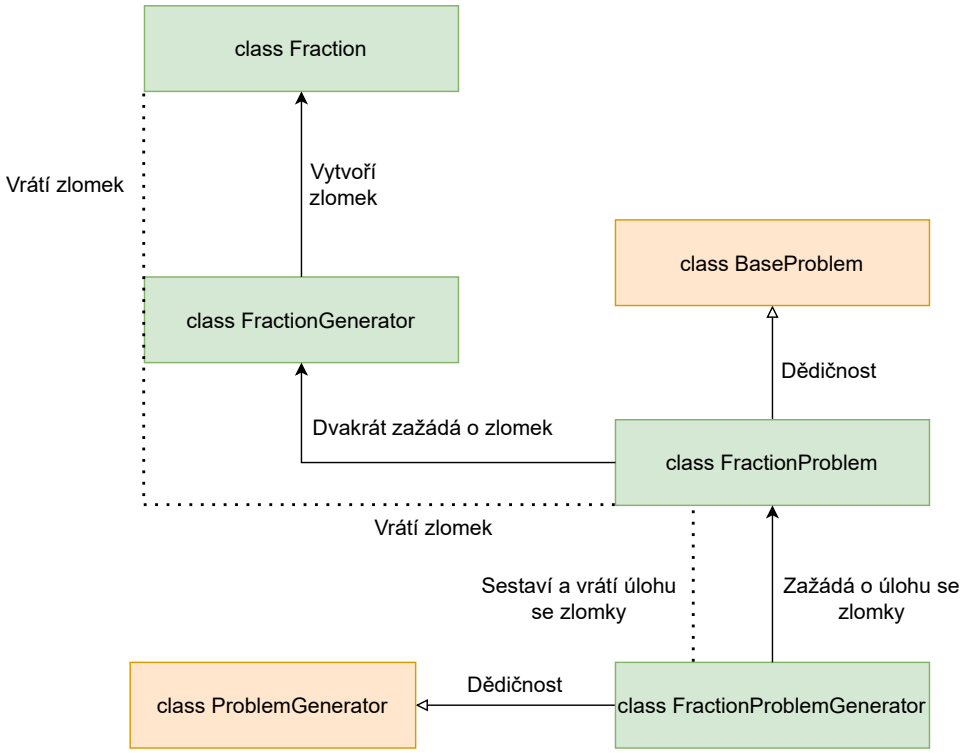
4.2.3 Třída FractionGenerator

Třída *FractionGenerator* zajišťuje generování zlomků. Její instance je vytvořena v momentě vytvoření objektu třídy *FractionProblem*. Následně je nad touto instancí zavolána metoda této třídy *get_problem()*, která vrací instanci třídy *Fraction*. V současné implementaci se proces opakuje dvakrát pro libovolnou operaci, tak abychom získali dva zlomky.

4.2.4 Třída Fraction

Třída *Fraction* se stará o operace se zlomky a vrací příslušné výsledky opět v podobě zlomku. Třída obecně pracuje se dvěma parametry: čitatelem a jmenovatelem. Pro vytvoření instance této třídy je potřeba zavolat její konstruktor v podobě *Fraction(numer1, number2)*, kde *number1* a *number2* jsou v rámci generátoru zlomků hodnoty čitatele a jmenovatele. Pro práci se zlomky má třída *Fraction* implementovány metody sčítání, odčítání, násobení a dělení dle matematických principů, a výsledkem je nový zlomek. Dále jsou implementovány metody k úpravě jednotlivých zlomků, které však nebyly využity pro výstup této práce. Libovolný zlomek lze s jejich pomocí převést do základního tvaru (metoda *normalize()*), usměrnit (metoda *rationalize()*) nebo provést jeho inverzi (metoda *inverse()*). K budoucí kontrole výsledků je pak vytvořena metoda srovnání (metoda *is_equal()*), která porovná dva zlomky tak, že porovná poměr čitatele a jmenovatele prvního zlomku s druhým. Pokud se poměry rovnají, jsou si zlomky rovny. To by mělo význam především u příkladů, kde mohou studenti výsledky určit v různých, ale přesto správných tvarech. V neposlední řadě je k dispozici

metoda (metoda *___repr___()*), jež slouží k reprezentaci daného zlomku v podobě textového řetězce.



