VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

AUTOMATIZOVANÉ TESTOVÁNÍ GUI POMOCÍ KONTEJNERŮ

AUTOMATED GUI TESTING USING CONTAINER TECHNOLOGY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JURAJ SOJČÁK
AUTHOR

VEDOUCÍ PRÁCE Ing. ALEŠ SMRČKA, Ph.D.
SUPERVISOR

BRNO 2016

Abstrakt

Cieľom bakalárskej práce je vytvorenie systému na automatizované testovanie GUI desktopových aplikácií, ktorý sa dá jednoducho nasadiť ako testovacie prostredie. Tento systém pozostáva z nástroja, v ktorom je užívateľ schopný vytvárať testovacie skripty, a interpretu, ktorý spúšťa dané testovacie skripty a vyhodnocuje ich. Testovacie skripty užívateľ vytvára pomocou jazyka, ktorý je tiež popísaný v tejto práci. Jednoduché nasadenie systému zabezpečuje použitie kontajnerovej virtualizácie.

Abstract

The aim of this Bachelor thesis is to create a system for automated testing of GUI desktop applications that may be easily used as a test environment. The system consists of, first, an application in which the user is able to create the test scripts, and second, the interpreter that runs and evaluates the relevant test scripts. The user creates the test scripts using a language that is also described in this thesis. Simple deployment of the system ensures the use of container-based virtualization.

Kľúčové slová

automatizované testovanie GUI, kontajnerová virtualizácia, Xpresser, Docker

Keywords

automated GUI testing, container-based virtualization, Xpresser, Docker

Citácia

SOJČÁK, Juraj. *Automatizované testování GUI pomocí kontejnerů.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Smrčka Aleš.

Automatizované testování GUI pomocí kontejnerů

Prehlásenie

Prehlasujem, že túto bakalársku prácu som vypracoval samostatne pod vedením pána Ing. Aleša Smrčku, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Juraj Sojčák
17. mája 2016

Poďakovanie

Ďakujem môjmu vedúcemu Ing. Alešovi Smrčkovi, Ph.D. za jeho pomoc a odborné rady poskytnuté pri písaní tejto práce.

Obsah

Kapitola 1

Úvod

Testovanie softvéru je jednou z hlavných častí softvérového inžinierstva, avšak tento proces sa v minulosti dosť zanedbával. Postupom času ako sa softvér stával neoddeliteľnou súčasťou všedného života, tak na testovanie softvéru sa kládli čoraz väčšie nároky. Tento proces odhaľuje a poskytuje informácie o chybách v softvéri. Kým budeme vyvíjať softvér, tak isto v ňom budeme produkovať chyby. Samozrejme, že nie vedome. Proces testovania sa snaží tieto vady odhaliť a tým spôsobom zvyšuje kvalitu vývoja produktu a kvalitu samotného produktu.

Grafické užívateľské rozhranie, ďalej označované ako GUI (angl. Graphical User Interface), je najrozšírenejšia metóda na interakciu užívateľa s aplikáciou. GUI pozostáva z objektov, ako sú napríklad tlačítka, zaškrtávacie polia, rozbaľovacie menu a tieto objekty sú vykresľované na obrazovku ako príslušné okná (angl. window). Užívateľ pomocou klávesnice, myši a iných vstupných zariadení, posiela týmto objektom vstupnú udalosť alebo sekvenciu udalosti, ktoré program spracuje a týmto spôsobom dochádza ku zmene stavu aplikácie. Napríklad v aplikácií WordPad od spoločnosti Microsoft je možné vykonať 325 operácií[8] a to prináleží množstvu stavov, do ktorých sa môže aplikácia dostať. Otestovanie týchto prechodov medzi stavmi je veľmi náročný proces, preto je testovanie GUI zložitejšie oproti testovaniu konvenčného softvéru.

Teoretický základ, z ktorého vychádzala práca, je priblížený v kapitole 2. Opisuje technológiu kontajnerovej virtualizácie, ktorá je kľúčová pri tvorbe tejto práce, bližšie objasňuje obor testovania GUI a zároveň oboznamuje čitateľa o nástrojoch, ktoré sa pri tomto procese najbežnejšie využívajú. Vlastnosti, ktoré bude musieť systém spĺňať sú špecifikované v kapitole 3. Táto kapitola obsahuje aj návrh systému, ktorého cieľom je ukázať ako sa budú jednotlivé požiadavky realizovať. Je tu popísaná štruktúra systému a jeho jednotlivých časti. Taktiež je tu znázornený diagram kolaborácie, ktorý znázorňuje interakciu medzi jednotlivými modulmi systému a užívateľmi. V kapitole 4 sú prebraté niektoré implementačné detaily systému. Softvérovým produktom tejto práce je nástroj na testovanie GUI a jeho zhodnotenie popsiuje kapitola 5.

Kapitola 2

Technológie súvisiace s automatizovaným testovaním grafického užívateľského rozhrania

Táto kapitola najprv priblížuje pojem virtualizácie a bližšie vysvetľuje kontajnerovú virtualizáciu. Druhá časť kapitoly sa venuje testovaniu GUI a nástrojom, ktoré sa pri tomto testovaní využívajú.

2.1 Virtualizácia

Výsledný testovací systém využíva pre automatizáciu možnosti virtualizácie. V prvej časti je prinesený celkový náhľad na virtualizáciu, kde budú uvedené základné pojmy a taktiež popísané niektoré druhy virtualizácie, v druhej časti je bližšie popísaný projekt Docker, ktorý výsledný systém práve využíva.

2.1.1 Základný princíp

Virtualizáciu môžeme chápať ako technológiu, ktorá vytvára vrstvu nad fyzickými prostriedkami počítača (procesor, operačná pamäť, ...), ktorá umožňuje tieto prostriedky ďalej zdieľať [7]. S virtualizáciou sú spojené nasledujúce pojmy [1]:

- Hostiteľský počítač.

  Tento pojem označuje fyzický počítač, na ktorom je vykonávaná virtualizácia. Na tomto počítači sú následne spúšťané klientské počítače.

- Virtuálny počítač (klientský počítač).

  Chová sa ako fyzický počítač, avšak reálne je reprezentovaný iba kolekciou súborov na hostiteľskom počítači.

- Hypervízor.

  Tento pojem označuje softvér alebo firmvér, ktorý virtualizuje fyzické prostriedky počítača a simuluje prostriedky pre virtuálne počítače. Taktiež sa využíva na celkovú správu virtuálnych počítačov (vytváranie, spúšťanie, zastavovanie). Hypervízor rozdeľujeme na dva typy podľa toho, či je spustený priamo na hardvéri hostiteľského

počítača, alebo či je spustený ako aplikácia, alebo služba v operačnom systéme hostiteľského počítača [22].

Virtualizácia nám umožňuje pomerne jednoducho spravovať virtuálne počítače nezávisle od seba. V praxi to väčšinou vyzerá tak, že na hostiteľskom počítači je nainštalovaný hypervízor ako aplikácia v operačnom systéme a pomocou tejto aplikácie je možné spravovať virtuálne počítače a následne sa v nich môžu inštalovať operačné systémy. Správou virtuálneho počítača sa myslí, jeho vytvorenie, spúšťanie, vypínanie, poprípade zastavovanie. Týmto počítačom je možné prideľovať dynamicky rôzne zdroje ako procesor, operačná pamäť, šírka prenosového pásma. Existuje niekoľko rôznych druhov virtualizácie[7].

Plná virtualizácia

Plná virtualizácia je technika, pri ktorej sa vykonáva úplná simulácia hardvéru pre virtuálny počítač. To znamená, že všetky rôzne vlastnosti ako inštrukčná sada, alebo aj BIOS musia byť virtualizované, čo prináša väčšie hardvérové nároky pre hostiteľský počítač. To prináša výhodu úplnej izolácie virtuálneho počítača od hostiteľského počítača a operačný systém nemusí byť žiadnym spôsobom modifikovaný pre beh vo virtuálnom počítači, čo prináša ďalšiu výhodu pri migrácii, keďže operačný systém sa neskôr môže nasadiť aj na fyzickom hardvéri.

Najznámejšie riešenia, ktore ponúkajú plnú virtualizáciu sú napríklad *VritualBox*, *VMware* a *KVM*.

Paravirtualizácia

Pri tomto druhu virtualizácie sa využíva iba čiastočná simulácia hardvéru a taktiež sa využíva upravené jadro operačného systému vo virtuálnom počítači. Následne toto dovoľuje, aby tento operačný systém dokázal komunikovať s hypervízorom, čo pozitívne vplýva na hardvérové nároky, keďže niektoré operácie môže virtuálny počítač vykonávať priamo na hostiteľskom počítači.

Kontajnerová virtualizácia

Kontajnerová virtualizácia sa ináč označuje ako aj virtualizácia na úrovni operačného systému. V tomto prípade pri virtualizácii nevzniká nový virtuálny počítač, ale uzatvorené prostredie, ktoré sa nazýva kontajner. Kontajner je skupina procesov, ktoré zdieľajú množinu parametrov s jedným alebo viacerými podsystémami [3].

Podrobnejšie popísaný projekt *Docker*[1], ktorý danú technológiu značne spopularizoval [19] a je využívaný v tejto práci, je popísaný v kapitole 2.1.3.

Chroot[2] (change root) je systémové volanie, pomocou ktorého je možné izolovať spúšťaný proces a jeho potomkov od zvyšku súborového systému. Pomocou tohoto volania sa posunie koreň súborového systému do nového vytvoreného adresára, a tak tento adresár sa stáva koreňovým adresárom pre nový proces, pred ktorým chceme skryť zvyšok systému. Využíva sa hlavne na bezpečnostné zabezpečenie, testovanie. Taktiež sa považuje za predchodcu kontajnerovej technológie [11].

[1] Projekt Docker: https://www.docker.com/
[2] Linuxová manuálová stránka volania chroot: http://linux.die.net/man/2/chroot

BSD jail je mechanizmus, ktorý nadväzuje a rozširuje koncept volania chroot, ale v tomto prípade už hovoríme o virtualizácii na úrovni operačného systému. BSD jail dokážeme vytvoriť bezpečné prostredie s vlastným prístupom do súborového systému, s množinou užívateľov a s vlastnou IP adresou [20].

Linuxové kontajnery (LXC) je virtualizačná metóda, ktorá umožňuje vytvárať na hostiteľskom počítači kontejnery, ktoré sú vlastne linuxové systémy, ale zdieľajú jadro s hostiteľským počítačom, ktoré plní úlohu hypervízoru, a teda stará sa o prideľovanie fyzických prostriedkov pre daný kontajner. Túto funkcionalitu zabezpečuje pomocou vlastností linuxového jadra zvanej kontrolné skupiny (angl. cgroups[3]) a menné priestory (angl. namespaces[4]). Prvá menovaná vlastnosť dokáže obmedziť spotrebu fyzických prostriedkov pre určitú skupinu procesov. Druhá vlastnosť slúži na izolovanie fyzických zdrojov pre procesy, ktoré sa nachádzajú v tzv. menných priestoroch.

2.1.2 Nástroje projektu Libvirt

Libvirt je súbor nástrojov na správu virtuálnych počítačov, avšak neposkytujú služby hypervízora, ale spravujú virtuálne počítače hypervízorov, ktoré sú nainštalované na hostiteľskom počítači[5]). Tento súbor nástrojov zahŕňa aplikačné rozhranie, démona (*libvirtd*) a program *virsh* [13].

Virsh sa využíva ako konzolová aplikácia na vytváranie, pozastavenie a vypínanie virtuálnych počítačov, listovanie všetkými virtuálnymi počítačmi atď [2]. Táto aplikácia využíva k svojmu fungovaniu práve aplikačné rozhranie *libvirt*, ktoré využívajú aj ďalšie aplikácie[6], napr. desktopová aplikácia *virt-manager*, ktorá je tiež schopná spravovať virtuálne počítače avšak za pomoci grafického užívateľského rozhrania.

Virtuálne počítače sa označujú ako domény a sú reprezentované pomocou XML formátu. Koreňový element je vždy `<domain>`, ktorý obsahuje atribút `type` pre špecifikovanie hypervízoru, ktorý sa využíva pre spúšťanie danej domény. Taktiež môže obsahovať atribút `id`, ktorý identifikuje spustenú doménu unikátnym číslom (nebežiaca doména neobsahuje daný atribút).

