VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

ROZŠÍŘENÍ EDITORU ECLIPSE CHE O MODUL PRO UI TESTY

EXTENSION OF THE ECLIPSE CHE EDITOR FOR UI TESTING MODULE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MARIÁN LORINC**
AUTHOR

VEDOUCÍ PRÁCE **Doc. Mgr. ADAM ROGALEWICZ, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Lorinc Marián**
Program: Informační technologie
Název: **Rozšíření editoru Eclipse Che o modul pro UI testy**
Extension of the Eclipse Che Editor for UI Testing Module
Kategorie: Analýza a testování softwaru

Zadání:

1. Prostudujte architekturu vývojového prostředí Eclipse Che, zaměřte se na vytváření nových rozšíření.
2. Prostudujte nástroj vscode-extension-tester umožňující integraci testů ve vývojovém prostředí VS code.
3. Navrhněte modul umožňující automatizovanou integraci testů určených pro vscode-extension-tester do prostředí Eclipse Che.
4. Navržený modul implementujte.
5. Funkčnost demonstrujte na reálném projektu obsahujícím alespoň 10 různých testovacích scénářů pro vscode-extension-tester.
6. Diskutujte možná další rozšíření.

Literatura:

- Dokumentace vscode-extension-tester: https://github.com/redhat-developer/vscode-extension-tester/wiki
- Dokumentace Eclipse Che: https://www.eclipse.org/che/docs/

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Rogalewicz Adam, doc. Mgr., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Cieľom tejto práce je navrhnúť a naprogramovať testovací modul na testovanie užívateľských rozhraní vývojového prostredia Eclipse Che. Najhlavnejšou prioritou tejto práce je zaistiť kompatibilitu testovacieho modulu pre Eclipse Che s testovacím modulom pre editor Visual Studio Code. Aby bolo možné zaistiť kompatibilitu medzi oboma editormi, bolo navrhnuté spoločné programátorské rozhranie pre grafické komponenty editorov. Vytvorené riešenie umožňuje vývojárom rozšírení editora Visual Studio Code používať testy grafického užívateľského rozhrania aj vo vývojovom prostredí Eclipse Che s minimálnymi úpravami. Prínosom tejto práce je zníženie nárokov na údržbu testov grafického rozhrania a jednotný zdrojový kód pre rozšírenie a testy.

Abstract

The goal of this thesis is to design and implement module for testing graphical user interfaces of integrated development environment Eclipse Che. The biggest priority of this thesis is to make the module compatible with module for testing graphical user interfaces of editor Visual Studio Code. In order to ensure compatibility between both editors, new module was created to define common application programming interface for graphical components. Created solution enables Visual Studio Code extension developers to use existing user interface tests in Eclipse Che IDE with minimal effort. As result source code can be shared which reduces maintenance costs.

Kľúčové slová

testovanie užívateľského rozhrania, Selenium, Eclipse Che, Eclipse Theia, Che-Theia, Visual Studio Code, rozšírenia, zásuvné moduly, Kubernetes, Kubernetes-native, VS Code Extension Tester, vscode-extenion-tester, theia-extension-tester, cyklus udalostí, JavaScript, TypeScript

Keywords

user interface testing, Selenium, Eclipse Che, Eclipse Theia, Visual Studio Code, extensions, Kubernetes, Kubernetes-native, VS Code Extension Tester, vscode-extenion-tester, theia-extension-tester, event loop, JavaScript, TypeScript

Citácia

LORINC, Marián. *Rozšíření editoru Eclipse Che o modul pro UI testy.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Mgr. Adam Rogalewicz, Ph.D.

Rozšíření editoru Eclipse Che o modul pro UI testy

Prehlásenie

Vyhlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Doc. Mgr. Adama Rogalewicza Ph.D. Uviedol som všetky literárne zdroje, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Marián Lorinc
10. mája 2021

Poďakovanie

Týmto by som sa chcel poďakovať pánovi Doc. Mgr. Adamovi Rogalewiczovi Ph.D. za vedenie tejto bakalárskej práce, konzultantovi Ing. Dominikovi Jelínekovi za sprostredkovanie komunikácie s tímom Eclipse Che, pánovi Janovi Richterovi za poskytnuté informácie o testovacom nástroji VS Code Extension Tester a ostatným osobám, ktoré prispeli spätnou väzbou k vylepšeniu tejto práce.

Obsah

1 Úvod **2**

2 Úvod do editora Eclipse Che **4**
2.1 Architektúra 4
2.2 Platforma na vývoj integrovaných programovacích prostredí Eclipse Theia . 9
2.3 Nasadenie zásuvných modulov platformy Eclipse Theia a rozšírení Visual Studio Code do Eclipse Che IDE 13

3 Testovanie rozšírení v editore Visual Studio Code **14**
3.1 VS Code Extension Tester 14

4 Úvod do programovacieho jazyka JavaScript **17**
4.1 Úvod do dedičnosti 17
4.2 Asynchrónne programovanie 18
4.3 Cyklus udalostí 22

5 Zhodnotenie súčasného stavu a návrh projektu **27**
5.1 Zhodnotenie testovacieho nástroja VS Code Extension Tester 27
5.2 Návrh testovacieho nástroja Theia Extension Tester 28

6 Implementácia **31**
6.1 Vstupný bod programu 31
6.2 Ovládač Selenium WebDriver 31
6.3 Riešenie problémov spojené s výpočtovým výkonom 34
6.4 Riešenie problémov nekompatibility testov 37
6.5 Komponent TheiaElement 38
6.6 Komponent ScrollableWidget 41
6.7 Komponent príkazovej palety 42
6.8 Komponent DefaultTreeSection 44

7 Testovanie **46**
7.1 Overenie funkčnosti komponentov 46
7.2 Integrácia testovacieho modulu s testami pre rozšírenia editora Visual Studio Code 46
7.3 Testovanie s potenciálnymi používateľmi 47

8 Záver **48**

Literatúra **49**

Kapitola 1

Úvod

Grafické používateľské rozhrania sa stali najužívanejšou formou užívateľských rozhraní v moderných aplikáciách. Ľudia sa stretávajú s grafickými používateľskými rozhraniami každodenne, či už pri práci na počítači, pri platbe kartou na termináli, alebo trávením voľného času na smartfóne. Ich dôležitosť je natoľko veľká, že zlyhanie grafického rozhrania môže spôsobiť nemalé finančné škody a stratu zákazníkov.

Nie je to inak ani v odvetví podnikových programov. Firmy medzi sebou súťažia nielen o to, kto vytvorí robustnejší program, ale sústredia sa aj na použiteľnosť programu. V tejto oblasti zohrávajú dôležitú rolu vývojárske nástroje. Vývojárske nástroje dokážu pomocou grafických prvkov zjednodušiť prácu s celkovým programom a tým pádom podporiť jeho predaj.