Obrázek 4.2: Schéma generování úloh se zlomky

4.3 Proces generování úloh třídy WordProblem

Tato sekce popisuje funkci a komunikaci jednotlivých struktur, které zajišťují generování slovních úloh.

4.3.1 Třída WordProblemGenerator

Třída *WordProblemGenerator* zajišťuje generování slovních úloh. Konstruktor objektu této třídy přijímá dva parametry a to *problem_type*, jenž představuje typ slovní úlohy a *problem_index*, což je index konkrétní slovní úlohy daného typu. Na objekt této třídy pak lze zavolat její metodu

get_problem, jež vytvoří instanci třídy *WordProblem* a vrátí vygenerovanou slovní úlohu s konkrétními náhodně vygenerovanými hodnotami dosazenými za parametry.

4.4 Třída WordProblem

Třída *WordProblem* definuje způsob práce s parametrizovanými slovními úlohami. Konstruktor objektu této třídy přijímá dva parametry od objektu třídy *WordProblemGenerator* a to *problem_type* a *problem_index*. Následně je v konstruktoru vytvořena instance třídy *ProblemRepository*, na níž je zavolána její metoda *get_problem_data()* s parametry třídy *WordProblem*. Tím je vrácena požadovaná parametrizovaná slovní úloha (viz Kód 4.3), s níž objekt třídy *WordProblem* pracuje dál. Úloha je následně s využitím regulárních výrazů rozklíčována a jsou z ní separovány postupně vzorec pro výpočet neznámé veličiny a parametry známých veličin, kterým jsou přiřazeny náhodně vygenerované hodnoty. Hodnota neznámé veličiny je pak vypočtena dosazením hodnot parametrů do vzorce pomocí metody *evaluate()* a tato hodnota je uložena jako výsledek přidružený k dané úloze.

Třída má dále implementované metody k vlastní textové reprezentaci (metoda *___repr___()*) a převodu do podoby kompatibilní s LaTeXem (metoda *to_tex()*).

4.4.1 Třída WordProblemRepository

Třída *WordProblemRepository* zajišťuje provázání parametrické sbírky s generátorem příkladů. Její metoda *get_problem_data()* přijímá 2 parametry a to typ neboli téma slovní úlohy a její index. Je-li zavolána, prohledá soubor *problem_collection.json*, kde najde odpovídající parametrizovanou slovní úlohu (viz Kód 4.3) a tu vrátí uloženou v proměnné *task*. Přímo v konstruktoru třídy je přednastavená základní cesta ke sbírce úloh. Využití jiné sbírky, než výchozí, je v současné době možné pouze s přímým zásahem do kódu, nicméně do budoucna by mělo být možné poskytnout uživateli možnost tuto sbírku měnit dynamicky.

4.4.2 Kolekce parametrizovaných úloh problem_collection

Parametrická sbírka *problem_collection* je kolekce různorododých matematických a fyzikálních slovních úloh, kde každá úloha je složena z výplňového textu, tj. text, který slovně popisuje zadaný problém, z libovolného počtu

```
{
    # toto je téma úlohy
    "constant_velocity": [

    # úloha s indexem 0
    "Jak rychle <velocity; unknown; km/h> musí vozidlo jet,
    ↪ aby urazilo vzálenost <distance; s; km> kilometrů za
    ↪ <time; t; h> hodin?; unknown=s/t",

    # úloha s indexem 1
    "Za jak dlouho <time; unknown; h> urazí vozidlo
    ↪ vzdálenost <distance; s; km> kilometrů, pojede-li
    ↪ rychlostí <velocity; v; km/h> km/h?; unknown=s/v",

    # úloha s indexem 2
    "Jak velkou vzdálenost <distance; unknown; km> ujede
    ↪ vozidlo za <time; t; h> hodin, jestliže pojede
    ↪ rychlostí <velocity; v; kn/h> km/h?; unknown=v*t"
    ],

    # zde pokračují další témata slovních úloh
}
```