Pre pripojenie na konkrétny hypervízor libvirt využíva špecifické URI, ktoré umožňuje taktiež pripojenie cez sieť na vzdialený hypervízor. Podmienkou prístupu na vzdialený hypervízor je použitie zabezpečeného spojenia a taktiež to, že na vzdialenom počítači musí byť nainštalovaný libvirt démon [14]. Po úspešnom pripojení je možné využívať rovnaké aplikačné rozhranie, ktoré je rovnaké pre všetky podporované hypervízory [12].

2.1.3 Projekt Docker

Docker je jednou z implementácií kontajnerovej virtualizácie a dopĺňa túto technológiu s vlastnými pracovnými postupmi a nástrojmi pre vývoj aplikácií v kontajneroch. Je to open-source projekt pod licenciou Apache a je vo verzii 1.11.1 [7]. Podobne ako LXC, tiež využíva vlastnosti cgroups a namespaces a pôvodne na to využíval práve rozhranie LXC, ale od verzie 0.9 prešiel na vlastné rozhranie, ktoré môže byť prevádzkované súčasne s LXC [6]. Základom tohoto rozhrania je knižnica *libcontainer*, ktorá je napísaná v jazyku Go [10].

[3]Cgroups: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt
[4]Manuálová stránka namespaces: http://man7.org/linux/man-pages/man7/namespaces.7.html
[5]Podporované hypervízory: http://libvirt.org/drivers.html
[6]http://libvirt.org/apps.html
[7]https://github.com/docker/docker/blob/master/CHANGELOG.md#1111-2016-04-26

Pre vytvorenie kontajnerov Docker využíva šablóny zvané obrazy (angl. images). Tieto obrazy môže užívateľ sťahovať priamo z repozitárov na to určených, môže ich vytvárať už z vytvorených kontajnerov alebo ich vytvára pomocou súboru *Dockerfile*. Obraz je tvorený z niekoľkých vrstiev. Prvou vrstvou je základný obraz (base image) a môže to byť napríklad systém Ubuntu. Ďalšie vrstvy sú tvorené krokmi, kde jeden krok znamená jednu inštrukciu. Tieto inštrukcie sa využívajú práve v súbore *Dockerfile* a sú to napríklad:

- FROM obraz

  Inštrukcia FROM musí byť vždy prvou inštrukciou v *Dockerfile* a pomocou nej sa nastavuje základný obraz pre novo vznikajúci obraz. Príklad použitia: *FROM ubuntu:latest*.

- RUN príkaz

  Pomocou tejto inštrukcie sa vykoná akýkoľvek príkaz v novej vrstve a výsledok príkazu sa použije do ďalšej vrstvy. Príklad použitia *RUN apt-get update*.

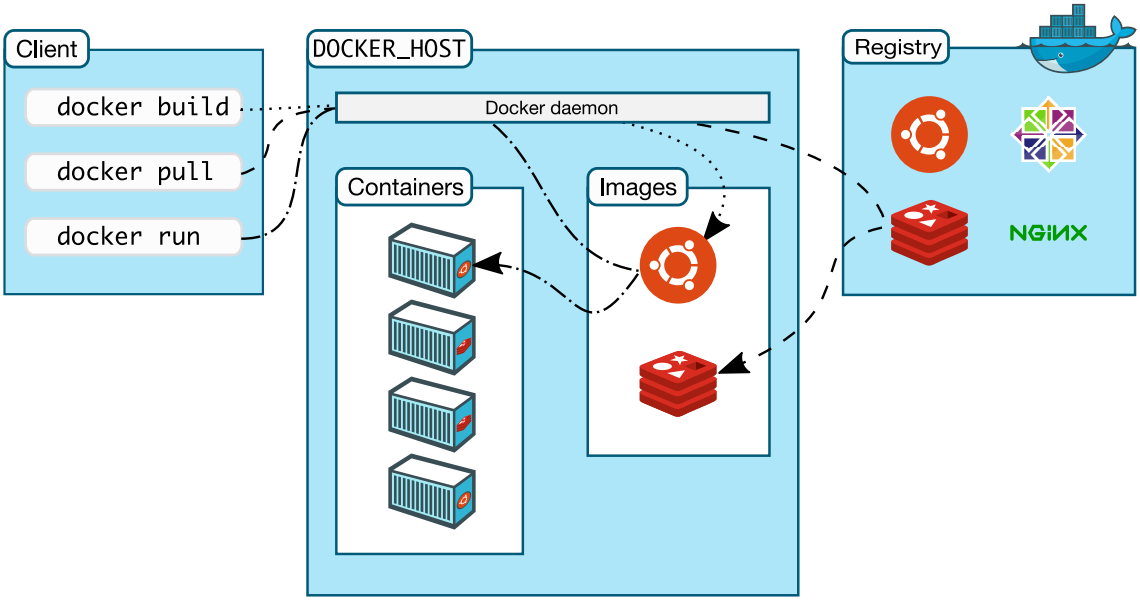
- CMD príkaz

  Táto inštrukcia definuje príkaz, aký sa vykoná pri spustení kontajneru. Ak je v súbore *Dockerfile* viac inštrukcií *CMD*, tak iba poslená z nich nadobúda účinnosť a ostatné sa ignorujú.

- USER meno_pouzivatela

  Táto inštrukcia nastavuje používateľa pre nasledujúce príkazy *RUN* alebo *CMD*.

Docker funguje na princípe klient-server a skladá sa z troch častí: Docker klient, Docker démon a Docker register. Na obrázku 2.1 je zobrazená architektúra systému, kde sú zobrazené všetky tri časti.



Obr. 2.1: Architektúra Docker systému. Prevzaté z [10]

.

Démon Docker je spustený na hostiteľskom počítači a jeho úlohou je spravovanie obrazov, ktoré sa nachádzajú lokálne na počítači, popr. dokáže stiahnuť obrazy s Docker registra. Užívateľ pristupuje k démonu pomocou Docker klienta, ktorý komunikuje s démonom pomocou soketov alebo pomocou REST API.

2.2 Testovanie GUI

Väčšina dnešných aplikácií využíva k svojmu ovládaniu GUI a zásluhu má na tom hlavne fakt, že oproti textovému užívateľskému rozhraniu, je GUI intuitívnejšie a jednoduchšie na ovládanie pre bežného užívateľa. Avšak testovanie GUI je oproti testovaniu konvenčného softvéru omnoho náročnejší proces, keďže prechod medzi dvoma stavmi aplikácie môže byť ovplyvnený mnohými užívateľskými vstupmi a počet týchto sekvencií udalostí je veľmi vysoký [21].

2.2.1 Manuálne testovanie

Tento druh testovania vykonávajú samotný testeri. Cieľom je otestovať určité zásady zobrazených prvkov, taktiež sa kontroluje správne zobrazenie pri rôznom rozlíšení, pri rôznej veľkosti okna aplikácie. Výhodami tohoto prístupu je väčšia šanca objavenia chyby a na ich základe sa môžu nájsť podobné chyby. Na druhej strane, je treba vynaložiť viac úsilia a taktiež tento druh testovanie je časovo náročnejší proces.

2.2.2 Automatizované testovanie

Automatizované testovanie je rozdelené na dva prístupy:

Testovanie zo znalosti ovládacích prvkov

Pri tomto druhu testovania sa používajú nástroje, ktoré využívajú pre svoju činnosť napojenie sa do štruktúry určitého GUI frameworku a následne sa zavolá callback pre simulovanie užívateľských vstupov. Najpoužívanejšie frameworky pre tvorbu GUI majú minimálne jeden nástroj, ktorým sa dá vykonávať daný druh testovania.

Medzi najznámejšie nástroje patria:

- Selenium[8] je súbor nástrojov na testovanie webových aplikácií, ktoré využívajú znalosť DOM. Dokážu nahrať užívateľské vstupy a následne ich vyexportovať do určitého formátu. Testy je možné ďalej upravovať ako bežný kód, keďže má väzbu na rôzne programovacie jazyky.

- Dogtail[9] je knižnica určená na testovanie GTK+ a QT aplikácií. Pre túto činnosť využíva prístup k asistenčným technológiám.

- Squish [10]je komerčný nástroj na testovanie GUI, ktorý sa dá využiť na testovanie aplikácii bežiacich na rôznych platformách (PC, mobilné zariadenia, vstavané systémy).

[8] http://www.seleniumhq.org/
[9] https://fedorahosted.org/dogtail/
[10] https://www.froglogic.com/

Testovanie pomocou rozpoznávania objektov

Pri tomto druhu testovania sa na obrazovke vyhľadávajú objekty pomocou rôznych algoritmov alebo vzorcov, následne po úspešnom vyhľadaní sa na príslušné miesto simuluje užívateľská akcia. Na vyhľadávanie objektov slúži napríklad knižnica *OpenCV*, popr. na rozpoznávanie textu sa využívajú knižnice implementujúce metódy OCR (angl. Optical Character Recognition). Výhodou tohoto testovania je, že sa testuje to, čo by užívateľ naozaj videl. Nevýhodou je samotný proces vyhľadávania, keďže je časovo náročný a taktiež výsledky vyhľadávania nie sú vždy presné. Taktiež takýto proces testovania nevie zareagovať na nepredvídateľné udalosti, napr. vyskakujúce okná.

2.3 Nástroje na testovanie GUI pomocou rozpoznávania objektov

V tejto časti sú popísané najznámejšie nástroje pre testovanie GUI. Najprv je popísaný projekt Sikuli vyvíjaný na univerzite MIT, druhým z nich je modul Xpresser a nakoniec komerčný nástroj s názvom Applitools Eyes.

2.3.1 Sikuli

Sikuli je systém na automatizáciu činnosti na obrazovke, čo sa využíva pri testovaní aplikácií. Tento systém je dostupný na operačných systémoch Windows, Linux a Mac a je nutné mať nainštalovanú Javu 6+ [17]. Skladá sa z dvoch častí [16]:

- Sikuli Script a
- Sikuli IDE.

Sikuli Script je knižnica, ktorá automatizuje interakciu (udalosti klávesnice a myši) s objektami grafického užívateľského rozhrania, ktoré vyhľadá pomocou obrázkových vzorov. Táto knižnica je napísaná v Jave a využíva sa na zasielanie udalosti k daným objektom balíček `java.awt.Robot`. Poloha na obrazovke, kde sa má zaslať daná udalosť, sa určí práve podľa nájdenej zhody a slúži na to knižnica OpenCV, ktorá je napísaná v C++ a je spojená s Javou pomocou JNI[11]. Ďalej táto knižnica poskytuje pre užívateľa množinu jednoduchých príkazov.

Sikuli IDE slúži na tvorbu a interpretovanie zdrojových kódov Sikuli. Kombinuje možnosti textového editora a nástroja na zachytávanie obrazovky. V textovom editore je možné vytvárať skripty, ktoré sú tvorené príkazmi ako napríklad `click()` alebo `find()` a užívateľ ma prehľad o tom, na aký objekt sa aplikuje daná udalosť, keďže priamo v textovom editore je zobrazený obrázok daného objektu. Pomocou nástroja na zachytávanie je užívateľ schopný vytvárať obrázky daných objektov priamo na obrazovke, kde je Sikuli IDE spustené.

2.3.2 Xpresser

Xpresser je modul jazyka Python vytvorený Gustavom Niemeyerom v roku 2010. Tento projekt vznikol vďaka myšlienke testovania aplikácii pomocou rozpoznávania objektov z projektu *SikuliX* a taktiež autor chcel vytvoriť testovací nástroj na testovanie GUI aplikácii na operačnom systéme Ubnutu [18].

[11] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/

Činnosti, ktoré modul vykonáva:

- vytváranie screenshotov obrazovky,
- rozpoznávanie objektov a
- generovanie užívateľských udalostí.

Hlavnou úlohou modulu je rozpoznávanie objektov. Na to je potrebné mať zachytenú obrazovku, v ktorej chceme vyhľadávať daný objekt. Modul na zachytávanie obrazovky využíva knižnicu Gtk a cairo, pomocou ktorých vytvára a následne ukladá zachytenú obrazovku ako dočasný PNG súbor.

Vyhľadávané objekty musí užívateľ vytvoriť tiež ako súbor PNG a ďalej ich musí uložiť do pripravenej zložky, ktorú musí špecifikovať *Xpressru* [4].

