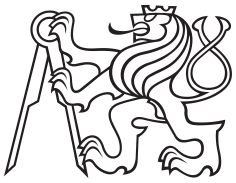
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Návrh a implementace nástroje pro automatické black-box testování desktopové aplikace s využitím OCR a virtualizace

Tomáš Musil

Vedoucí: RNDr. Ladislav Serédi
Studijní program: Otevřená informatika
Květen 2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Musil** Jméno: **Tomáš** Osobní číslo: **491993**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Návrh a implementace nástroje pro automatické black-box testování desktopové aplikace s využitím OCR a virtualizace

Název diplomové práce anglicky:

Design and implementation of an automatic black-box texting tool using OCR and virtualization for desktop applications

Pokyny pro vypracování:

Prostudujte stávající systémy pro black-box testování, implementace OCR nástrojů a metody pro virtualizaci desktopových aplikací. Diskutujte vlastnosti black-box testování, jeho příklady a rozdíly od white-box testování.
Navrhněte vlastní systém pro black-box testování desktopových aplikací s přihlédnutím k systému IDP (Information Data Processing) ŘLP. Při návrhu nástroje se zaměřte zejména na následující aspekty:
- možnost paralelizace běhu více testů,
- efektivní využití OCR pro validaci vizuálního výstupu aplikace,
- ukládání a evidenci výsledků prováděných testů.

Implementujte takto navržený systém a ověřte jeho funkcionalitu v reálném provozu. Výsledky testů vyhodnoťte.
Navrhněte a implementujte webovou aplikaci pro správu testovacích scénářů, jejich spuštění a validace výsledků. Proveďte jeho neformální testování a diskutujte dosažené výsledky.

Seznam doporučené literatury:

A Comparative Study of Black Box Testing and White Box Testing Techniques, Manish Kumar, Santosh Kumar Singh2, Dr. R. K. Dwivedi, 2015, online na adrese: https://www.vbu.ac.in/ftpwebapps/vbu/resources/vbu_web/dept/mca/A%20Comparative%20Study%20of%20Black%20Box%20Testing%20and%20White%20Box%20Testing%20Techniques.pd
XTEST Extension Library, Kieron Drake, online na adrese: https://www.x.org/releases/X11R7.7/doc/libXtst/xtestlib.html
OCR in 2023: Benchmarking Text Extraction/Capture Accuracy, Cem Dilmegani, online na adrese: https://research.aimultiple.com/ocr-accuracy/
Tesseract Documentation, online na adrese: https://tesseract-ocr.github.io/tessapi/5.x/

Jméno a pracoviště vedoucí(ho) diplomové práce:

RNDr. Ladislav Serédi kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **01.02.2024** Termín odevzdání diplomové práce: **24.05.2024**

Platnost zadání diplomové práce: **21.09.2025**

RNDr. Ladislav Serédi podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____ _____

Datum převzetí zadání Podpis studenta

Poděkování

Děkuji RNDr. Ladislavu Serédimu za vedení mé diplomové práce, zároveň tak za poskytnuté konzultace a podnětné návrhy, které práci obohatily.

Dále děkuji kolegům z týmu Information Data Processing, že mi umožnili projekt realizovat a poskytli cenné rady při implementaci.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškerou použitou literaturu.

V Praze, 20. května 2024

Abstrakt

Diplomová práce představuje návrh řešení a následnou implementaci nástroje pro automatizované black-box testování desktopové aplikace, využívané pro řízení letového provozu, spolu s webovou aplikací pro správu testů. V první části se seznámíme se zadáním projektu a se základními pojmy jako jsou virtualizace a OCR. Následně se podíváme na existující řešení a vydefinujeme si sadu knihoven, které budu v projektu integrovat. Dále detailně rozebereme architekturu mého návrhu a jeho klíčové funkcionality. Na to přímo navazuje poslední část, která se věnuje samotné implementaci nástroje a v neposlední řadě také testování rychlosti, přesnosti a uživatelské přívětivosti.

Klíčová slova: black-box testování, automatické testování, OCR, virtualizace

Vedoucí: RNDr. Ladislav Serédi
Praha, Resslova 9, E-429

Abstract

The master thesis presents the design of a solution and subsequent implementation of a tool for automated black-box testing of a desktop application used for air traffic control, together with a web application for test management. In the first part we introduce the project brief and basic concepts such as virtualization and OCR. We then take a look at existing solutions and define the set of libraries that I will integrate in the project. Next, we will discuss in detail the architecture of my proposal and its key functionalities. This is directly followed by the last part, which is dedicated to the actual implementation of the tool and last but not least to testing its speed, accuracy and user-friendliness.

Keywords: black-box testing, automated testing, OCR, virtualization

Title translation: Design and implementation of automatic black-box testing tool with the use of OCR and virtualization

Obsah

Obrázky

Tabulky

Ukázky kódu

Kapitola 1

Úvod

Přestože testování je po psaní dokumentace snad nejméně oblíbenou aktivitou každého vývojáře, jedná se nezpochybnitelně o klíčovou část rozvoje a udržování systému ve funkčním stavu. Testování je o to důležitější, pokud se jedná o systém, který je používaný a vyvíjený mnoho let, a ještě o to důležitější, pokud jde o kritický systém, který musí být vždy funkční.

Systémy pro řízení letového provozu splňují všechny tyto podmínky a dovádí je do extrému, jelikož stejný systém je v produkci i několik desítek let a po celou dobu je aktivně rozvíjený i několika generacemi programátorů. Stejně tak nezpochybnitelný je fakt, že na tyto systémy jsou vyvíjeny největší nároky jak na správnost jejich výstupu, tak na alespoň minimální provozní funkčnost za jakýchkoli okolností. Je tedy zřejmé, že testování těchto systémů musí být skutečně důkladně realizované.

V praxi má testování těchto systémů několik úrovní. V první řadě probíhají vývojářské testy při tvorbě nové funkcionality, následně je aplikace předána testovacímu oddělení, které sepíše a provede interní uživatelské testy na základě požadavků od zákazníka. Po předání systému zákazníkovi (Řízení letového provozu České republiky) spolu s dokumentací a seznamem testů se systém nasadí to testovacího provozu přímo v ŘLP, kde zpravidla půl roku probíhají detailní testy nové verze a následně trénink řídících na nové verzi. V každé z těchto fází nalezení chyby znamená podle velikosti cokoli od rychlé opravy až po znovu započetí celého procesu od vývoje. Pokud nová verze systému projde všemi těmito kroky a je adekvátně certifikována, může být nasazena do provozu a sloužit řídícím dalšího půl roku, než přijde nová verze, která ji nahradí, a cyklus se opakuje.

Takto rozsáhlé systémy je velice složité komplexně testovat, hlavně když je často vyvíjejí firmy menší, než by čtenář očekával. Proto se tento dokument bude zaměřovat na automatizaci jedné z forem testování, konkrétně na uživatelské testování, které provádí testovací oddělení ve vývojářské firmě.

1.1 Zadání

Na konci roku 2020 jsem dostal příležitost stát se členem týmu, který jeden z těchto systémů pro ŘLP vyvíjí, a přestože jsem nastoupil původně jako tester, velice rychle se ze mě stal vývojář a u toho jsem již zůstal. Testování se mi však samozřejmě nevyhýbá, jelikož je potřeba provádět testy již při vývoji nové funkcionality a následně asistovat testovacímu oddělení při tvorbě jejich testů. Už při testování mého ASTERIX dekodéru pro systém IDP, kterému jsem se věnoval ve své bakalářské práci, jsem narazil na problém s tím, jak věrohodně otestovat, že letadla se skutečně zobrazují přesně na správných pozicích, protože jsem neměl způsob, jak zobrazovat pozice letadel dekódovaných starým dekodérem a novým dekodérem na stejné obrazovce. Samozřejmě jsem byl schopen zkontrolovat shodnost hodnot vycházejících z obou dekodérů, ale jelikož výstup dekodérů ani neměl být stoprocentně identický, musel jsem se spolehnout na vizuální kontrolu.

V tu chvíli jsem dostal nápad na systém, který by tuto validaci byl schopen udělat automaticky s přesností na pixely a využíval by to, co uživatel skutečně vidí na obrazovce a nejen data, která jsou na výstupu dekodéru. Hlavním problémem však je to, že se jedná o desktopový systém a nemohu se tedy spolehnout na nejrůznější knihovny, které testují uživatelské rozhraní webových aplikací. Z předchozí zkušenosti s malými automatizačními scripty jsem věděl, že existují knihovny, které umožňují hýbat s myší, psát na klávesnici a hledat shody mezi obrázkem a tím, co je na monitoru, ale chtěl jsem tento koncept posunout na vyšší úroveň.

Hlavní myšlenkou je, že tento nástroj bude umět právě to, co umí běžný uživatel, když sedí u fyzického stroje, dalo by se tedy říci, že se bude jednat o jakéhosi „umělého testera“. Překladem do angličtiny a trochou inspirace z řecké mytologie vznikl projekt ARTEMIS (ARtificial TEster Monitoring Information Systems).

1.1.1 Požadavky na systém

S nárůstem počtu neuronových sítí, které jsou schopny zvládat nejrůznější úkony, jsem si byl optimisticky jist, že bude existovat nějaký nástroj, který bude schopen „číst“ text z obrazovky stejně jako běžný uživatel a nebude potřebovat přístup k interní funkcionalitě systému. Bude se tedy jednat o formu black-box testování. Těmto nástrojům se říká OCR (Optical Character Recognition) a umí převádět obrázky na text. Tím bych měl podchycenou asi největší překážku v realizaci takového projektu.

V ideálním případě by provádění testu nemělo blokovat fyzický stroj, aby bylo možné ho spouštět paralelně a jako součást interních CI/CD[1] procesů, proto je zapotřebí do testovacího nástroje zahrnout i nějakou formu virtualizace.

Kromě interakce s displayem, myší a klávesnicí by měl nástroj umožňovat i pracovat s konfigurací testovaného systému a dynamicky ji měnit za dobu běhu testu. Jelikož systémy, které bude tento nástroj testovat, jsou hodně závislé na příchozích datech po síti, je nezbytné, aby nástroj uměl posílat požadavky po síti a také poslouchat na příchozí a odchozí komunikaci.

V neposlední řadě všechny aplikace, které bude nástroj testovat, jsou postavené na Linuxu, konkrétně na CentOS 7[cen]. To mi umožňuje využít širokou škálu nástrojů, hlavně pro virtualizaci, ale jedná se o jisté omezení, protože Linux není natolik rozšířený jako např. Windows a může se stát, že některé zajímavé nástroje budou pouze pro Windows.

1.1.1.1 Funkční požadavky

FR1. Nástroj musí pracovat na virtualizovaném stroji bez přístupu k fyzické obrazovce.

FR2. Nástroj musí umožňovat validaci dat na obrazovce.

FR3. Nástroj musí umožňovat interakci s myší a klávesnicí.

[1] Continuous Integration / Continuous Delivery(Deployment) představuje sérii opakovatelných kroků potřebných k nasazení softwaru.

FR4. Nástroj musí umožňovat posílání dat po síti a validaci dat přicházejících/odcházejících po síti.

FR5. Nástroj musí umožňovat úpravu konfigurace testované aplikace v průběhu testu.

FR6. Nástroj musí umožňovat zobrazení postupu a výsledku provedeného testu.

FR7. Nástroj musí umožnovat práci v rámci jednoho okna stejně jako v rámci celé obrazovky.

FR8. Nástroj musí fungovat na operačním systému CentOS 7.

FR9. Nástroj nesmí vyžadovat, aby testovaná aplikace byla napsána ve specifickém jazyce nebo knihovně.

1.1.1.2 Nefunkční požadavky

NFR1. Nástroj pracuje alespoň tak rychle, jako pracuje běžný manuální tester.

NFR2. Nástroj provádí všechny kroky exaktně a nezanáší žádnou míru náhody.

NFR3. Nástroj garantuje opakovatelnost testu.

NFR4. Nástroj je uživatelsky přívětivý a umožní vytvářet testy i uživateli s minimální znalostí programování.

1.1.2 Integrace

Aby bylo možné nástroj efektivně využívat a případně ho i zaintegrovat do CI/CD procesů, bude nezbytné, aby byl dostupný v rámci interní sítě a tedy aby byl umístěný na některém z firemních serverů. Aby se k nástroji dalo pohodlně přistupovat a používat ho, je nutné zamyslet se nad tím, jak bude nástroj dostupný pro uživatele, případně jaké další integrace by měl poskytovat. Zvolený způsob integrace také výrazně ovlivní architekturu projektu, proto je potřeba zabývat se tímto tématem co nejdříve. V této kapitole představím několik možností, které lze realizovat.

1.1.2.1 Přístup přes konzoli

První a nejjednodušší variantou je přístup přes konzoli, tedy nástroj je pouze ve formě skriptu umístěn na některém z firemních serverů. Uživatel se k serveru musí připojit přes konzoli, dodat všechny potřebné materiály a parametry a spustit nástroj s daným testem ručně. Tento přístup je sice nejjednodušší pro mě jako autora ARTEMIS, ale rozhodně není uživatelsky přívětivý.

1.1.2.2 TestLink

Lepší alternativou je napojit ARTEMIS na systém pro evidenci testů – v naší firmě se využívá TestLink[tesf]. Tyto systémy typicky poskytují nějakou formu API, na které by se mohl nástroj dotázat a získat tak testy z centrální databáze. Tím odpadá potřeba kopírovat testy na server, ale je stále potřeba připojit se na server přes konzoli, spustit ARTEMIS ručně a předat jí identifikaci testů, které se mají provést. ARTEMIS by však už sama mohla výsledky těchto testů vložit zpět do systému.

Problémem tohoto přístupu je, že tímto efektivně svážu ARTEMIS s aplikací TestLink a pro její další využití bude potřeba velkou část kódu přepsat pro jiné API, což automaticky zhoršuje šanci tohoto projektu na úspěch. V neposlední řadě také bude potřeba vymyslet způsob zápisu testů v TestLinku tak, aby jim ARTEMIS rozuměla, protože TestLink je inherentně aplikace pro evidenci manuálních testů, ne automatických testů.

1.1.2.3 Vlastní webová aplikace

Nejlepší, ale také nejsložitější variantou je vytvořit vlastní webovou aplikaci na správu automatizovaných testů pro ARTEMIS. Nástroj by tím pádem fungoval jako webový server, který poskytuje API pro psaní a spouštění testů. Webová aplikace by také umožňovala snadné sdílení funkcí pro jejich přepoužití i psaní samotných testů pomocí drag-and-drop podobně, jako funguje např. programovací jazyk Scratch[scr]. Tím by se dosáhlo skutečně vysoké míry uživatelské přívětivosti i pro testery bez programátorských znalostí. Také by to nesvazovalo ARTEMIS s žádnou jinou aplikací a bylo by možné ji používat samostatně.

Volba vlastní webové aplikace zní velmi lákavě a dává projektu mnohem větší šanci na úspěch a také mnohem větší uživatelskou přívětivost. Proto součástí tohoto projektu bude vytvoření webové aplikace, která bude interagovat s nástrojem popsaným v kapitole Požadavky na systém. Webová aplikace i samotný nástroj provádějící testy jsou tedy nadále označovány pod společným názvem ARTEMIS.

Kapitola 2

Technologie

Pojďme se nyní zaměřit na seznámení se s technologiemi a přístupy, které budou v tomto projektu aplikované. Nejprve si představíme pojem black-box testování, který určuje strukturu celému projektu a v podstatě diktuje sadu nástrojů, které bude potřeba využít. Následně se podíváme na fungování OCR enginů a jejich způsob využití neuronových sítí v kategorizaci písmen. Na závěr se seznámíme s konceptem virtualizace strojů a vysvětlíme si, proč je zrovna virtualizace, stejně jako OCR, klíčová pro tento projekt.

2.1 Black-box testování

Abychom skutečně pochopili, proč je ARTEMIS strukturovaná tak, jak je, musíme se seznámit s pojmem black-box testování. V rámci black-box testování pracuje tester se systémem, který testuje, jako s tzv. černou skříňkou (black box), do které nevidí a pouze zná její vstupy a výstupy. Konkrétním příkladem je právě testování uživatelského rozhraní, kdy tester např. klikne na tlačítko a očekává, že se otevře nové okno se správnými hodnotami, ale už ho nezajímá, jestli tlačítko zavolalo správnou funkci se správnými parametry, nebo jestli přišla správná odpověď na požadavek poslaný po síti.[KSD15]

Pokud pracujeme s manuálními testery (tedy živými lidmi) je poměrně jednoduché provádět black-box testování, protože to je základní lidský přístup ke každému systému. Naopak při naprogramovaných testech je mnohem jednodušší provádět tzv. white-box testing, kdy nezkoumáme, co se objevilo

Obrázek 2.1: Ilustrace konceptu black-box

uživateli na obrazovce, ale právě to, jestli byly zavolány správné funkce se správnými parametry a v podstatě předpokládáme, že pokud jsou všechna data správně, tak i uživatelský výstup bude správně. Komplikace nastávají právě ve chvíli, kdy chceme po člověku, aby prováděl white-box testování, nebo aby program prováděl black-box testování. Člověk, většinou vývojář, má k dispozici nejrůznější debuggovací nástroje, které mu umožní pozastavit běh programu a podívat se na aktuální stav všech hodnot. Pokud však chceme, aby program uměl právě to, co umí běžný uživatel, je potřeba mít možnost replikovat pohyby myší, psaní na klávesnici a hlavně schopnost „vidět“ obrazovku a vyhodnotit, co se na ní děje, což vyžaduje poměrně komplexní sadu funkcí.

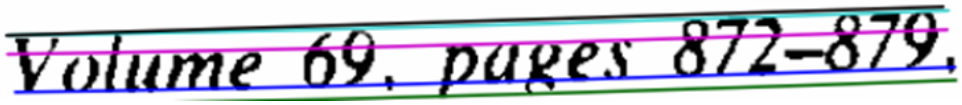
Hlavní výhodou black-box testování prováděného programem je jeho přesnost, tedy že pokud do něj aktivně nezaneseme nějakou míru nejistoty, bude program provádět stejný test pokaždé na pixel přesně tak, jako dříve, což je u běžného člověka nemožné. Program je také schopen otestovat určité limity systému, které jsou nad běžnou lidskou schopnost, třeba psaní velmi rychle na klávesnici, rychlé klikání apod., které mohou odhalit chyby, na které by tester nepřišel, ale mohly by se v krajním případě projevit.

2.2 Optical Character Recognition

OCR je proces rozpoznávání písmen z obrázku. Nástroje pro OCR (také nazývány „enginy“) jsou používány například pro digitalizaci tištěných knih, protože jsou schopny z naskenované stránky vytvořit stránku textu. Stejně tak je tedy možné číst text z obrazovky pomocí zachycení jejího snímku a předáním tohoto snímku OCR enginu, který je schopen z něj vytáhnout veškerý text, který je následně možné porovnat s očekávanými hodnotami. Jelikož interní procesy těchto enginů jsou velmi komplexní a často nejsou veřejně známé, nastíním zde proces jednoho z nich, s názvem Tesseract.

2.2.1 Preprocessing

Prvním krokem je rozpoznání jednotlivých řádků pomocí proložení spodních hran jednotlivých blobů (uskupení pixelů připomínající písmeno nebo písmena) linií, na které všechny bloby „sedí“. Následně se řádky rozdělí nejprve do jednotlivých slov podle velikosti mezer mezi bloby, jelikož mezera mezi slovy je typicky větší než mezera mezi dvěma písmeny ve slově. Každé slovo složené z několika blobů je rozděleno do jednotlivých písmen. Většinou jeden blob odpovídá jednomu písmenu, ale při práci s obrázkem ve špatné kvalitě nebo s nekvalitním tiskem textu se může více písmen spojit do jednoho blobu. Stejně tak se může i stát, že jedno písmeno je rozděleno do více blobů, např. pokud pracujeme s poškozeným nebo jinak nekompletním textem.[Smi07]



(a) rozpoznání řádků

(b) rozdělení blobů

(c) sloučení blobů

Obrázek 2.2: Vizualizace preprocessingu v Tesseractu[Smi07]

2.2.2 Klasifikace

Následně je potřeba každé písmeno klasifikovat a v tuto chvíli přichází na scénu neuronová síť naučená na velkém množství různých písem a jazyků, která ho zkouší klasifikovat pomocí několika desítek vlastností každého písmene. Na tomto procesu se podílí dva klasifikátory – statický a adaptivní. Statický umí generalizovat a není závislý na konkrétním písmu, to však znamená, že některá písmena neumí přesně rozpoznat. Proto je zde adaptivní klasifikátor, který se snaží rozpoznat písmo, nebo ho alespoň připodobnit k nějakému známému, a radí statickému klasifikátoru. Jelikož adaptivní klasifikátor se v průběhu klasifikace textu učí a na konci textu tedy může mít znalosti, které neměl na začátku, je tento klasifikační proces dvouprůchodový, aby měl adaptivní klasifikátor příležitost „vyjádřit“ se ke každému písmenu se všemi znalostmi.[Smi07]

Tento proces je obecně velmi robustní, překvapivě výkonově nenáročný (oproti jiným neuronovým sítím) a relativně spolehlivý, protože moderní OCR enginy dosahují běžně nad 95% přesnosti pro tištěné materiály. Všechny

tyto faktory přispívají k tomu, že OCR je aplikovatelné pro využití v rámci black-box testování.

2.3 Virtualizace

Hlavní myšlenkou virtualizace je abstrakce hardwaru počítače a jeho částečná emulace pomocí softwaru. Díky tomu jsme schopni vytvořit virtuální počítač, který se tváří jako fyzický stroj, a i on sám si myslí, že je fyzickým strojem. V praxi nám tedy umožňuje rozdělit jeden fyzický stroj na více oddělených počítačů. To lze využít například pro spuštění více různých operačních systémů na jednom počítači nebo rozdělení jednoho výkonného počítače (typicky serveru) na více slabších počítačů, které mohou sloužit pouze jednomu účelu a jsou mezi sebou izolované. Jelikož je celý virtuální stroj řízený pomocí programu, jsme také schopni virtuální stroj pozastavit nebo přesunout na jinou lokaci, aniž by se programy ve virtuálním stroji nějak porušily nebo vyžadovaly restart. Vytváření a mazání těchto virtuálních strojů je také výrazně jednodušší než u fyzických strojů a pro jejich vytváření je možné použít šablony, které zajistí, že každý nový virtuální stroj je identický tomu předchozímu.

Pro automatizaci testování jsou však klíčové dva faktory:

- Jeden stroj je možné rozdělit na mnoho dalších, což pomůže paralelizaci prováděných testů.
- Pomocí šablon se dají vytvořit identické kopie systému velmi snadno, což zajišťuje konzistenci testů napříč jejich spouštěními.

Kapitola 3

Rešerše

Po úvodní analýze zadání je na místě podívat se na nástroje, které bych mohl při vývoji využít a zaměřit se také na existující systémy, které by mohly kompletně řešit celé zadání. V rámci rešerše je potřeba se zaměřit na několik oblastí, konkrétně způsob virtualizace, OCR engine, nástroj pro automatizaci pohybu myší a psaní na klávesnici, nástroj pro záznam obrazovky a nástroj pro zpracování záznamů obrazovky před tím, než se předá OCR enginu. Zde také stojí za to zmínit, že v zadání projektu sice není specifikováno, v jakém jazyce má být nástroj napsaný, ale jelikož velká část programů a nástrojů v našem týmu je psaná v Pythonu, dává největší smysl zvolit tento jazyk. V rámci rešerše se zaměřím převážně na nástroje integrovatelné do Pythonu.