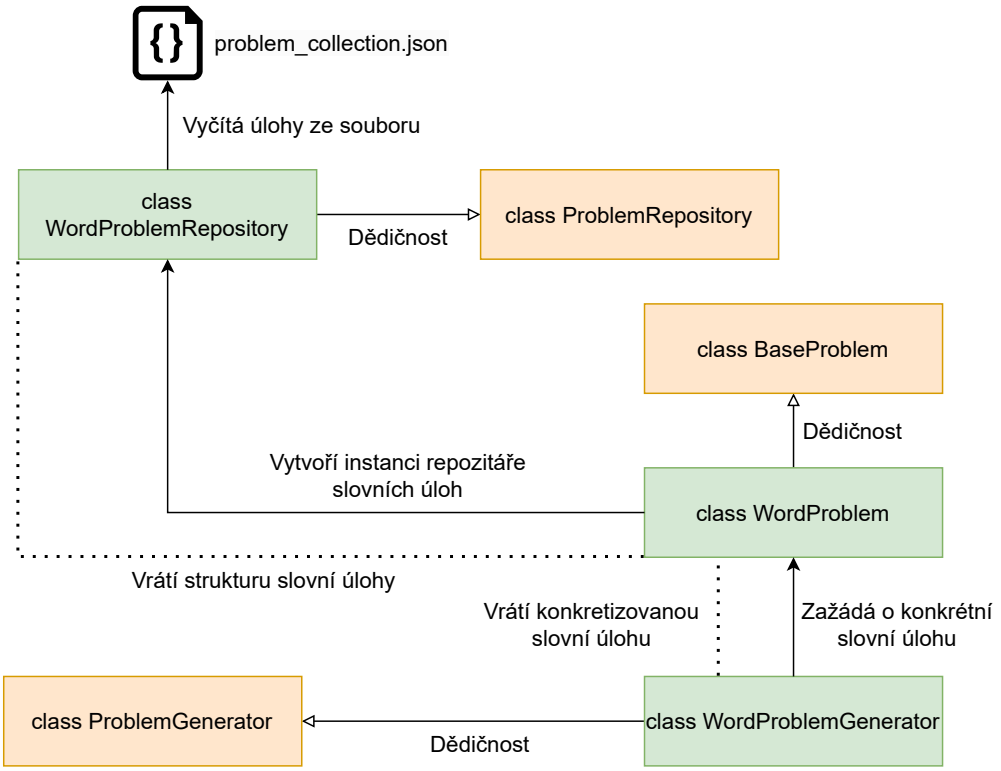
Kód 4.3: Ukázka struktury slovních úloh ze sbírky (viz Subsekce 4.4.2) na téma konstantní rychlost

tripletů pro příslušné veličiny a ze vzorců pro výpočet dané úlohy. Triplety jsou tvořeny následujícím zápisem: `<quantity_name; value; unit>`, kde *quantity_name* je název veličiny, *value* je parametr její hodnoty a *unit* je jednotka, ve které je uvedena. Jediná výjimka nastává u veličiny, jejíž hodnota je neznámá a je potřeba ji vypočítat z ostatních zadaných údajů. Hodnota této veličiny není opatřena parametrem jako ostatní, ale výrazem `unknown`, který symbolizuje, že se jedná o neznámou (viz Kód 4.3). Triplety jsou umístěny v textu a definují tak, kde se mají nacházet dané parametry po dosazení.

Aby každé úloze mohl být přiřazen výsledek, je na konci úlohy vzorec výpočtu výsledku, například pro úlohu o dvou parametrech a jedné neznámé, ve tvaru `unknown=parameter1<operation>parameter2`, kde *parameter1* a *parameter2* jsou hodnoty parametrů příslušných veličin. V pří-

padě použití různých jednotek, u kterých je zapotřebí převodu, stačí vzorec pro výsledek doplnit potřebnými konstantami.

Například v úloze zaměřené na výpočet rychlosti ze zadané dráhy s parametrem `s` a čas s parametrem `t`, kde dráha je zadána v kilometrech a čas v sekundách a výsledek je požadován v metrech za sekundu, může výsledný vzorec vypadat následovně: `unknown=1000*s/t`. Takto sestavená úloha je pak uložena v parametrické sbírce a může být zavolána generátorem příkladů.