Na samotné vyhľadávanie objektov sa využíva funkciu `matchTemplate()` z knižnice OpenCV. Tejto funkcii sa poskytne zdrojový obrázok, čo je v našom prípade obrazovka, a obrázok, ktorý sa má vyhľadávať a metóda, pomocou ktorej sa má uskutočniť proces vyhľadávania. Existuje šesť metód a Xpresser využíva metódu `CV_TM_CCOEFF_NORMED`. Výstupom tejto funkcie je zoznam nájdených oblastí s pozíciou nájdenia a s mierou zhody zvanej prah (angl. threshold).

Na generovanie užívateľských udalostí používa knižnicu *python-pyatspi*, ktorá slúži na prístup k asistenčným technológiám. Modul dokáže vygenerovať nasledujúce udalosti (za pomlčkou je názov príslušnej metódy):

- stlačenie ľavého tlačítka - click(),
- dvojité stlačenie ľavého tlačítka - double_click(),
- stlačenie pravého tlačítka - right_click(),
- presunutie kurzoru - hover() a
- písanie na klávesnici - type().

Z praktického hľadiska to vyzerá, že užívateľ si vytvorí PNG súbory daných objektov do vytvoreného priečinka. Vytvorí skript v jazyku Python kde poskytne objektu Xpresser daný priečinok a pomocou volania daných metód špecifikuje, aké udalosti sa majú vykonať v zdrojovom okne. Tie pri spustení volajú funkciu na rozpoznávanie objektov a vypočíta sa na akej pozícii v zdrojovom okne sa má vykonať daná udalosť. Tato pozícia je hodnota `x` a `y` a je to väčšinou stred nájdaného objektu.

Priečinok, ktorý je určený na obrázky pre Xpresser môže obsahovať aj konfiguračný súbor pomenovaný `xpresser.ini`. Tento súbor môže obsahovať informácie, ktoré bližšie určujú dve vlastnosti pre hľadané objekty. Prvou z nich je prah, z ktorým sa má daná oblasť vyhľadávať, keďže explicitný prah je 0.98 (98%). Druhou vlastnosťou je možné upraviť pozíciu vykonávanej udalosti relatívne k objektu.

Výhodou je, že tento modul je open-source a má kvalitné spracovanie rozhrania, ktoré je veľmi jednoducho použiteľné a dokazuje to aj fakt, že posledná modifikácia modulu Xpresser je niekedy z roku 2012. Taktiež je možné vyhľadať rôzne rozšírenia modulu[12].

Nevýhodou je rozpoznávanie objektov, ktoré je v niektorých prípadoch často nepresné a časovo náročné. Nevýhodou môže byť aj skutočnosť, že je nutné nainštalovať ďalšie závisle knižnice, napr. OpenCV a SimpleCV.

[12] https://code.launchpad.net/xpresser

2.3.3 Applitools Eyes

Je to komerčný nástroj na statickú validáciu obsahu aplikácie, ktorú vykonáva na základe porovnávania screenshotov, ktoré zachytáva v testovaných aplikáciách. Takýmto spôsobom poukazuje na vykonané zmeny užívateľského prostredia. Pomocou tohoto nástroja je možné testovať desktopové aplikácie, webové aplikácie a taktiež mobilné aplikácie a to vďaka tomu, že dokáže spolupracovať s frameworkami na testovanie týchto aplikácií [9].

Pre používanie tohoto nástroja je nutné sa registrovať na internetovej stránke tejto spoločnosti. Na výber je účet zadarmo (je značne obmedzený počtom spustení testov), 30-dňová skúšobná verzia alebo platená verzia. Po registrácií si užívateľ môže vybrať s akým frameworkom na testovanie pracuje a v akom jazyk a pre každú možnosť je ukážka kódu ako vytvoriť prvý test. Sú tu s konkrétnymi závislosťami, ktoré užívateľ musí nainštalovať na svoj počítač ako aplikačné rozhranie. Pre používanie tohoto rozhrania je každému užívateľovi vygenerovaný jednoznačný kľúč, ktorý musí používať pri spúšťaní testovacích skriptov.

Pri spustení testovacieho skriptu nástroj najprv zaznamená počiatočný screenshot aplikácie. Tester následne simuluje užívateľské vstupy pomocou testovacieho frameworku a zachytáva ďalšie screenshoty na porovnanie z počiatočným screenshotom. Toto porovnanie sa dá vykonať po skončení vykonávania testovacej sady pomocou vygnerovaného testovacieho reportu vo webovej aplikácii, ktorá je dostupná po prihlásení do vytvoreného užívateľského účtu. Takýto testovací report sa skladá z vytvorených screenhotov pomocou aplikačného rozhrania a z počiatočného screenshotu, ktorý vytvoril samotný nástroj. Aplikácia sama poukáže na rozdiely v obsahu aplikácie. Tieto rozdiely môžu odzrkadľovať zmeny farby prvkov, zmeny v pozícii prvku, zmeny písma atď. Na tieto rozdiely aplikácia upozorňuje vyfarbením oblasti, ktorej sa rozdiel týka. Užívateľ tieto rozdiely môže prehlásiť za validné, alebo naopak ich označí ako nájdenú chybu aplikácie.

Výhodou tohoto nástroja je prístupné užívateľské rozhranie pre testovacie reporty, takže na overovanie týchto reportov môže byť nasadený aj bežný užívateľ.

Kapitola 3

Špecifikácia požiadaviek a návrh testovacieho systému

V tejto kapitole sú zahrnuté dve etapy vývoja výsledného testovacieho systému - analýza požiadaviek a návrh systému. Čitateľ by sa mal dozvedieť, čo bude výsledný systém spĺňať, teda jeho funkcionálne a nefunkcionálne požiadavky a tiež ako budú tieto požiadavky realizované.

3.1 Špecifikácia požiadavkov

V nasledujúcej časti budú popísané dôležité vlastnosti navrhnutého systému.

Funkcionálne požiadavky

P 1. Systém musí byť schopný testovať aplikácie, ktoré sú na hostiteľskom počítači alebo vo virtualizovanom počítači.

V oboch prípadoch musí byť možné sa pripojiť k testovanému systému pripojiť ako VNC klient.

P 2. Systém bude umožňovať návrat ku predchádzajúcim stavom testovaného systému.

To bude umožnené iba v prípade, ak testovaný systém bude možné spustiť vo virtualizovanom počítači (ESX, VirtualBox, QEMU).

P 3. Systém bude schopný rozpoznávať vyhľadávané objekty v zdrojovom okne.

Vyhľadávaný objekt musí byť obrázok ovládacieho prvku aplikácie alebo oblasť, ktorú chceme verifikovať a na danom obrázku musí byť zachytená aj časť okolia tohoto objektu, čo umožní presnejšie určenie zhody v zdrojovom okne. Nebude možné rozpoznať text v zdrojovom okne.

V prípade, že by sa zhoda nenašla by mal systém danú situáciu oznámiť. Ak by došlo k mnohonásobnej zhode, systém by mal taktiež túto skutočnosť oznámiť.

P 4. Systém bude schopný simulovať najčastejšie používané užívateľské vstupy.

Systém by mal umožňovať manipulovanie s myšou a klávesnicou. Konkrétne interakcie s myšou by mali byť kliknutie pravým alebo ľavým tlačidlom, dvojité kliknutie, stlačenie pravého alebo ľavého tlačidla, presunutie kurzoru, uvoľnenie pravého alebo

ľavého tlačidla. Interakcia s klávesnicou zahŕňa písanie textových reťazcov a zadávanie špeciálnych kláves (funkčné klávesy, modifikátory).

P 5. Užívateľ bude schopný vytvoriť testovacie skripty pomocou vytvoreného jazyka.

Užívateľ bude vedieť zadávať inštrukcie simulujúce bežnú prácu s aplikáciou. Inštrukcie by mali byť také isté ako dané užívateľské vstupy (kliknutie, písanie ..), pomocou jazyka by sa mali dať naplánovať aj určité body v skripte, ktoré uložia aktuálny stav aplikácie a ku ktorým sa bude môcť navracať.

Tieto skripty by sa mali dať vytvárať vo vytvorenom textovom editore, ktorý bude súčasťou testovacieho systému.

P 6. Užívateľ bude môcť pomocou testovacieho systému ovládať testovanú aplikáciu.

V aplikácií pri vytváraní testovacích skriptov, by mal užívateľ mať možnosť pripojiť sa k danému VNC serveru, kde sa nachádza testovaný systém.

P 7. Užívateľ bude môcť pomocou testovacieho systému zachytiť užívateľom definovanú časť obrazovky, v ktorej je spustená testovaná aplikácia, následne sa daná časť obrazovky exportuje ako obrázok vo formáte PNG.

Táto funkcionalita by mala byť dostupná v prípade, že užívateľ vytvára testovacie skripty a je pripojený k aplikácií ako VNC klient, vtedy má možnosť zachytiť určitú oblasť, ktorú chce využiť na testovacie účely.

P 8. Pomocou systému bude možné spúšťať vytvorené testovacie skripty.

P 9. Testovací systém bude musieť vygenerovať testovací report po vykonaní testovacieho skriptu.

Nefunkcionálne požiadavky

- Na hostiteľskom počítači musí byť nainštalovaný 64-bitový operačný systém Linux s nainštalovaným nástrojom Docker.

  Užívateľ, ktorý chce systém používať musí mať práva root alebo musí byť členom používateľskej skupiny docker.

- Na hostiteľskom počítači musí byť nainštalovaný démon libvirtd.

Požiadavky na testovaný systém

- Testovaný systém musí byť 2D okienková aplikácia.

  Testovaný systém by nemal byť grafický editor, editor na 3D modelovanie a CAD, GIS, GPS, počítačové hry, multimediálny program. Testovací systém dokáže vyhľadávanie obdĺžnikových oblastí (napr. ako tlačidlá, logá, textové polia). Testovaný systém by mal byť kancelársky nástroj, jednoduchá webová prezentácia.

- Testovaný systém musí byť spustiteľná na bežných desktopových operačných systémoch, resp. na OS, ktoré je možné spustiť vo virtuálnom počítači.

  Nemali by sa testovať aplikácie určené pre mobilné telefóny.

3.2 Návrh systému

Nasledujúca časť sa venuje popisu realizácie špecifikovaných požiadaviek výsledného systému. Na ich základe sa dá predpokladať, že systém bude pozostávať z dvoch aplikácií. Jednou z nich bude nástroj na vytváranie testovacích sád a druhá bude interpret, ktorý bude vytvorené testovacie sady spúšťať a generovať výsledky. Najprv je uvedená štruktúra obidvoch aplikácií. V poslednej časti kapitoly je princíp činnosti, v ktorej je uvedený diagram kolaborácie.

3.2.1 Jazyk pre popis testovacích sád

V tejto sekcii je popis všetkých príkazov, ktoré sa budú môcť využiť pre vytváranie testovacích skriptov (P 5.). Príkazy sú odvodené od metód použitých v module Xpresser, pre generovanie užívateľských udalostí a taktiež overenie existencie danej oblasti. Tieto príkazy sú doplnené o príkaz na vytváranie obrazov spustených domén a o príkaz na návrat k vytvorenému obrazu.

Celkový zoznam vytvorených príkazov:

- click(),
- double_click(),
- right_click(),
- hover(),
- type(),
- find(),
- wait(),
- create_snapshot() a
- revert_to_snapshot().

Prvé štyri uvedené príkazy slúžia na simulovanie užívateľských akcií vykonaných myšou. Parametrom týchto príkazov môže byť názov obrázka, popr. dve celé čísla oddelené čiarkou, určujúce pozíciu `x` a `y` v zdrojovom okne, kde sa má príslušná udalosť vykonať. V prípade, ak je parameter obrázok, tak príslušná pozícia `x` a `y` sa určí pomocou vyhľadania daného objektu v zdrojovom okne. Príkaz `type()` slúži na simulovanie vstupu klávesnice a parameter tohoto príkazu je textový reťazec uvedený v úvodzovkách. Príkazy `find()` a `wait()` slúžia na vyhľadanie objektu a ich parametrom je vyhľadávaný obrázok, ktorý sa má vyhľadať, a pri príkaze `wait()` je možné druhým parameterom určiť čas v sekundách, ako dlho má systém vyčkať na daný objekt. Predposledným príkazom `create_snapshot()` je možné vytvoriť obraz domény a parametrom sa určí názov tohoto obrazu. Posledným príkazom `revert_to_snapshot()` sa systém dokáže vrátiť do predchádzajúceho stavu, ktorý je definovaný pomocou vytvoreného obrazu, parametrom príkazu je názov obrazu, do ktorého sa má systém navrátiť (P 2.). Taktiež je možnosť písať aj riadkové komentáre začínajúce znakom #.