3.1 Existující řešení

Na internetu je dostupná široká škála komerčních i nekomerčních nástrojů, poskytujících automatizované testování desktopových aplikací. Když se ale zaměříme na konkrétní funkcionalitu, ani jeden z těchto nástrojů neposkytuje veškerou funkcionalitu, kterou musí obsahovat ARTEMIS. Identifikoval jsem čtyři hlavní kategorie těchto nástrojů, podle toho, jaký požadavek nesplňují. V každé této sekci zmíním jedno řešení jako příklad.

3.1.1 Řešení pro Windows

Největší část nástrojů poskytuje automatizaci pouze pro aplikace na Windows. Jsou tedy závislé na knihovnách pro interakci s myší a klávesnicí pro Windows. Příkladem takového nástroje je AutoIt[aut], který jinak vypadá velmi slibně. Dodržuje principy black-box testování, protože využívá pouze pohyb myší a psaní na klávesnici, umožňuje psaní testů pomocí scriptů, umožňuje manipulovat s procesy a mnoho dalšího, ale pouze pro Windows. Tato kategorie nástrojů je tedy pro nás nepoužitelná.

3.1.2 Řešení bez black-box

Druhá sada se snaží tomuto problému vyhnout a propaguje se jako multiplatformní, pro Windows, Linux, Android atd. Svojí multiplatformity však tyto nástroje dosahují pomocí spolehnutí se na mupliplatformitu knihoven, na kterých staví. Příkladem takového nástroje je Ranorex[ran], který podporuje testování i na Linuxu, ale pouze proto, že se spoléhá na knihovny jako je Swing nebo Qt, které fungují jak na Windows, tak na Linux. Tento nástroj nedodržuje princip black-box, protože kdyby byla testovaná desktopová aplikace napsaná v nějaké knihovně, která není podporovaná, nástroj by se nedal využít.

3.1.3 Řešení bez virtualizace

Zatím nejslibnější sadou nástrojů jsou ty, které skutečně jsou multiplatformní bez závislosti na knihovnách, ale bohužel nepodporují virtualizaci, respektive nejsou tzv. headless, tedy vyžadují připojený display. Nejzajímavějším nástrojem v této sekci se SikuliX[sik], který je zdarma, funguje na Windows, Linux i MacOS, dokonce má i OCR engine Tesseract, ale v návodu jasně specifikuje, že vyžaduje mít připojený display. Tohle by nebyl nepřekonatelný problém, pokud by stačilo spouštět testy sekvenčně na dedikovaném stroji, jako když je pouští vývojáři, ale pro komplexní testování není možné tento nástroj škálovat podle potřeby.

3.1.4 Řešení bez vyžadovaných modulů

Sada, která se nejvíce blíží představě popsané v zadání, je ta, která je jak skutečně multiplatformní, tak podporuje virtualizaci. Příkladem je nástroj ZAPTEST[zap], který je zjevně velice komplexní a podporuje vše od jednoduchých testů přes no-code testy a komplexní reporty až po integrace s nejrůznějšími CI/CD nástroji jako Jira nebo Jenkins. Jedinou nevýhodou, a v tomto případě kritickou, je nepřítomnost modulů pro práci s konfigurací a pro posílání a přijímání dat po síti, což je pro ARTEMIS zásadním požadavkem. Proto ani tyto nástroje není možné použít.

3.1.5 Závěr

Bohužel žádný z nástrojů dostupných na internetu není z různých důvodů vhodným adeptem pro nasazení. Čemu jsem se zatím vůbec nevěnoval je fakt, že komerční nástroje jsou velmi drahé, pohybující se v tisících dolarů za jednu licenci, zatímco vývoj vlastního řešení by vyšel rozhodně levněji v poměru k počtu uživatelů, kteří budou ARTEMIS využívat.

3.2 Nástroje pro virtualizaci

Prvním klíčovým parametrem pro směr vývoje ARTEMIS je zvolit správný způsob virtualizace. Existuje mnoho typů virtualizace, ale přesnější by bylo mluvit spíše o úrovních virtualizace, jelikož se typicky liší v tom, jakou úroveň stroje virtualizují, od instrukční sady až po aplikaci. V této kapitole si nejprve představíme jednotlivé typy a následně se podíváme na možné adepty a popíšeme si jejich výhody a nevýhody. Před popisem jednotlivých úrovní je nutno poznamenat, že existuje mnoho různých pohledů na úrovně virtualizace, já zde tedy budu popisovat ten, který se zabývá pouze hlavními úrovněmi, ale v praxi lze každou z těchto úrovní dále dělit.[RHFN+12]

3.2.1 Úrovně virtualizace

Nejnižší úrovní virtualizace je virtualizace instrukční sady procesoru. Tato virtualizace se používá např. pro emulaci starých herních konzolí na moderních

strojích. Protože tyto konzole využívaly procesory dramaticky odlišné od moderních, je potřeba v podstatě virtualizovat samotný procesor včetně ostatních částí hardwaru původní konzole. V případě ARTEMIS však není potřeba zacházet tak „hluboko“ s virtualizací, protože aplikace, které bude testovat, fungují na běžných moderních procesorech.

Druhou, a asi nejrozšířenější, úrovní je virtualizace hardwaru počítače. To může na první pohled znít hodně podobně jako první vrstva, ale zde už se nezabýváme konkrétní instrukční sadou, ale pouze rozdělujeme jednu část hardwaru, např. procesor, na více částí, které jsou však funkčně identické s fyzickou verzí. Jelikož moderní hardware je velice výkonný, je možné rozdělit jeho výkon mezi více virtuálních strojů, které si „myslí“, že mají svůj dedikovaný hardware. Běžný počítač je možné takto rozdělit orientačně na jednotky strojů, ale výkonné servery lze rozdělit i na stovky až tisíce virtuálních strojů. Tato úroveň virtualizace by se dala využít pro potřeby ARTEMIS a jelikož je velmi rozšířená, existuje pro ni i mnoho nástrojů.

Další úrovní, a v posledních letech také velmi populární, je virtualizace operačního systému. Zde už nás nezajímá, jaký hardware je ve stroji, ale pouze jaký operační systém k tomuto hardwaru přistupuje. Tato úroveň virtualizace umožňuje jednu instanci operačního systému efektivně rozdělit na více částí, které přistupují ke sdílenému hardwaru. I tato úroveň virtualizace by byla pro ARTEMIS vhodná, protože stačí virtualizovat jen danou aplikaci a není potřeba řešit, jestli procesor, paměť nebo disk přiřazený této aplikaci je sdílený nebo není.

Nejvyšší úrovní virtualizace je takzvaná aplikační. O tuto úroveň se stará jazyk, ve kterém je daná aplikace napsaná. Tyto virtualizační platformy slouží často pro výše zmíněnou multiplatformitu, protože umožňují stejný kód (tedy stejnou aplikaci) spustit na jakémkoli stroji nezávisle na jeho architektuře a operačním systému. Stačí, když daný jazyk podporuje danou architekturu a operační systém a je schopen překládat příkazy přicházející z aplikace do příkazů pro operační systém. Tato úroveň není vhodná právě proto, že nesplňuje požadavek na black-box, jelikož je závislá na jazyku, ve kterém je aplikace napsaná, zatímco ARTEMIS musí umět testovat aplikace psané v různých jazycích.

3.2.2 Virtualizace hardwaru

Mezi hlavní výhody virtualizace hardwaru patří hlavně možnost virtualizovat jiný operační systém, než je ten, který virtualizaci hostuje. Další výhodou je

rozšířenost tohoto typu virtualizace, protože existuje spousta různých nástrojů, V naší firmě využíváme konkrétně nástroj VMware[vmw] pro virtualizaci aplikačních serverů.

Možnost různého operačního systému je určitě hodnotná, ale jelikož v našem případě virtualizační server bude mít jistě operační systém Linux, stejně jako naše testované aplikace, není to pro ARTEMIS klíčovým parametrem. Druhou nevýhodou je, že virtualizace hardwaru je časově a zdrojově náročnější než vyšší vrstvy. Jelikož bude virtualizovaný stroj pouze na jedno použití pro daný test, aby se zajistila konzistentnost testů, jedná se o velkou režii v poměru k využitelnosti. V neposlední řadě VMware je komerční nástroj, přidali bychom tedy potenciálně firmě velké náklady s dalšími licencemi, nebo museli nasadit alternativu, která bude zdarma.

3.2.3 Virtualizace operačního systému

Virtualizace OS řeší mnoho problémů, které má virtualizace hardwaru, aniž by z toho vznikalo příliš mnoho nevýhod. Tato úroveň virtualizace je časově a zdrojově poměrně nenáročná, je tedy reálné vytvořit virtuální stroj pouze na pár minut pro spuštění testu, aniž by jeho tvorba a následné rušení trvalo déle než samotný test.

Jelikož virtualizační server i virtuální stroje jsou postavené na Linuxu, je možné využít nástroj Docker[Doc], který se pro toto využití perfektně hodí. Umožňuje použít různé distribuce Linuxu, což povoluje dostatečnou variabilitu, a podporuje i komplexní strukturu šablon pro snadné vytváření nových strojů. Dalším bonusem je to, že existuje knihovna pro Python, která umožňuje správu virtuálních strojů, je tedy možné v rámci ARTEMIS zahrnout snadno i tuto část.

Nevýhodou Dockeru je to, že není možné použít „obyčejnou“ verzi zvolené distribuce Linuxu, ale je potřeba mít verzi vytvořenou přímo pro využití v Dockeru. Jelikož je Docker volně dostupný a open-source, existuje velká komunita jeho uživatelů a všechny operační systémy, které ve firmě používáme, mají svou verzi pro Docker. Jedinou skutečnou nevýhodou Dockeru je to, že musíme věnovat úsilí přípravě těchto šablon pro samotné aplikace a následně šablony upravit pro potřeby ARTEMIS, ale jelikož část našich aplikací už takto převedena je, stačí dokončit zbývající. Pro potřeby tohoto projektu tedy využiji Docker jako virtualizační platformu. Psaní základních šablon nebude součástí této práce, pouze nadstavba pro potřeby ARTEMIS.

3.3 OCR enginy

Stejně jako u virtualizace je i u OCR enginů široké spektrum možných kandidátů na integraci. Pro zvolení správného enginu je klíčové vybrat takový, který je dostatečně přesný, rychlý a nenáročný. V ideálním případě by zvolený engine měl být zdarma a kvůli bezpečnostním pravidlům musí podporovat lokální nasazení, nemůže se tedy jednat o tzv. cloudové řešení.

3.3.1 Porovnání enginů

Pokud se zaměříme na přesnost a rychlost jednotlivých OCR enginů, jednoznačně vyhrávají již zmíněná cloudová řešení od Googlu, Amazonu nebo Microsoftu.[Dil23] Ty však nemohu využít kvůli bezpečnostnímu riziku, nemluvě o ceně za zpracování jednoho obrázku.

Existují i komerční nástroje, jako třeba Nuance OmniPage, které v určitých případech převyšují svou přesností i některá cloudová řešení, ale licence těchto softwarů nejsou nejlevnější a ne vždy podporují nahrávání souborů pomocí API, protože se soustředí na manuální převod PDF souborů do textu.[Tom17]

Existuje také velké množství nekomerčních open-source řešení jako GOCR nebo CuneiForm, která však nedosahují takové přesnosti, jako ostatní řešení. [Tom17] Na druhou stranu je potřeba uznat, že se většinou jedná jen o několikaprocentní rozdíl, kdy všechny nástroje mají přesnost nad 90%. Z množiny open-source enginů však vyvstává jeden – Tesseract. Konzistentně dosahuje lepších výsledků než jiné nekomerční i komerční enginy a v některých případech překonává i cloudová řešení.[RJN95][Tom17][Dil23]

3.3.2 Tesseract

Tesseract je pouze samotný OCR engine, je tedy velmi kompaktní a poskytuje integraci přes své API, které mohu využít. Existuje pro něj i knihovna v Pythonu. Jedinou nevýhodou, kterou zmiňují některé zdroje, je jeho rychlost.[Tom17] Vzhledem ke své přesnosti velmi pravděpodobně obětuje více času pro lepší výsledek než konkurenční enginy. V rámci velice rychlého testu jsem si však

ověřil, že jeho rychlost je více než přijatelná, protože zpracování jednoho obrázku trvá pouze zlomek vteřiny, což je pro mé využití dostatečné.[1]

3.3.2.1 Historie

Tesseract byl původně vyvinut ve společnosti HP mezi roky 1984 a 1994 a začal jako doktorská práce na Univerzitě v Bristolu. Jeho původní název byl HP Labs OCR a pod tímto názvem se také poprvé objevil v dokumentu srovnávajícím přesnost OCR enginů v roce 1995, kde předčil všechna dosavadní komerční řešení. Jeho plánované využití bylo jako integrace pro tehdejší skenery HP, ale k tomu nakonec nedošlo. V roce 1996 došlo k převodu Tesseractu z platformy HP Workstation na Windows a následně v roce 1998 k jeho přepisu do C++.[Smi07]

V roce 2005 vydalo HP Tesseract jako open-source software a jeho správu převzal Google, který ho dále rozvíjel až do roku 2018. Dnes je jeho kód dostupný na GitHubu s poslední verzí 5.0.0 vydanou v roce 2021.[tesc]

3.3.2.2 TessBaseAPI

Jak jsem již zmínil v kapitole Historie, Tesseract je aktuálně napsaný v C++ a v něm poskytuje i své oficiální API s názvem TessBaseAPI[tesd]. Toto API poskytuje široké spektrum funkcí, které umožňují konfigurovat interní procesy v Tesseractu. Hlavní výhodou však je, že poskytuje několik formátů výstupu – jednak čistý text a jednak formátované výstupy obsahující pozici každého slova a jeho „confidence“, tedy jak moc si je Tesseract jistý, že rozpoznal dané slovo správně.

Hlavní nevýhodou TessBaseAPI je, že je napsané v C++, což znamená, že by bylo poměrně komplikované ho zaintegrovat. Na druhou stranu vím, že to není nemožné a mohu se o integraci pokusit, pokud by to bylo nutné.

[1] Rychlostí Tesseractu se budu věnovat detailněji v budoucnu, proto zde toto téma dále nerozvíjím.

3.3.2.3 Knihovna Pytesseract

Knihovna Pytesseract[pyt] řeší problém integrace TessBaseAPI do kódu v Pythonu, je tedy velice jednoduché ji zaintegrovat. Jelikož je volně dostupná přes package manager PIP, není složité ji nainstalovat. Ke svému běhu vyžaduje pouze „runtime“ verzi Tesseractu, kterou není složité nainstalovat v nejnovější verzi na cílový stroj. Jedinou drobnou nevýhodou je fakt, že neposkytuje tak detailní funkcionalitu jako TessBaseAPI, ale i tak poskytuje formátovanou verzi výstupu s pozicemi a „confidence“, což se bude hodit pro implementaci v ARTEMIS.

3.4 Nástroje pro automatizaci

Interakce s myší a klávesnicí jsou ty nejdůležitější funkce, které musí ARTEMIS ovládat, protože se jedná o hlavní způsob práce s desktopovou aplikací. V této sféře se nabízí několik možných řešení, jelikož projekt bude realizovaný v Pythonu, zaměříme se tedy na knihovny s ním kompatibilní. Požadavky na tyto knihovny jsou poměrně jednoduché, knihovna musí umožňovat stisknutí klávesy či tlačítka, musí umět hýbat s kurzorem a v neposlední řadě musí spolupracovat s nástroji, které byly vybrané v předchozích kapitolách.

3.4.1 Knihovna PyAutoGUI

Jednou z nejpopulárnějších knihoven zaměřenou na automatizaci je PyAutoGUI [pya], která umožňuje jednak pomocí jednoduchého API interagovat s myší a klávesnicí, ale také zaznamenávat obrazovku, k tomu se však dostaneme v kapitole Nástroje pro záznam obrazovky. Prozatím je klíčové, že PyAutoGUI lze snadno integrovat do jakéhokoli projektu v Pythonu a osobně s ní mám už předchozí zkušenosti.

Původně jsem předpokládal, že právě PyAutoGUI bude jednoznačná volba, ale narazil jsem na v podstatě nepřekonatelný problém. Pro jeho vysvětlení se musíme vrátit až k tématu virtualizace. Jak bylo zmíněno, jednou z klíčových vlastností ARTEMIS má být, že nevyžaduje připojení fyzického monitoru. V praxi to znamená, že takto virtualizovaný systém musí umět pracovat s virtuálním displayem. Právě s tím má ale knihovna PyAutoGUI problém, kvůli svému modulu na snímání obrazovky. Tento problém je tak vážný, že i když

jsem nezkoušel snímat obrazovku a pouze hýbat kurzorem, tak se PyAutoGUI odmítalo spustit a blokovalo tím celý zbytek testu. Po neúspěšném hledání alternativního řešení jsem musel přijmout fakt, že PyAutoGUI jednoduše nebude možné integrovat s našimi systémy virtualizovanými pomocí Dockeru.

3.4.2 Knihovna XTest

Druhou a mnohem méně známou variantou pro automatizaci je knihovna XTest[xte]. Tato knihovna je primárně napsaná v C, ale existuje pro ni tzv. wrapper v Pythonu, který by bylo možné integrovat. Specificky knihovna XTest má výhodu oproti jiným alternativám v tom, že emuluje myš a klávesnici přímo na úrovni XServeru, což je nejrozšířenější nástroj pro zobrazování GUI v Linuxu, a v podstatě všechny populární distribuce Linuxu ho využívají nebo alespoň podporují. To znamená, že mohu využít výhody přímého přístupu na poměrně nízkou úroveň zpracování uživatelského vstupu, aniž bych musel porušit úroveň black-boxu, na rozdíl od nástrojů popsaných v kapitole Řešení bez black-box. Další výhodou knihovny XTest je, že je v podstatě automaticky integrovatelná s ostatními nástroji, protože pokud jsem schopen zprovoznit systém běžící na XServeru v Dockeru, pak i tato knihovna bude spolupracovat s tímto způsobem virtualizace.

3.5 Nástroje pro záznam obrazovky

Víme již, jakým způsobem budeme číst data z obrazovky, ale OCR enginy samy o sobě nepodporují snímání obrazovky a počítají s tím, že na vstupu už dostanou připravený obrázek. Je tedy potřeba najít nástroj, resp. knihovnu, která bude umožňovat uložit snímek obrazovky jako obrázek, který bude následně předán Tesseractu ke zpracování.

3.5.1 Knihovna PyAutoGUI

Jak jsem již zmiňoval v předchozí kapitole o PyAutoGUI v rámci automatizace, tato knihovna podporuje i snímání obrazovky. Nejen to, ale umožňuje i na obrazovce vyhledat pozici nějakého obrázku. Tato funkce se využívá pro automatizaci interakce s nějakým objektem na obrazovce, například s tlačítkem. Stačí totiž mít obrázek daného tlačítka a PyAutoGUI je schopné

toto tlačítko na obrazovce najít a kliknout na něj. Právě tato funkce se často využívá jako primitivní verze „čtení obrazovky“ bez použití OCR, protože v praxi poskytuje velmi podobnou funkcionalitu, pokud uživateli stačí vyhledávání objektů a nepotřebuje přímo čtení textu.

Stejně jako v případě automatizace však není možné knihovnu PyAutoGUI využít, protože právě modul pro snímání obrazovky nespolupracuje se strukturou projektu, a je tedy potřeba najít alternativu.

3.5.2 Knihovna Pillow

Velice slibnou alternativou je knihovna Pillow[pil], která je v popularitě efektivně na druhém místě za knihovnou PyAutoGUI pro snímání obrazovky. Tato knihovna poskytuje širokou škálu funkcí pro práci s obrázky, pro jejich vytváření, úpravu a zobrazování. Mimo to poskytuje i nástroj pro vytvoření snímku obrazovky, který následně vrátí jako datovou strukturu, kterou je možné buď uložit do souboru nebo s ní dále pracovat.

Nejen, že podporuje Linux, ale právě ve funkci na vytvoření snímku obrazovky odkazuje na integraci s XServerem, což znamená velkou pravděpodobnost interoperability, kterou jsem si následně ověřil i v jednoduchém testu.

3.6 Nástroje pro image processing

Aktuálně spíše bonusovou sekcí jsou nástroje na image processing. Image processing lze využít pro úpravu obrázků po tom, co jsou zaznamenány knihovou Pillow a před tím, než přejdou do Tesseractu. Tato úprava může často pomoct s přesností výstupu OCR nástrojů a je klíčová pro jiné způsoby analýzy obrázků, ale Tesseract umí pracovat i s neupravenými snímky obrazovky, proto není aktuálně nutné mít nástroj na image processing, ale v budoucnu se bude pravděpodobně hodit pro získání lepších výsledků.

3.6.1 Knihovna OpenCV

Ve sféře image processingových knihoven v Pythonu existuje de facto jediná možnost, kterou je knihovna OpenCV[opeb]. Tato knihovna je funkčně velmi rozsáhlá a poskytuje v podstatě veškeré funkce využívané v moderním image processingu. Na druhou stranu tato funkčnost přichází na úkor uživatelské přívětivosti, protože účelem této knihovny je spíše umožnit profesionálům detailní zpracování obrazových dat než jen běžnou úpravu obrázků, kterou poskytuje knihovna Pillow. Bonusem této knihovny je její headless verze, vytvořená speciálně pro využití na serverech nebo v rámci virtualizace, kdy není připojený fyzický monitor, je tedy snadno kompatibilní s mým navrženým prostředím.

3.7 Nástroje pro získání předpisu funkce

Pro zlepšení uživatelské přívětivosti a pro zjednodušení tvorby uživatelské dokumentace jsem se rozhodl automaticky generovat seznam funkcí poskytovaných nástrojem ARTEMIS. Aby bylo možné dynamicky poskytovat seznam dostupných funkcí a testů, nestačí pouze načítat názvy souborů ve složce, ale je nutné z daného Python modulu získat seznam funkcí, které poskytuje, jejich parametry a jejich návratové hodnoty. Kromě toho je potřeba být schopen rozpoznat jednotlivé řádky testu tak, aby se z nich dal vyrobit blok pro drag-and-drop.

Jednou z variant řešení je vzít Python modul jako text a pomocí vlastní znalosti syntaxe rozpoznávat jednotlivé parametry a volání. Avšak existují knihovny, které tuto funkci poskytují a které velmi pravděpodobně budou mít mnohem kvalitnější porozumění syntaxi, než co bych byl schopen napsat já. Pojďme se tedy podívat na možné kandidáty.

3.7.1 Knihovna Inspect

Knihovna Inspect[ins] poskytuje možnost velice snadno získat tzv. signature funkce, tedy její název, parametry a návratovou hodnotu. Díky tomu lze sestavit dynamicky dokumentaci, která zobrazuje všechny volatelné funkce v rámci ARTEMIS.