Obrázek 4.3: Schéma generování slovních úloh

4.5 Třída LatexExporter

Třída *LatexExporter* je určena k exportování vygenerovaných zkoušek do LaTeXové šablony. Konstruktor třídy přijímá jako parametr cestu k šabloně jménem *jinja_test_pre_template* ve formátu *.tex* (viz Kód 2), jež je ve stejném adresáři jako tato třída. V rámci třídy je implementována metoda

export() definující prostředí šablony. Metoda jako jeden z parametrů přijímá seznam problémů a na základě počtu přítomných problémů připraví strukturu šablony pro testy v rámci vygenerované zkoušky. Objekt třídy *LatexExporter* je vytvořen při zavolání metody *export_to_latex()* na objekt třídy *Exam*, načež jsou jednotlivé testy dané zkoušky vysázeny do připravených šablon.

```
def __init__(
    self,
    template_path: pathlib.Path = pathlib.Path(__file__)
     .resolve().with_name("jinja_test_pre_template.tex")
):
```

Kód 4.4: Konstruktor třídy LatexExpoerter

Kapitola 5

Praktické testování

V této kapitole jsou popsány postupy, jimiž byl celý projekt otestován a jsou zde prezentovány některé ukázkové výstupy. Testování bylo prováděno pouze ve formě uživatelského testování. Vzhledem k účelu a rozsahu práce bylo určeno, že není potřeba pro výstup práce vytvářet automatizované testy. V rámci testování byly stanoveny a různě modifikovány jak počty žáků, tak i konfigurace testu, a výstupy důkladně analyzovány, aniž by byly shledány problémy.

Funkčnost projektu byla v průběhu práce pravidelně testována, na základě čehož byl průběžně zdokonalován. V závěru, kdy byla hotova implementační část, byl celý projekt otestován jako celek a byla ověřena správná funkčnost všech dílčích částí. Některé výstupy testování jsou uvedeny a okomentovány níže.

Generování testu bylo testováno pomocí spustitelného souboru generovaného v příkazovém řádku v cílovém adresáři příkazem *pyinstaller main.spec*. V rámci testování bylo vyzkoušeno několik scénářů, nicméně zde bude popsán průběh pouze jednoho z nich.

Při spuštění je uživatel nejprve dotázán na počet studentů, kde byl pro jednoduchost zvolen počet 2. Nechce-li uživatel tento počet stanovit, popřípadě zapomene-li na tento krok, je použita výchozí hodnota 1, jež je uvedena v hranatých závorkách. Následně je uživatel dotázán, jakou konfiguraci testu si přeje. Zde je výchozí hodnotou soubor s předpřipravenou konfigurací formátu *JSON*, jež je uveden v hranatých závorkách. Tento soubor je vnitřní součástí celé aplikace a není možné k němu zvenku přistupovat. Pro daný test byla použita výchozí konfigurace v podobě zobrazené v Kódu 5.1.

Na základě zvolené konfigurace byly vygenerovány dvě LaTeXové šablony testů s rozložením typových úloh odpovídajících konfiguraci testu,

avšak jednotlivé úlohy se mezi testy numericky liší. Získané šablony pak lze již snadno pomocí *LaTeXového* kompilátoru převést do formátu *PDF*. Stejný proces byl opakován s různými počty studentů a libovolně modifikovanou konfigurací testu.

První z vygenerovaných testů, na základě dříve uvedených kritériích, je prezentován Kódem 5.2 a druhý Kódem 5.3. Jednotlivé příklady jsou automaticky vsazeny do předpřipravené *LaTeXové* šablony. V Příloze C můžeme vidět zkompilované šablony ve formátu *PDF*.

Další z uvedených testů v Příloze C ukazuje výsledek generování testu dle konfigurace zobrazené Kódem 5.4. Jeho *PDF* reprezentace byla kompilována z Kódu 5.5. V tomto případě bylo v konfiguraci vyžádáno pouze pět příkladů na sčítání zlomků, které byly následně bez problémů vygenerovány a jak je vidět, každý z příkladů se liší v numerických hodnotách.

```
{
  "fraction": { # typ problému: operace se zlomky
    "+": 1, # jeden příklad na sčítání zlomků
    "-": 1, # jeden příklad na odčítání zlomků
    "*": 1, # jeden příklad na násobení zlomků
    "/": 1  # jeden příklad na dělení zlomků
  },
  "word_problem": { # typ problému: slovní úloha
    "constant_velocity": [1, 2], # druhý a třetí příklad na
    ↪  konstantní rychlost
    "ohm_law": [2], # třetí příklad na Ohmův zákon
    "newton_second_law_of_motion": [] # žádný příklad na
    ↪  Newtonův druhý zákon
  }
}
```