3.2.2 Interpret jazyka

Pred začiatkom návrhu si bolo treba uvedomiť, že daný interpret bude vykonávať celkovú svoju funkcionalitu v uzatvorenom kontajnery z projektu Docker. Stojí to za pripomenutie, že pri vytváraní Docker kontajneru sa spúšťa iba jeden proces. S požiadaviek vychádza, že s interpretom musí byť konkurentne spustený VNC klient (P 1.), ktorý sa stará o pripojenie na obrazovku domény, tak aby mohol interpret z touto obrazovkou ďalej pracovať.

Existuje niekoľko variant, ktoré som mohol zvoliť pri navrhovaní interpretu a jeho:

- Vytvorenie viac kontajnerov, ktoré by boli spustené súbežne.
- Vytvorenie jedného kontajneru, v ktorom by bol skript, ktorý by sa staral o spustenie daných aplikácii.
- Vytvorenie jedného kontajneru, v ktorom by sa samotný interpret sa staral o spúšťanie VNC klienta.

Pri riešení tohoto problému som zvolil tretiu variantu, kedy je možné mať väčšiu kontrolu spustených procesov vrámci daného interpretu a aby som sa vyvaroval väčšej réžii pri spustení viacerých kontejneroch.

Vyššie spomínaný VNC klient potrebuje k svojmu spusteniu obrazovku, v ktorej sa následne vykresľuje. Na vykreslenie okien na fyzický monitor sa v Linuxových OS stará X server. Daný kontajner by nemal mať prístup k fyzickému monitoru, tak na vykreslenie VNC klienta sa použije VNC server, ktorý sa stará o vytvorenie zobrazovacej plochy v medzipamäti počítača.

Interpret som rozčlenil na niekoľko častí, ktoré by mali spĺňať uvedené požiadavky na systém. Každá časť predstavuje jednu triedu v systéme. Na obrázku 3.1 je možné vidieť navrhnutý diagram tried a ďalej uvádzam popis týchto tried.

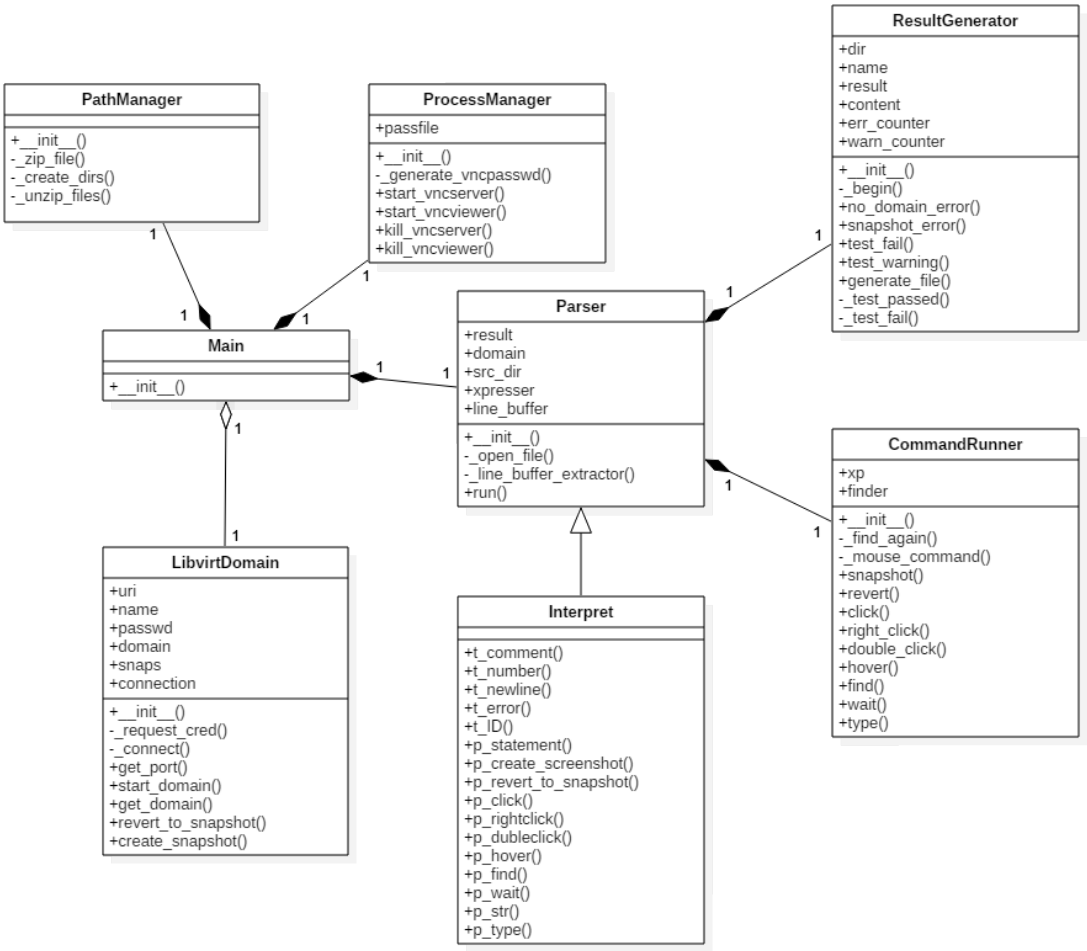
Interpret

Táto trieda by sa mala starať o jednotlivé načítanie tokenov zo vstupného súboru pomocou lexikálneho analyzátora. Taktiež sa musí postarať o syntaktické overenie príkazov. Tieto príkazy sú následne vykonávané a potom sa spracuje príslušný návratový kód. Po vykonaní testovacieho prípadu by mal tento objekt vygenerovať výsledok do súboru(P 8.).

PathManager

Otvára archív z testovacím skriptom, pripravovať štruktúru zložiek pre rozbalenie testovacieho archívu, taktiež v tejto zložke po skončení interpretácií by sa mal vytvárať priestor pre uloženie testovacieho reportu.

Archív obsahuje súbor s testovacím skriptom, ktorý má príponu `.test`. Interpret predpokladá iba jeden takýto súbor v jednom archíve, avšak uvažujem ako budúce možné rozšírenie existenciu viacerých možných súborov, čo by viedlo k vytváraniu komplexnejších testovacích sád. Pri tomto súbore sa ďalej nachádza konfiguračný súbor, ktorý nesie údaje potrebné pre pripojenie na VNC server a démona libvirt. Pri VNC serveri sú to informácie o IP adrese a porte, poprípade aj hesle, ak sa jedná o zabezpečený server. Taktiež je tu informácia o rozlíšení daného VNC serveru. K pripojeniu na libvirt démon obsahuje súbor informácie o autentifikačných údajoch a je tu aj URI konkrétneho hypervízoru. V archíve sa nachádza aj zložka, v ktorej sa môžu nachádzať obrázky vo formáte PNG na vyhľadávanie objektov, ktoré obrázky reprezentujú. Pri týchto obrázkoch sa môže nachádzať konfiguračný súbor určený pre modul `xpresser.ini`



Obr. 3.1: Navrhnutý diagram tried.

ProcessManager

Stará o vytvorenie procesov VNC serveru a VNC klienta pred začatím interpretácie testovacej sady(P 1.). Následne po jej skončení by mal dané procesy ukončiť. Pri vytváraní VNC serveru tento objekt využíva informáciu o rozlíšení z konfiguračného súboru popísaného vyššie. VNC klient sa spúšťa v celoobrazovom režime a využíva informácie o VNC serveri z konfiguračného súboru.

LibvirtDomain

Táto trieda zabezpečuje pripojenie na daný hypervízor a prácu z konkrétnou doménou, v ktorej sa nachádza testovaný systém. Táto práca zahŕňa spúšťanie danej domény, taktiež vytváranie obrazov domény a návrat k vytvoreným obrazom domény (P 2.).

CommandRunner

Vykonáva konkrétne príkazy tým, že volá metódy z Xpresser modulu (P 3., P 4.), popr. z vytvoreného modulu LibvirtDomain, ktoré už priamo generujú užívateľské akcie. Taktiež odchytáva prípadné vygenerované výnimky z modulu Xpresser, ktoré môžu byť spôsobené chybovým správaním testovanej aplikácie. Tieto výnimky sa generujú iba pri rozpoznávaní obrazu. Tu je zoznam stavov, na ktoré interpret upozorní a zároveň je popísaná reakcia, ako na daný stav reaguje interpret:

- Upozornenie o nájdení objektu, ale s nižším prahom ako je definovaný

  Tento stav nastane, ak sa objekt najprv nenájde zo zadaným prahom, ktorý bol definovaný v súbore `xpresser.ini`. V tom prípade sa interpret pokúsi o nájdenie nového maximálneho prahu, ktorý je už nižší od pôvodného a vykoná daný príkaz znova. V prípade, že interpret klikol niekde, kam pôvodne nemal, tak je dosť pravdepodobné, že sa pri nasledujúcom príkaze dostane do chybového stavu.

- Chyba, prípade nenájdenia vyhľadávanej oblasti

  V tomto prípade interpret nenašiel objekt s požadovaným prahom a nenašiel ani nový nižší prah, vtedy nemôže vykonať daný príkaz a tak interpret vygeneruje testovaciu správu a končí predčasne.

- Upozornenie o mnohopočetnom výskyte vyhľadávaného objektu

  V tomto prípade interpret našiel viac objektov z rovnakým prahom, tak vykoná operáciu na prvý z nich, ktorý našiel.

- Upozornenie o nájdení objektu s nižším prahom a zároveň s mnohopočetným výskytom.

  Tento stav je iba kombináciou dvoch vyššie uvedených stavov.

ResultGenerator

Vytvára štrukturovaný súbor, v ktorom bude výsledok testovania(P 9.). Výsledok testovania je vo forme HTML súboru a obsahuje informácie o defektoch, ktoré nastali pri rozpoznávaní objektov v testovanej aplikácii. Tieto defekty, ktoré môže zaznamenať do súboru, sú popísané vyššie (3.2.2) a o každom z nich značí tieto informácie:

- Názov stavu, ktorý nastal
- Príkaz, pri ktorom chybový stav nastal
- Obrázok hľadanej oblasti
- Obrázok zachytenej obrazovky domény, v prípade ak ide iba o upozornenie, tak v tomto obrázku je zaznačená oblasť, ktorá sa našla ako zhoda k hľadanému objektu, popr. všetky nájdené oblasti, ak sa jedná o viac násobnú zhodu.

Main

Hlavná trieda, ktorá sa stará o vytváranie ďalších objektov a volanie príslušných metód.

3.2.3 Nástroj na vytváranie testovacích sád

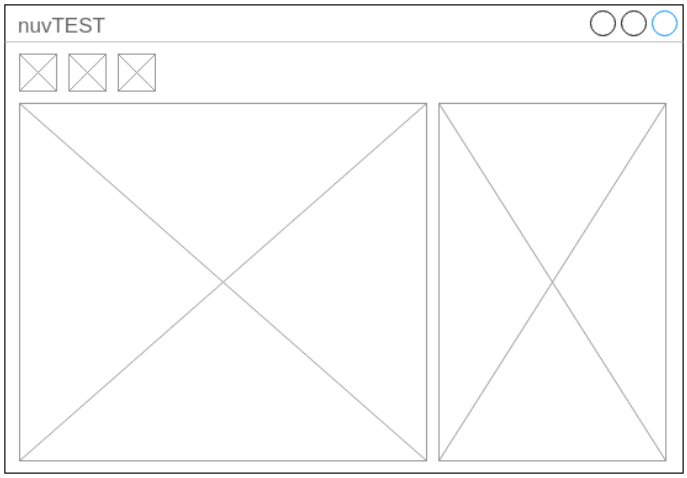
Tento nástroj by mal užívateľovi poskytnúť podobnú funkcionalitu ako *Sikuli IDE*, popísaný v druhej kapitole, takže sa bude jednať o aplikáciu s grafickým užívateľským rozhraním. V prvom rade by mala obsahovať textový editor na vytváranie testovacích sád a taktiež užívateľ by mal mať prehľad o možných príkazoch, ktoré je možné aplikovať. Aplikácia by ďalej mala poskytovať služby VNC klienta, aby užívateľ mohol ovládať danú testovanú aplikáciu a zároveň mohol zachytávať oblasti, ktoré budú súčasťou testovacej sady. Užívateľ by mal mať taktiež prehľad o všetkých zachytených oblastiach.