Má však dvě nevýhody. Jednak neumí načíst informace o obecném řádku kódu, rozdělování do bloků pro drag-and-drop by tedy muselo být řešené jinak, a hlavně pro velkou část svých funkcí vyžaduje, aby byl zkoumaný modul importovaný. To na první pohled nezní jako problém, ale je potřeba si uvědomit, že tyto moduly jsou psané tak, aby fungovaly uvnitř Docker kontejneru s testem, takže samy o sobě obsahují nevalidní importy. Jedním typem jsou špatné cesty k modulům, protože složková struktura na serveru se může lišit od složkové struktury v kontejneru. Druhým a výraznějším typem jsou knihovny, které jsou nainstalované v kontejneru, ale nejsou nainstalované na serveru a v některých případech ani nejdou nainstalovat, protože např. vyžadují alespoň virtuální display (viz. kapitola XTest), který ale na serveru rozhodně nebude.

3.7.2 Knihovna Abstract Syntax Tree

Knihovna AST[ast] řeší problém z jiného směru a mnohem obecněji. Přijímá totiž vstup ve formě textu (který je možné získat načtením souboru bez importování) a následně předpokládá, že se jedná o kód v Pythonu, který po jednotlivých řádcích načte a vytvoří list objektů, které je reprezentují. Práce s AST je sice komplikovanější, ale poskytuje mnohem širší spektrum funkcí, tedy jak získání signature funkce, tak identifikaci každého řádku v dané funkci a rozdělení do jednoho z mnoha typů. Jelikož nevyžaduje import modulu, tak nemá problém s nevalidními importy uvnitř tohoto modulu. A celkově nevyžaduje ani předložení kompletního modulu a umí rozpoznat i jeden samotný řádek. Tím pádem pokrývá všechny mé požadavky.

3.8 Nástroje pro webový server

Jak jsem již zmínil v kapitole Vlastní webová aplikace, součástí projektu je také webová aplikace, která se bude napojovat na API a přes něj spravovat a spouštět testy. Toto API bude realizované webovým serverem, který bude psaný v Pythonu, aby mohl snadno interagovat s ostatními částmi nástroje, které jsou taky v Pythonu. Co však není přesně definované je, jakou knihovnu pro tvorbu webového serveru budu využívat, pojďme se tedy podívat na možné kandidáty.

3.8.1 Django

Knihovna Django[dja] je robustní a zavedená knihovna pro tvorbu webových serverů v Pythonu. Její hlavní výhodou je dobrá škálovatelnost a už v základu poskytuje autorizaci a umožňuje tvořit i frontend webových aplikací.

V tomto konkrétním případě však nemají její výhody příliš velký přínos. Aplikaci bude používat omezený počet lidí (firma čítá cca 30 lidí, z nichž jen část bude využívat ARTEMIS), takže škálovatelnost není klíčová. Aplikace také bude fungovat pouze v interní síti a nebude poskytovat žádné funkce specifické pro konkrétního uživatele, takže ani autorizace není potřeba. Frontend ARTEMIS chci vytvořit sám v jiném frameworku a v neposlední řadě, Django je velice komplexní knihovna a je tedy složité pro nezkušeného programátora do ní proniknout. Jelikož nemám s Djangem žádnou předchozí zkušenost a hledám co nejjednodušší variantu, není pro mě vhodným kandidátem.

3.8.2 Flask

Knihovna Flask[fla] řeší spoustu problémů Djanga, jelikož je jednoduchá, je snadné integrovat ji do stávající aplikace a umožňuje rychle iterovat při vývoji. Nelze říci, že by Flask měl nějaké konkrétní nevýhody, které by se dotýkaly tohoto projektu, ale přesto jsem se rozhodl ji nezvolit, protože existuje jedna mírně lepší varianta.

3.8.3 FastAPI

Knihovna FastAPI[fasa] sdílí všechny výhody Flasku, je tedy jednoduchá na použití, nevyžaduje příliš kódu pro integraci a je přívětivá k novým uživatelům. Navíc však v základu poskytuje možnost psát asynchronní funkce, což se hodí při volání dalších API, práci s vlákny nebo při práci s datovými toky, které je často potřeba tzv. awaitovat[2]. FastAPI umožňuje psát nejvyšší úroveň funkcí asynchronně a je tedy možné zmiňované asynchronní volání realizovat. Další velkou výhodou je automatické generování OpenAPI[opea] dokumentace podle kódu a následné vytvoření interaktivního Swagger UI[swa], přes které je možné vytvořené API testovat.

[2]Funkce await umožňuje kódu počkat na dokončení asynchronní funkce, která by se „spustila na pozadí“ a nečekalo by se na její dokončení. Await však lze použít jen v další asynchronní funkci.

3.9 Nástroje pro webový frontend

Po zvolení knihovny pro tvorbu webového serveru je potřeba zvolit knihovnu pro tvorbu webového frontendu, se kterým bude uživatel interagovat. V tomto prostředí existuje nepřeberné množství variant, ale jelikož nejsem příliš zkušený frontend vývojář, zvolil jsem ten framework, se kterým mám dobré zkušenosti a který také poskytuje širokou sadu knihoven, které budu chtít v projektu využít.

3.9.1 React

Knihovna React je velice populární knihovnou pro tvorbu moderních webových aplikací. Pro tento projekt je klíčové, že React není příliš komplikovaný a umožňuje tvorbu i relativně jednoduché webové aplikace. Co je však ještě důležitější, jsou dostupné moduly poskytující konkrétní komponenty, které budu v projektu integrovat. Konkrétně bych zde chtěl zmínit dvě, které budou poskytovat hlavní funkcionalitu aplikace.

Prvním modulem je CodeMirror[cod], který poskytuje velice elegantní řešení pro webovou verzi IDE[3], která se bude hodit pro správu testů. Kromě základního editoru poskytuje i možnosti vlastního napovídání při psaní textu, které je možné využít pro jakousi integrovanou dokumentaci. Dále poskytuje i možnost vlastního lintingu, tedy statické kontroly kódu a zvýrazňování chyb, což může být využito pro validaci správnosti testu již na straně klienta. V neposlední řadě podporuje zobrazení MergeView, které vizuálně zobrazuje rozdíly mezi dvěma verzemi souboru, což lze využít pro kontrolu provedených změn před uložením.

Druhým modulem je React DnD[rea], který umožňuje pracovat s drag-and-drop elementy v rámci určeného prostoru na stránce. Konkrétním využitím v aplikaci bude možnost skládat kroky testu jako jednotlivé drag-and-drop elementy, představené v kapitole Vlastní webová aplikace.

[3]Integrated Development Environment je pro potřeby tohoto projektu brán jen jako typ textového editoru zaměřený na psaní kódu.

3.10 Závěr

Ukázali jsme si, že žádné již existující nástroje nesplňují všechny požadavky kladené na systém ARTEMIS a je tedy nezbytné vytvořit vlastní nástroj. Jako virtualizační nástroj využiji Docker, jelikož umožňuje dostatečnou míru oddělení virtualizovaných strojů spolu s jednoduchostí integrace. Jako OCR engine jsem zvolil Tesseract, který, přestože je open-source, často dosahuje stejných i lepších výsledků než komerční řešení. Knihovnu PyAutoGUI není možné využít, proto jsem musel najít alternativy v oblasti automatizace (knihovna XTest) a snímání obrazovky (knihovna Pillow). Představil jsem knihovnu OpenCV, která sice nepokrývá žádný konkrétní požadavek, ale její funkce v oblasti image processingu se mohou hodit pro zpřesnění výstupu z Tesseractu. K tvorbě dynamické dokumentace využiji jako nejvhodnější knihovnu Abstract Syntax Tree, jelikož nevyžaduje import zkoumaného modulu a umí identifikovat jednotlivé řádky kódu. Pro tvorbu webového serveru jsem zvolil knihovnu FastAPI pro její jednoduchost a integraci se Swaggerem. V neposlední řadě jsem ukázal, proč bude knihovna React nejvhodnější pro tvorbu frontendu pro tento projekt. Na základě těchto znalostí se mohu přesunout k fázi návrhu architektury systému ARTEMIS.

Kapitola 4

Architektura

Po výběru všech potřebných knihoven a nástrojů se můžeme zaměřit detailněji na architekturu celého projektu. V této kapitole si nejprve představíme ukázkové nasazení a popíšeme si fyzické rozmístění jednotlivých částí projektu. Dále se zaměříme na jádro projektu, na jeho části a jak komunikují jednak mezi sebou a jednak s knihovnami, které jsme zvolili. Pak si představíme celou sadu funkcí, kterou ARTEMIS poskytuje, podíváme se detailně na vybraný proces uvnitř projektu a představíme si datový tok s ním spojený. Na závěr si popíšeme strukturu testů, požadavky na šablonu pro Docker a strukturu API, na které se bude napojovat frontend webové aplikace.

4.1 Nasazení

Klíčovou částí k pochopení nasazení projektu ARTEMIS je seznámení se s tzv. manager-agent modelem, který je pro tento projekt ideální. Následně si popíšeme jeho konkrétní implementaci v rámci ARTEMIS.

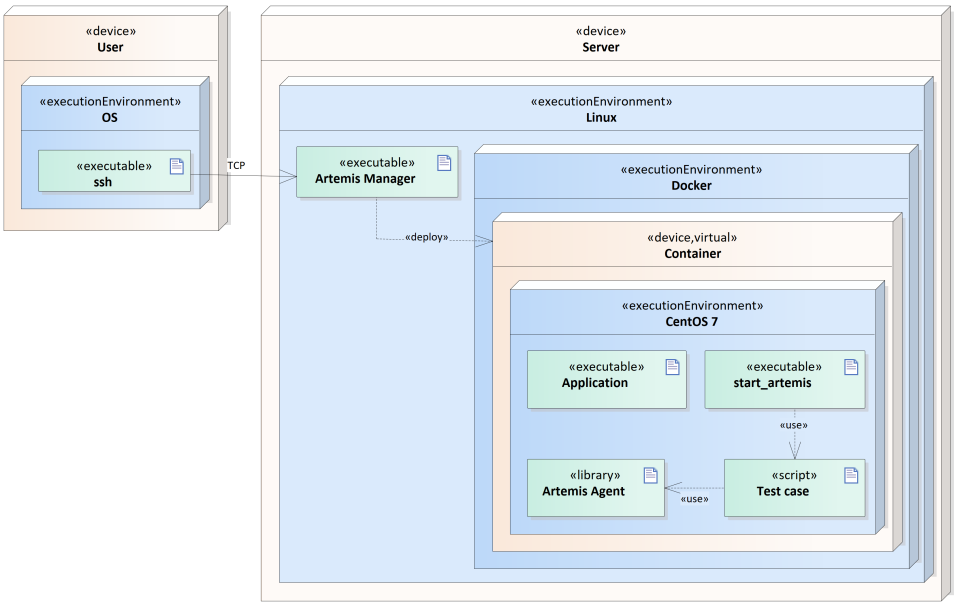
4.1.1 Manager-agent model

Manager-agent model (nebo také master-slave model) vychází z principu jednoho hlavního uzlu (managera), který spravuje a rozděluje práci mezi

podřízené uzly (agenty). Klíčovou vlastností je, že manager komunikuje se všemi agenty, ale agenti mezi sebou nekomunikují. Tento model je velmi užitečný pro paralelizování procesů, kde lze velké množství dat rozdělit do menších bloků, ty nezávisle zpracovat a následně spojit mezivýsledky do jednoho finálního výstupu.

4.1.2 Využití modelu v projektu

V případě ARTEMIS můžeme brát jako vstup sadu testů, které je potřeba provést. Testy na sobě nejsou nijak závislé, takže je možné je spustit paralelně a pouze posbírat výsledky jednotlivých testů. Manager-agent model se pro tento problém velmi dobře hodí, proto jsem ho zvolil ve spojení s virtualizací pomocí Dockeru.

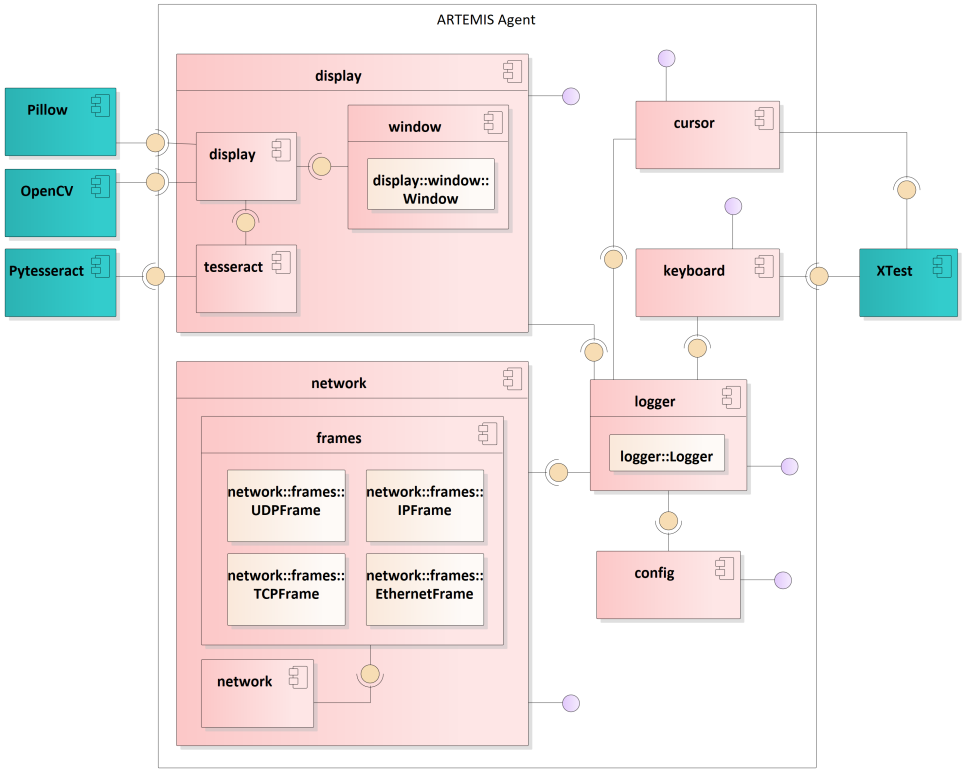


Obrázek 4.1: Diagram nasazení projektu

ARTEMIS manager je jako aplikace umístěný na serveru a dostupný pro uživatele prostřednictvím webového API (resp. webové aplikace). Uživatel předá manageru seznam testů, které chce spustit a manager pro každý test zvlášť založí vlastní virtuální instanci testované aplikace (v rámci Dockeru nazývané kontejner). Tento kontejner obsahuje kromě testované aplikace i ARTEMIS agenta ve formě knihovny, předpis testu a aplikaci, která test spustí. Manager sleduje stav všech kontejnerů, kterých může být paralelně spuštěných několik (limitem je teoreticky pouze výkon serveru) a pokud kontejner skončí, znamená to, že skončil i jeho test, proto z něj získá výsledek testu, který následně reportuje uživateli.

4.2 Moduly

Pojďme nyní o úroveň níže a zaměřme se na strukturu ARTEMIS agenta, který je jako knihovna vložen do každého kontejneru. V rámci požadavků jsem identifikoval šest hlavních funkčních bloků: práce s myší (cursor), práce s klávesnicí (keyboard), práce s obrazovkou (display), práce se sítí (network), práce s konfigurací (config) a záznam progresu (logger).



Obrázek 4.2: Diagram struktury modulů

Každý z těchto bloků bude reprezentovaný jedním Python modulem, který vystavuje určitou sadu funkcí ve vlastním API. Nabízelo by se jít cestou OOP a nevytvářet pouze moduly, ale konkrétní třídy, které by poskytovaly všechny funkce jako metody. Jelikož je ale cílem mít API co nejjednodušší a uživatelsky nejpříjemnější, je snazší odkazovat se na funkce v rámci modulu, než muset vytvářet instance tříd a následně volat jejich metody.

Přesto však jsou v některých modulech užitečné třídy, například třída Logger, která pouze doplňuje svůj název před samotnou zprávou, stejně jako je to zvykem u existujících loggerů. Dále třída Window poskytuje vybrané funkce celého modulu display, ale zabaluje je do vlastního kontextu. Pokud tedy chci přečíst text v daném okně, a ne na celé obrazovce, mohu na to

využít instanci třídy Window. V rámci modulu network bylo užitečné vytvořit si základní strukturu síťových framů pro jejich snazší parsování. U všech těchto tříd ale platí, že jsou buď čistě interní (Logger a *Frame) anebo že jsou instancovány interně a uživatel přistupuje pouze k jejich objektům (Window), nemá to tedy negativní dopad na čitelnost testů.

Všechny moduly interagují s modulem logger, protože skrz něj zaznamenávají všechny svoje aktivity. Moduly cursor a keyboard dále interagují s knihovnou XTest, přes kterou simulují vybrané eventy. Modul display interaguje také s knihovnou Pytesseract pro OCR, knihovnou Pillow pro záznam obrazovky a knihovnou OpenCV pro preprocessing.

Submoduly uvnitř hlavních modulů nejsou pro uživatele nijak viditelné a pouze vylepšují strukturu projektu.

4.3 Sada funkcí

Každý představený modul poskytuje svou sadu funkcí uživateli, což společně tvoří API projektu. Pojďme si nyní zběžně představit jednotlivé funkce, se kterými se detailněji seznámíme v průběhu implementace.

Počítačová myš má v podstatě velmi omezenou sadu funkcí: stisknutí tlačítka, pohyb nebo kombinace předchozích dvou. Stisknutí tlačítka však můžeme rozdělit na krátké tzv. kliknutí, delší stisk (např. po dobu jedné vteřiny) a na dlouhodobý stisk tlačítka, dokud ho neuvolníme. Funkce scrollování je oddělená, protože jako běžný uživatel ji vnímám odděleně od stisknutí tlačítka, přestože interně v tom není žádný rozdíl. Kombinací držení tlačítka a pohybu myší dostaneme tzv. dragování.

Pokud má myš omezenou sadu funkcí, tak klávesnice jich má ještě méně, protože umí pouze stisknout klávesu. Proto u ní stačí pouze stisk a držení/uvolnění klávesy. Bonusem je funkce Type, která umožní uživateli zadat celý text, který chce napsat, aniž by musel zadávat každou klávesu zvlášť.

Většina funkcí modulu display je spojená se čtením textu nebo obecně rozpoznáváním, co se na obrazovce děje. Kromě klasického čtení je důležitá funkce vyhledávání pozice textu na obrazovce. Dále může být užitečné zjistit, jaké části obrazovky se změnily za poslední chvíli a jakou barvu má konkrétní

Obrázek 4.3: Diagram uživatelských funkcí uvnitř modulů

pixel nebo oblast. V neposlední řadě display umožňuje uložit snímek obrazovky do souboru jako součást výstupu testu.

V rámci komunikace po síti je klíčové umět poslat zprávu jak sám sobě, tak na server a poslouchat komunikaci. Modul network dále pro jednodušší spouštění příkazů jak lokálně, tak na serveru poskytuje funkce pro jejich volání.

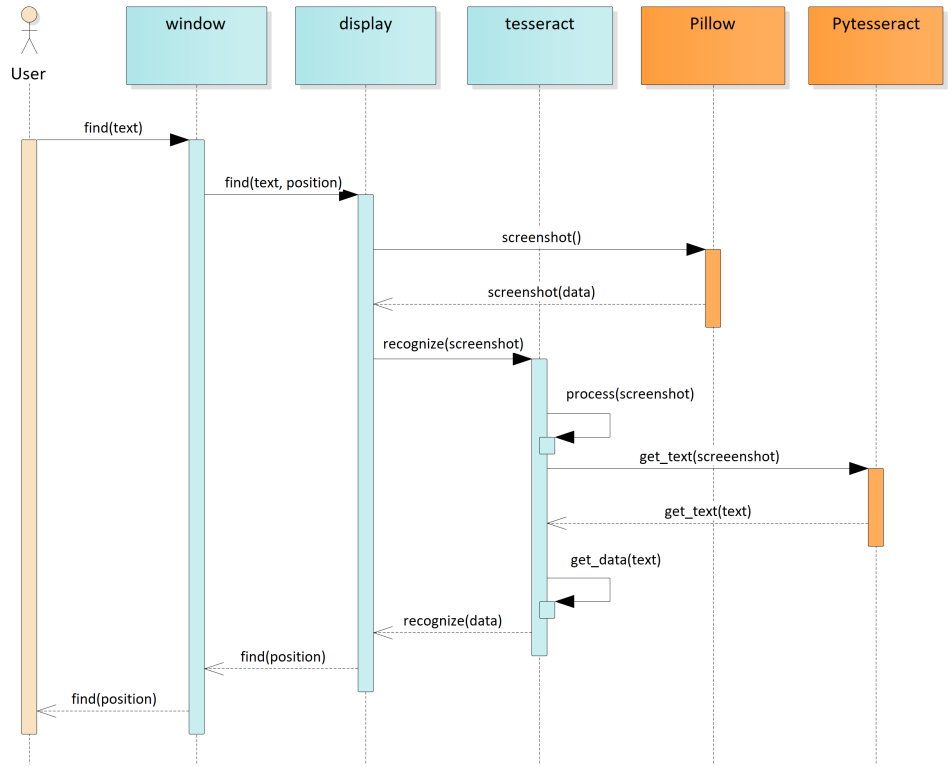
Pro práci s konfigurací testované aplikace (např. pro změnu barvy pozadí) je klíčové umět ji přečíst a následně upravit, přepsat nebo smazat. Doplňkovou funkcí je možnost restartovat klienta (sebe) nebo server, protože tento restart je nezbytný pro znovunačtení konfigurace.

V neposlední řadě v rámci logování je potřeba mít pokryté hlavní úrovně

logů (debug, info, error, warn atd.) a kromě toho mít i způsob, jak zapsat úspěch či neúspěch testu, což pak kontroluje manager. Užitečnou funkcí je také možnost zkontrolovat, jestli jakýkoli log (nejen ten generovaný v rámci testu) obsahuje nějakou zprávu, protože se jedná o podobnou úroveň kontroly jako poslouchání aktivity na síti, ale uvnitř aplikace.

4.4 Procesy

Pro lepší představu procesů uvnitř ARTEMIS agenta si detailně představme jednu z klíčových funkcí: vyhledávání textu na obrazovce. Tuto funkci lze využívat jak k vyhledávání pozice tlačítek a formulářových polí, tak pro obecnou kontrolu přítomnosti nějakého textu.



Obrázek 4.4: Diagram sekvence kroků pro funkci find

Uživatel vytvoří požadavek na nalezení textu v konkrétním okně. Okno tento požadavek předá modulu display a obohatí ho o svou pozici a rozměry. Modul display si nejprve od knihovny Pillow vyžádá snímek obrazovky, který v případě potřeby ořízne. Tento snímek přejde do submodulu tesseract, který slouží jako adaptér mezi knihovnou Pytesseract a zbytkem aplikace. Submodul obrázek nejprve předzpracuje a převede do formátu, kterému rozumí

Pytesseract. Následně tento upravený snímek předá knihovně Pytesseract, která vrátí sadu formátovaných dat. Je možné zvolit z několika formátů, ale v podstatě všechny obsahují nalezená slova a k nim jejich pozici, velikost a jistotu. Tato data submodul tesseract zpracuje do struktury srozumitelné dalším částem systému a předá je zpět modulu display. Ten následně data projde a zjistí, zda obsahují vyhledaný text a jestli jistota daného textu je dostatečně vysoká. Pokud tomu tak je, vrací pozici a velikost textu oknu, které je následně vrací uživateli.

Tento proces kombinuje několik dalších funkcí, které jsou uživateli dostupné, jako záznam obrazovky a čtení textu z obrázku. Pokud by uživatel chtěl přečíst snímek celé obrazovky nebo nějaké oblasti neodpovídající konkrétnímu oknu, může přistupovat přímo k funkci modulu display.