Kód 5.1: Konfigurace testu se smíšenými úlohami. Dva vygenerované testy (viz Kódy: 5.2 a 5.3) jsou po zkompilování do formátu *PDF* uvedeny v Příloze C.

```
\begin{questions}

    \question Vyřešte: $\frac{2}{1} + \frac{5}{9} =$
    \vspace{\stretch{1}}

    \question Vyřešte: $\frac{7}{6} - \frac{6}{9} =$
    \vspace{\stretch{1}}

    \question Vyřešte: $\frac{5}{6} * \frac{2}{2} =$
    \vspace{\stretch{1}}

    \question Vyřešte: $\frac{1}{6} / \frac{8}{4} =$
    \vspace{\stretch{1}}

    \question Za jak dlouho urazí vozidlo vzdálenost 4
    ↪  kilometrů, pojede-li rychlostí 4 km/h?
    \vspace{\stretch{1}}

    \question Jak velkou vzdálenost ujede vozidlo za 6 hodin,
    ↪  jestliže pojede rychlostí 2 km/h?
    \vspace{\stretch{1}}

    \question Jaký je odpor spotřebiče, kterým při napětí 2
    ↪  V prochází proud 1 A?
    \vspace{\stretch{1}}

\end{questions}
```

Kód 5.2: Část zdrojového kódu prvního vygenerovaného testu v *LaTeXu*. Zkompilovaný do formátu *PDF* je uveden v Příloze C.

```
\begin{questions}

    \question Vyřešte: $\frac{6}{5} + \frac{3}{10} =$
    \vspace{\stretch{1}}

    \question Vyřešte: $\frac{9}{7} - \frac{6}{3} =$
    \vspace{\stretch{1}}

    \question Vyřešte: $\frac{8}{2} * \frac{9}{7} =$
    \vspace{\stretch{1}}

    \question Vyřešte: $\frac{4}{6} / \frac{8}{5} =$
    \vspace{\stretch{1}}

    \question Za jak dlouho  urazí vozidlo vzdálenost 9
    ↪  kilometrů, pojede-li rychlostí 8 km/h?
    \vspace{\stretch{1}}

    \question Jak velkou vzdálenost ujede vozidlo za 9 hodin,
    ↪  jestliže pojede rychlostí 2 km/h?
    \vspace{\stretch{1}}

    \question Jaký je odpor spotřebiče, kterým při napětí 7
    ↪  V prochází proud 9 A?
    \vspace{\stretch{1}}

\end{questions}
```

Kód 5.3: Část zdrojového kódu druhého vygenerovaného testu v *LaTeXu*. Zkompilovaný do formátu *PDF* je uveden v Příloze C.

```
{
  "fraction": {
    "+": 5, # pět příkladů na sčítání zlomků
    "-": 0,
    "*": 0,
    "/": 0
  }
}
```

.

Kód 5.4: Konfigurace testu obsahujícího pouze úlohy na sčítání zlomků. Uvedená čísla představují počet příkladů, na daný typ úlohy se zlomky. Zde je stanoveno pět příkladů na sčítání zlomků a žádný od ostatních operací, tudíž vygenerovaný test bude obsahovat právě pět příkladů na sčítání zlomků. To lze vidět v Kódu 5.5

```
\begin{questions}

    \question Vyřešte: $\frac{3}{10} + \frac{10}{5} =$
    \vspace{\stretch{1}}

    \question Vyřešte: $\frac{1}{7} + \frac{8}{10} =$
    \vspace{\stretch{1}}

    \question Vyřešte: $\frac{5}{5} + \frac{3}{9} =$
    \vspace{\stretch{1}}

    \question Vyřešte: $\frac{1}{1} + \frac{7}{8} =$
    \vspace{\stretch{1}}

    \question Vyřešte: $\frac{8}{6} + \frac{9}{3} =$
    \vspace{\stretch{1}}

\end{questions}
```

Kód 5.5: Část zdrojového kódu vygenerovaného testu s úlohami na sčítání zlomků v *LaTeXu*. Zkompilovaný do formátu *PDF* je uveden v Příloze C.