Tieto hlavné funkcie aplikácie som použil na vytvorenie jednotlivých tried, ktoré sú súčasťou architektonického návrhu:

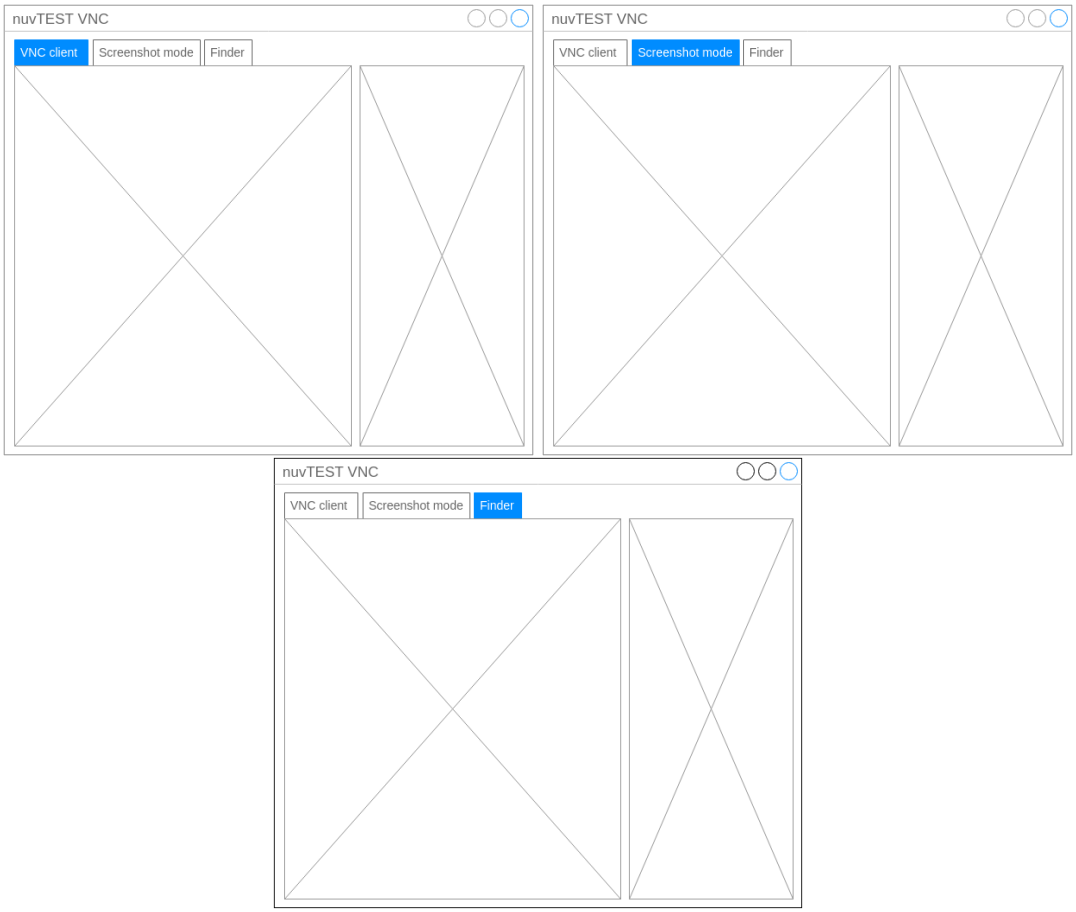
- MainWindow - je to hlavné okno aplikácie a vytvára sa po bezprostrednom spustení aplikácie. Jeho návrh je možné vidieť na obrázku 3.2.
- VNCWindow - okno sa vytvára iba na podnet užívateľa, keď chce využívať služby VNC klienta. Návrh tohoto okna je na obrázku 3.3. Okrem VNC klienta, je možné aj zachytávať objekty a je možné tieto objekty následne vyhľadať a overiť si nájdený prah.
- VNCConnection - je objekt, ktorý vzniká pri pripojení na VNC server.
- PicsList - štruktúra kde sa nachádzajú všetky potrebné informácie o vytvorených objektoch. Táto štruktúra je základom pre zobrazovací prvok, ktorý je v pravej časti na obrázkoch 3.2 a 3.3.

Následne jednotlivé časti architektonického návrhu som využil na vytvorenie prototypu GUI, ktorý prinášal iba jednoduchú funkčnosť, ale daný prototyp mi ďalej pomohol pri vytvorení štrukturovaného návrhu. V tomto návrhu pôvodné časti z architektonického návrhu som vložil do jedného modulu s názvom `nuvtest.py`. V ďalšom module `libvirtconnection.py` sa nachádza trieda `Libvirt`, ktorá sa stará o pripojenie na démona libvirt a je takmer totožná s triedou `LibvirtDomain`, avšak v tomto nástroji by nemala byť funkcionalita pre vytváranie obrazov domén. Modul `finder.py` by mal poskytovať informácie o nájdených objektoch. V tomto module je prebratá trieda `OpenCVFinder` z modulu *Xpresser*. V poslednom module `dialogs.py` sú triedy, ktoré predstavujú jednotlivé dialógové okná aplikácie.

Výstupom nástoja je archív v podobe, ako je popísaný v kapitole 3.2.2.



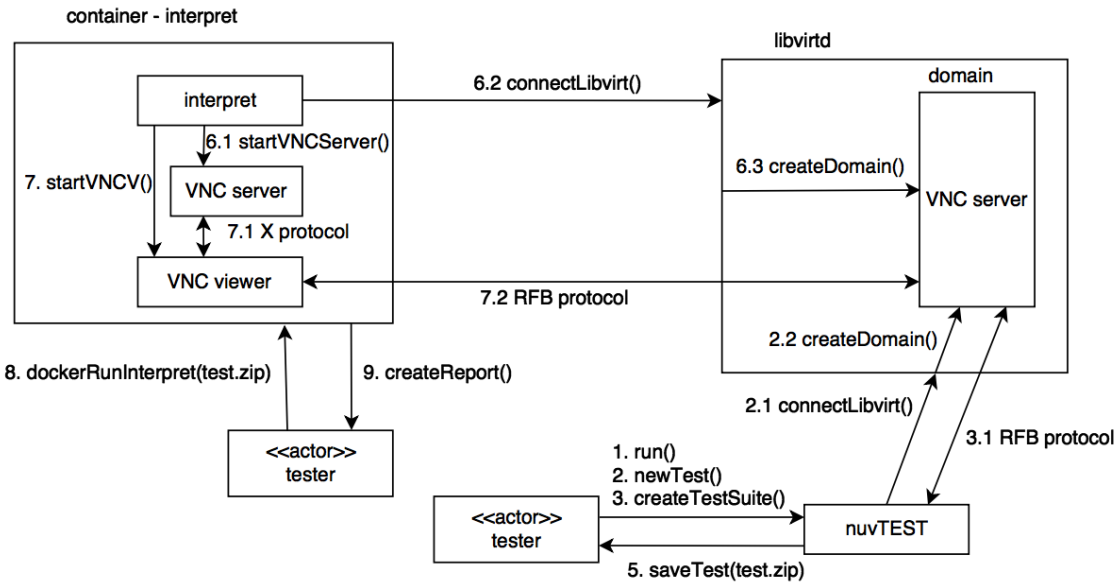
Obr. 3.2: Návrh hlavného okna aplikácie. V hornom paneli sú tlačítka na vytváranie nové skriptu, ukladanie a otváranie VNC klietna. V jeho ľavej časti má užívateľ priestor pre písanie skriptov a v pravej časti má prehľad o vytvorených objektoch.



Obr. 3.3: Návrh VNC klienta a jeho ďalších funkcií: nástroj na vytváranie objektov a nástroj na vyhľadávanie objektov.

3.2.4 Princíp činnosti systému

Táto časť popisuje činnosť systému. Na začiatok je uvedený UML diagram kolaborácie 3.4, ktorý zobrazuje interakcie medzi účastníkmi systému, teda testermi, ktorý do neho vstupujú a danými časťami systému. Táto interakcia je zobrazená ako sekvencia udalostí, ktoré sú náležite očíslované. Užívateľ pre vytvorenie testovacieho skriptu musí použiť nástroj na ich tvorbu (skr. nuvTEST). Pri vytváraní nového skriptu musí užívateľ definovať na akú doménu sa chce pripojiť. Nástroj následne komunikuje s VNC severom pomocou RFB protokolu[1], ktorý sa využíva pri VNC technológií. Keď užívateľ vytvorí testovací skript, môže ho vyexportovať, kedy sa vygeneruje archív. Tento archív môže užívateľ poskytnúť interpretu, ktorý je uzatvorený v kontajneri, ten podobne ako nuvTEST sa pripojí na existujúcu doménu a spustený VNC klient v kontajneri komunikuje s VNC serverom pomocou RFB protokolu. VNC klient sa vykresľuje na spustený VNC server pomocou X protokolu a interpret zasiela užívateľské udalosti na tento VNC server, ktoré sa následne interpretujú na VNC server testovaného systému, ktorý je v doméne. Po dokončení interpretácie sa vygeneruje testovací report.



Obr. 3.4: Diagram kolaborácie

Nástroj na vytváranie testovacích sád

Pomocou tohoto nástroja sa musí užívateľ pripojiť na existujúci VNC server podľa IP adresy a portu, popr. pripojiť sa na konkrétny hypervízor pomocou URI a vybrať doménu, ktorú chce využiť na testovacie účely. Nástroj sa následne sám pripojí na VNC server, ktorý musí mať doména špecifikovaný vo svojom XML popise. Ak sa nástroj úspešne pripojí na VNC server, tak užívateľ bude mať k dispozícií okno na písanie testovacích sád. Po stlačení pravým tlačítkom v tomto okne sa mu zobrazí ponuka možných príkazov, ktoré sú popísané v kapitole 3.2.1.

V prípade potreby si užívateľ môže otvoriť okno s VNC klientom. V tomto okne sa nachádza aj nástroj na zachytávanie oblastí. Pomocou menu treba najrpv vytvoriť screenshot

[1] http://www.realvnc.com/docs/rfbproto.pdf

aktuálnej obrazovky VNC klienta a v tomto screenshote môže užívateľ pomocou stlačenia ľavého tlačítka vybrať oblasť ktorú chce zachytiť. Takýmto spôsobom užívateľ vytvorí objekty, ktoré bude chcieť v danej aplikácii otestovať. Pri vytváraní objektov im užívateľ zadá jedinečný identifikátor. Týmto objektom následne užívateľ môže nastaviť prah s akým ich má interpret vyhľadávať. Prah je automaticky nastavený na najvyšší možný v rozmedzí od 0.0 až po 1.0. Ak užívateľ nie je s prahom spokojný, tak môže vybraný objekt vymazať a vytvoriť nový.

Po napísaní testovej sady ju užívateľ uloží ako *ZIP* archív, ktorý poskytne interpretu.

Interpret

Užívateľ pri spustení kontajneru s interpretom musí špecifikovať testovací archív a zložku, do ktorej sa má vygenerovať testovací report. Po spustení interpretu sa vytvorí priečinok na rozbalenie testovacieho archívu. Tento priečinok je pomenovaný ako `archiv_denMesiacRok_hod:min:sek` a do tohoto priečinka sa vytvorí ďalší s názvom `source`, do ktorého sa rozbalia už súbory z archívu. Keď má interpret prístup ku konfiguračnému súboru, ktorý je popísaný v kapitole 3.2.2, tak môže vytvoriť spojenie na príslušný hypervízor pomocou libvirt démonu, ktorý je nainštalovaný na hostiteľskom počítači. Po tomto pripojení sa vytvorí proces s VNC serverom, ktorého jediný argument je rozlíšenie z konfiguračného súboru. Následne sa môže spustiť proces VNC klienta, ktorý sa pripojí na VNC server v spustenej doméne. V tomto momente sa začína parsovanie testovacieho skriptu a vykonávanie samotných príkazov z modulu *Xpresser*, popr. volanie metód na správu obrazov domény. Test končí neúspešne v prípade, že modul *Xpresser* nenájde požadovanú oblasť. Test končí úspešne v prípade vykonania všetkých príkazov. Užívateľ musí skontrolovať testovací report, v ktorom môžu byť ohlásené prípadne defekty, ktoré sú bližšie popísané v kapitole 3.2.2.