4.5 Struktura testů

Po představení všech dostupných modelů je na řadě přestavit si obecnou strukturu testů, které budou uživatelé psát. ARTEMIS poskytuje široké spektrum funkcí, ale napsat jeden test na vyplnění formuláře může být i na několik desítek řádků. To není samo o sobě problém, ale pokud je toto vyplnění jen jedním z více kroků, například pro nastavení letové hladiny je potřeba nejprve poslat let, ověřit pozici letadla, otevřít editační okno, vyplnit všechny položky, uložit formulář a ověřit provedenou změnu, začíná velikost jednoho testu rychle narůstat.

Abychom se tomuto problému vyhnuli, můžeme použít doplňkové předepsané funkce, které budou do výsledného testu importovány jako knihovny. Tyto funkce mohou být relativně komplexní a obsahovat velké množství kontrol, ale jejich použití by mělo v testu být velmi jednoduché a celá sada kroků popsaných v předchozím odstavci může být shrnutá do dvou „vytvoř let“ a „změň letovou hladinu na X“. Cílové testy, které mohou psát i lidé s minimální znalostí programování, se tedy budou skládat pouze z funkcí, které mohou být napsány zkušenými programátory a umístěny do nějaké sdílené složky. Pokud bychom tuto myšlenku posunuli ještě dál, bylo by možné v rámci jednoho testu spouštět i další pod-testy, jelikož není konkrétní rozdíl mezi testem a funkcí. Tato struktura zajistí čitelnost a přepoužitelnost i komplikovaných testů napříč testovací sadou.

4.6 Šablona pro Docker

Cílem ARTEMIS je, aby testovaná aplikace nevěděla o tom, že není v běžném nasazení, je tedy cílem co nejméně zasahovat do struktury Docker image, který se pro testování využívá. Veškeré úpravy jsou tedy pouze aditivní, jedná se o instalaci Tesseractu a Python knihoven jako je OpenCV apod. Kromě těchto úprav není potřeba provádět další změny a díky neinvazivnosti úprav je snadné tyto kroky adaptovat pro jakýkoli image, jedinou predispozicí je nainstalovaný Python a přístup k internetu.

4.7 API

ARTEMIS musí ve svém API poskytovat možnost jak spravovat testy a funkce, jak spouštět testy a jak zjistit výsledky provedených testů. Aby bylo možné tvořit testy a funkce, je potřeba poskytovat také seznam funkcionalit dostupných ze základních modulů ARTEMIS, funkcí a testů. To určuje základní skupiny endpointů:

- core - poskytuje seznam funkcí dostupných přímo z kódu ARTEMIS
- function - poskytuje seznam uživatelských funkcí[1] a endpointy pro jejich správu
- test - poskytuje seznam testů a endpointy pro jejich správu
- run - umožňuje spouštění testů
- results - poskytuje výsledky provedených testů

Dále budou k dispozici dvě doplňkové sady endpointů, konkrétně common, který poskytuje seznam veškeré dostupné funkcionality z jádra ARTEMIS, uživatelských funkcí a testů, a validator, který poskytuje endpointy pro validaci kódu na straně klienta.

Sady function a test budou poskytovat základní create-read-update-delete endpointy pro konkrétní testy, což pokrývá požadavek na správu testů a uživatelských funkcí. Sadu run si můžeme představit jako interaktivní část sady results, která je ale záměrně oddělená, jelikož bude napřímo interagovat s Dockerem, zatímco results je podobnější ostatním sadám, které pouze pracují s daty na disku.

[1] „funkce v uživatelských funkcích“ je poměrně nešťastné názvosloví, ale bohužel mě nic lepšího nenapadlo

Při rozmýšlení architektury webového serveru jsem se inspiroval aplikací Jenkins[jen], která umožňuje správu a spouštění CI/CD procesů. Přestože je Jenkins velice rozsáhlá a komplexní aplikace poskytující velké množství dat, není napojená na žádnou databázi a všechna data udržuje jako soubory ve složkové struktuře. To dává smysl, jelikož většina dat v Jenkinsu jsou buď logy z proběhlých prací, výsledky sestavení (tedy binární soubory) nebo definice CI/CD procesů, které mohou být také uložené v souborech. ARTEMIS v tomto smyslu poskytuje velice podobnou sadu funkcí, které nevyžadují žádnou databázi, proto bude celý webový server pracovat se soubory na disku, ve kterých bude mít uložené všechny potřebné informace.

Kapitola 5

Design

Pojďme se nyní podívat na poslední přípravnou část tohoto projektu před samotnou aplikací. Tou je design webové aplikace, tedy jak bude aplikace vypadat a jak se bude používat. Nejprve se zaměříme na logo, které určuje identitu celého projektu ARTEMIS a následně si představíme několik ilustrativních návrhů částí webové aplikace.

5.1 Logo

Obrázek 5.1: Logo projektu ARTEMIS

Při vytváření loga jsem se zaměřoval hlavně na jednoduchost, ale také jedinečnost a vypovídající hodnotu. Jako primární barvu celého projektu jsem zvolil tmavě červenou až vínovou barvu, která bude dostatečně kontrastní ke zbytku aplikace v odstínech šedi. Tato barva byla zvolená Pantone barvou roku 2023 a má název Viva Magenta.

Logo znázorňuje několik prvků. V první řadě se jedná o kombinaci dvou písmen „A“, jedno složené ze tří listů v primární barvě a jedno z prostoru mezi listy. Dále celkový tvar loga je inspirován hrotem šípu, jelikož řecká bohyně lovu Artemis má jako jeden ze symbolů šíp.

5.2 Webová aplikace

V první fázi jsem se rozhodl vytvořit velice hrubý návrh aplikace, abych si ověřil své představy o základním rozložení a barevných odstínech. Také jsem potřeboval znázornit, jak bude vypadat prostor pro vytváření drag-and-drop testů. Na základě tohoto konceptu jsem následně vytvořil detailní návrh vzhledu viditelný v příloze B.

Obrázek 5.2: Koncept vzhledu webové aplikace

Cílem bylo vytvořit jednoduchou, moderně vypadající aplikaci, která se odliší od ostatních interních aplikací v rámci firmy, které jsou většinou designově zastaralé (např. TestLink). Důležitým faktorem bylo, aby design nepředcházel funkcionalitě, jelikož tato aplikace bude využívaná interně a musí být v první řadě použitelná.

Hlavní navigace v aplikaci probíhá přes vždy přítomné horní menu, aplikace má čtyři hlavní obrazovky. V levém menu je typicky seznam prvků daného typu, tedy seznam testů, funkcí nebo proběhlých sad testů. U testů je v pravém menu seznam použitelných funkcí, které se pomocí drag-and-drop dají přesunout do hlavního okna, které definuje kroky testu. Test lze editovat i jako

kód, zmíněné bloky v pravém menu tedy slouží jako interaktivní dokumentace díky svým tooltipům (ty nejsou na návrzích zobrazeny). Stránka pro spuštění testů poskytuje možnost vybrat jednak testy, které chce uživatel spustit a jednak Docker image, na kterém mají testy běžet. Na stránce výsledků testů je ještě pod-menu, které zobrazuje seznam testů v dané sadě s jejich výsledky, seznam screenshotů pořízených za běhu testů a s informacemi o provedených testech.

Kapitola 6

Implementace

Po výběru všech potřebných nástrojů a knihoven v kapitole Rešerše a po konkretizaci projektu v kapitole Architektura se můžeme pustit do samotné implementace. V ní se budu zabývat nejprve detailnějším popisem jednotlivých modulů a jejich funkcí a rozeberu implementaci vybraných metod. Následně se podíváme na webový server, jeho způsob práce s jednotlivými moduly pomocí knihovny AST a interakci s Dockerem. Na závěr projdeme tvorbu webového frontendu, převážně pak využití knihoven, které tvoří hlavní část funkcionality.

6.1 Moduly

Na úvod se pojďme zaměřit na jednotlivé moduly a jejich funkce, které jsme si nastínili při návrhu. Nejprve projdeme základní moduly pro kurzor a klávesnici, dále se zaměříme na konfiguraci, logování a síť a na závěr si představíme největší modul, zabývající se prací s displayem. U každého modulu také popíšu ukázkové využití.

6.1.1 Kurzor

Práce s kurzorem je jedna ze základních operací potřebná k práci s počítačem. V případě aplikací testovaných pomocí ARTEMIS se bude jednat převážně

o klikání na tlačítka či pole, posouvání po mapě nebo přesouvání oken po obrazovce.

Pro simulaci funkčního kurzoru je potřeba simulovat eventy, které vytváří běžná interakce s myší. XServer umožňuje simulaci těchto eventů pomocí knihovny XTest. Tato knihovna je však dostupná pouze pro C++ a bylo tedy potřeba obalit si její objekty a funkce do Pythonu. To není nijak dramaticky složité, protože knihovna ctypes poskytuje pro Python všechny typy používané v C/C++ a umožňuje tedy pracovat se strukturami z C++ knihovny jako by byla napsaná v Pythonu.

```
_XTestFakeButtonEvent = _xtest.XTestFakeButtonEvent
_XTestFakeButtonEvent.restype = c_int
_XTestFakeButtonEvent.argtypes = [_Display, c_uint, c_int, c_ulong]
```

Kód 6.1: Vytvoření Python obalu pro C++ funkci

Následně stačí tyto funkce obalit do vlastního, uživatelsky přívětivého API, případně je zkombinovat a vytvořit komplexnější funkce jako dragování (stisk tlačítka, posun kurzoru, uvolnění tlačítka). Jelikož kolečko myši se bere jako dvě tlačítka (jedno pro posun nahoru a jedno pro posun dolů), lze ho simulovat stejně jako každé jiné tlačítko na myši. Konkrétně levé tlačítko má index 1, kliknutí kolečkem má index 2, pravé tlačítko má index 3, otočení kolečkem nahoru má index 4 a otočení kolečkem dolů má index 5.

Abych vytvořil co nejrealističtější simulaci interakce s myší, rozhodl jsem se zvolit vybrané základní hodnoty podle průměru lidského uživatele. První hodnotou je doba trvání jednoho kliknutí, které trvá průměrně 85ms.[SOG22] Další hodnotou je doba mezi jednotlivými kliknutími, které používám hlavně pro několikanásobné otáčení kolečkem myši – to jsem zvolil jako 100ms. U pohybu myší jsem také vytvořil možnost plynulého přechodu ze startovní pozice na koncovou pozici. Pro tento pohyb jsem musel definovat rychlost kurzoru. Ta se samozřejmě dramaticky liší v závislosti na vzdálenosti, zkušenosti atd. neexistuje tedy jedna konkrétní hodnota. Empirickým testováním jsem tedy nalezl hodnotu 1 500 px/s jako rozumnou rychlost pro běžnou interakci s myší. Poslední hodnotou je rychlost snímání myši. Zde opět velmi záleží na konkrétním modelu myši, protože ty nejvýkonnější snímají svůj pohyb až 8 000krát za vteřinu. Průměrná kancelářská myš však snímá zhruba pouze 100krát za vteřinu, což je pořád více než obnovovací frekvence typického monitoru (60krát za vteřinu) a jelikož se jedná spíše o doplňkovou funkci, dá se předpokládat, že 100 aktualizací za vteřinu je dostačující. Zajímavostí právě u této postupné simulace je způsob jejího generování. Pomocí knihovny XTest totiž vytvoříte frontu událostí a definujete jejich časové rozestupy a až následně pomocí zavolání funkce XFlush tyto události skutečně provedete.

```
for i in range(int(steps)):
            _XTestFakeMotionEvent(_display, -1, int(x+step_x), int(y+step_y),
     1000 // polling_rate)
            x += step_x
            y += step_y
        _XFlush(_display)
```

Kód 6.2: Tvoření fronty událostí

6.1.2 Klávesnice

Klávesnice má jednak zřejmé využití pro vyplňování formulářů, ale specificky v systémech pro řízení letového provozu je typické, že spousta funkcí se ovládá pomocí klávesových zkratek, spíše než pomocí otevírání menu. Jelikož vycházíme z předpokladu, že řídící jsou s těmito zkratkami seznámeni, je na dostupnost funkce přes zkratku kladen větší důraz, než na dostupnost přes menu nebo tlačítko.

Modul pro práci s klávesnicí staví na podobném principu jako kurzor, tedy využívá knihovnu XTest pro simulaci stisknutí kláves. Usnadněním je, že stačí simulovat pouze stisknutí klávesy, protože klávesnice neumožňuje žádný pohyb, naopak ale představuje komplikaci s mapováním mnohem většího počtu kláves oproti několika tlačítkům na myši. Základní knihovna pro práci s XServerem poskytuje funkci pro převod symbolu klávesy (tzv. key sym) na kód klávesy (tzv. key code), ale nejprve je potřeba se z klasického písmene, které zadá programátor v testu, dostat na zmiňovaný symbol. K tomu jsem využil knihovnu XLib (psaná v Pythonu), která mimo jiné poskytuje i mapování všech možných znaků, které je člověk schopen napsat na klávesnici, na symbol, kterému rozumí XServer. Výhodou této knihovny je, že všechny tyto znaky jsou ve formě proměnných a je tedy možné se na ně přímo odkazovat i v samotném testu. Programátor tedy nemusí přemýšlet, jestli se klávesa pravého Controlu jmenuje „RCtrl“, „Control_R“ nebo „RC“, ale pouze využije proměnnou keyboard.Control_R.

Tímto přístupem jsem však podchytil pouze stisk konkrétních kláves, ale psaní jakéhokoli delšího textu by bylo velice zdlouhavé, protože by ho programátor musel definovat po jednotlivých klávesách. Proto jsem vytvořil funkci, která přijímá celý text ve formě stringu a ten následně po písmenech převádí na klávesové symboly za pomoci knihovny XLib a to včetně možnosti modifikátorů jako je Shift pro velká písmena apod.

V neposlední řadě bychom si měli představit hodnoty, které rychlost simulovaného psaní přibližují rychlosti psaní člověka. První z nich je doba stisku

klávesy, která je průměrně o něco delší než stisk tlačítka myši, konkrétně 100ms.[KM09] Další hodnotou je doba mezi jednotlivými stisky kláves, kterou jsem, stejně jako u kurzoru, nastavil na 100ms.

6.1.3 Konfigurace

Jelikož naše firma dodává stejný software v různých variantách a tyto varianty se mohou od sebe velmi lišit, mají naše systémy poměrně rozsáhlou strukturu konfigurací. Příkladem využití tohoto modulu je ověření, že daná funkcionalita funguje ve všech variantách systému. Alternativně se může jednat o banální kontrolu toho, že daný konfigurační parametr (např. na změnu barvy pozadí) skutečně funguje tak, jak má.

Konfigurace našich systémů je realizovaná pomocí jednoho nebo více konfiguračních souborů. Načtení správných hodnot z jednotlivých souborů je poměrně komplexní proces, jelikož některé hodnoty jsou závislé na jiných konfiguračních hodnotách a některé jsou společné pro všechny konfigurační typy aplikace. Mým cílem rozhodně není tento proces replikovat, proto načítám konfigurační parametry z jednoho souboru, který vznikne při startu aplikace a obsahuje již předzpracované hodnoty, které se skutečně použijí pro aktuální běh testované aplikace. Abych však mohl konfiguraci upravovat, musím ji zapisovat do jednoho z originálních souborů, ne až do toho vytvořeného při startu aplikace. Kromě této komplikace je zapisování konfigurace poměrně jednoduché a sestává pouze z různé kombinace načítání aktuální hodnoty, její úpravy a následného zápisu. V rámci firmy je tento proces dostatečně konzistentní pro všechny aplikace, ale pokud by se měl projekt ARTEMIS poskytnout jiné firmě, tento modul by bylo nutné upravit pro potřeby konkrétní konfigurační struktury.

Kromě této úpravy souborů umožňuje tento modul také restartovat testovanou aplikaci, protože až tak se projeví změny v konfiguraci.

6.1.4 Logování

Logovací modul slouží převážně pro evidenci kroků provedených v rámci testu a využívá ho přímo ARTEMIS, ale monitorování logů testované aplikace nám může prozradit interní kroky, které bychom běžným pozorováním nebyli schopni otestovat.

Modul pro logování by se mohl zdát poněkud zbytečný, když existují knihovny, které tuto funkci poskytují. Pro potřeby tohoto projektu jsem však potřeboval klasický koncept logování trochu rozšířit a než upravovat existující řešení, rozhodl jsem se radši napsat si vlastní jednoduchý logger. Logger poskytuje základní sadu funkcí jako „debug“, „info“, „warn“, „error“ a „trace“ jednak ve formě globálních funkcí, které bude využívat programátor v testu a jednak ve formě třídy, kterou instancují jednotlivé moduly a logují tedy „pod svým jménem“. Všechny tyto zprávy jsou zapisovány do společného souboru, který je sdílený i mimo samotnou instanci testu.

Kromě toho jsem však přidal dvě další úrovně logování a to „success“ a „fail“, které fungují stejně jako třeba „info“, ale ARTEMIS manager po skončení testu hledá v logu testu právě jednu z těchto úrovní a podle ní určuje výsledek testu. Právě proto je klíčové, aby se k souboru bylo možné dostat i mimo průběh samotného testu, specificky po jeho skončení. Další funkcionalitou je možnost zkontrolovat přítomnost nějaké zprávy v jakémkoli logu, ať už od ARTEMIS nebo od testované aplikace, což slouží jako možnost kontroly provedení nějaké akce. Tato kontrola probíhá jednoduchým průchodem celého souboru a vyhledáním konkrétního textu.

6.1.5 Síť

Aplikace, které bude ARTEMIS testovat, jsou převážně tzv. zobrazovací, tedy dostávají velké množství informací po síti ze serveru (typicky pozice jednotlivých letadel) a na základě uživatelského vstupu mohou generovat požadavky, které jdou na server (např. změna letové hladiny daného letadla). V průběhu testování se nepočítá s tím, že by testovaná aplikace přijímala zprávy od serveru (přestože to není výslovně nemožné), takže ARTEMIS musí být schopna funkci serveru zastoupit. Typickým příkladem testu tedy je, že v rámci testu se odešle po síti zpráva obsahující identifikaci a pozici letadla (jelikož ARTEMIS je na stejném stroji jako testovaná aplikace, probíhá komunikace v lokální síti) a následně se ověří, že se letadlo objevilo na obrazovce. Poté se v testu provedou kroky pro změnu letové hladiny a aplikace vygeneruje požadavek, který pošle na adresu, o které se domnívá, že je server. Jelikož však žádný server není, aby mohl odpovědět, chceme, aby ARTEMIS alespoň ověřila, že požadavek odešel na správnou adresu a se správným obsahem, protože pak můžeme prohlásit tuto část testu za úspěšnou.

Síťový modul v první řadě poskytuje funkce pro posílání zpráv po síti na server a sám sobě. Právě pomocí posílání zpráv sám sobě je možné snadno simulovat vybranou leteckou aktivitu. Jelikož většina komunikace mezi serverem a klientem je realizována pomocí UDP, je tato funkcionalita

realizována pomocí otevření UDP socketu a odeslání dané zprávy na konkrétní adresu a port. Dále tento modul poskytuje možnost spouštět příkazy na serveru a na klientovi. Pro spouštění příkazů je využívána knihovna subprocess. Pro připojení na server se používá protokol ssh, je ale důležité poznamenat, že běžné Docker kontejnery neumožňují připojení přes ssh, pokud by tedy i server byl virtualizovaný přes Docker, je potřeba na něm ssh zprovoznit. Pro spouštění příkazů na klientovi (tedy na lokálním stroji) není potřeba žádná síťová komunikace, ale pro udržení konzistence je i tato funkce v modulu network.

Nejzajímavější funkcí tohoto modulu je však možnost poslouchání po síti. Některé akce provedené v aplikaci vytvoří požadavek poslaný na server, a i když zrovna server není k dispozici, chceme jistě otestovat, že tento požadavek byl alespoň vytvořen a odeslán, přestože nepřijde odpověď. K tomu slouží funkce „sends_message“, která přijímá mimo jiné další funkci, kterou nejprve spustí a následně čeká určenou dobu na to, jestli se na síti objeví packet směřující na definovanou adresu a port. Aby se však nestalo, že se zpráva pošle dříve, než stihne modul poslouchat, spouští se tento poslech ve vlastním vlákně před samotným spuštěním funkce, která má zprávu vytvořit. Stejným způsobem funguje i funkce „receives_message“ s tím rozdílem, že ta očekává zprávu přicházející z definované adresy a portu, což je užitečné pro kontrolu odpovědi serveru, pokud je dostupný. Pro tyto funkce jsem definoval jednotlivé síťové „framy“ představené na diagramu modulů. Tyto „framy“ mi totiž umožňují postupně konkretizovat jednotlivé síťové packety, až se dostanu na úroveň TCP/UDP, kde mohu zkontrolovat zdrojový a cílový port packetu.

6.1.6 Display

Přestože modul pro práci s displayem neposkytuje žádné konkrétní funkce pro provádění testů (jako pohyb myši nebo poslání packetu), je klíčový pro ověření, že se všechny kroky provedly správně, a pro vyhledávání objektů. Aby totiž kurzor věděl, kam se přesunout, když má kliknout na tlačítko „Uložit“, je potřeba tlačítko nejprve najít na obrazovce. Aby se ověřilo, že klávesová zkratka otevřela správné okno, je potřeba najít okna na obrazovce, přečíst jejich názvy a zkontrolovat, jestli odpovídají předpokladu. Aby bylo jisté, že konfigurace barvy pozadí skutečně funguje, musí se zkontrolovat barvy v konkrétních oblastech obrazovky a porovnat je se skutečnou hodnotou v konfiguraci.

Hlavní funkcionalitou tohoto modulu je čtení textu z obrazovky, ke kterému využívá OCR engine Tesseract. Tyto funkce jsou konkrétně „read“, která

vrátí veškerý text v dané části obrazovky, a funkce „find“, která vyuzívá funkci „read“ a na rozdíl od celého textu vrací pozici konkrétního textu, který uživatele zajímá. Tesseract je založený na neuronové síti a pracuje tedy s pravděpodobnostmi, specificky s tzv. confidence indexem, který určuje, s jakou jistotou bylo dané slovo správně rozpoznané. Může se stát, že některé slovo bude špatně rozpoznané a bude mít třeba jen 30% confidence index a bude tedy narušovat navazující kontrolu. Proto tyto dvě funkce pracují s uživatelsky definovanou minimální hodnotou confidence indexu, aby bylo možné tyto nepřesné identifikace ignorovat. To přispívá nejen přesnosti výsledného textu, ale také přesnosti pozice, protože některé slovo může být správně rozpoznané, ale jelikož rozpoznaná oblast zasahuje ještě do dalšího slova, má velmi nízký confidence index a jeho pozice je nepřesná. Proto, kdybychom měli na takové slovo třeba kliknout, mohlo by se stát, že kurzor mine.

Modul však poskytuje i funkce, které nejsou spojené s Tesseractem, třeba funkce pro uložení snímku obrazovky, což se hodí pro vizuální kontrolu stavu aplikace v dané situaci, nebo funkci pro získání barvy obrazovky na daném pixelu nebo v dané oblasti. Dále také poskytuje funkci pro získání všech oblastí, které se změnily od daného momentu. Tato funkce přijímá snímek obrazovky, který byl pořízen někdy v minulosti a porovnává ho na úrovni pixelů s aktuálním děním na obrazovce. Oblasti pixelů, které se změnily, následně „zabalí“ do bounding boxů (obdélníků), jejichž pozice vrátí jako výstup. Tyto pozice se mohou použít např. pro situaci, kdy chceme zkontrolovat, že aktuální čas na obrazovce se skutečně mění jednou za sekundu, aniž bychom museli kontrolovat přesnou hodnotu hodin, která se může lišit mezi dobou rozpoznání a dobou validace hodnoty.