Závěr

Cílem práce byl návrh a implementace parametrické sbírky a generátoru matematických úloh. Prvním krokem bylo prozkoumání a popsání různých studijních portálů, generátorů příkladů a školních systémů, jež jsou z hlediska výuky matematiky a přírodních věd zajímavé (viz. Kapitola 1). Následovalo vytvoření jednoduchého konceptuálního návrhu projektu, kde byly vytyčeny hlavní cíle práce a rozebrány důležité body, se kterými se měla práce potýkat (viz. Kapitola 2). K technické realizaci byl v převážné míře využit programovací jazyk Python, avšak k plnohodnotné implementaci bylo zapotřebí i dalších podpůrných nástrojů (viz. Kapitola 3). Na základě všech předchozích bádání a analýz bylo implementováno řešení, jenž je popsáno a znázorněno v Kapitole 4. Poslední částí práce bylo otestování vytvořeného generátoru podle rozličných potenciálních uživatelských požadavků (viz. Kapitola 5).

Výstupem práce je pak funkční generátor implementovaných matematických úloh a plně modifikovatelná parametrická sbírka příkladů. Obě části dohromady umožňují snadné přípravy testů s individuálně numerizovanými hodnotami jednotlivých úloh pro libovolný počet studentů. Tímto řešením je dosažena férovost jednotlivých testů a současně ušetřen čas k jejich přípravě.

Velmi důležitou částí mé práce bylo nalezení vhodných nástrojů, programů k celkové implementaci projektu a vytvoření vhodného uživatelského rozhraní, k práci s celým systémem, jež by nevyžadovalo od uživatele podrobně rozumět zbytku práce. Momentálně uživateli stačí přepsat v konfiguraci testu počty jednotlivých typových úloh, dle svých představ a zadat počet studentů. Vše ostatní už systém obstará sám.

Do budoucna předpokládám rozšíření projektu v několika úrovních. Za prvé bych chtěl zvětšit množství příkladů v představené sbírce tak, aby byla

využitelná v běžné praxi, a současně plánuji rozšířit sbírku i pro další přírodovědné předměty, jako je chemie, fyzika, atd. Z pohledu technického je nutné vylepšit uživatelské rozhraní tak, aby bylo přívětivé i pro běžné uživatele. Mimo jiné jde o vytvoření grafického editoru pro konfiguraci testu, které umožní uživateli měnit jednotlivé parametry, bez toho aby musel přímo editovat konfigurační soubor. Dalším nutným krokem je integrování vlastní služby pro kompilaci LaTeXových šablon, které zbaví uživatele při exportování testů a zjednoduší celý proces. V neposlední řadě je cílem vystavit celý systém jako webovou aplikaci dostupnou pro případné uživatele přes internet.

Celkově výsledek práce jakožto koncept splnil má očekávání a přesvědčil mě k jeho zdokonalování a rozvoji. Věřím, že finální řešením které vznikne na základě tohoto konceptu má potenciál pozitivně přispět ke kvalitě výuky přírodovědných předmětů a současně usnadnit práci učitelů při přípravě vhodných studijních materiálů.

Literatura

[1] Systém Bakaláři. Dostupné z: https://www.bakalari.cz/

[2] Nápověda systému Bakaláři. Dostupné z: https://napoveda.bakalari.cz/

[3] Webová aplikace systému Bakaláři. Dostupné z: https://www.bakalari.cz/Static/webapp

[4] Edookit. Dostupné z: https://edookit.com/cs/examples

[5] Systém IS/STAG. Dostupné z: https://is-stag.zcu.cz/

[6] The WeBWorK Software. Dostupné z: https://openwebwork.org/what-is-webwork/

[7] Systém WeBWork. Dostupné z: https://webwork.maa.org/wiki/WeBWorK_Main_Page

[8] Introduction to WeBWorK for students. Dostupné z: https://www.math.wustl.edu/~blake/wwdocs/students-intro.html

[9] WolframProblemGenerator. Dostupné z: https://www.wolframalpha.com/pro/problem-generator

[10] Ukázka vygenerovaného testu webovým generátorem MathsBot. Dostupné z: https://mathsbot.com/questionGenerator

[11] Webový generátor matematických úloh MathsBot. Dostupné z: https://mathsbot.com/

[12] Petáková, J.: *Matematika*. Praha: Prometheus, druhé vydání, 2020, ISBN 978-80-7196-487-2.