Kapitola 4

Implementačné detaily testovacieho systému

V tejto kapitole sa nachádzajú niektoré detaily implementácie systému, taktiež sú tu popísané niektoré knižnice použité pri implementácii. Pre implementáciu všetkých častí projektu bol využitý jazyk Python vo verzii 2.7, hlavne kvôli faktu, že v tomto jazyku je napísaný taktiež modul Xpresser.

Na začiatku je uvedený popis interpret, ktorý je vložený do kontajneru vďaka systému Docker. Pomocou tohoto kontajneru sa interpret zo všetkými závislosťami chová ako jeden celok a nie je nutné inštalovať žiadne dodatočné knižnice na hostiteľský počítač. Problémom je vyhľadávajúca funkcia modulu Xpresser, kedy niekedy nepresne označí vyhľadávaný objekt.

4.1 Jadro systému

V tejto časti je bližšie popísaný vytvorený kontajner a taktiež potrebné náležitosti pre fungovanie interpretácie. Súbor *Dockerfile*, ktorý slúži na vytvorenie Docker obrazu, je uvedený v prílohe B.

Najprv je nutné vytvoriť testovací archív. Následné spustenie kontajneru musí vyzerať takto: `docker run -it -v /zlozka/a.zip:/test/a.zip -v /zlozka/:/result interpret`. Zložky `test` a `result` musia byť v zápise príkazu zachované, keďže po vykonaní interpretácie bude v priečinku `zlozka` vytvorený ďalší priečinok, v ktorom sa budú nachádzať rozbalené súbory z archívu a taktiež vygenerovaný testovací report.

Pre správne fungovanie interpretu je nutné aby démon *libvirt* na hostiteľskom počítači bol správne nastavený pre používanie cez sieť [14], tak aby sa kontajner na neho dokázal pripojiť. Doména, ktorá sa využíva ako testovaný systém musí mať vo svojom XML popise definovaný VNC server ako zobrazovacie zariadenie a IP adresa tohoto serveru musí byť rovnaká so sieťovým rozhraním démona Docker. Interpret pre pripojenie na VNC server domény využíva klient vncviewer[1]. Tento klient sa následne zobrazí vo VNC serveri[2], ktorý sa spúšťa s interpretom. Pre je ho spustenie je nutné mať definované heslo v zložke */home/interpret/.vnc/passwd*. Toto heslo sa generuje pri každom spustení kontajneru a vypisuje sa na štandardný výstup a užívateľ je schopný sa pripojiť na tento VNC server a sledovať vykonávanú interpretáciu.

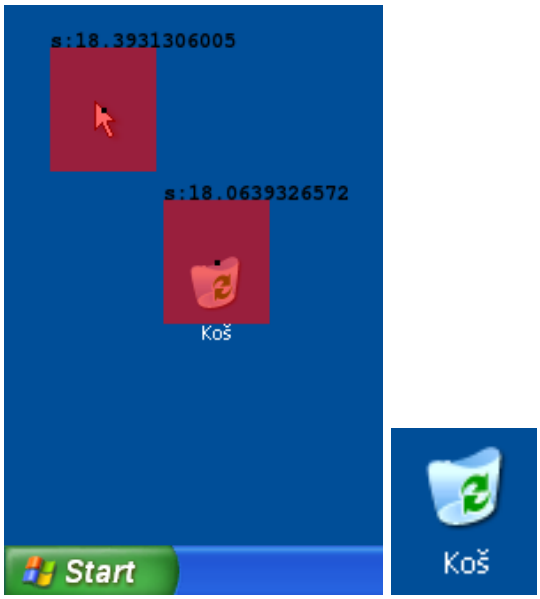
[1] https://www.realvnc.com/products/open/4.1/man/vncviewer.html
[2] https://www.realvnc.com/products/open/4.1/man/vncserver.html

Pre návrat testovaného systému do predchádzajúcich stavov interpret využíva volania z knižnice *libvirt*, keďže testovaný systém musí byť vo virtuálnom počítači. Pre vytvorenie obrazu domény sa využíva metóda `snapshotCreateXML()`, ktorej parametrom je krátky text vo formáte XML, ktorého koreňovou značkou je `<domainsnapshot>` a synovskými značkami `<name>`, ktorá definuje meno zachyteného obrazu a značka `<memory>`, ktorá popisuje či sa má uložiť stav operačnej pamäte virtuálneho počítača [15]. Po vytvorení obrazu domény, je možný návrat k tomuto obrazu. Najprv je však nutné pomocou metódy `snapshotLookupByName()` tento obraz vyhľadať na základe mena, ktoré je jedinečné. Následne pomocou metódy `revertToSnapshot()`, ktorej parametrom je vyhľadaný obraz, sa systém dostane do pôvodného stavu.

Na implementáciu samotnej triedy *Interpret*, ktorá vykonáva príkazy testovacích skriptov, bola použitá knižnica *PLY*, ktorá je implementáciou nástrojov lex a yacc pre jazyk Python.

4.2 Problém vyhľadávania objektov

Už od začiatku práce s Xpresser modulom som si všimol fakt, že proces vyhľadávania objektov je často omylný. Chybovosť sa dá znížiť napríklad tým, že keď chceme vyhľadať napr. tlačítko, tak pri vytváraní obrázku tohoto tlačítka zachytíme aj časť jeho okolia (časť iného tlačítka, časť textu, ..), čo pomáha pri zvyšovaní presnosti vyhľadávania. Ale aj zachytenie správnej mieri okolia si vyžadovalo niekoľko iterácií udalostí, kedy som musel zachytiť objekt a overiť jeho nájdený prah.



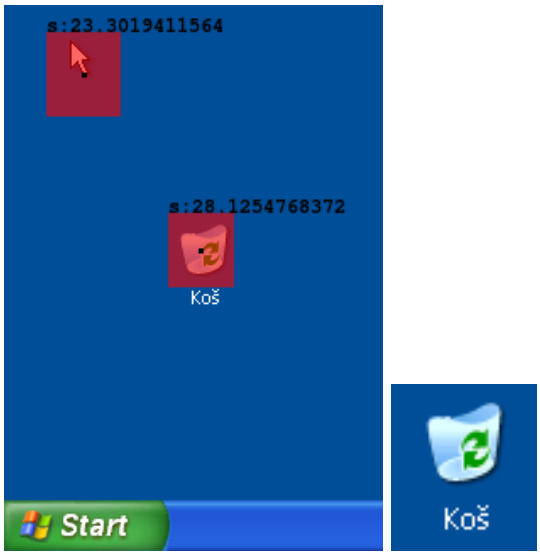
Obr. 4.1: Ukážka nepresnosť vyhľadávacej metódy. Ľavý obrázok znázorňuje nájdené zhody a vpravo sa nachádza vyhľadávaný objekt.

Väčšia nepresnosť sa objavovala hlavne pri prehľadávaných zdrojových objektoch, v ktorých sa vyskytovalo minimum hrán, napr. ako na obrázku 4.1, kde som sa snažil o vyhľadanie ikony koša na ploche operačného systému Windows XP, ktorého pozadie bola iba modrá farba. Daný objekt bol vyhľadaný, ale s menšou presnosťou ako kurzor a to bol dosť závažný problém (kurzor sa našiel s presnosťou 18.39%, ikona koša sa našla s presnosťou 18.06%).

V tomto prípade stačilo zachytiť menšiu časť okolia koša, ale to by prinieslo znova ďalšie vynaložené úsilie.

```
1  def find(self, screen_image, area_image):
2      matches = self.find_scaled_image(screen_image, area_image)
3      matches = filter(lambda x:
4                  x.similarity == matches[0].similarity), matches)
5
6      if len(matches) == 1:
7              return matches[0]
8      elif len(matches) > 1:
9          raise MultipleMatchWarn(area_image, screen_image, matches)
10
11     return None
```

Obr. 4.2: Metóda `find` po aplikovaní zmeny.



Obr. 4.3: Po aplikovaní zmeny v module Xpresser. Ľavý obrázok znázorňuje nájdené zhody a vpravo sa nachádza vyhľadávaný objekt.

Úpravou v module *Xpresser* je zmena metódy `find()` v triede `OpenCVFinder`. Táto metóda sa volá v prípade vyhľadávnia objektu v zdrojovom okne. Práve vstupom metódy je hľadaný objekt a screenshot obrazovky a výstupom je nájdená jedna zhoda, ktorá je reprezentovaná objektom `ImageMatch`. Metóda volá ďalšiu metódu `_find()`, ktorá slúži na vyhľadávanie (za použitia metódy findTemlate()) z modulu SimpleCV) a vracia list nájdených objektov. Zmenu som vykonal tak, že do pôvodnej triedy som vložil metódu `find_scaled_image()`, ktorú volám z metódy `find()` a nová metóda volá metódu `_find()`. Nová metóda slúži na opakované vyhľadávanie daného objektu. Najprv sa vyhľadá objekt v pôvodnej veľkosti, výsledky hľadania sa uložia a následne sa vykoná hľadanie objektu, ale už s orezanou plochou. Opakované hľadanie sa vykonáva tri krát a to s orezanou plochou na 95%, 80% a 70% ku pôvodnému objektu. Ako je možné vidieť na obrázku 4.3, tak došlo

k miernemu spresneniu vyhľadávacieho procesu, avšak za cenu pomalšieho spracovania. Za zmienku stojí aj fakt, že triedu `OpenCVFinder` využívam aj pri nástroji na vytváranie testovacích sád a aj pri interprete.

V prípade mnohonásobného výskytu nájdeného objektu, pôvodná metóda `find()` vracala iba prvý prvok zo zoznamu nájdených objektov čo predstavovalo jedinú zhodu, čo pre testovacie účely nebolo vhodné. Metódu som upravil tak, aby z vyhľadaných objektov vytvorila nový list, do ktorého uloží objekt z najlepším prahom a ku nemu vyfiltruje ďalšie objekty, ktoré majú zhodný prah. V prípade nájdenia viacerých objektov z rovnakým prahom metóda vyhodí výnimku *MultipleMatchWarn*, ktorá sa spracuje ako upozornenie v interprete.

4.3 Vytvorenie testovacieho reportu

```
1  # Test report #
2  ## Test: Example ##
3  Result: PASSED
4  * * *
5
6  WARNING at :
7      1: create_snapshot(test2)
8      2: find(btn_name2)
9      3: type("test␣string")
10     >>> 4: find(btn_name3)
11
12 This image was not found with similarity ‘0.98‘,
13 but was found with similarity ‘0.65‘:
14 ![Warn image](img/1_btn_name3_warn_template.png)
15 in this screen:
16 ![Warn screenshot](img/1_btn_name3_warn_source.png)
17 * * *
```

Obr. 4.4: Na prvých dvoch riadkoch je text ohraničený znakom `#`, resp. `##`. Po vytvorení HTML dokumentu tieto riadky budú prevedené na nadpis prvej úrovne, resp. nadpis druhej úrovne. Na štvrtom riadku sú znaky `* * *`, ktoré znamenajú horizontálnu čiaru. Odsadený text na riadkoch 7 až 10 znamená blok kódu a v HTML sú tieto riadky ohraničené značkami `<pre><code>` a `</code></pre>`. Poslednú značku, ktorú je vidieť v ukáže, je `![Warn image](img/1_btn_name3_warn_template.png)`, ktorá v HTML dokumente sa prevedie na značku `<img>` s obrázkom uloženým v zložke *img*, s názvom *1_btn_name3_warn_template* a popisom *Warn image*.

Výstupom interpretu je testovací report. Tento report je súbor vo formáte HTML, avšak ku vytváraniu tohoto súboru nepoužívam priamo značky HTML. Pri interpretácií sa vytvára textový reťazec pričom využívam syntax *MarkDown*. MarkDown [5] je syntax navrhnutá na vytváranie ľahko čitateľných formátovaných textov. Názov MarkDown taktiež označuje nástroj, ktorý formátovaný text pomocou syntaxe MarkDown prevedie na HTML

dokument. V interprete využívam na túto činnosť modul s názvom *markdown*[3]. Pomocou volania `markdown.markdown(self.content)` vygenerujem telo HTML dokumentu a následne pri ukladaní pripojím vlastnú hlavičku s definovaným štýlom a pätu dokumentu. Ukážka jednoduchého vygenerovaného testovacieho reportu v syntaxi MarkDown je ukázaný a popísaný v 4.4.