Táto práca vznikla v spolupráci s firmou Red Hat Czech s.r.o., ktorá má široké portfólio vyvíjaných programov. Jedným z týchto odvetví je aj vývoj programátorských nástrojov pre editor VS Code[1] na podporu podnikových programov. Tieto programátorské nástroje editora VS Code (ďalej len „rozšírenia“) využívajú grafické komponenty editora VS Code, aby čo najviac uľahčili prácu vývojárom.

Tieto podporné rozšírenia editora VS Code treba pravidelne testovať, aby sa zamedzilo škodám spôsobenym chybami rozšírenia. Testovanie rozšírení sa realizuje pomocou nástroja VS Code Extension Tester[2], ktorý simuluje ľudské interakcie s užívateľským rozhraním. Nástroj funguje na princípe, že programátorovi poskytuje kolekciu tried, ktoré reprezentujú daný grafický komponent v editore VS Code. Programátor je na základe týchto tried je schopný vytvoriť automatizované testy.

Aktuálnym trendom sa stáva čoraz viac používanie cloudových aplikácií. Jednou z týchto aplikácií je aj editor Eclipse Che[3]. Eclipse Che je editor postavený na cloudových technológiách, ktorý vďaka vlastnej modularite umožňuje užívateľovi pokročilú konfiguráciu editoru. V aplikácii Eclipse Che je možné používať editor Eclipse Theia[4]. Eclipse Theia je webový editor, ktorý ma podobné grafické užívateľské rozhranie ako editor VS Code. Taktiež rozšírenia naprogramované pre editor VSCode je možné použiť v editore Eclipse Theia. Z tohto vyplýva, že by mohlo byť možné použiť nástroj VSCode Extension Tester na testovanie rozšírení, avšak to nie je možné z technologických dôvodov.

[1] Visual Studio Code – https://code.visualstudio.com/
[2] VS Code Extension Tester – https://github.com/redhat-developer/vscode-extension-tester
[3] Eclipse Che – https://www.eclipse.org/che/
[4] Eclipse Theia – https://theia-ide.org/

Cieľom tejto práce je prekonať tieto technologické dôvody a vytvoriť testovací modul grafických používateľských rozhraní pre aplikáciu Eclipse Che, ktorý bude kompatibilný s nástrojom VS Code Extension Tester.

Práca je rozčlenená do kapitol, ktoré uvedú čitateľa do problematiky. V kapitole číslo 2 bude čitateľ uvedený do problematiky editora Eclipse Che a súvisiacich technológií. Kapitola 3 oboznámi čitateľa o problémoch testovania s predvoleným nástrojom editora Visual Studio Code a jeho existujúcom riešení. Kapitola 4 uvedie čitateľa do dedičného modelu programovacieho jazyka a problematiky asynchrónneho modelu. Kapitola 5 oboznámi čitateľa o aktuálnom stave testovacieho nástroja VS Extension Tester a návrhu nového testovacieho nástroja. Kapitola 6 sa zaoberá implementáciou riešení problémov a implementáciou zaujímavých nástrojov. V kapitole číslo 7 bude popísaný postup testovania nového nástroja.

Kapitola 2

Úvod do editora Eclipse Che

Eclipse Che je voľne šíriteľné integrované programátorské prostredie (v praxi je bežne používaná anglická skratka „IDE“), vyvinuté na cloudových technológiách. Eclipse Che sa taktiež označuje aj ako aplikácia typu Kubernetes-native. Aplikácie typu Kubernetes-native sa vyznačujú tým, že je ich možné prevádzkovať u viacerých poskytovateľov cloudových služieb. Títo poskytovatelia cloudových služieb musia podporovať štandard orchestrácie zhlukov kontajnerov Kubernetes (v angličtine sa pojem zhluk kontajnerov označuje ako „cluster“). [33, 2]

Integrované programovacie prostredie Eclipse Che funguje na centralizovanom princípe. Centralizovaný prístup umožňuje administrátorom integrovaného programovacieho prostredia Eclipse Che mať väčšiu kontrolu, z čoho plynú nasledujúce výhody: [33]

- Manažovanie a vytváranie nových technologických konfigurácií (anglicky sa používa pojem „stack“), ktoré môžu vývojári spustiť bez potrebnej inštalácie závislosti alebo znalosti danej konfigurácie.
- Vývojári, ktorí používajú rovnakú konfiguráciu (stack), sa nemusia obávať nekonzistencií programátorského prostredia.
- Integrované programátorské prostredie Eclipse Che môže fungovať aj ako alternatívne riešenie technológie VDI[1]. Administrátor má právomoc zmazať dáta užívateľa (napríklad bývalého zamestnanca).

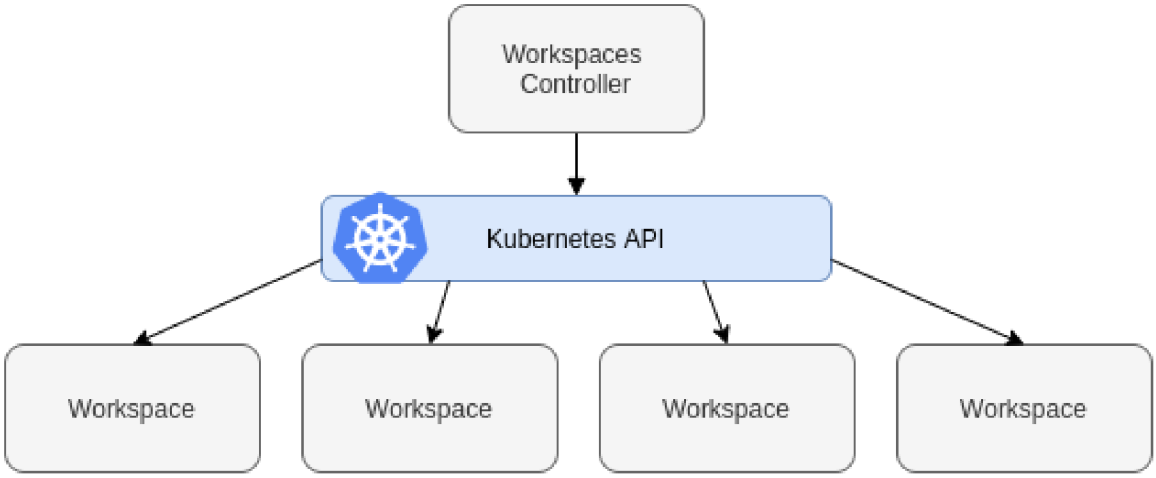
Nasledujúce podkapitoly budú zamerané na uvedenie čitateľa do problematiky architektúry integrovaného programátorského prostredia Eclipse Che a tvorbu nových rozšírení Eclipse Che IDE.