Testované aplikace budou většinou obsahovat kromě hlavního okna ještě doplňková okna jako jsou formuláře nebo letové informace. Aby se s těmito okny dalo lépe pracovat a nebylo nutné si pamatovat přesné pozice všech oken, poskytuje tento modul vybrané funkce tzv. v kontextu konkrétního okna. Může se totiž stát, že v jednu chvíli budou na obrazovce dvě tlačítka „Uložit“ a bylo by složité kontrolovat, které je které, ale když vyhledáme toto tlačítko pouze v kontextu (oblasti) aktivního okna, které testujeme, je malá šance, že by existovalo více výskytů stejného slova. Pro tuto práci v kontextu slouží objekty třídy Window. Zde je potřeba zmínit, že různé aplikace mohou vytvářet různé typy oken, jedna aplikace vytváří okna přímo v systému pomocí XServeru, zatímco jiná, psaná ve Swingu, vytváří vlastní okna, která nejsou dostupná ze systému. Je tedy potřeba pro každý tento typ oken vytvořit vlastní implementaci pro jejich identifikaci. XServer poskytuje funkci pro získání všech aktivních elementů včetně oken a tato funkce je uživatelsky spustitelná přes příkazovou řádku, ale např. právě Swing žádnou takovou funkci nemá a pokud se není možné napojit na interní struktury v aplikaci, což nechceme, je potřeba implementovat vlastní způsob identifikace oken, např. pomocí pravidelného logování všech otevřených oken, nebo vystavení

endpointu, který tyto informace poskytuje. Je tedy zřejmé, že konkrétní implementace je potřeba přizpůsobit testované aplikaci. Alternativně by se aktivní okna dala poznávat přímo z obrazovky aplikace, ale tím nemusíme identifikovat všechny (mohou se překrývat) a hlavně vizuál oken se bude také lišit napříč aplikacemi, bylo by tedy stejně nutné implementovat konkrétní rozpoznávací algoritmus.

6.2 Webový server

Na první pohled by se mohlo zdát, že server pro ARTEMIS nebude nijak náročný, když ani nepřistupuje do databáze a pouze pracuje se soubory. Ze začátku to tak vypadalo i na druhý pohled, ale tato část projektu poměrně rychle vyrostla v jednu z nejkomplikovanějších částí, hlavně kvůli rozpoznávání obsahu modulů pomocí knihovny AST. V této kapitole se tedy nejprve ponoříme do světa rozpoznávání syntaxe, následně si nastíníme integraci Pythonu s Dockerem a cachování ve FastAPI a na závěr představím využití websocketů v tomto projektu.

6.2.1 Rozpoznávání syntaxe pomocí AST

Celá sekce rozpoznávání syntaxe existuje pouze z toho důvodu, že jsem se rozhodl mít možnost skládat testy pomocí bloků v rámci drag-and-drop. Aby se totiž drag-and-drop dal realizovat, jsou zapotřebí dvě věci: a) znát seznam možných bloků, které lze použít a b) umět daný blok použít jako volání funkce, tedy nastavit konkrétní parametry. Přestože to původně nebylo v plánu, využil jsem tuto funkcionalitu i mimo drag-and-drop, konkrétně k validaci kódu a k napovídání, kterému se budu věnovat v kapitole Frontend webové aplikace.

6.2.1.1 Načítání dokumentace

Pro vytvoření dokumentace z kódu je potřeba projít všechny moduly, které je možné využít, a získat z nich definice funkcí, které poskytují. Pro každou funkci se pak musí načíst její parametry a typy těchto parametrů, její návratová hodnota a její popis. V Pythonu se jako standardizovaná forma popisu funkce bere komentář hned pod názvem funkce (tedy jako první řádek funkce).

Pokud jsou parametry otypované a případně mají i výchozí hodnotu, je možné vytvořit kompletní předpis funkce. Objektová reprezentace těchto parametrů je v knihovně AST relativně komplikovaná, a hlavně se dramaticky liší podle toho, jestli se jedná o primitivní typ jako *str* nebo *int* a hlavně jakého typu je případná výchozí hodnota (konkrétní hodnota vs. globální proměnná). Tato fakta bylo potřeba zapracovat do kódu webového serveru, který zmíněnou dokumentaci vytváří a poskytuje klientovi ve formátu JSON.

Kód v Pythonu

```
def move_to(to_x: int, to_y: int, smooth: bool=False) ->
    None:
    """move cursor to position (smoothly)"""
    ...
```

Objekt v AST

```
FunctionDef(name='move_to', args=arguments(posonlyargs=[],
    args=[arg(arg='to_x', annotation=Name(id='int', ctx=Load
    ())), arg(arg='to_y', annotation=Name(id='int', ctx=Load
    ())), arg(arg='smooth', annotation=Name(id='bool', ctx=
    Load()))], kwonlyargs=[], kw_defaults=[], defaults=[
    Constant(value=False)]), body=[...], decorator_list=[],
    returns=Constant(value=None))
```

Výstup v JSONu

```
{
 "name": "move_to",
 "module": "cursor",
 "description": "move cursor to position (smoothly)",
 "args": [
  {
   "name": "to_x",
   "type": "int",
   "default": null
  },
  {
   "name": "to_y",
   "type": "int",
   "default": null
  },
  {
   "name": "smooth",
   "type": "bool",
   "default": 0
  }
 ],
 "returns": null,
 "type": "core"
}
```

Tabulka 6.1: Rozpoznání předpisu funkce pomocí AST

6.2.1.2 Převod kódu testu do bloků

Dalším krokem je schopnost převádět jednotlivé kroky testu (tedy typicky volání funkcí) do bloků podobných těm, které se vrací v dokumentaci tak, aby s nimi mohl frontend pracovat. Tento proces je velmi podobný vytváření dokumentace, ale místo načítání objektů typu *FunctionDef* je potřeba vejít do takové funkce (test typicky obsahuje jednu funkci, která se spouští) a rozpoznat každý řádek zvlášť. Typů těchto řádků je velké množství, od volání funkcí, přes nastavování proměnných a podmínky, až po *for* cykly a *while* cykly. V rámci ARTEMIS jsem se rozhodl omezit rozpoznávané typy na volání funkce, nastavení proměnné a kontrolu hodnoty pomocí *assert*. Tento výběr vychází z toho, že typický scénář pro manuálního testera se skládá z „proveď úkon“ (tedy zavolej funkci) a „zkontroluj stav“ (tedy *assert*). Všechny „komplexnější“ operace jako podmínky a cykly lze psát do uživatelských funkcí, které není možné tvořit přes drag-and-drop. Díky tomu zajistím, že testy budou stručné a přehledné a budou sestávat hlavně z volání „high-level“ uživatelských funkcí.

Běžné volání funkce nám typicky neřekne všechny údaje o dané funkci, jelikož nemusí nutně obsahovat všechny parametry, téměř určitě nebude obsahovat typy parametrů a jistě nebude obsahovat popis volané funkce. Proto je potřeba každé volání funkce spárovat s předpisem funkce, získaným při vytváření dokumentace. Alternativně, pokud není možné najít předpis funkce (např. se jedná o generickou funkci *time.sleep*, která není v dokumentaci), předávají se uživateli alespoň známé informace o takové funkci.

Pokud se tedy uživatel dotáže na obsah daného testu, nedostane jeho kód, ale jeho reprezentaci pomocí těchto bloků spolu s dodatečnými informacemi jako je popis atd.

Kód v Pythonu

```
cursor.move_to(100,200) # volani funkce z dokumentace
keyboard.press_duration = 200 # nastaveni promenne
assert cursor.x == 150 # kontrola hodnoty
time.sleep(1) # volani nezname funkce
```

Objekt v AST

```
Expr(value=Call(func=Attribute(value=Name(id='cursor', ctx=
    Load()), attr='move_to', ctx=Load()), args=[Constant(
    value=100), Constant(value=200)], keywords=[]))
Assign(targets=[Attribute(value=Name(id='keyboard', ctx=Load
    ()), attr='press_duration', ctx=Store())], value=
    Constant(value=200))
Assert(test=Compare(left=Attribute(value=Name(id='cursor',
    ctx=Load()), attr='x', ctx=Load()), ops=[Eq()],
    comparators=[Constant(value=150)]))
Expr(value=Call(func=Attribute(value=Name(id='time', ctx=
    Load()), attr='sleep', ctx=Load()), args=[Constant(value
    =1)], keywords=[]))
```

Výstup v JSONu

```
[
 {
  "name": "move_to", "module": "cursor",
  "description": "move cursor to position (smoothly)",
  "args": [
   {"name": "to_x", "type": "int", "default": null, "value": 100,
    "isExec": false},
   {"name": "to_y", "type": "int", "default": null, "value": 200,
    "isExec": false},
   {"name": "smooth", "type": "bool", "default": 0, "value": null,
    "isExec": false}
  ],
  "returns": null, "type": "core", "returnType": null
 },
 {
  "left": "keyboard.press_duration",
  "right": "200",
  "type": "assign"
 },
 {
  "assertion": "cursor.x == 150",
  "type": "assert"
 },
 {
  "name": "sleep", "module": "time", "description": null,
  "args": [
   {"name": null, "type": null, "default": null, "value": 1,
    "isExec": false}
  ],
  "returns": null, "type": "undefined", "returnType": null
 }
]
```

Tabulka 6.2: Převod kódu do bloků v JSONu pomocí AST

6.2.1.3 Validace

Knihovna AST nám poskytuje ještě jednu užitečnou funkci. Pokud totiž na vstupu dostane syntakticky nevalidní kód, vrátí stejnou chybu, kterou by vrátil interní proces Pythonu při importování modulu. To lze využít k validaci kódu, která sice neodhalí chyby vzniklé při běhu programu, ale odhalí překlepy, chybějící závorky či uvozovky apod. Tato funkcionalita je ideální pro validaci příchozích uživatelských funkcí, které uživatel píše přímo jako kód. Testy mají kromě syntaktické validity ještě jeden požadavek a to, že se dají převést do již zmiňovaných bloků pro drag-and-drop. Zde lze opět využít již existující funkcionalitu popsanou v kapitole Převod kódu testu do bloků, který také vrací chybu, pokud narazí na kód, který nejde zkonvertovat. Díky tomu lze validovat příchozí kód ještě před jeho uložením na disk a zajistit tím, že všechny testy a funkce budou alespoň spustitelné.

6.2.2 Integrace s Dockerem

Aby bylo možné spustit ARTEMIS v Docker kontejneru, potřebujeme definovat tři věci: jaký základní image se používá, jaký test se bude provádět a jaké další moduly bude test potřebovat. Základní image je poměrně jednoduchý, jelikož Docker je schopen si ho stáhnout z firemní registry[1], stačí tedy dostat jeho název. Test, resp. všechny další spustitelné soubory mapuji do kontejneru jako volumes[2]. V případě testů se tedy jedná o namapování samotného souboru s testem, souboru pro zápis logů (to je potřeba, aby logy nezanikly se zánikem kontejneru) a složky pro screenshoty (stejný důvod jako u logů). Dále je potřeba namapovat všechny závislosti testu. Těmi je vždy celé jádro ARTEMIS, které se importuje najednou, a pak všechny další testy a uživatelské funkce, které jsou v testu importované. Jelikož daná uživatelská funkce může importovat jinou uživatelskou funkci, je potřeba získat seznam všech modulů pro import rekurzivně. Tento proces importuje pouze moduly týkající se ARTEMIS a ostatní knihovny už musí být nainstalované uvnitř image.

V jednu chvíli (resp. v rámci jedné sady testů) je možné pustit více testů, které běží paralelně, každý ve svém kontejneru. Manažer pak pravidelně sleduje status běžících kontejnerů a jakmile všechny testy doběhnou, vytvoří jednoduchý report. Ten neobsahuje (ne)úspěch testů, ale pouze informace o

[1] Docker registry je místo, kde jsou uložené volně dostupné Docker image.

[2] Docker volume je složka na hostitelském stroji, která je mapovaná do složkové struktury kontejneru před jeho startem. Všechny ostatní soubory musí být uložené již v Docker image, ze kterého kontejner vzniká.

času spuštění, délce běhu a použitém image. Výsledky testu jsou načítané z logu daného testu, jelikož ten musí obsahovat buď záznam „SUCCESS“ nebo „FAIL“. Pokud neobsahuje ani jeden, předpokládá se, že test stále běží.

6.2.3 Cachování odpovědí

Velká část požadavků směřovaných na webový server vrací statickou odpověď, která se příliš často nemění nebo se mění po řízené změně. Nabízí se tedy využít nějakou formu cache, která by zrychlila dobu odpovědi serveru. Jelikož jednu instanci webového serveru bude aktivně využívat jen malé množství uživatelů, nejedná se o vážný problém, ale přesto jsem se rozhodl tento směr prozkoumat. FastAPI neposkytuje žádnou formu cache, ale existují knihovny od třetích stran, specificky jedna nejvyužívanější - fastapi-cache2[fasb]. Ta po prvním zavolání endpointu automaticky ukládá jeho odpověď pro budoucí použití, dokud není záznam smazaný nebo neskončí doba jeho platnosti.

Mně úplně nevyhovovalo řešit živostnost cache, jelikož jsem schopen identifikovat přesný okamžik, kdy se každý záznam má vyprázdnit (např. při přidání nového testu se musí smazat všechny cache záznamy ohledně testů), takže jsem se rozhodl nastavit maximální možnou dobu platnosti a spravovat si mazání cache sám. Každý endpoint lze zařadit do tzv. namespace, přes který je pak možné ho smazat, tj. všechny endpointy ohledně testů mají namespace „test“ atd. je tedy velice snadné všechny záznamy promazat jedním příkazem.

V tomto principu jsem udělal jedinou výjimku, a to v případě mazání cache po úpravě testu. Nejprve je ale potřeba udělat malou odbočku a připomenout si, že při dotazu na obsah testu se nevrací jeho kód, který by nebylo potřeba cachovat, ale jeho struktura v blocích popsaných v kapitole Převod kódu testu do bloků. Tento princip neplatí pro uživatelské funkce, u kterých se vrací celý obsah souboru a není potřeba ho cachovat. Když se tedy aktualizuje obsah jednoho testu, není potřeba mazat cache obsahu všech testů, ale pouze toho, který se aktualizoval. Aby bylo možné správně identifikovat nevalidní záznam v cachi, vytvořil jsem pro tento endpoint vlastní implementaci vytváření klíče cachovaného záznamu tak, abych byl schopen klíč znovu vytvořit a následně smazat správný záznam, fastapi-cache2 totiž nevrací klíč k danému záznamu.

6.2.4 Využití WebSocketů

V průběhu vytváření frontendu pro ARTEMIS jsem narazil na problém, že nemám jak poznat, že běžící sada testů doběhla, aniž bych musel pravidelně obnovovat stránku. Zjevným řešením tohoto problému bylo zaintegrovat WebSockety, které jsou ve FastAPI dostupné již v základu, stejně jako ve frontendovém JavaScriptu. Jelikož jsem však nechtěl přijít o potenciál cachování již proběhlých testů, bylo potřeba několik endpointů vytvořit jak ve „statické“ tak „WebSocketové“ verzi. WebSockety jsou tedy využívány ke zjištění stavu testu a k zobrazení (a automatické aktualizaci) logů probíhajícího testu. Webový server může snadno zjistit, jestli daný test již doběhl a následně zavřít daný WebSocket.

6.3 Frontend webové aplikace

Pojďme nyní uzavřít implementaci celého projektu představením hlavních částí webového frontendu ARTEMIS. Vzhled aplikace jsme si již ukázali v kapitole Design, zde se chci tedy zaměřit hlavně na integraci dvou knihoven, které jsou stěžejní ve funkci celého frontendu - CodeMirror pro psaní kódu a React DnD pro práci s drag-and-drop elementy. Na závěr si se všemi nabytými znalostmi představíme poněkud netradiční cyklus v komunikaci mezi frontendem a backendem při psaní testů.

6.3.1 Psaní kódu pomocí CodeMirror

CodeMirror je velice silná a rozšiřitelná knihovna, která poskytuje webovou verzi IDE. Nejen, že poskytuje prostředí podobné klasickým IDE spolu se zvýrazňováním syntaxe podle vybraného jazyku, čísly řádků apod., ale umožňuje vytvořit si vlastní linting (tedy statickou validaci kódu), napovídání vlastních funkcí nebo zobrazení, kde uživatel vidí všechny provedené změny. Všechny tyto funkce jsem se rozhodl do ARTEMIS zaintegrovat, abych co nejvíce zlepšil uživatelskou přívětivost při psaní kódu uživatelských funkcí nebo testů.

Linting mi velice dobře zapadl do již existující validace příchozího kódu, kterou jsem popsal v kapitole Validace. Stačilo pouze vystavit tuto funkcionalitu jako endpoint v API a všechny chyby, které by normálně šly přímo

ke klientovi, přetvořit do objektů, kterým porozumí linter v CodeMirror. Výsledkem je, že pokud uživatel udělá syntaktickou chybu v testu nebo funkci, je daný řádek zvýrazněný a doplněný o vysvětlení vzaté přímo ze *SyntaxError* v Pythonu. Pokud uživatel v testu napíše syntakticky správný kód, který ale nelze zkonvertovat do bloků, je řádek opět zvýrazněn a doplněn o vysvětlení, že daný blok je nevalidní. Pokud jsou v kódu chyby jakéhokoli typu, pak není možné soubor uložit.

I napovídání (známější pod názvem autocomplete) mi zapadlo do již existující funkcionality webového serveru. V rámci testů i uživatelských funkcí je totiž většina kódu tvořena voláním předem známých funkcí, které jsou známé díky dokumentaci. Opět tedy stačilo vzít tento výstup a pouze ho přemapovat do objektů, kterým rozumí CodeMirror a vznikl velice kvalitní napovídač. Napovídání závisí pouze na aktuálním řádku, nejedná se tedy o tak komplexní systém, jakým je IntelliSense[int], ale přesto by měl pokrýt většinu základních požadavků a sloužit jako interaktivní verze dokumentace.

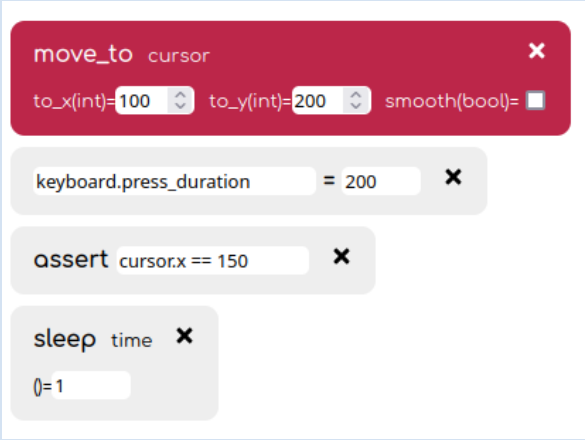
Aby uživatel viděl, jaké změny provedl v daném souboru od posledního uložení, může si zobrazit tzv. MergeView, což je pohled na dva CodeMirror editory vedle sebe, kdy v jednom je obsah souboru uložený na serveru a v druhém aktuální verze na frontendu uživatele. Mezi těmito editory je znázorněné jaké řádky byly přidány, odebrány nebo změněny a uživatel může snadno vrátit provedené změny. Aby tento pohled mohl fungovat, bylo potřeba do serverové odpovědi na dotaz o obsahu testu přidat kromě bloků ještě celý obsah souboru jako text. To se může zdát jako zbytečné, ale zajišťuje se tím to, že uživatel skutečně uvidí přesně tu verzi souboru, která neprošla konverzí na frontendu. Obecně není důvod, aby se verze dramaticky lišily, ale typicky se může jednat o přidání nebo smazání prázdných řádků či jiných syntakticky nevýznamných změn, přesto jsem však chtěl zobrazit skutečný obsah souboru.

6.3.2 Skládání testu pomocí React DnD

React DnD poskytuje spoustu různých variant využití drag-and-drop, od základního přesouvání karet uvnitř definovaného boxu, přes rozřazování karet do různých boxů až po změnu pořadí seřazených karet. Právě tu poslední funkcionalitu jsem se rozhodl využít pro tvorbu editoru testů. React DnD poskytuje funkce zobrazení karet pod sebou podle jejich pořadí a možnost prohazovat je mezi sebou pomocí drag-and-drop, ale blíže nespecifikuje obsah těchto karet. Z mé strany bylo potřeba naimplementovat dvě věci - obsah karet a možnost přidávání nových karet pomocí drag-and-drop.

Obsah karet vychází z bloků, které vrátí server po dotazu na obsah testu. Kromě zobrazení základních informací jako je název funkce a název modulu bylo nejdůležitější vytvořit editovatelné argumenty funkce. Každý argument má svůj typ, podle kterého je možné vybrat správný typ pole pro vložení hodnoty, tedy argument typu *str* bude zobrazený jako textové pole, zatímco argument typu *bool* bude zobrazen jako zaklikávací pole. Pokud funkce něco vrací, zobrazí se okno pro zadání názvu proměnné, do které se má výsledek uložit. Typ argumentu sice určuje typ pole, ale neurčuje žádnou formu validace, jelikož každý argument může být reprezentovaný proměnnou, takže není možné argument typu *int* omezit pouze na číselný vstup. Pokud bychom tedy využili ukázku z tabulky 6.3, bude výsledné zobrazení vypadat následovně.

Kód v Pythonu	`cursor.move_to(100,200) # volani funkce z dokumentace`
	`keyboard.press_duration = 200 # nastaveni promenne`
	`assert cursor.x == 150 # kontrola hodnoty`
	`time.sleep(1) # volani generic ke funkce`

Bloky v editoru	

Tabulka 6.3: Zobrazení kódu ve frontendu webové aplikace

Dále bylo potřeba realizovat možnost přidávat nové bloky do existujícího kódu, na což jsem využil funkci rozřazování karet mezi boxy. Vytvořil jsem si druhý box vedle samotného editoru, který obsahuje vzorové karty, vycházející z dokumentace. Pokud uživatel přesune vzorovou kartu do boxu editoru, karta se zkopíruje do editoru a zůstane v boxu se vzorovými kartami pro další použití. Jediným nedostatkem tohoto přístupu je, že nové karty je možné přidávat pouze na konec testu, nikoli mezi již existující karty. Jakmile se však karta zkopíruje, je možné ji v editoru zatřídit podle potřeby kamkoli mezi ostatní karty.

6.3.3 Cyklus komunikace při psaní testů

Jak už víme, server předává frontendu test ve formě bloků. Frontend tyto bloky následně zobrazí jako drag-and-drop elementy. Ale frontend zároveň umožňuje editovat ten stejný test i jako běžný kód a pak musí test v nějakém formátu serveru předat pro uložení. Jak tedy tento cyklus vypadá?