4.4 Nástroj umožňujúci vytváranie testovacích sád

Pre vytvorenie nuvTEST bolo nutné si zvoliť framework na vytváranie GUI, keďže existuje mnoho možností[4], tak v prvých fázach riešenia projektu som sa snažil nájsť čo najjednoduchší framework, ktorý by poskytoval potrebnú funkcionalitu, kvôli snahe uložiť daný nástroj do kontajneru, ktorý by bol hardvérovo nenáročný, ale nakoniec som sa rozhodol pre framework Gtk 3, kvôli existencii balíku gtk-vnc[5], vďaka ktorému som implementoval VNC klient. Gtk je multiplatformová sadá nástrojov napísaná v jazyku C a pôvodne bola vytvorená na bitmapový editor GIMP. GTK má veľmi dobré spracovanú dokumentáciu a aj návod na použitie v jazyku Python, ktorý som využíval hlavne na začiatku vývoja aplikácie.

Na vytvorenie samotného GUI aplikácie som využil nástroj *Glade*[6], ktorý vygeneruje XML súbor, v ktorom sú popísané všetky prvky užívateľského rozhrania. V programe následne využívam objekt `GtkBuilder`[7], ktorý načítava vytvorené XML súbory a slúži na jednoduché pristupovanie k vytvoreným prvkom UI. Výsledný systém je možné vidieť v prílohe C.

[3] https://pypi.python.org/pypi/Markdown
[4] https://docs.python.org/2/faq/gui.html#gtk
[5] https://lazka.github.io/pgi-docs/GtkVnc-2.0/index.html
[6] https://glade.gnome.org/
[7] https://developer.gnome.org/gtk3/stable/GtkBuilder.html

Kapitola 5

Zhodnotenie testovacieho systému

V tejto kapitole je zhodnotenie implementovaného testovacieho systému. Pomocou testovacieho systému som navrhol a otestoval GUI nástroja umožňujúceho vytváranie testovacích sád. Pre potreby testovania som vytvoril virtuálny počítač s OS Ubuntu, kde som umiestnil testovaný subjekt.

Pomocou nuvTEST som najprv vytvoril testovací skript (v prílohe D a taktiež je na priloženom CD A), tzn. pripojil som sa na VNC server vytvorenej domény, zachytil som potrebné objekty ako obrázky a napísal som k nim testovací skript, ktorý by mal overiť funkcionalitu testovaného subjektu a následne som tento skript exportoval ako ZIP archív. Testovací archív som predal interpretu ako vstup a po vykonaní testovacieho skriptu bol výsledný testovací report prázdny, tzn. bez nájdenej chyby v testovanom subjekte.

Požiadavky na systém boli zväčša splnené, keďže sa mi úspešne podarilo vytvoriť skript aj s potrebnými náležitosťami a následne tento skript bol úspešne interpretovaný. Avšak slabinou stále zostáva vyhľadávajúca schopnosť systému (P 3.), síce sa pri spustení skriptu žiadne vady nevyskytli, tak tento problém stále pretrváva. Taktiež nebolo možné systémom otestovať požiadavku P 7., keďže systém neumožňuje užívateľskú akciu *drag and drop*.

Kapitola 6

Záver

V rámci tejto práce som najprv naštudoval informácie ohľadom kontajnerovej virtualizácie a testovaní GUI, kde som sa aj zoznámil s niektorými nástrojmi. Špecifikoval som požiadavky pre testovací systém a následne som tento systém navrhol. Systém sa skladá z troch častí: interpret testovacích skriptov, jazyk pre popis testovacích skriptov a nástroj umožňujúci vytváranie testovacích skriptov. Interpret som implementoval pomocou kontajnerovej virtualizácie, čo zabezpečilo možnosť jednoduchého nasadenia. Taktiež som implementoval nástroj na vytváranie testovacích skriptov, ktorý mi umožnil zrýchlenie tvorby týchto skriptov. Nakoniec som celú funkčnosť systému overil pri testovaní GUI nástroja, ktorý bol v práci navrhnutý.

6.1 Možnosti budúceho vývoja systému

V budúcnosti by som chcel dospieť k zlepšeniu rozpoznávacej schopnosti systému, ideálne do takej miery, aby sa systém blížil k nulovej chybovosti. V predchádzajúcej kapitole som opísal implementáciu jednoduchej úpravy, ktorá priniesla mierne zlepšenie, avšak nedostačujúce pre použitie v praxi. Najpresnejší výsledok by prinieslo porovnávanie, kedy by sa hľadaný objekt posúval pixel po pixeli v zdrojovom okne a na každej tejto pozícii by sa porovnali všetky pixely hľadaného objektu s pixelmi plochy zdrojového okna, ktorú by hľadaný objekt prekrýval. Takýto algoritmus by priniesol vysokú časovú zložitosť, avšak jeho použitie si viem predstaviť, v prípade, že by som obmedzil vyhľadávaciu plochu, napr. pomocou pôvodného algoritmu. Na obrázku 4.1 kde je vidieť chybne nájdený objekt, po aplikovaní metódy porovnávania pixel po pixeli na nájdené oblasti, ktoré sú vyznačené červenou farbou, by algoritmus hľadaný objekt z určitosťou našiel.

Pri vytváraní testovacích sád mi chýbala možnosť vykonávať niektoré užívateľské akcie pomocou klávesových skratiek (napr. prepínanie medzi oknami vo vytvorenom VNC kliente), čo by prinieslo minimálne zrýchlenie tvorby skriptov.

Systém by sa mohol ďalej rozšíriť a používať aj na statické overenie GUI, podobne ako produkt Applitools Eyes, ktorý je vhodný na regresné testovanie.

Literatúra

[1] Campbell, S.; Jeronimo, M.: An Introduction to Virtualization. online, 2006 [cit. 2016-05-05].
URL https://software.intel.com/sites/default/files/m/d/4/1/d/8/An_Introduction_to_Virtualization.pdf

[2] Clift, J.; Blake, E.; A. Dadhania, N.; aj.: Libvirt FAQ. online, [cit. 2016-05-05].
URL http://libvirt.org/sources/virshcmdref/html/

[3] Corbet, J.: Process containers. online, 29.5.2007 [cit. 2016-05-05].
URL http://lwn.net/Articles/236038/

[4] Gagnon, C.: Xpresser. online, 22.12.2012 [cit. 2016-05-05].
URL https://wiki.ubuntu.com/Xpresser/

[5] Gruber, J.: Markdown. online, [cit. 2016-05-16].
URL https://daringfireball.net/projects/markdown/

[6] Hykes, S.: Docker 0.9: introducing execution drivers and libcontainer. online, 10.3.2014 [cit. 2016-05-05].
URL https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/

[7] Jones, M. T.: Virtualization. online, 25.5.2010 [cit. 2016-05-05].
URL http://www.datamation.com/features/article.php/3884091/Virtualization.htm

[8] A. M. Memon, M. E. Pollack, M. L. Soffa: Using a Goal-driven Approach to Generate Test Cases for GUIs. *ICSE '99 Proceedings of the 21st international conference on Software engineering*, 1999: s. 257–266.

[9] Applitools Team: Applitools Features. online, [cit. 2016-05-05].
URL https://applitools.com/features/

[10] Docker Team: Understand the architecture. online, [cit. 2016-05-05].
URL https://docs.docker.com/engine/understanding-docker/

[11] Gentoo Team: Linux Containers. online.
URL https://wiki.gentoo.org/wiki/LXC

[12] Libvirt Team: libvirt API support matrix. online, [cit. 2016-05-05].
URL http://libvirt.org/hvsupport.html

[13] Libvirt Team: Libvirt FAQ. online, [cit. 2016-05-05].
URL http://wiki.libvirt.org/page/FAQ

[14] Libvirt Team: Remote support. online, [cit. 2016-05-12].
URL https://libvirt.org/remote.html

[15] Libvirt Team: Snapshot XML format. online, [cit. 2016-05-16].
URL https://libvirt.org/formatsnapshot.html

[16] Sikuli Team: How Sikuli Works. online, [cit. 2016-05-05].
URL http://doc.sikuli.org/devs/system-design.html

[17] Sikuli Team: SikuliX - the basics. online, [cit. 2016-05-05].
URL http://sikulix-2014.readthedocs.io/en/latest/basicinfo.html

[18] Niemeyer, G.: Xpresser – Python library for GUI automation with image matching. online, 18.05.2010 [cit. 2016-05-05].
URL http://blog.labix.org/2010/05/18/xpresser-python-library-for-gui-automation-with-image-matching/

[19] Ponticello, M.: Docker Community Passes Two Billion Pulls! online, 10.02.2016 [cit. 2016-05-05].
URL https://blog.docker.com/2016/02/docker-hub-two-billion-pulls/

[20] Riondato, M.: Chapter 14. Jails. online, 28.1.2016 [cit. 2016-05-05].
URL http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails.html

[21] Smrčka, A.: *ITS - Testování aplikací využívající síť, testování grafického uživatelského rozhraní*. Fakulta informačních technologií. Vysoké učení technické v Brně., 2016-03-17.

[22] Tholeti, B. P.: Hypervisors, virtualization, and the cloud. online, 23.9.2011 [cit. 2016-05-05].
URL
https://www.ibm.com/developerworks/cloud/library/cl-hypervisorcompare/

Prílohy

Zoznam príloh

Príloha A

Obsah priloženého CD

V tejto prílohe je uvedený popis a obsah adresárovej štruktúry priloženého nosiča:

- **src-interpret** - zložka obsahuje zdrojový súbor implementovaného interpretu a taktiež súbor Dockerfile.
- **src-nuvtest** - zložka obsahuje zdrojové súbory nástroja umožňujúceho vytváranie testovacích skriptov.
- **example** - zložka obsahuje rozbalený testovací archív s testovacím reportom.
- **text** - v zložke sa nachádza zdrojový kód tejto práce.

Príloha B

Súbor Dockerfile na vytvorenie obrazu s interpretom

Táto príloha obsahuje Dockerfile, pomocou ktorého je možné vytvoriť kontajner, ktorý slúži ako interpret testovacích sád.