2.1 Architektúra

Integrované programátorské prostredie Eclipse Che, ako bolo spomenuté v úvode kapitoly, je aplikácia typu Kubernetes-native. Z tohoto poznatku vyplýva, že štandard Kubernetes bude základným blokom architektúry. Zjednodušená verzia architektúry integrovaného programátorského prostredia Eclipse Che sa skladá z blokov uvedených na ďalšej strane, ktoré budú vysvetlené v nasledujúcich podkapitolách (vizuálnu formu architektúry je možné vidieť na ilustrácii číslo 2.1). Uvedené bloky čerpajú zo zdroja [31].

[1] VDI – virtuálny stolný počítač

- Kubernetes API – Aplikačné programátorské rozhranie štandardu Kubernetes (integrované programátorské prostredie Eclipse Che využíva túto vrstvu, avšak táto vrstva nie je súčasťou projektu Eclipse Che, a preto nebude rozobratá do hĺbky).
- Workspaces Controller – Centrálna služba, ktorá funguje po celú dobu životnosti štandardu Kubernetes. Služba má na starosti riadenie užívateľských objektov Workspace (pracovné prostredie editora) cez blok Kubernetes API.
- Workspace – Kontajnerová inštancia nakonfigurovaného editora, ktorá sa po ukončení práce vývojára deaktivuje.



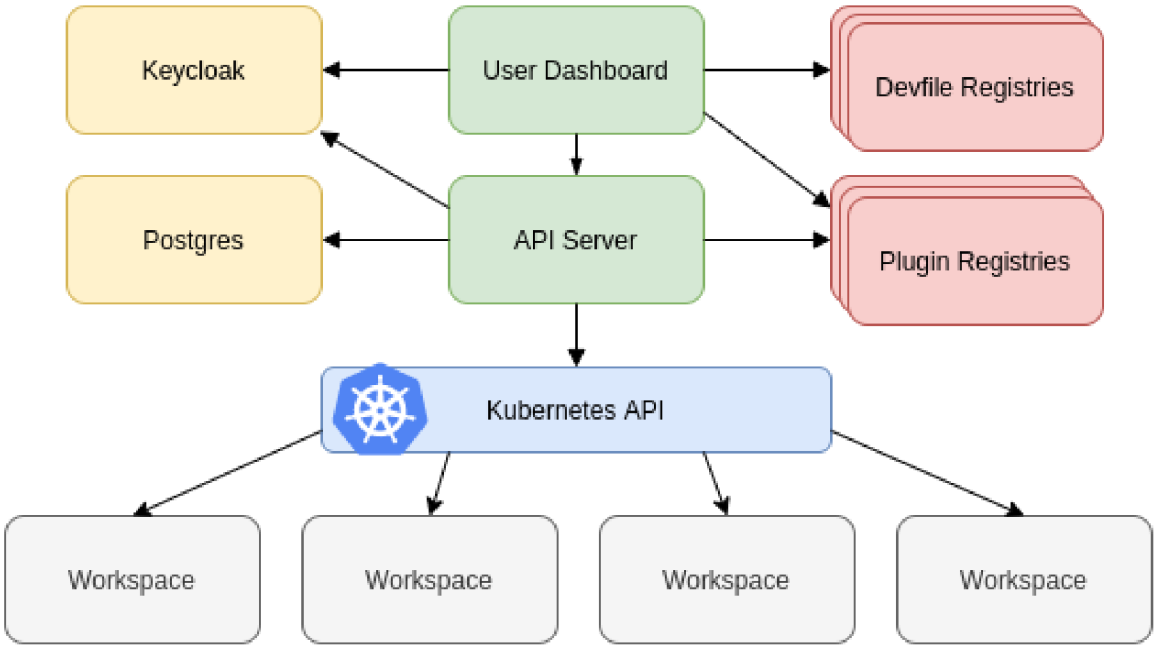
Obr. 2.1: Zjednodušená forma architektúry integrovaného programovacieho prostredia Eclipse Che. Ilustrácia bola prevzatá zo zdroja číslo [31].

Architektúra bloku Workspaces Controller

Nasledujúca text podkapitoly čerpá zo zdroja číslo [36], pokiaľ nie je uvedené inak.

Architektonický blok Workspaces Controller má na starosti orchestráciu nových pracovných prostredí, zabezpečenie, správu technických konfigurácií a spracovávanie príkazov od užívateľov prostredníctvom grafického rozhrania alebo REST API. Architektúru bloku Workspaces Controller tvorí viacero menších celkov, ktoré je možné vidieť na ilustrácií číslo 2.2. Ich význam bude ďalej vysvetlený pod uvedenou ilustráciou. Architektúra je ovplyvnená nasledujúcimi módmi nasadenia integrovaného programátorského prostredia Eclipse Che:

- Mód jedného používateľa – V móde jedného používateľa je deaktivovaný bezpečnostný modul a tým pádom nie je potrebná autentifikačná služba. Mód jedného používateľa je vhodný na prácu na lokálnom počítači, pretože Eclipse Che IDE v tomto móde vyžaduje menej systémových prostriedkov.
- Mód viacerých používateľov – Mód viacerých používateľov je vhodný na komerčné alebo podnikové použitie. V tomto móde je už autentifikácia vyžadovaná, čo vedie k tomu, že bezpečnostný modul je aktivovaný. Eclipse Che IDE v tomto móde používa viacej systémových prostriedkov, než je to v móde jedného používateľa.



Obr. 2.2: Architektúra bloku Workspaces Controller. Ak je integrované prográmovacie prostredie Eclipse Che nasadené v móde jedného užívateľa, blok Keycloak nie je aktívny. Ilustrácia bola prevzatá zo zdroja číslo [36].

API Server

API Server je najhlavnejšou službou architektúry na strane servera. Služba má na starosti správu vytváranie nových pracovných prostredí. V prípade, že Eclipse Che funguje v móde viacerých používateľov, služba má na starosti aj správu užívateľov. Služba obsluhuje požiadavky vytvorené používateľmi editora prostredníctvom rozhrania služby User Dashboard alebo požiadavky odoslané na službu REST API.

User Dashboard

User Dashboard je služba, ktorá prevádzkuje úvodnú stránku integrovaného programátorského prostredia Eclipse Che. Z úvodnej stránky je možné pomocou grafického prostredia vytvárať príkazy na vytvorenie nového pracovného prostredia, nasadiť existujúceho deaktivovaného pracovného prostredia alebo otvoriť aktívne pracovné prostredie. Taktiež je možné zmeniť konfiguráciu pracovných prostredí alebo ich natrvalo zmazať (pritom sa vyžaduje, aby pracovné prostredie bolo deaktivované).