Server převádí kód testu do bloků, jelikož rozumí jeho syntaxi pomocí knihovny AST. Tyto bloky se zobrazí uživateli, ale pokud uživatel přejde na zobrazení kódu, musí frontend převést tyto bloky do textu. K tomu se používá šablona testu, kterou obdrží frontend spolu s bloky, tato šablona obsahuje věci mimo kód samotné funkce uvnitř testu jako jsou obecné importy (typicky ARTEMIS agent) a kód pro ukončení kontejneru po skončení testu. Frontend tuto šablonu doplní jednak o nové importy využitých uživatelských funkcí a testů, o nastavené proměnné, a hlavně o kroky samotného testu. Tato konverze bloků do textu je poměrně jednoduchá a dá se shrnout do *„<výstupní-proměnná> = <modul>.<název>(<argument1>, <argument2>, ...)“*, což může dělat frontend sám. V průběhu psaní testu ve formě kódu se frontend pravidelně dotazuje na linting (dotaz na napovídání proběhne pouze jednou na začátku editace). Pokud uživatel chce přejít zpět do zobrazení drag-and-drop bloků, je nutné kód poslat na server, který ho převede do bloků. Při uložení testu se na server posílá textová reprezentace, aby server nemusel znovu konvertovat bloky do kódu.

Celý tento proces vypadá zbytečně komplikovaně, ale umožňuje editovat stejný dokument ve dvou různých pohledech, mezi kterými lze dynamicky přecházet bez potřeby uložení testu do souborového systému. Při rozsáhlejším nasazení by jistě stálo za to zamyslet se nad optimalizací a třeba vymyslet způsob, jak převod textu do bloků přesunout na frontend a snížit tím nároky na výkon serveru a celkově počet dotazů na server. Jelikož však aplikace poběží na rozumně moderním hardwaru pouze v interní síti a bude ji používat omezené množství lidí, nejedná se o akutní problém, který by bylo potřeba řešit.

Kapitola 7

Testování

V průběhu implementace jsem prováděl testování jednotlivých částí projektu, abych ověřil kvalitu výsledného systému. Pojďme se nyní podívat na výsledky tohoto testování, které průběžně ovlivňovaly proces implementace. Testování se zaměřovalo na tři hlavní kategorie: rychlost OCR, přesnost OCR a uživatelskou přívětivost. V této kapitole se budu věnovat jak procesu testování, tak jeho výsledkům a dopadům na kvalitu projektu.

7.1 Rychlost OCR

Jak jsem zmiňoval již v rešerši v kapitole Tesseract, označilo několik zdrojů TesseractOCR za relativně pomalý oproti konkurenci. Při výběru jsem vycházel z předpokladu, že mé nároky na rychlost nejsou nijak přísné a nebude tedy tato pomalost znatelná. Velice mě tedy zarazilo, když jsem zjistil, že má prvotní implementace čeká na výsledek od Tesseractu několik vteřin a to ještě samotný výsledek není příliš kvalitní. Věděl jsem, že kvalitu budu zlepšovat, ale také mi bylo jasné, že každý další „požadavek“ na Tesseract, který by mohl zpřesnit výsledek, bude prodlužovat celý proces o drahé vteřiny. Nemohu říct, že by existoval konkrétní časový limit na test, který musí ARTEMIS splnit, ale jako vlastní limit jsem si nastavil, že ARTEMIS by měla být alespoň tak rychlá jako manuální tester, což aktuálně nevypadalo příliš slibně (představme si, že by se manuální tester deset vteřin díval na obrazovku a hledal tlačítko „Uložit“, to není příliš reálné). V následujících kapitolách představím, proč tato pomalost není ve skutečnosti chybou Tesseractu, jak jsem ji vyřešil a na závěr porovnám všechny varianty řešení.

7.1.1 Integrace pomocí knihovny Pytesseract

Pokud budete hledat na internetu způsob, jak integrovat Tesseract do Pythonu, jistě neminete knihovnu Pytesseract. Té jsem se již věnoval v rešerši, kde jsem ji chválil za to, že využívá „runtime“ verzi Tesseractu, kterou je snazší distribuovat do kontejnerů. Právě v tom ale tkví její hlavní nevýhoda, jelikož s Tesseractem interaguje pomocí příkazové řádky a musí přes ni předat všechny parametry (hlavně pak obrázek k přečtení). Proces je tedy následující: Pytesseract obdrží obrázek jako objekt v paměti, obrázek uloží do dočasného souboru, cestu tohoto souboru předá Tesseractu, ten ho převede do textu, text uloží do souboru (název je vstupním parametrem), Pytesseract tento soubor přečte a výsledek vrátí uživateli. Tento proces je vidět v interní funkci Pytesseractu *run_and_get_output*, kterou jsem doplnil o vlastní komentáře:

```
def run_and_get_output(
    image,  # obrazek jako objekt v pameti
    extension='',
    lang=None,
    config='',
    nice=0,
    timeout=0,
    return_bytes=False,
):

    # obrazek se ulozi na disk, ziska se jmeno obrazku a vystupniho souboru
    with save(image) as (temp_name, input_filename):
        kwargs = {
            'input_filename': input_filename,
            'output_filename_base': temp_name,
            'extension': extension,
            'lang': lang,
            'config': config,
            'nice': nice,
            'timeout': timeout,
        }

        # volani Tesseractu pres prikazovou radku
        run_tesseract(**kwargs)
        filename = f"{kwargs['output_filename_base']}{extsep}{extension}"
        # nacteni vystupniho souboru a vraceni obsahu
        with open(filename, 'rb') as output_file:
            if return_bytes:
                return output_file.read()
            return output_file.read().decode(DEFAULT_ENCODING)
```

Kód 7.1: Proces interakce mezi Pytesseract a Tesseract

Ukládání a načítání souborů jak na straně Pytesseractu, tak na straně Tesseractu je velmi časově náročné a nutnost zakládat podprocesy při komunikaci přes příkazovou řádku rychlosti také nepomůže. Knihovna Pytesseract je velmi užitečná pro jednorázové načtení, či práci s většími dokumenty, kde

bude Tesseract skutečně pracovat déle, ale rozhodně se nejedná o škálovatelné řešení. Konkrétní dobu běhu ukážu později v kapitole Porovnání rychlostí různých implementací.

7.1.2 Paralelní volání Pytesseractu

Už při úvodním testování funkcí Tesseractu jsem si všiml, že má lepší výsledky pokud na vstupu dostane obrázek pouze daného slova či řádku, spíše než větší části obrazovky, na které musí hledat text. Detailněji se tomuto fenoménu budu věnovat při testování přesnosti, ale pro tuto chvíli je klíčové, že jeden obrázek z aplikace se rozdělí až na několik desítek podobrázků, které se zvlášť předávají Tesseractu pro rozpoznání. Když jsem tedy narazil na problém s rychlostí popsaným v předchozí kapitole, jako první řešení mě napadlo tento proces paralelizovat.

Místo sériového volání Pytesseractu jsem tedy zkusil volání paralelizovat na počtu vláken v mém procesoru. Tento pokus byl velice neúspěšný a místo jednotek vteřin jsem čekal až desítky. To mě velice překvapilo, jelikož jsem čekal při nejhorším stejný čas jako u sériového řešení.

S takto katastrofálním výsledkem jsem pokus rovnou zahodil, aniž bych se příliš zamýšlel nad důvodem tohoto problému, a určitou nápovědu jsem dostal až mnohem později v procesu vývoje při paralelním pouštění několika kontejnerů. Tesseract běží na procesoru a nevyžaduje tedy grafickou kartu, na rozdíl od většiny moderních neuronových sítí. To ale znamená, že při paralelním zatížení, kdy na procesoru běží několik instancí Tesseractu, nedosahuje procesor takového výkonu, jakého by mohla dosáhnout grafická karta. Jakmile totiž několik kontejnerů najednou spustí svou instanci Tesseractu, je paralelní čas horší, než kdyby stejné požadavky proběhly sériově. Nejedná se o dramatické zpoždění a tento problém nastává pouze pokud souběžné testy vyžadují funkci OCR ve zhruba stejnou chvíli, takže se nejedná o tak závažný problém jako při paralelizaci OCR v rámci jednoho kontejneru. Tesseract má tedy evidentně problém s paralelním během, respektive jeho výkon klesá rychleji než kapacita procesoru, kterou má k dispozici.

7.1.3 Integrace pomocí TessBaseAPI

Po předchozích pokusech mi došlo, že není možné dál pokračovat s Pytesseractem a že bude potřeba obrátit se na nativní TessBaseAPI a to zaintegrovat do

ARTEMIS. Tím jsem si však vytvořil dva problémy: a) jak zaintegrovat C++ knihovnu do Pythonu a b) jak získat knihovnu Tesseractu pro CentOS 7.

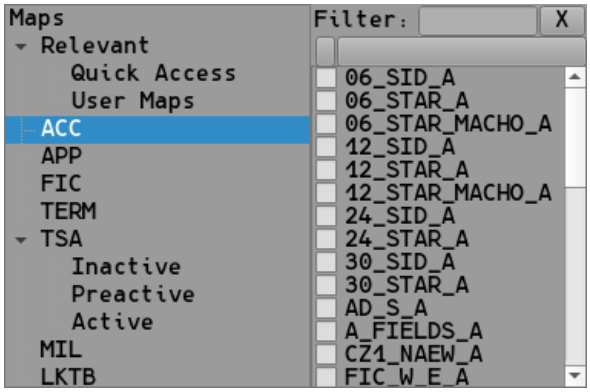
První problém se mi podařilo vyřešit relativně snadno, jelikož jsem na internetu našel již existující projekt, v rámci kterého tento wrapper existoval.[tese] V průběhu implementace jsem ho lehce modifikoval, konkrétně jsem dodal obaly funkcí pro TessBaseAPI verze 5, ale za většinu kódu vděčím právě tomuto projektu.

Získání knihovny Tesseractu byl větší oříšek. Zatímco balíček runtime verze Tesseractu byl dostupný na oficiálních zdrojích (do verze Tesseractu 3) i na neoficiálních[tesb] (verze Tesseractu 4, tu jsem využil pro integraci s Pytesseractem), knihovnu Tesseractu jsem si ale pro CentOS 7 musel sestavit sám.[tesa] Když jsem měl tu možnost, rozhodl jsem se využít nejnovější verzi Tesseractu, tedy 5.0.6. Kromě samotné knihovny Tesseract jsem musel sestavit i vlastní verzi knihovny Leptonica[lep], kterou Tesseract využívá a jejíž potřebná verze není dostupná pro CentOS 7.

Výsledné zrychlení však rozhodně stálo za námahu, jelikož jsem se z několika vteřin dostal na několik desetin vteřiny, což mělo celkově velký pozitivní dopad na pocitovou rychlost celého systému.7.2

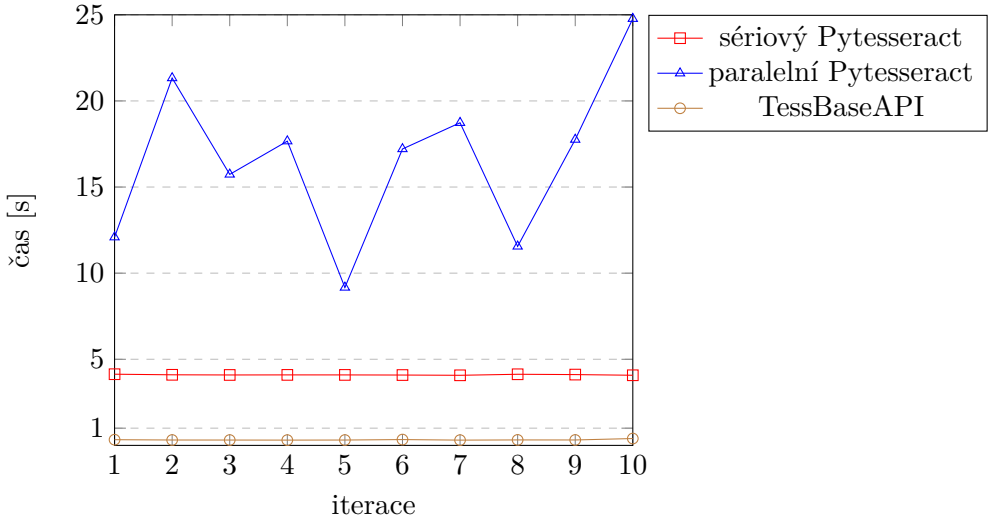
7.1.4 Porovnání rychlostí různých implementací

Po představení všech možných implementací se pojďme podívat na konkrétní čísla, na která jsem se doposud vágně odkazoval. Jako vstupní obrázek jsem zvolil okno pro výběr zobrazených map v systému.7.1 Tento obrázek obsahuje velké množství textu, rozděluje se proto na velké množství podobrázků a slouží tedy jako dobrý indikátor rychlosti rozpoznání. Zároveň má relativně rozumný kontrast mezi textem a pozadím, mezi jednotlivými verzemi Tesseractu tedy neexistuje rozdíl v přesnosti jeho rozpoznání, což by mohlo mít vliv na rychlost.



Obrázek 7.1: Okno pro výběr zobrazených map

Pro testování jsem napsal jednoduchou metodu, která rozdělí obrázek do menších bloků a následně každý blok předá Tesseractu pomocí Pytessractu nebo TessBaseAPI. Pro každou implementaci jsem spustil jedenáct iterací s tím, že první iteraci jsem bral jako rozehřívací a dalších deset jsem bral jako směrodatné.



Obrázek 7.2: Porovnání rychlostí funkcí find a read pro Pytesseract a TessBaseAPI

V obrázku 7.2 je vidět porovnání běhu všech těchto testů. Bez příliš dlouhého zkoumání člověk pozná, jak velký rozdíl je mezi jednotlivými iteracemi. Zajímavá je vizualizace nekonzistentnosti výsledků u paralelního Pytesseractu, která pravděpodobně souvisí s celkovou nevhodností paralelizace Tesseractu. Tento test jsem několikrát opakoval a paralelní implementace Pytesseractu byla vždy takto nekonzistentní.

7.2 Přesnost OCR

V kapitole Rychlost OCR jsme vytvořili základ pro úspěšné čtení textu z obrázků, ale nyní je potřeba se zaměřit na přesnost výsledků, které Tesseract poskytuje. Využití pro čtení textu z obrazovky nějaké aplikace totiž není hlavní účel typického OCR. Většina OCR se zaměřuje spíše na digitalizaci knih, čtení PDF souborů nebo rozpoznávání účtů, ale čtení textu z aplikací má určitá specifika, kterými se většina OCR jednoduše nezabývá. Jedním příkladem za všechny je barva pozadí. U knihy či dokumentu můžeme předpokládat, že pozadí bude téměř vždy konzistentní barvy a většinou bílé, zatímco u aplikace je potřeba pracovat s překrývajícími se okny, barevným pozadím, barevnými tlačítky apod.

Aby tedy Tesseract dosahoval co nejlepších výsledků, je zapotřebí mu trochu napomoci. V této kapitole si tedy představíme jednotlivé kroky preprocessingu[1] před tím, než se samotný obrázek dostane do Tesseractu. Nejprve se podíváme, jak segmentovat text z obrázku pomocí tresholdingu, dilatace a eroze a následně si představíme kroky, které tuto segmentaci zpřesňují, konkrétně odstraňování čar, hit-or-miss a barevnou segmentaci.

7.2.1 Thresholding

Při práci s Tesseractem jsem velice rychle zjistil, že pracuje výrazně přesněji, když mu předám obrázek s jedním slovem či řádkem, spíše než když mu předám obrázek celého okna a nechám Tesseract „hledat“ text. Tento přístup je podpořen i tím, že Tesseractu lze předat tzv. *PageSegmentationMode*, který mu napoví, co má na obrázku očekávat, typicky „jedno písmeno“, „jeden řádek“, „jeden odstavec“ či „volný text“. Tesseract tuto hodnotu využívá k zpřesnění svého výstupu. Já jsem na začátku používal „volný text“, kde Tesseract musí podle vlastní segmentace najít text na obrázku, ale následně jsem přešel na vlastní segmentaci a nyní předávám obrázky konkrétních slov s typem „jeden řádek“.

Abych se k takové segmentaci dostal, je prvním krokem segmentovat text pomocí tzv. thresholdingu, tedy metody, kdy se využívá předpoklad, že text je barevně kontrastní od svého pozadí a vytvoří se tedy barevný threshold (hranice), kdy všechny tmavší pixely jsou text a světlejší pixely jsou pozadí

[1] Preprocessing označuje kroky provedené před hlavní částí zpracování daného objektu. Opakem je postprocessing, který probíhá po hlavní části zpracování.

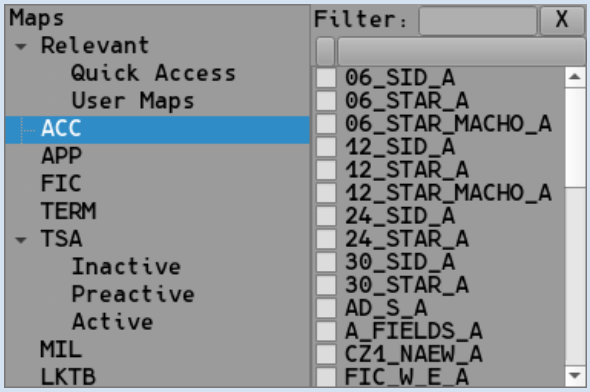
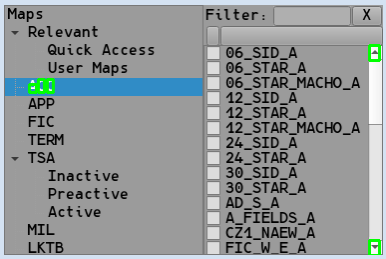
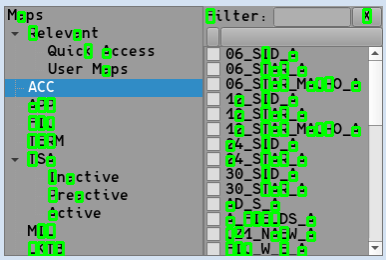
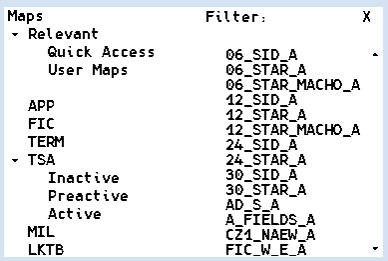
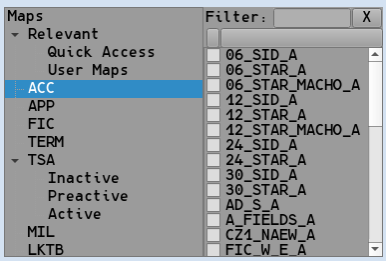
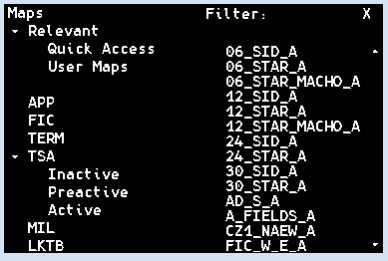
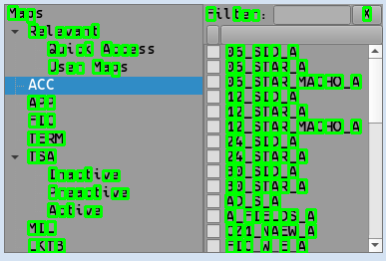
(pro tmavý text na světlém pozadí) nebo naopak (pro světlý text na tmavém pozadí). K vytvoření tohoto thresholdu se mi hodila knihovna OpenCV, kterou jsem prozkoumal již na začátku projektu. V první fázi jsem využil Otsuovu metodu[Ots79], která v mém pozorování dosahovala nejkonzistentnějších výsledků ze všech thresholdingových metod. Tato metoda se (alespoň v implementaci OpenCV) zaměřuje na segmentaci světlého popředí z tmavého pozadí, proto jsem jako druhý threshold zvolil inverzní Otsuovu metodu, která segmentuje tmavé pixely ze světlého pozadí. Jako doplňkové thresholdy jsem zvolil ještě bílou a černou masku, které segmentovali pouze tyto dvě barvy, ale v krajních případech segmentovaly text, který jiné metody „neviděly“. V ukázce kódu 7.2 můžeme vidět nejprve převod obrázku do stupňů šedi a následnou tvorbu thresholdingových předloh pro následnou segmentaci.

```
def get_thresholds(img):
    # vstupni obrazek se prevede na stupne sedi, tedy hodnoty 0 az 255
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # bila maska segmentuje pouze pixely s hodnotou 245 az 255
    white_mask = cv2.inRange(gray, np.asarray([245]), np.asarray([255]))
    # cerna maska segmentuje pouze pixely s hodnotou 0 az 10
    black_mask = cv2.inRange(gray, np.asarray([0]), np.asarray([10]))
    # otsu segmentace a prevod do binarniho obrazku, ktery ma pouze hodnoty
    0 nebo 255
    ret, otsu = cv2.threshold(gray, 0, 255, cv2.THRESH_OTSU + cv2.
    THRESH_BINARY)
    # otsu segmentace a prevod do inverzniho binarniho obrazku
    ret, otsu_inv = cv2.threshold(gray, 0, 255, cv2.THRESH_OTSU + cv2.
    THRESH_BINARY_INV)
    return [otsu, otsu_inv, white_mask, black_mask]
```

Kód 7.2: Metoda pro získání thresholdingových předloh pomocí Otsuovy metody

V tabulce 7.1 vidíme jednak vstupní obrázek a následně pro každou z výše zmíněných metod její masku[2] a následnou segmentaci nalezeného textu. Každý zelený obdélník značí samostatnou oblast nalezeného textu. Na první pohled vidíme, že Otsuova metoda nenašla žádný světlý text na tmavém pozadí, zato její inverzní verze našla velkou část textu v obrázku. Černá maska nalezla pouze vybraná písmena a bílá maska jako jediná našla text „ACC“, který je psaný bílým textem na (alespoň podle Otsua) světlém pozadí, takže žádná jiná metoda ho nenašla. Kombinací výsledků těchto čtyř metod je tedy možné relativně spolehlivě identifikovat veškerý text na obrázku. Je však také zjevné, že aktuální výstup není použitelný, jelikož nechci segmentovat jednotlivá písmena, ale spíše slova či řádky, je tedy potřeba preprocessing dále zlepšovat.

[2]Maska je taková verze původního obrázku, kde jsou bílé jen ty pixely, které splňují parametry dané metody. Všechny ostatní pixely jsou černé.

Vstupní obrázek		
	Maska	**Oblasti textu**
Bílá maska		
Černá maska		
Otsu		
Otsu inverzní		

Tabulka 7.1: Thresholdingové metody a jejich nalezené oblasti textu

Zde ještě stojí za to se krátce pozastavit nad tím, jak se z masky vytvářejí jednotlivé zelené obdélníky v tabulce 7.1. Využívám k tomu metodu OpenCV, která z masky umí získat seznam kontur, tedy v tomto případě seznam oblastí bílých pixelů, ohraničených černými pixely. Následně pro každou konturu vezmu její *bounding box*, tedy obdélník opsaný této oblasti, a pokud není příliš malý ani příliš velký, což by značilo chybnou konturu, identifikuje tento obdélník oblast textu, kterou můžu předat Tessseractu. Pro lepší vizualizaci pouze tento obdélník vykresluji v původním obrázku pomocí zelené barvy.