[13] Běloun, F.: *Sbírka úloh z matematiky pro základní školu*. Praha: Prometheus, 8 vydání, 2008, ISBN 978-80-7196-104-8.

[14] Webový portál Isibalo. Dostupné z: `https://isibalo.com/`

[15] Webový portál Mathematicator. Dostupné z: `https://mathematicator.com/`

[16] Webový portál OnlineSchool. Dostupné z: `https://onlineschool.cz/`

[17] Matematické Fórum. Dostupné z: `https://forum.matweb.cz/`

[18] Webová aplikace Moodle. Dostupné z: `https://www.linuxexpres.cz/software/moodle-open-source-vyuka-pres-internet`

[19] Regular-Expressions. Dostupné z: `http://www.regular-expressions.info/tutorial.html`

[20] Programovací jazyk Python. Dostupné z: `https://docs.python.org/3/library/re.html`

[21] Python Regular Expressions. Dostupné z: `https://developers.google.com/edu/python/regular-expressions`

[22] Python RegEx. Dostupné z: `https://www.programiz.com/python-programming/regex`

[23] Object-oriented programming (OOP). Dostupné z: `https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP`

[24] Object-Oriented Programing. Dostupné z: `https://dev.to/jburroughs/stay-in-the-l-oop-with-object-oriented-programming-basics-1dn`

[25] On-line manuál - Úvod do objektově orientovaného programování v Pythonu. Dostupné z: `https://www.itnetwork.cz/python/oop/python-tutorial-uvod-do-objektove-orientovaneho-programovani`

[26] Object-Oriented Programming (OOP) in Python 3. Dostupné z: `https://realpython.com/python3-object-oriented-programming/`

[27] Matthes, E.: *Python crash course*. San Francisco: No Starch Press, druhé vydání, [2019]., ISBN 978-1-59327-928-8.

[28] Introducing Classes. Dostupné z: https://docs.python.org/3/tutorial/classes.html

[29] Introducing JSON. Dostupné z: https://www.json.org/json-en.html

[30] Template system Jinja. Dostupné z: https://jinja.palletsprojects.com/en/3.1.x/

[31] Jinja2 Explained in 5 Minutes! Dostupné z: https://codeburst.io/jinja-2-explained-in-5-minutes-88548486834e

[32] Jinja Blocks. Dostupné z: https://documentation.bloomreach.com/engagement/docs/jinjablocks

[33] The LaTeX Project. Dostupné z: https://www.latex-project.org/about/

[34] Webové IDE Overleaf. Dostupné z: https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes

[35] At These Coordinates. Dostupné z: https://atcoordinates.info/2015/04/03/inserting-data-into-templates-with-python-and-jinja/

[36] Modul Click. Dostupné z: https://click.palletsprojects.com/en/8.1.x/

[37] Modul Pathlib. Dostupné z: https://docs.python.org/3/library/pathlib.html

[38] Real Python. Dostupné z: https://realpython.com/python-pathlib/#the-problem-with-python-file-path-handling

[39] PyInstaller Manual. Dostupné z: https://pyinstaller.org/en/stable/

Příloha A

Seznam použitých zkratek

HTML Hypertext Markup Language

IDE Integrated Development Environment

JSON JavaScript Object Notation

PDF Portable Document Format

XML Extensible Markup Language

Příloha B

Struktura přiloženého ZIP souboru

Následující stránka prezentuje strukturu přiloženého souboru ZIP. Soubor obsahuje tři adresáře: adresář se spustitelnou aplikací, adresář zdrojových kódů projektu a adresář ukázkových testů v PDF formátu.