Devfile registries

Služba Che Devfile registry má za úlohu uchovávať a poskytovať súbory *devfile.yml* z databázy. Vloženie novej technologickej konfigurácie je možné vykonať upravením a nasadením kontajnera s názvom *quay.io/eclipse/che-devfile-registry*. Predvolené technické konfigurácie je možné nájsť v repozitári *eclipse-che/che-devfile-registry*[2].

[2] https://github.com/eclipse-che/che-devfile-registry/tree/master/devfiles

Plugin Registries

Služba Che Plugin registry má za úlohu uchovávať a poskytovať metadáta o rozšíreniach (anglicky plug-in) a dostupných editorov (Eclipse Che podporuje viacero verzií editorov). Na metadáta rozšírení sa odkazujú súbory devfile.yml. Predvolené metadáta publikovaných rozšírení je možné nájsť v repozitári *eclipse-che/che-plugin-registry*[3].

Postgres

Modul Postgres uchováva v databáze užívateľské dáta, čo napríklad zahŕňa nasledujúce položky:

- nastavenia pre pracovné prostredia
- metadáta inštancie editora
- prihlasovacie údaje do programu Git
- údaje o užívateľovi

Integrované programátorské prostredie Eclipse Che používa ako predvolenú relačnú databázu PostgreSQL[4]. Eclipse Che podporuje znova použitie existujúcej inštancie databázy PostgreSQL.

Keycloak

Služba Keycloak zastrešuje autentifikáciu v integrovanom programovacom prostredí Eclipse Che. Tento modul je povinný, ak prostredie Eclipse Che funguje vo viacpoužívateľskom móde. Tak ako to bolo v predošlom module Postgres, modul Keycloak podporuje znova použitie existujúcej inštancie služby Keycloak.

Architektúra bloku Workspace

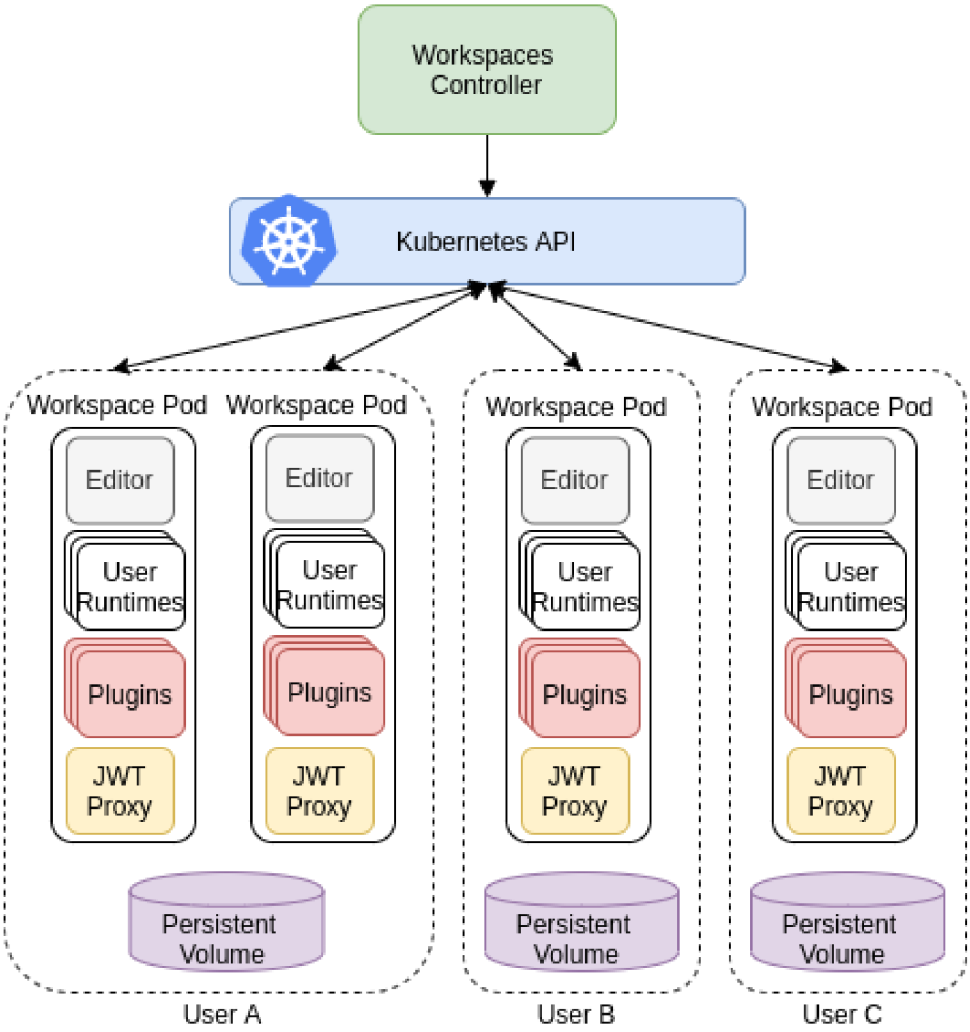
Nasledujúci text podkapitoly čerpá zo zdroja číslo [37], pokiaľ nie je uvedené inak.

Táto časť architektúry má na starosti časť editora, s ktorým vývojár pri programovaní pracuje. Vývojársky editor je naprogramovaný ako webová aplikácia, ktorá využíva ďalšie služby. Služby editora sú spustené v kontajneroch prostredia Kubernetes. Medzi tieto služby patria napríklad: služba samotného editoru, služba asistenta dopĺňania textu programovacích jazykov, nástroje na opravu chýb v programoch, ďalšie služby definované v súboroch devfile.yml a plugin.yml (niekedy sa súbor označuje aj ako meta.yml).

Ďalšou úlohou tejto časti architektúry je správa súborov v súborovom systéme. Zdrojové kódy a ostatné súbory inštancií pracovných prostredí sa ukladajú do virtuálnych diskov v prostredí Kubernetes. Do virtuálneho diskového priestoru majú prístup aj iné bežiace služby v tejto architektúre. Ukladanie zdrojových kódov a súborov je možné vypnúť pri vytváraní nových pracovných prostredí.

V nasledujúcich podkapitolách bude vysvetlená funkcionalita architektonických blokov z ilustrácie číslo 2.3.