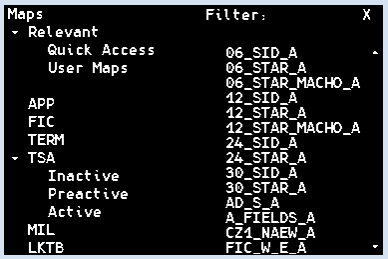
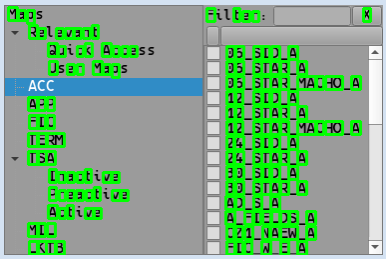
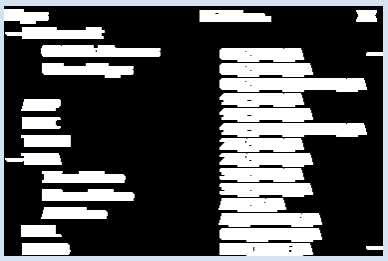
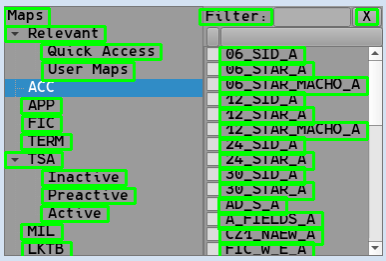
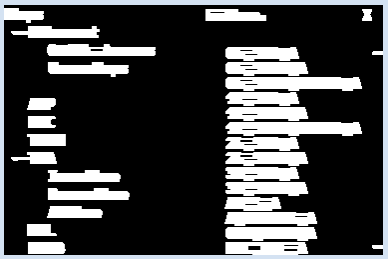
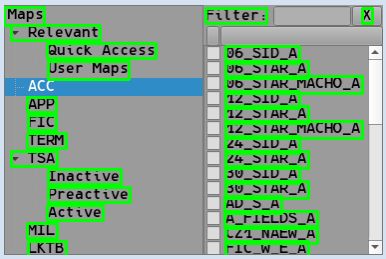
7.2.2 Dilatace a eroze

Abychom z jednotlivých písmen nalezených pomocí thresholdingu složili slova, je zapotřebí každou nalezenou konturu jednoho písmene sloučit s okolními konturami a vytvořit jednu konturu pro celé slovo. Na to se velmi dobře hodí metoda dilatace, která slouží právě ke slučování sousedních kontur. Dilataci je možné si představit tak, že každou bílou oblast v masce obkreslím v určité vzdálenosti a vnitřek vybarvím bíle, tím pádem zvětším původní oblast a pokud byla v blízkosti jiná bílá oblast, pak tyto oblasti nejspíše splynou do jedné. V praxi je tento proces prováděný pro každý pixel zvlášť, tedy každý černý pixel, který má v dané vzdálenosti bílý pixel, se nastaví jako bílý.

Dilatací se nám podařilo spojit sousední písmena do slov, ale krajní písmena tím zbytečně rozšířila oblast celého slova na část, kde žádné písmeno není. Proto je potřeba doplnit proces o tzv. erozi, která je přesným opakem dilatace. Opět si ji můžeme představit tak, že každou bílou oblast v masce obtáhneme v určité vzdálenosti, ale tentokrát zevnitř, a všechny bílé pixely mimo tuto oblast vybarvíme černě. V praxi je eroze opět prováděná pro každý pixel zvlášť, tedy každý bílý pixel, který má v dané vzdálenosti černý pixel, se nastaví jako černý.

Přestože eroze je přesným opakem dilatace, tak se nejedná o inverzní funkci, jelikož dilatací se spojí dvě oblasti, které se již erozí znovu nerozdělí, protože jejich spoj již není na okraji nově vytvořené oblasti. Kombinací těchto dvou metod je možné sloučit sousední písmena do slov, ale zároveň zachovat původní velikost slova.

V tabulce 7.2 vidíme jako vstup masku Otsuovy metody a z ní vytvořené oblasti textu, které jsme získali v kapitole Thresholding. Dále vidíme masku po jednom cyklu dilatace a po jednom následném cyklu eroze. Zelené obdélníky vytvořené z masky po erozi správně obsahují celé slovo či řádek tak, jak by člověk očekával při vyhledávání v textu, ale zároveň nejsou zbytečně široké

	Maska	Oblasti textu
Vstup		
Po dilataci		
Po erozi		

Tabulka 7.2: Proces dilatace a eroze na masce inverzní Otsuovy metody

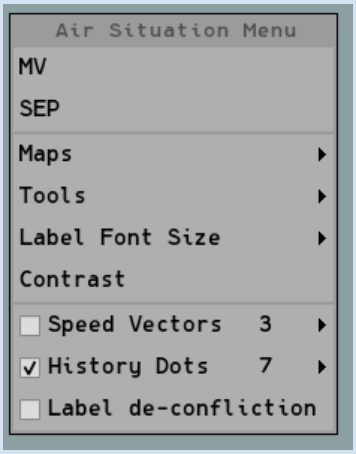
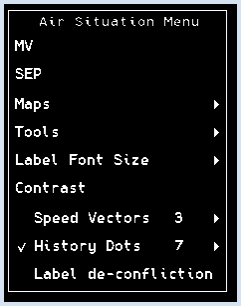
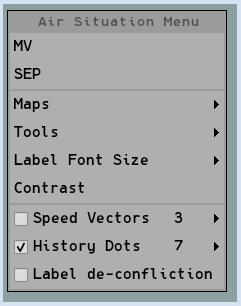
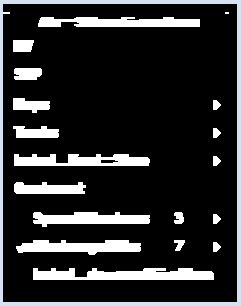
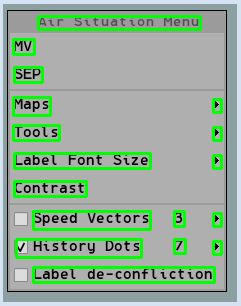
a nezasahují do okolních prvků, jako jsou checkboxy nebo scrollbar. Tato míra segmentace je dostačující pro čtení textu z oken, jako je okno pro výběr aktivních map.

7.2.3 Odstraňování čar

Ne všechen text je však možné číst z jasně definovaných oken, jako jsem ukazoval doposud. Příkladem jsou kontextová menu, která se otevřou někde v blízkosti kurzoru, ale není možné přesně definovat jejich polohu a velikost. Kvůli tomu je potřeba udělat větší snímek obrazovky a s tím přichází problém okrajových čar kolem takového menu. Obecně jakýkoli obrázek, který kromě textu obsahuje i svislé či vodorovné čáry ve stejné barvě jako text, bude mít tyto čáry segmentované při thresholdingu a tím se naruší následná segmentace textu.

Po thresholdingu tedy odstraňuji všechny svislé a vodorovné čáry nad

určitou velikost. Jelikož písmena jsou také složena z čar, bylo nutné zvolit experimentálně takové velikosti, aby nedocházelo k poškození čitelnosti textu.

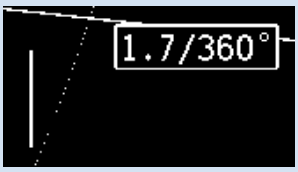
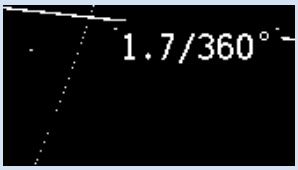
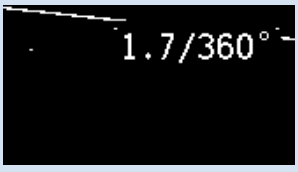
Vstupní obrázek			
	Maska	**Po dilataci a erozi**	**Oblasti textu**
Otsu inverzní			
Po odstranění čar			

Tabulka 7.3: Maska a nalezené oblasti textu před a po odstranění čar

V tabulce 7.3 můžeme vidět, že bez odstranění čar nenajde můj algoritmus v obrázku vůbec žádný text. To je způsobeno tím, že dilatací a erozí splynou oblasti textu se svislými čárami a vytvoří tím jednu velkou konturu, která zahrnuje celou oblast menu a není tedy možné rozpoznat kontury jednotlivých slov. Pokud ale před dilatací a erozí odstraníme svislé a vodorovné čáry, další kroky probíhají podle představ a algoritmus nalezne veškerý text. Díky této metodě mám pokryté všechny hlavní oblasti textů, které potřebuji segmentovat.

7.2.4 Hit-or-miss

Metoda hit-or-miss je spíše doplňková k předešlým vylepšením, jelikož jsem nenarazil na konkrétní situaci, kde by bez ní segmentace nefungovala a s ní ano, ale rozhodně se jedná o metodu vylepšující celkovou přesnost segmentace. Hit-or-miss lze použít pro odstranění samostatných pixelů, které netvoří žádný konkrétní útvar, ale po dilataci a erozi by mohly narušovat další kroky. Obecně se tato metoda využívá pro tzv. denoising, který může vizuálně vyčistit obrázek od drobných nedokonalostí. Já tuto metodu využívám hlavně při segmentaci textu z map, kdy může být text nad čárou či symbolem, který je hit-or-miss schopen odstranit. Při implementaci jsem však často narážel na problém, že hit-or-miss odstraňoval i části textu, konkrétně okraje písmen, takže jsem musel zvolit odstraňování pouze velice malých oblastí.

Vstupní obrázek	
Otsu inverzní	
Po odstranění čar	
Po hit-or-miss	

Tabulka 7.4: Maska před a po hit-or-miss

Tabulka 7.4 ukazuje, jak metoda hit-or-miss odstranila pozůstatky vertikální šikmé čáry. Tato čára neovlivňuje přímo samotnou segmentaci, ale jedná se o určitou formu vyčištění obrázku.

7.2.5 Barevná segmentace

Posledním a také doplňkovým vylepšením preprocessingového algoritmu je možnost barevné segmentace podle výběru uživatele. Velká část textu na obrazovce aplikace je záměrně nekontrastní, aby nedocházelo k namáhání očí řídících. Jako metodu poslední záchrany, pokud žádná z výše popsaných metod nepomůže autorovi testu segmentovat potřebný text, může manuálně zadat barvu textu, který chce segmentovat a rozpětí barev, které mají být součástí výběru a tím je zaručené, že bude možné text segmentovat. Tato segmentace funguje podobně jako při vytváření bílé a černé masky, ale místo stupňů šedi pracuje s celým barevným spektrem. Autor testu si tedy zvolí barvu ve formátu RGB reprezentovanou třemi čísly v rozmezí 0 až 255 a rozpětí reprezentované jedním číslem v rozpětí od 0 do 255. Rozpětí se odečte od barvy a tím se vytvoří spodní hranice segmentace (ekvivalent hodnoty 245 pro bílou masku). Přičtením rozpětí k barvě se vytvoří horní hranice segmentace (ekvivalent hodnoty 255 pro bílou masku). Rozdílem oproti bílé a černé masce je to, že tato barevná segmentace není využívána jen pro identifikaci oblastí s textem, ale je přímo vstupem do Tesseractu, jelikož i ten může mít problémy se segmentací takto problémového textu.

Vstupní obrázek	
Otsu	
Otsu inverzní	
Barevná segmentace podle šedého textu	

Tabulka 7.5: Barevná segmentace nekontrastního textu

Pro ilustraci tohoto problému jsem si vytvořil vlastní obrázek, který je specifický tím, že text je v jedné části světlejší než pozadí a v druhé části tmavší než pozadí. Pro lidské oko není nepřekonatelný problém přečíst nápis „TEST“, ale Otsuova metoda není schopná korektně zvolit hranici a při nejlepším segmentuje pouze „ST“. Pokud však správně zvolíme segmentační barvu jako barvu textu a nastavíme malé rozpětí, je možné segmentovat úspěšně celý text, jak je vidět v tabulce 7.5. V tuto chvíli se mi nepodařilo nalézt ani vymyslet žádnou situaci, ve které by můj algoritmus nebyl schopen segmentovat text z obrázku v rámci možností testované aplikace.

7.3 Uživatelská přívětivost

Závěrečným bodem testování je ověřit, že ARTEMIS splňuje požadavky na uživatelskou přívětivost. Tu je potřeba ověřit na dvou frontách: jednak u programátorů, kteří s ní budou ověřovat novou funkcionalitu, a jednak u testerů, kteří s ní budou provádět testy před předáním nové verze zákazníkovi. Každá z těchto skupin má své specifické požadavky, proto jsem se rozhodl rozdělit je do vlastních kategorií. V průběhu implementace jsem dostával drobné připomínky, které jsem většinou rovnou zapracoval, proto se v této kapitole budu zaměřovat na hlavní architektonické koncepty a jejich dopady na uživatele ARTEMIS.

7.3.1 Hodnocení programátorů

Programátoři umí samozřejmě pracovat s kódem, proto pro ně nebyla stěžejní funkcionalita drag-and-drop, ale velice oceňovali možnost psát testy i funkce přímo ve webovém rozhraní, které jim umožňovalo i napovídání, linting a možnost vidět seznam posledních změn. Stejně tak jim přišlo velice užitečné mít dostupné API ARTEMIS, přes které je možné jak automatizovaně spouštět testy v pravidelných intervalech, tak získat výsledky testů pro tvorbu reportů.

Jednou z připomínek bylo umístění testů a funkcí pouze na disku serveru kde běží ARTEMIS a rádi by viděli integraci se systémem pro verzování kódu, tedy že testy a funkce budou uložené ve verzovaném repozitáři a ARTEMIS k nim bude přistupovat klonováním těchto souborů na disk. Díky tomu by totiž bylo možné testy spravovat i mimo webovou aplikaci a umožnit vrácení se k předchozí verzi testu či funkce.

7.3.2 Hodnocení testerů

Testeři ocenili funkci drag-and-drop pro psaní testů a celkovou jednoduchost a přehlednost webové aplikace. Překvapivě také ocenili možnost editovat testy jako kód, jelikož jim to umožní snadné kopírování testů, což v sobě nemá webová aplikace jinak zabudované. Chválili strukturu jednotlivých funkcí ARTEMIS, převážně pak celou funkcionalitu OCR, jelikož aktuálně disponují několika automatizovanými testy, které v sobě mají napevno zapsané pozice jednotlivých tlačítek, takže při změně aplikace je potřeba často tyto hodnoty

znovu počítat, přestože se tlačítku nezměnil text ani funkce, pouze jeho pozice.

Nedostatek viděli v ploché struktuře testů, funkcí a výsledků a rádi by viděli možnost zatřídění těchto elementů do složek, které mohou odpovídat jednotlivým funkčním blokům nebo verzím aplikace. Plochá struktura bez složek komplikuje orientaci v těchto sekcích hlavně při vysokých desítkách či stovkách testů, které v budoucnu mohou v ARTEMIS být.

Kapitola 8

Závěr

V této práci jsem představil svůj projekt ARTEMIS, který slouží pro automatizaci testování desktopových aplikací za pomoci virtualizace v Dockeru a OCR enginu Tesseract. Nyní bych se rád ohlédl za úspěšností stávajícího řešení a následně představil plány pro jeho budoucí rozšíření.

8.1 Úspěch projektu

Celkově vnímám projekt jako velmi úspěšný, jelikož se mi podařilo pokrýt všechny hlavní požadavky jak v teoretické, tak praktické rovině a vytvořil jsem prostor pro další rozšíření jak funkcionality, tak integrace. ARTEMIS vnímám jako validní alternativu k existujícím produktům s vlastním přístupem ke konkrétním problémům jako je práce s displayem nebo se sítí. ARTEMIS je aktuálně nasazená na jednom z firemních serverů a probíhá na ní testovací provoz, kdy se kolegové prakticky seznamují se všemi funkcemi.

Mým osobním úspěchem je i samostatné naplánování a provedení celého projektu od počáteční myšlenky až k celostnímu řešení. Vnímám, že jsem byl schopen zužitkovat znalosti nasbírané v rámci mého studia jak v oblastech analýzy a vedení projektu, tak v řešení specifických problémů na relativně nízké úrovni. Velice mne potěšily komentáře o ARTEMIS jako o produkční aplikaci, která má přesah nad rámec samotné diplomové práce. To platilo i pro můj bakalářský projekt, ale ARTEMIS je samostatný systém, který má díky názvu, logu a webové aplikaci svou vlastní identitu.

8.2 Budoucí rozšíření

Proces vývoje ARTEMIS je sice u konce v rámci této diplomové práce, ale mám v plánu na projektu dál pracovat a existuje několik oblastí, kterým se chci věnovat. Nejvíce nasnadě jsou připomínky uživatelů z kapitoly o testování uživatelské přívětivosti, tedy zapracovat využívání verzovacího systému pro kód testů a funkcí a přidat možnost zařazování testů, funkcí a výsledků do složek. Kromě toho však existuje možnost vylepšit možnost ARTEMIS integrovat do jiných systémů například tím, že bude poskytovat strukturované výsledky testů nad rámec struktury pro webové rozhraní. S tím souvisí i celková možnost vytváření reportů přímo z ARTEMIS, které mohou být dále zařazené do výstupů pro zákazníka. V neposlední řadě se chci zaměřit na možnost testovat pomocí ARTEMIS i další firemní aplikace a celkově zrobustnit testovací procesy uvnitř systému ARTEMIS. Pokud se mi tato rozšíření podaří zapracovat, není nereálné, že by se ARTEMIS nabízela jako samostatný produkt ostatním firmám ve stejné sféře.

Příloha A

Literatura

[ast] *Abstract Syntax Tree* [online]. URL: https://docs.python.org/3/library/ast.html [zobrazeno 20.05.2024].

[aut] *AutoIt* [online]. URL: https://www.autoitscript.com/site/ [zobrazeno 20.05.2024].

[cen] *The CentOS Project* [online]. URL: https://www.centos.org/ [zobrazeno 20.05.2024].

[cod] *CodeMirror* [online]. URL: https://codemirror.net/ [zobrazeno 20.05.2024].

[Dil23] Cem Dilgemany. *OCR in 2024: Benchmarking Text Extraction/Capture Accuracy* [online]. 2023. URL: https://research.aimultiple.com/ocr-accuracy/ [zobrazeno 20.05.2024].

[dja] *Django* [online]. URL: https://www.djangoproject.com/ [zobrazeno 20.05.2024].

[Doc] *Docker* [online]. URL: https://www.docker.com/ [zobrazeno 20.05.2024].

[fasa] *FastAPI* [online]. URL: https://fastapi.tiangolo.com/ [zobrazeno 20.05.2024].

[fasb] *fastapi-cache2* [online]. URL: https://pypi.org/project/fastapi-cache2/ [zobrazeno 20.05.2024].

[fla] *Flask* [online]. URL: https://flask.palletsprojects.com/en/3.0.x/ [zobrazeno 20.05.2024].

[ins] *Inspect* [online]. URL: `https://docs.python.org/3/library/inspect.html` [zobrazeno 20.05.2024].

[int] *IntelliSense* [online]. URL: `https://code.visualstudio.com/docs/editor/intellisense` [zobrazeno 20.05.2024].

[jen] *Jenkins* [online]. URL: `https://www.jenkins.io/` [zobrazeno 20.05.2024].

[KM09] Kevin S. Killourhy and Roy A. Maxion. *Comparing Anomaly Detectors for Keystroke Dynamics. Proceedings of the 39th Annual International Conference on Dependable Systems and Networks*, pages 125–134, 2009.

[KSD15] Manish Kumar, Santosh Kumar Singh, and Dr. R. K. Dwivedi. *A Comparative Study of Black Box Testing and White Box Testing Techniques. International Journal of Advance Research in Computer Science and Management Studies*, 3:32–44, 2015.

[lep] *Leptonica* [online]. URL: `http://www.leptonica.org/` [zobrazeno 20.05.2024].

[opea] *OpenAPI* [online]. URL: `https://www.openapis.org/` [zobrazeno 20.05.2024].

[opeb] *OpenCV Python* [online]. URL: `https://github.com/opencv/opencv-python` [zobrazeno 20.05.2024].

[Ots79] Nobuyuki Otsu. *A Threshold Selection Method from Gray-Level Histograms. IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, 1979. `doi:10.1109/TSMC.1979.4310076`.

[pil] *Pillow* [online]. URL: `https://python-pillow.org/` [zobrazeno 20.05.2024].

[pya] *PyAutoGUI* [online]. URL: `https://github.com/asweigart/pyautogui` [zobrazeno 20.05.2024].

[pyt] *Python Tesseract* [online]. URL: `https://github.com/madmaze/pytesseract` [zobrazeno 20.05.2024].

[ran] *Ranorex* [online]. URL: `https://www.ranorex.com/` [zobrazeno 20.05.2024].

[rea] *React DnD* [online]. URL: `https://react-dnd.github.io/react-dnd/about` [zobrazeno 20.05.2024].

[RHFN+12] Fernando Rodríguez-Haro, Felix Freitag, Leandro Navarro, Efraín Hernánchez-sánchez, Nicandro Farías-Mendoza, Juan Antonio Guerrero-Ibáñez, and Apolinar González-Potes. *A summary of virtualization techniques. Procedia Technology*, 3:267–272, 2012.

[RJN95] Stephen V. Rice, Frank Jenkins, and Thomas A. Nartker. *The Fourth Annual Test of OCR Accuracy*. 1995.

[scr] *Scratch* [online]. URL: `https://scratch.mit.edu/` [zobrazeno 20.05.2024].

[sik] *SikuliX* [online]. URL: `http://sikulix.com/` [zobrazeno 20.05.2024].

[Smi07] Ray Smith. *An Overview of the Tesseract OCR Engine. Ninth International Conference on Document Analysis and Recognition*, 2:629–633, 2007.

[SOG22] Sanket Salunke, Abdelkader Ouda, and Jonathan Gagne. *Transfer Learning for Behavioral Biometrics-based Continuous User Authentication. 2022 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6, 2022. `doi:10.1109/ISNCC55209.2022.9851764`.

[swa] *Swagger UI* [online]. URL: `https://swagger.io/tools/swagger-ui/` [zobrazeno 20.05.2024].

[tesa] *Compilation guide for various platforms* [online]. URL: `https://tesseract-ocr.github.io/tessdoc/Compiling.html` [zobrazeno 20.05.2024].

[tesb] *OCR-tesseract-on-Centos7* [online]. URL: `https://github.com/second-state/OCR-tesseract-on-Centos7` [zobrazeno 20.05.2024].

[tesc] *Tesseract* [online]. URL: `https://github.com/tesseract-ocr/tesseract` [zobrazeno 20.05.2024].

[tesd] *Tesseract Documentation* [online]. URL: `https://tesseract-ocr.github.io/tessapi/5.x/` [zobrazeno 20.05.2024].

[tese] *tesseract-ocr-wrapper* [online]. URL: `https://github.com/Altabeh/tesseract-ocr-wrapper` [zobrazeno 20.05.2024].

[tesf] *TestLink* [online]. URL: `https://testlink.org/` [zobrazeno 20.05.2024].

[Tom17] Martin Tomaschek. *Evaluation of off-the-shelf OCR technologies*. Bachelor's thesis, Masarykova univerzita, listopad 2017.

[vmw] *WMvare* [online]. URL: `https://www.vmware.com/` [zobrazeno 20.05.2024].