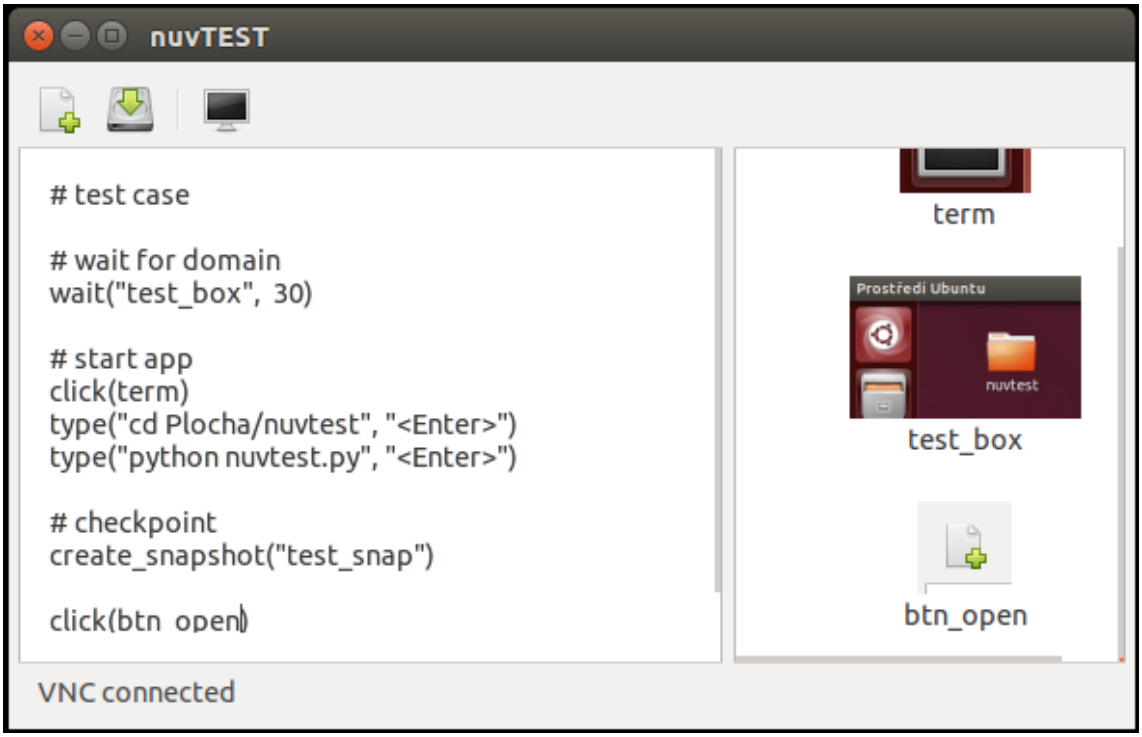
```
FROM ubuntu:12.04
MAINTAINER Sojcak Juraj <xsojca00@stud.fit.vutbr.cz>

RUN apt-get update

# get dependencies
RUN apt-get -f install -y \
        python-opencv \
        python-numpy \
        python-pyatspi \
        python-pyatspi2 \
        python-cairo \
        python-gi-cairo \
        python-pygame \
        python-scipy \
        python-setuptools \
        gir1.2-gdl-3 \
        python-dbus \
        tightvncserver \
        wmii \
        wget \
        python-pip \
        python-libvirt \
        dbus-x11 \
        vncviewer
# clean it
RUN apt-get clean \
        && rm -rf /var/lib/apt/lists/*
# another dependencies
RUN pip install svgwrite markdown ply \
```

```
    && https://github.com/sightmachine/SimpleCV/zipball/develop

WORKDIR /tmp/x

# download edited xpresser
RUN wget http://www.stud.fit.vutbr.cz/~xsojca00/IBP/ex.tar.gz \
        && tar -xf ex.tar.gz \
        && cd ex \
        && python setup.py install

WORKDIR /

RUN rm -rf /tmp/x

RUN useradd -ms /bin/bash interpret

USER root
ENV USER root

# create volumes
VOLUME ["/test"]
VOLUME ["/result"]


WORKDIR /home/interpret/

# download interpret
RUN wget http://www.stud.fit.vutbr.cz/~xsojca00/IBP/int.tar.gz \
        && tar -xf int.tar.gz \
        && rm int.tar.gz

WORKDIR /

# generate xauth file
RUN HOST=`hostname` && key=`perl -e 'srand; printf \
    int(rand(100000000000000000))'` && key=$key$key \
    && xauth add ${HOST}/unix:0 . $key

# set user
ENV USER interpret
USER interpret

CMD python /home/interpret/pl.py
```

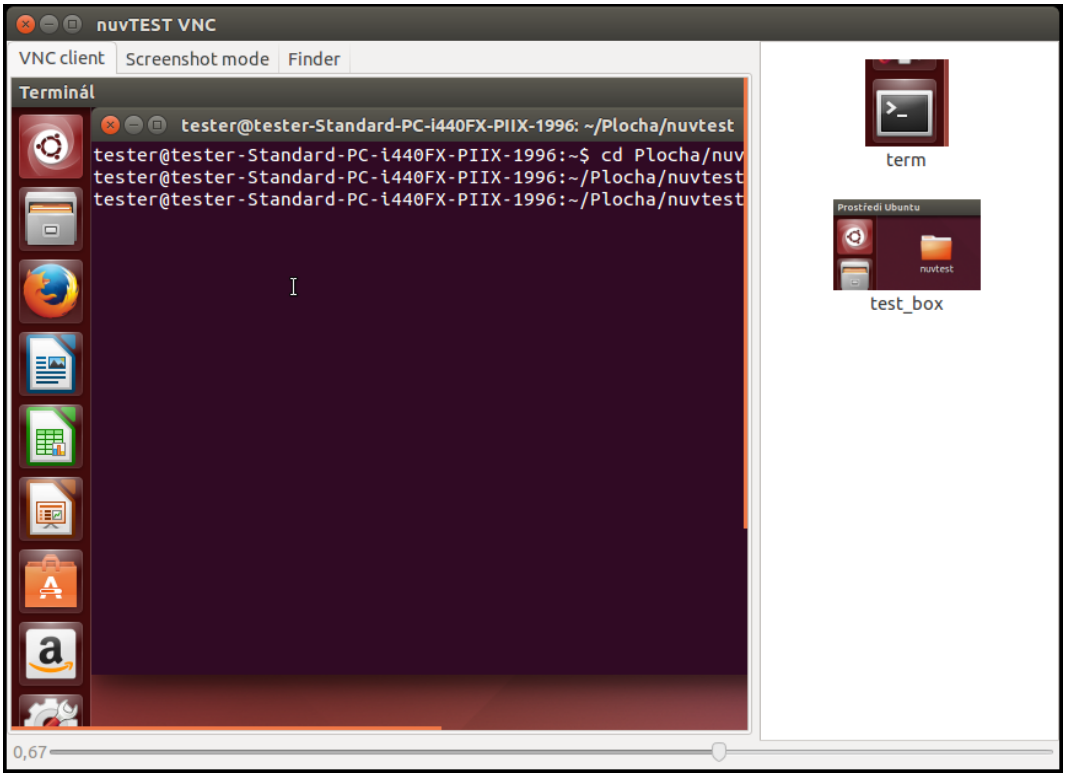
Príloha C

Nástroj umožňujúci vytváranie testovacích skriptov

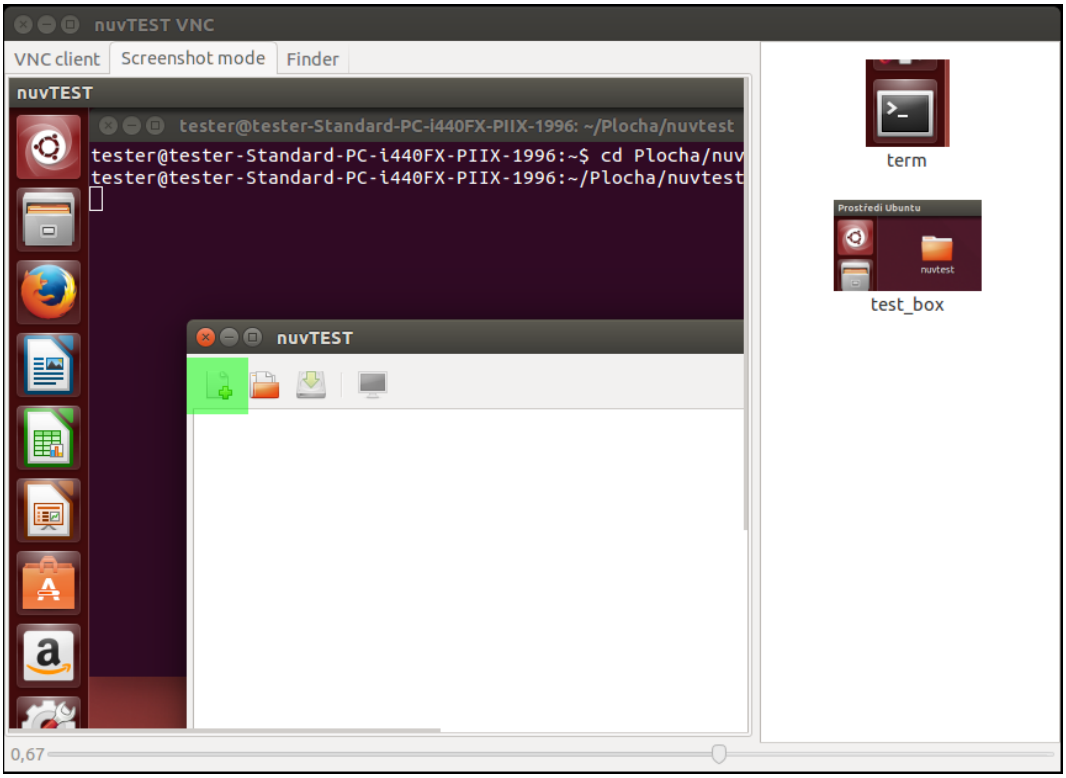
V tejto kapitole sú ukázané screenshoty z vytvorenej aplikácie.



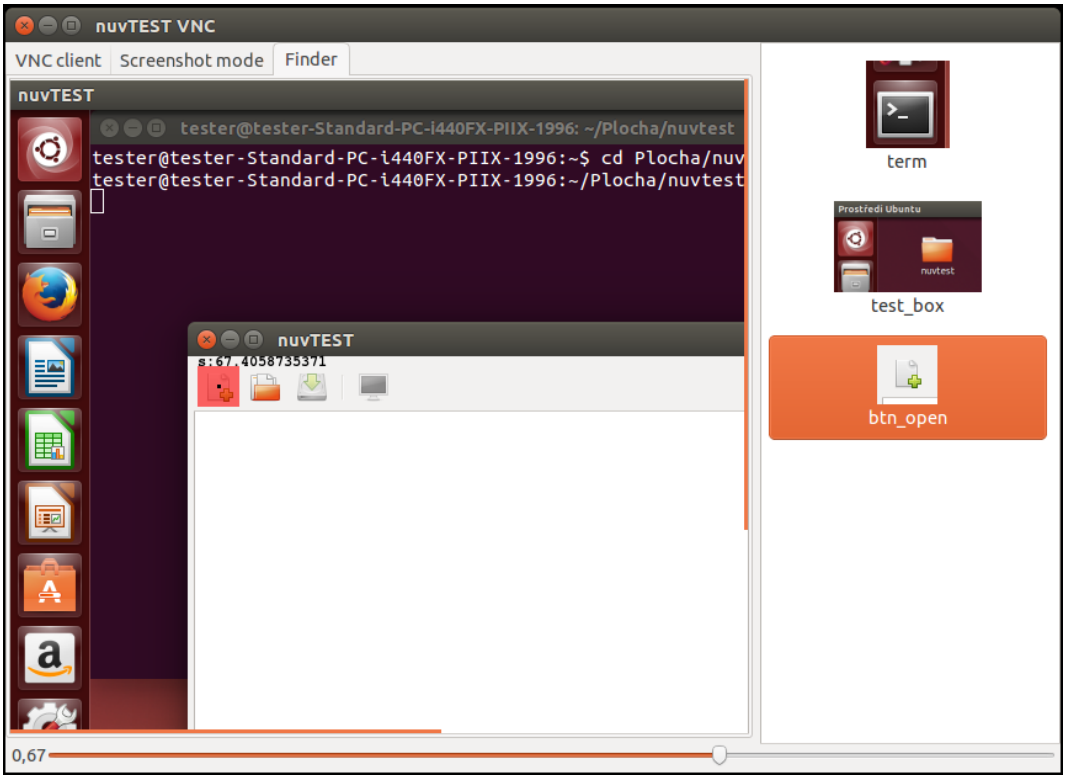
Obr. C.1: Hlavné okno nástroja umožňujúceho vytváranie testovacích skriptov



Obr. C.2: VNC klient umožňujúci užívateľovi ovládať doménu



Obr. C.3: Okno na vyberanie vyhľadávaných oblasti



Obr. C.4: Okno na overenie prahu hľadaného objektu

Príloha D

Vytvorený testovací skript na demonštráciu vlastností systému

Táto príloha obsahuje testovací skript vytvorený pomocou nuvTEST. Celý testovací archív sa nachádza na priloženom CD popísanom v prílohe A.

```
# test case

# wait for domain
find(test_box, 12)

# start app from terminal
click(term)
type("cd Plocha/nuvtest", "<Enter>")
type("python nuvtest.py", "<Enter>")

# checkpoint
create_snapshot(test_snap)

# create new test script
# connect to vnc
click(new_btn)

# click to text field, delete content and type new IP
click(ip_label)
type("<Back>", "<Back>", "<Back>", "<Back>", "<Back>", "<Back>")
type("127.0.0.1")

# and again
click(port_label)
type("<Back>", "<Back>", "<Back>", "<Back>")
type("5901")

# and again
click(pass_label)
type("<Back>", "<Back>", "<Back>", "<Back>", "<Back>", "<Back>")
```

```
type("sojcak")

# create new test
click(new_test_btn)

# try add comand
right_click(toolbar)
click(popup_panel)

# check command
find(click_text)

# check vnc client
click(vnc_btn)
find(vnc_panel)
click(close_vnc_tab)

#close app
click(close_tab)

# revert back to snapshot
revert_to_snapshot(test_snap)

# there must be a app again
find(toolbar)
```