[3] https://github.com/eclipse-che/che-plugin-registry
[4] https://www.postgresql.org/



Obr. 2.3: Schéma architektúry bloku Workspace. Na schéme je znázornený stav, keď v integrovanom programovacom prostredí Eclipse Che pracujú traja užívatelia. Užívateľ A má zapnuté dve inštancie pracovných prostredí editora. Ostatní užívatelia majú zapnutú iba jednu inštanciu pracovného prostredia editora. Ilustrácia bola prevzatá zo zdroja číslo [37].

Blok Editor

Tento blok reprezentuje samotný editor, jeho vzhľad, rozloženie grafického užívateľského rozhrania, a funkcionalitu. Integrované programovacie prostredie Eclipse Che je vysoko modulárne a umožňuje používanie ľubovoľných implementácií editorov. Predvoleným editorom Eclipse Che je Eclipse Theia, ktorý je detailnejšie vysvetlený v podkapitole číslo 2.2.

Blok Plugins (rozšírenia)

Rozšírenia v Eclipse Che IDE sú špeciálne služby, ktoré rozširujú funkcionalitu editoru. Každé rozšírenie môže byť zahrnuté vo vlastnom kontajneri, čo má niekoľko výhod. Jednou z výhod je izolácia od editora spolu s možnosťou konfigurácie limitov systémových prostriedkov. Ďalšou výhodou je, že kontajnery musia spĺňať štandardy, čo zľahčuje distribúciu rozšírení. Alternatívne, rozšírenia nemusia byť zahrnuté vo vlastnom kontajneri. V tomto

prípade sú rozšírenia súčasťou spoločného kontajneru ostatných rozšírení. Avšak tieto rozšírenia sú stále izolované od editora a ostatných rozšírení s vlastným kontajnerom.

Blok User Runtimes

V rámci tohto bloku vývojári rozšírení môžu pridať dodatočné závislosti rozšírenia do ďalšieho kontajnera. Prakticky sa to využíva napríklad, keď pracovné prostredia editora neobsahuje interpreter jazyka Python, ale projekt ho vyžaduje na prácu.

Blok JWT Proxy

Táto časť architektúry je zodpovedná za autentifikáciu a manažovanie tokenu formátu JWT. Ak integrované programátorské prostredie Eclipse Che funguje v móde jedného užívateľa, táto služba je deaktivovaná.

Blok Persistent Volume

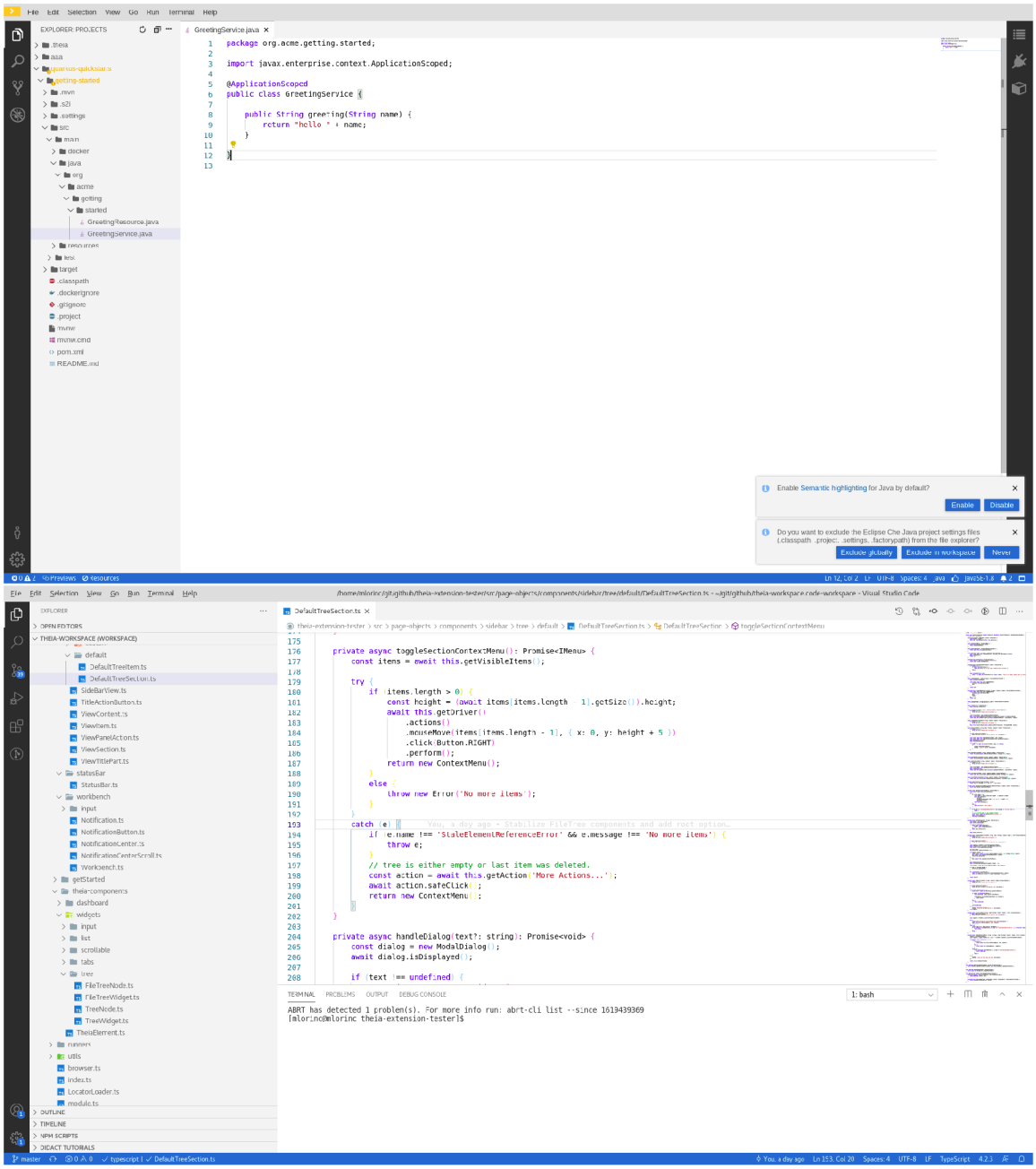
Tento blok je zodpovedný za ukladanie zdrojových kódov a súborov do virtuálnych diskov v prostredí Kubernetes. Ak je vypnuté ukladanie súborov, tento blok je neaktívny.