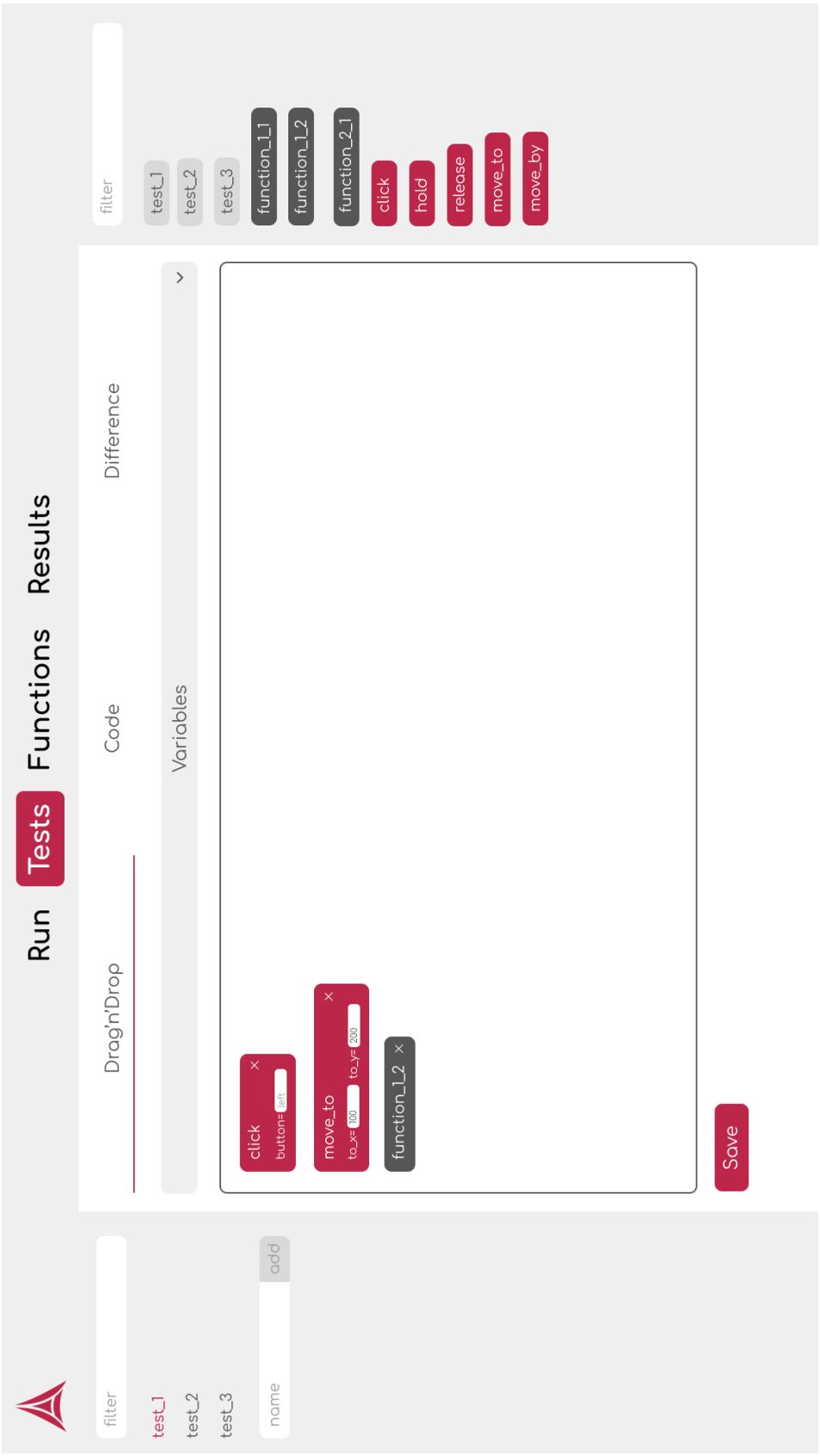
[xte] *XTest* [online]. URL: `https://www.x.org/releases/X11R7.7/doc/libXtst/xtestlib.html` [zobrazeno 20.05.2024].

[zap] *ZAPTEST* [online]. URL: `https://www.zaptest.com/` [zobrazeno 20.05.2024].

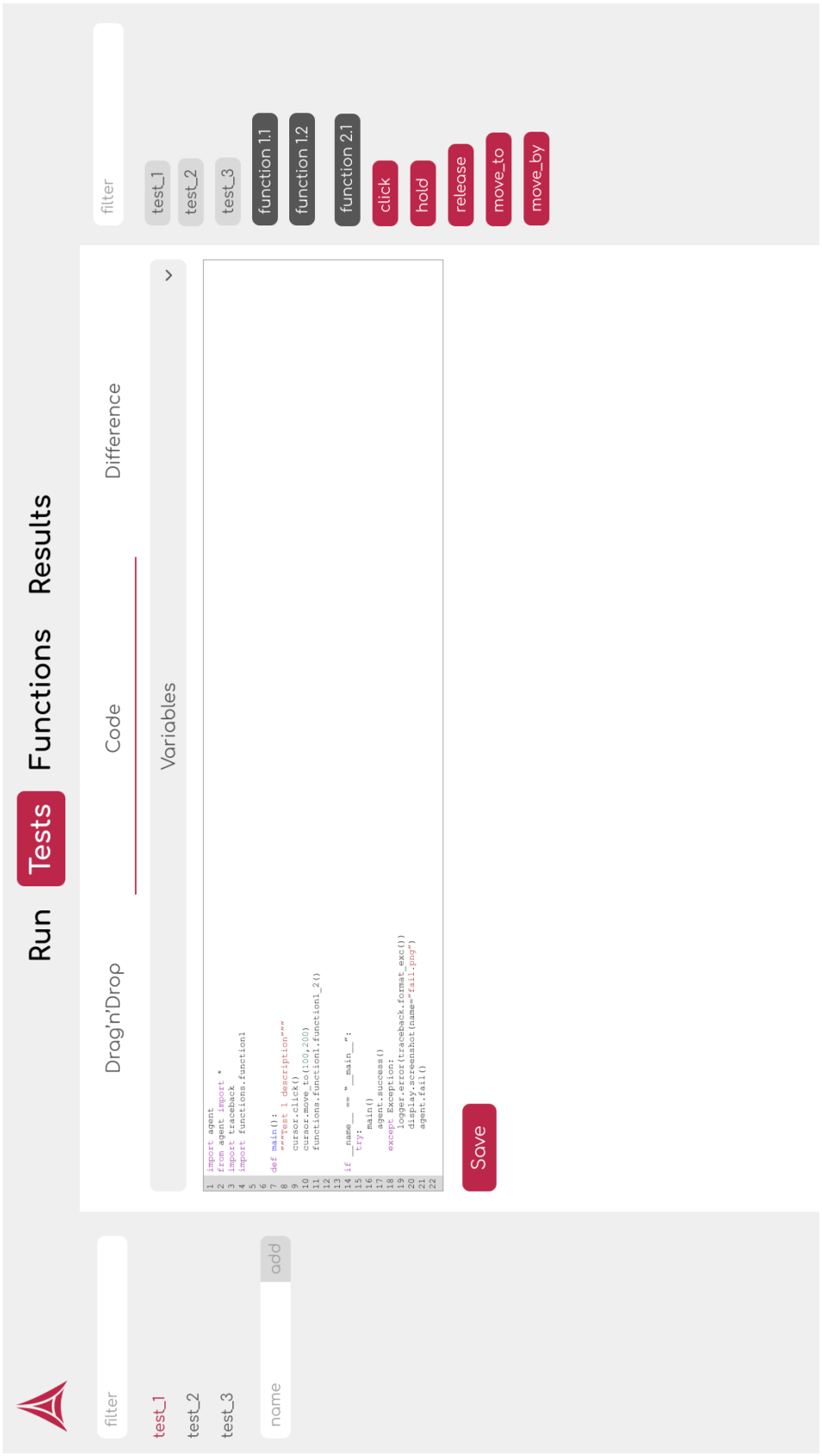
Příloha B

Návrh vzhledu webové aplikace

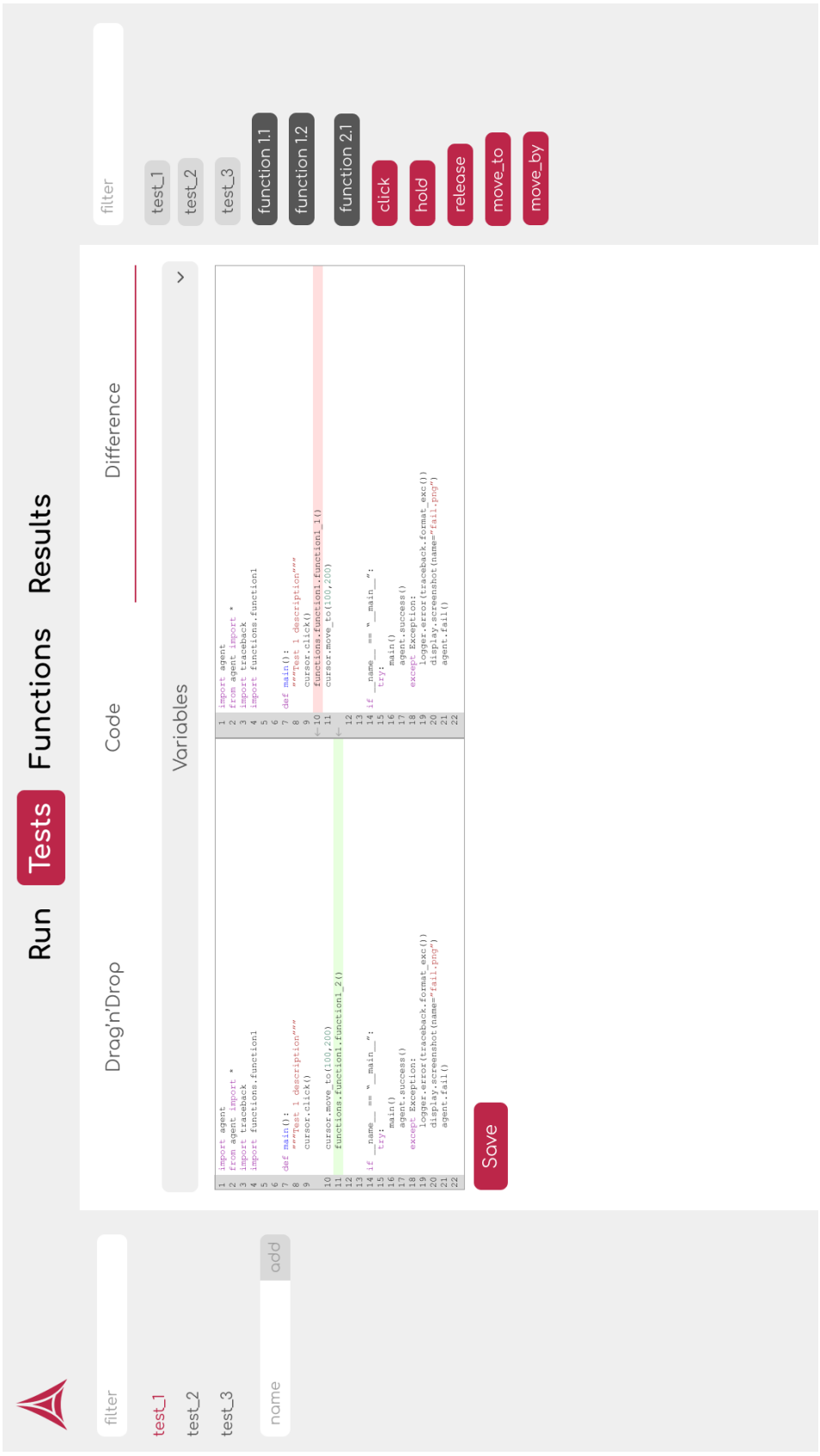
Obrázek B.1: Spuštění nové sady testů



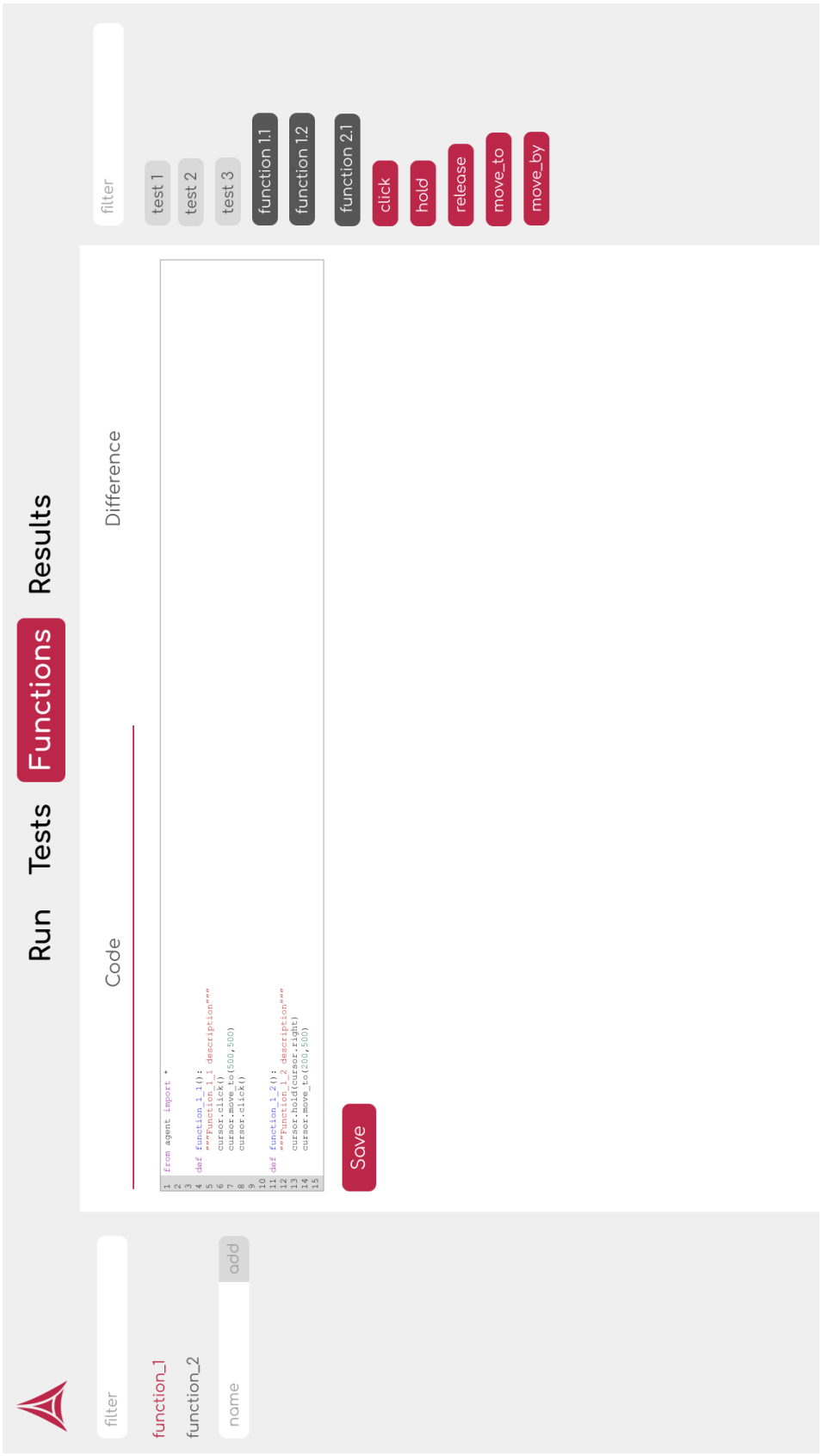
Obrázek B.2: Editace testu v režimu drag-and-drop



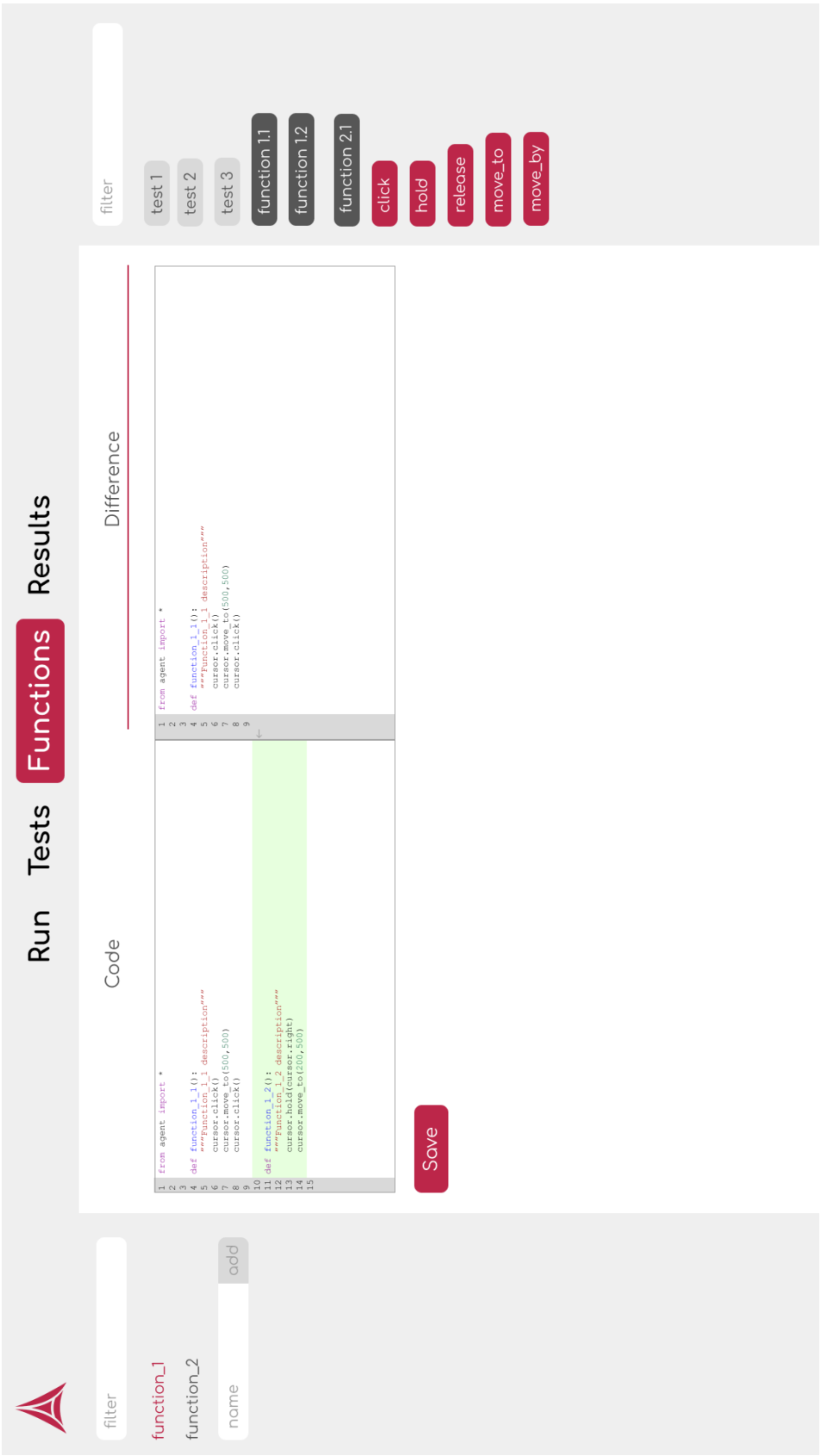
Obrázek B.3: Editace testu v režimu kódu



Obrázek B.4: Náhled na rozdíl kódu testu mezi aktuální verzí a verzí na serveru



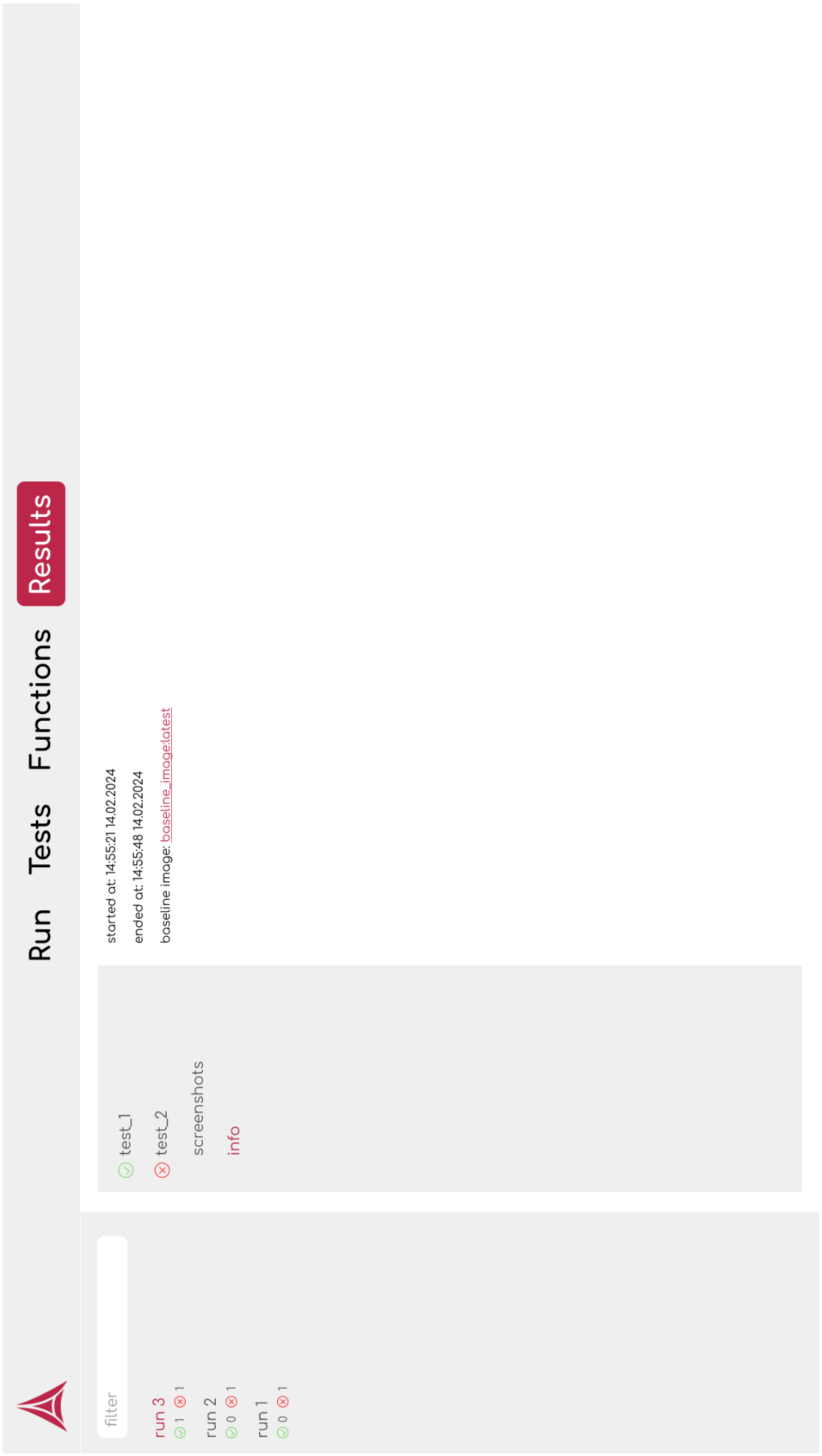
Obrázek B.5: Editace uživatelské funkce



Obrázek B.6: Náhled na rozdíl kódu uživatelské funkce mezi aktuální verzí a verzí na serveru

Obrázek B.7: Výpis logů proběhlého testu

Obrázek B.8: Seznam obrázků uložených v průběhu provádění sady testů



Obrázek B.9: Informace o provedené sadě testů