- app
  - TestGenerator.exe ..... spustitelná aplikace
  - test_config.json ..... konfigurační soubor testu
- src ..... adresář zdrojových kódů
  - exam
    - exam.py
  - main.py
  - main.spec
  - problem_generators
    - problems
      - base_problem.py ..... generická třída problémů
      - fraction_problem ..... úlohy na zlomky
        - fraction.py
        - fraction_generator.py
        - fraction_problem.py
        - fraction_problem_generator.py
      - problem_repository.py
      - word_problem ..... slovní úlohy
        - json_collection.json ..... sbírka úloh
        - word_problem.py
        - word_problem_generator.py
        - word_problem_repository.py
    - problem_generator.py .. generická třída generátorů problémů
  - README.md
  - requirements.txt ..... soubor závislostí
  - test_config_default.json ..... výzchozí konfigurace testu
  - test_generators ..... generátory testů
    - tests
      - test.py ..... třída pro reprezentaci testu
    - test_generator.py
  - utils
    - json_loader ..... skripty pro práci s JSON
      - json_loader.py
    - latex_exporter ..... skripty pro export do LaTeX
      - jinja_test_pre_template.tex .... generická šablona testu
      - latex_exporter.py
- tests ..... ukázkové testy
  - test_example.pdf
  - test_example2.pdf
  - test_example_fractions.pdf

Příloha C

Příklad generovaných testů

Přiložné testy jsou vyplněnými *LaTeXovými* šablonami (viz Kódy: 5.2, 5.3 a 5.5) prezentovanými v Kapitole 5 po zkompilování do formátu *pdf*.

První dva testy byly vygenerovány dle konfigurace prezentované Kódem 5.1. Je evidentní, že testy se vzájemně liší pouze v numerických hodnotách, zatímco struktura zadání a typové úlohy jsou zcela identické.

Třetí z testů byl generován dle konfigurace prezentované Kódem 5.4, kde bylo stanoveno, aby vzniklé testy obsahovaly pět příkladů na sčítání zlomků a žádné jiné úlohy. Jak je vidět jednotlivé příklady se liší v numerických hodnotách i přes to, že jsou stejného typu, z čehož je zřejmé, že aplikace neomezuje zadání testu na pouze jednu unikátní instanci každé z typových úloh.

Své odpovědi napište do volného místa ke každé jednotlivé úloze. V případě nedostatku místa pište na další strany.

Jméno a příjmení: _____

Třída: _____

1. Vyřešte: $\frac{2}{1} + \frac{5}{9} =$

2. Vyřešte: $\frac{7}{6} - \frac{6}{9} =$

3. Vyřešte: $\frac{5}{6} * \frac{2}{2} =$

4. Vyřešte: $\frac{1}{6} / \frac{8}{4} =$

5. Za jak dlouho urazí vozidlo vzdálenost 4 kilometrů, pojede-li rychlostí 4 km/h?

6. Jak velkou vzdálenost ujede vozidlo za 6 hodin, jestliže pojede rychlostí 2 km/h?

7. Jaký je odpor spotřebiče, kterým při napětí 2 V prochází proud 1 A?

Své odpovědi napište do volného místa ke každé jednotlivé úloze. V případě nedostatku místa pište na další strany.

Jméno a příjmení: _____

Třída: _____

1. Vyřešte: $\frac{6}{5} + \frac{3}{10} =$

2. Vyřešte: $\frac{9}{7} - \frac{6}{3} =$

3. Vyřešte: $\frac{8}{2} * \frac{9}{7} =$

4. Vyřešte: $\frac{4}{6} / \frac{8}{5} =$

5. Za jak dlouho urazí vozidlo vzdálenost 9 kilometrů, pojede-li rychlostí 8 km/h?

6. Jak velkou vzdálenost ujede vozidlo za 9 hodin, jestliže pojede rychlostí 2 km/h?

7. Jaký je odpor spotřebiče, kterým při napětí 7 V prochází proud 9 A?

Své odpovědi napište do volného místa ke každé jednotlivé úloze. V případě nedostatku místa pište na další strany.

Jméno a příjmení: _____

Třída: _____

1. Vyřešte: $\frac{3}{10} + \frac{10}{5} =$

2. Vyřešte: $\frac{1}{7} + \frac{8}{10} =$

3. Vyřešte: $\frac{5}{5} + \frac{3}{9} =$

4. Vyřešte: $\frac{1}{1} + \frac{7}{8} =$

5. Vyřešte: $\frac{8}{6} + \frac{9}{3} =$