2.2 Platforma na vývoj integrovaných programovacích prostredí Eclipse Theia

V prechádzajúcej podkapitole bolo uvedené, že Eclipse Theia je predvolený editor integrovaného programovacieho prostredia Eclipse Che. Tento výrok nie je až tak pravdivý, ale nie je ani ďaleko od pravdy. Platforma Eclipse Theia je nástroj, na vývoj integrovaných programovacích prostredí. Dá sa to chápať ako nástroj, pomocou ktorého dokáže programátor vyvinúť integrované programovacie prostredie na vlastnú mieru. Túto platformu používa aj editor Che-Theia. Editor Che-Theia je modifikovaná verzia platformy Eclipse Theia, v ktorej je pridaná nová funkcionalita zameraná na prácu s kontajnermi a s celkovou integráciou s Eclipse Che IDE. [34, 32]

Predvolená konfigurácia platformy Eclipse Theia sa vizuálne podobá editoru Visual Studio Code. Túto vizuálnu podobnosť je možné vidieť na ilustrácii číslo 2.4. Oba editory dokonca medzi sebou zdieľajú rovnakú závislosť, ktorou je knižnica Monaco Editor. Knižnica Monaco Editor pokrýva v oboch editoroch funkcionalitu textového editoru, príkazovej palety a niektoré komponenty spodnej lišty. [35, 9, 11]

Funkcionalitu editorov postavených na platforme Eclipse Theia je možné rozšíriť troma spôsobmi, ktoré budú bližšie vysvetlené v tejto podkapitole. Prvé dva spôsoby sa čisto viažu iba na platformu Eclipse Theia. Posledný spôsob je oveľa zaujímavejší, na základe ktorého vznikla motivácia vypracovať túto bakalársku prácu.



Obr. 2.4: V hornej časti obrázku sa nachádza snímka obrazovky, na ktorej je zobrazený editor Che-Theia. V dolnej časti obrázku sa nachádza snímka obrazovky editora Visual Studio Code. Na hornom snímku je vidno zopár rozdielov medzi editorom Che-Theia a predvolenou konfiguráciou platformy Eclipse Theia. Editor Che-Theia pridáva do platformy novú funkcionalitu ako je napríklad pravá lišta, ktorá má na starosti správu kontajnerov Eclipse Che IDE. Oranžové tlačidlo v ľavom hornom rohu, umožňuje užívateľovi vrátiť sa do hlavnej webovej aplikácie Eclipse Che IDE.

Rozšírenia platformy Eclipse Theia

Predtým ako prejdeme ku výkladu o rozšíreniach Eclipse Theia, je nutné spomenúť jednu dôležitú informáciu. Platforma Eclipse Theia používa inú terminológiu, čo sa týka vzťahu rozšírení a zásuvných modulov. V terminológií editora Visual Studio Code a celej tejto práce sa rozšírenie chápe ako program, ktorý je možné nainštalovať do editora počas jeho používania. Platforma Eclipse Theia tento druh rozšírení označuje v terminológie ako zásuvné moduly.

Rozšírenia na platforme Eclipse Theia majú najvoľnejší prístup ku editoru. Tento typ rozšírení je vhodné použiť vtedy, keď nie je možné naprogramovať nové rozšírenie prostredníctvom aplikačného programátorského rozhrania zásuvných modulov. Pôvodne to bol jediný spôsob, akým bolo možné rozšíriť editor založený na platforme Eclipse Theia. Celý princíp vytvárania nových rozšírení je založený na tom, že vývojár vytvorí modul, ktorý je od ostatných modulov platformy implementačne oddelený. Avšak komunikácia medzi modulmi je dôležitá pre správne fungovanie editora. Platforma Eclipse Theia tento problém rieši použitím návrhového vzoru „Dependency injection“. Modul rozšírenia môže takýmto spôsobom komunikovať s ostatnými modulmi cez ich verejné rozhranie bez toho, aby programátor modulu musel mať znalosti o životnosti ostatných modulov. [6]

Rozšírenia platformy Eclipse Theia sa delia na dve kategórie. Prvá kategória je v angličtine označovaná ako „back-end“ (v preklade chrbtová). Rozšírenia tejto kategórie fungujú na serverovej časti. To môže zahŕňať rozšírenia ako asistenti pri programovaní (anglicky „intellisense“), server debug alebo rozšírenie na synchronizáciu súborov s cloudovým poskytovateľom. Druhá kategória je v angličtine označovaná ako „front-end“ (v preklade popredná). Táto kategória funguje na klientskej časti v internetovom prehliadači. Tento typ rozšírení je vhodné použiť, ak nie je potreba pristupovať ku systémovým prostriedkom. [29]

Tento spôsob implementácie nových rozšírení má však značné nevýhody. Všetky rozšírenia musia byť nainštalované počas kompilácie editoru. Nie je možné ich nainštalovať počas behu editora. To znamená, že ak chce vývojár pridať alebo opraviť chybu v rozšírení, vývojár musí znovu vykonať všetky potrebné úkony pri kompilovaní editora. Ďalšou nevýhodou je príliš veľká sloboda rozšírení. Rozšírenia nie sú oddelené žiadnym spôsobom od editora. Nesprávne naprogramované rozšírenia majú potenciál pokaziť editor. Ďalším problémom väčšej slobody je, že inštalácia rozšírení od tretích strán predstavuje bezpečnostné riziko. Tretie strany môžu v najhoršom prípade editor infikovať škodlivým programom. Z tohto dôvodu vznikol ďalší typ rozšírení, ktoré sa v terminológii platformy Eclipse Theia nazývajú zásuvné moduly. [5]

Zásuvné moduly

Zásuvné module platformy Eclipse Theia vznikli na základe uvedených problémov v predošlej podkapitole. Zásuvné moduly sú izolované od editora a nie je ich potrebné inštalovať počas kompilácie programu. Izolovanosťou sa myslí izolácia zásuvných modulov od hlavného vlákna editora, ale aj od hlavných modulov editora, teda rozšírení. Izolácia od hlavných modulov je realizovaná formou verejného aplikačného programovacieho rozhrania, ktoré platforma zásuvným modulom poskytuje. [5]

Aplikačné programovacie rozhranie na vývoj zásuvných modulov je inšpirované existujúcim aplikačným programovacím rozhraním na vývoj rozšírení pre editor Visual Studio Code, čo je ďalším motivátorom vzniku tejto práce. Medzi aplikačnými programovacími rozhraniami sú malé rozdiely. Hlavným rozdielom je, že platforma Eclipse Theia zaostáva zopár aktualizácií za editorom Visual Studio Code. [5, 4]

Vytváranie zásuvných modulov

Zásuvné moduly platformy Eclipse Theia je možné vytvoriť pomocou generátora projektov *yo*. Generátor *yo* generuje projektu na základe šablón. Šablóna zásuvných modulov platformy Eclipse Theia nie je implicitne nainštalovaná v spomenutom generátore. Šablónu je nutné pred použitím nainštalovať. Príkazy na inštaláciu šablóny a vygenerovanie nového zásuvného modulu je možné vidieť na ilustrácii číslo 2.5. [7]

```
npm install -g yo @theia/generator-plugin
yo @theia/plugin
```

Obr. 2.5: Sekvencia príkazov, ktoré nainštalujú šablónu projektu zásuvných modulov a vygenerujú nový projekt zásuvného modulu.

Generátor zistí všetky potrebné informácie od programátora a na základe informácií vygeneruje nový zásuvný modul. Spúšťací súbor zásuvného modulu sa nachádza v súbore *<meno zásuvného modulu>-<kategória pluginu>-plugin.ts*. Daný súbor obsahuje dve funkcie: *start* a *stop*. Funkcia *start* sa zavolá, keď sa zásuvný modul úspešne nahrá do platformy Eclipse Theia. V tejto funkcií dochádza ku registrácií spätných volaní na udalosti a rôzne inicializačné úkony. Funkcia *stop* slúži na prípadne ukončenie otvorených prostriedkov, ktoré je potrebné zatvoriť. To môže napríklad zahrnovať: vypnutie lokálneho servera, ukončenie databázového pripojenia alebo zmazanie dočasných súborov. Funkcia *stop* je dobrovoľná a nemusí byť deklarovaná v súbore. [7, 30]

Integrácia Visual Studio Code rozšírení

Rozšírenia editora Visual Studio Code fungujú na rovnakom princípe ako zásuvné moduly platformy Eclipse Theia. Avšak prvý rozdiel je vidno už u spúšťacieho súboru. Zatiaľ čo zásuvne moduly platformy Eclipse Theia importujú modul npm[5] *@theia/plugin*, rozšírenia editora Visual Studio Code importuju z npm modulu *vscode*. Ďalším rozdielom je pomenovanie *start* a *stop* funkcií. Funkcia *start* sa v editor VS Code volá *activate*. Funkcia *stop* sa volá *deactivate*. [30, 8, 10]

Napriek tomuto rozdielu je možné rozšírenia VS Code používať na platforme Eclipse Theia. Platforma Eclipse Theia obsahuje dva rozšírenia, ktoré to umožňujú. Tieto rozšírenia sú nasledovné: *@theia/plugin-ext* a *@theia/plugin-ext-vscode*. Prvé rozšírenie pridáva podporu pre rozšírenia Visual Studio Code. Druhé rozšírenie umožňuje stiahnuť rozšírenie z registra rozšírení Visual Studio Marketplace[6]. Toto nie je vhodné na komerčné použitie, pretože inštalovanie rozšírení z registra VS Code Marketplace mimo editory VS Code nie je povolené licenčnými podmienkami. [8]

Na základe právnych problémov vznikol alternatívny otvorený register open-vsx.org[7]. Rozšírenia z tohto registra je možné nainštalovať aj na platforme Eclipse Theia. Aby bolo možné inštalovať rozšírenia (zásuvné moduly) z registra open-vsx.org, je najprv nutné aktivovať rozšírenie na platforme Eclipse Theia, ktoré sa volá *@theia/vsx-registry*. [8]

Posledným úkonom na zaručenie úspešnej integrácie rozšírení editora Visual Studio Code je overiť správnu funkčnosť rozšírenia na platforme Eclipse Theia, čo je ďalším dôvodom vzniku tejto práce.

[5] https://www.npmjs.com/
[6] https://marketplace.visualstudio.com/
[7] https://open-vsx.org/

2.3 Nasadenie zásuvných modulov platformy Eclipse Theia a rozšírení Visual Studio Code do Eclipse Che IDE

V úvode podkapitoly číslo 2.2 bolo uvedené, že integrované programovacie prostredie Eclipse Che používa editor Che-Theia, ktorý je postavený na platforme Eclipse Theia. Rozšírenia[8] editora Che-Theia fungujú na rovnakej podstate ako zásuvne moduly platformy Eclipse Theia.

Rozšírenia editoru Che-Theia podliehajú vyššej miere izolovanosti, než je to v prípade platformy Eclipse Theia. Každé jedno rozšírenie v editore Che-Theia musí bežať v rámci vlastného kontajnera alebo zdieľaného kontajnera. Vlastný kontajner nepotrebujú tie rozšírenia, ktoré nepoužívajú závislosti tretích strán. Ak rozšírenie používa závislosti tretích strán, musí byť vo vlastnom kontajneri, ktorého definícia je uložená v súbore meta.yml. Súbor meta.yml obsahuje metadáta o konkrétnom rozšírení, prípadne zoznam rozšírení, ak sa jedná o balíček rozšírení. Aby bolo možné použiť súbor meta.yml musí byť voľne stiahnuteľný z Internetu. Na súbory meta.yml sa odkazujú súbory devfile.yml, ktoré slúžia ako šablóna pri vytváraní pracovných prostredí v integrovanom programovacom prostredí Eclipse Che. Formáty oboch súborov z dôvodu objemnosti metadát nebudú v tejto práci ďalej vysvetlené. [38]

V rozšíreniach editora VS Code je pomerne bežné, že závislosti tretích strán sú súčasťou archívu zabaleného rozšírenia, alebo sú dodatočne stiahnuté po úspešnej inštalácii rozšírenia. Takýmto rozšíreniam je potrebné vytvoriť vlastný kontajner, ktorý sa v terminológii označuje ako „sidecar“.

Niektoré závislosti môžu byť službami. Dobrý príkladom je služba LSP[9]. Služby LSP je potrebné spustiť, ak nie sú ešte spustené pri aktivácií rozšírenia. V takomto prípade je možné v definícii kontajnera nastaviť štartovací príkaz [38].

[8] Zásuvné moduly, editor Che-Theia sa neriadi terminológiou platformy Eclipse Theia.
[9] https://microsoft.github.io/language-server-protocol/specification

Kapitola 3

Testovanie rozšírení v editore Visual Studio Code

Rozšírenia editora Visual Studio Code je program, ktorý rozširuje funkcionalitu editora. Zdrojový kód a návrh rozšírení môže byť jednoduchý alebo komplexný. Čím je rozšírenie komplexnejšie, tým je väčšia pravdepodobnosť zavedenia chyb pri aktualizáciách.

Testovaním rozšírení je možné tieto rizika do určitej miery eliminovať. Vývojári Visual Studio Code vytvorili pomocnú knižnicu vscode-test[1], ktorá pomáha testerovi so základnými úkonmi s editorom. So spomenutou knižnicou je možné stiahnuť čistú inštanciu editora Visual Studio Code, nájsť stiahnutý binárny súbor a spustiť integračné testy v inštancii editora [12].

Integračné testy môžu byť naprogramované v ľubovoľnej testovacej knižnici, avšak musia byť naprogramované v programovacom jazyku JavaScript. Testovací program môže manipulovať s inštanciou editora prostredníctvom aplikačného programátorského rozhrania, ktoré poskytuje editor Visual Studio Code na tvorbu rozšírení [12]. Ak chce tester testovať rozšírenie v rámci funkcionálnej stránky, tak toto riešenie nie je vhodné. Riešenie totiž používa to isté aplikačné programovacie rozhranie, s ktorým sa vyvíjajú nové rozšírenia. Netestuje sa funkcionalita rozšírenia, ale iba jeho aplikačné programovacie rozhranie.

Na vyriešenie problému s funkcionálnymi testovaním vznikol nástroj VS Code Extension Tester[2].

3.1 VS Code Extension Tester

Nástroj VS Code Extension Tester (skrátene Extension Tester) dokáže vykonávať podobné úkony ako spomenutá knižnica vscode-test. Avšak ako bolo spomenuté, tieto nástroje sa veľmi líšia. Obidva nástroje sú určené na iný typ testovania.

Nástroj Extension Tester pri testovaní rozšírenia simuluje interakcie užívateľa s grafickým užívateľským rozhraním editora. Je to umožnené jednou vlastnosťou editora Visual Studio Code. Editor je postavený na projekte ElectronJS [23]. ElectronJS je modifikovaný internetový prehliadač projektu Chromium, ktorý je jadrom internetového prehlia-

[1] https://www.npmjs.com/package/vscode-test
[2] https://github.com/redhat-developer/vscode-extension-tester

dača Google Chrome[3] [22]. Na základe týchto znalostí mohla byť vybratá knižnica Selenium WebDriver[4], ktorá sa používa na testovanie internetových prehliadačov.

Nástroj Extension Tester sa skladá z piatich základných blokov, ktoré budú vysvetlené v nasledujúcich podkapitolách[5].

Blok Native

Blok Native je experimentálna časť, ktorá ma na starosti ovládanie systémových dialógov. Časť vznikla ešte predtým, ako bolo možné používať modálne dialógy formou komponentu InputBox. Ovládanie systémových dialógov je pomerne komplexný problém, ktorý so sebou prináša veľa synchronizačných problémov (pri vyvíjaní sa muselo staticky uspávať vlákno). Taktiež systémové dialógy neponúkajú veľa možností ako ich efektívne ovládať, než je to v prípade dialógu typu InputBox. Z tohto dôvodu v blízkej budúcnosti bude tento experimentálny blok odstránený z nástroja.

Blok API handler

Tento blok bol kedysi odpoveďou na problémy so systémovými dialógmi. Princípom tohto bloku bolo obídenie systémového dialógu formou samotného rozšírenia editora. Rozšírenie reagovalo na príkazy od užívateľa a tie následne spustila vo vnútri editora. Principiálne je to rozšírenie typu proxy, ktoré umožňuje spúšťanie niektorých príkazov priamo v aplikačnom programovacom rozhraní rozšírení editora Visual Studio Code. Po predstavení dialógov formou komponentu InputBox, aj tento spôsob sa stal zastaralým.

Blok Locator

V bloku Locator sú uchovávané vyhľadávacie výrazy a iné súvisiace dáta, ktoré sa používajú pri vyhľadávaní elementov v HTML štruktúre editora Visual Studio Code. Vyhľadávacie výrazy sú fragmentované podľa verzií editora Visual Studio Code. Najstaršia verzia vyhľadávacích výrazov obsahuje všetky vyhľadávacie výrazy. Postupne s aktualizáciami editora dochádzalo ku zmene štruktúry HTML editora a to viedlo ku neaktuálnym výrazom. Neaktuálne vyhľadávacie výrazy bolo potrebné aktualizovať, avšak väčšinou neaktuálnych výrazov bolo málo. Z tohto dôvodu namiesto vypisovania všetkých vyhľadávacích výrazov od znova sa používajú fragmenty[6].

Každý ďalší fragment, okrem prvého fragmentu, obsahuje iba opravené vyhľadávacie výrazy od posledného fragmentu. O zrekonštruovanie fragmentov vyhľadávacích výrazov a iných dát je zodpovedný blok Monaco page objects.

Blok Monaco page objects

Ako bolo spomenuté v predošlej podkapitole, tento blok má na starosti zrekonštruovanie fragmentov. Rekonštruovanie fragmentov prebieha tak, že sa najprv prečíta, spracuje a uloží do pamäti prvý fragment, ktorý by mal obsahovať všetky vyhľadávacie výrazy a iné dáta. Následne algoritmus prechádza cez ďalšie fragmenty a aplikuje ich opravy do štruktúry,

[3] https://www.google.com/intl/sk_sk/chrome/

[4] https://www.selenium.dev/documentation/en/webdriver/

[5] Text podkapitol čerpá z diskusií tímov spomenutých v podkapitole číslo 7.3.

[6] fragment – súbor, ktorý obsahujú len nové vyhľadávacie výrazy

ktorú ma uloženú v pamäti. Algoritmus nemusí prejsť cez všetky fragmenty, ak ma nastavenú špecifickú verziu (neaplikuje fragmenty novších verzií).

Ďalšou úlohou bloku Monaco page objects je poskytovanie implementovaných komponentov. Komponentom sa myslí objekt programovacieho jazyka JavaScript, ktorý vytvára abstrakciu nad nejakým významným elementom HTML, s ktorým používateľ interaguje v editore Visual Studio Code. Významným elementom je taký element, s ktorý užívateľ editora priamo interaguje. To môžu byť napríklad komponenty ako InputBox (príkazová paleta a vstup od užívateľa), textový editor, notifikácie, atď. Abstrakcia komponentu je vytvorená nad objektom typu WebElement, ktorý patrí knižnici Selenium WebDriver.

Extester

Táto časť je najhlavnejšou častou celého nástroja. Časť Extester má na starosti orchestráciu ostatných blokov a správu samotného prehliadača. Prehliadač sa v tomto prípade chápe ako objekt programovacieho jazyka JavaScript, ktorý používa objekt Selenium WebDriver.

Testovanie rozšírení editora Visual Studio Code sa začína v tomto bloku. Spustiť testy je možné dvoma spôsobmi. Prvým spôsobom je použitie poskytnutého aplikačného programovacieho rozhrania. Druhý spôsobom je spustenie skriptu, ktorý exportuje modul Extester. V oboch prípadoch dochádza ku stiahnutiu potrebných závislostí (editor a príslušný Chrome WebDriver), inicializácii prehliadača a spúšťača testov. Prehliadač má na starosti inicializáciu Selenium WebDriver a vyhľadávacích výrazov. Následne prehľadávač spustí testy cez spúšťač testov. Spúšťač testov má na starosti inicializáciu testov Mocha[7]. Spúšťač ďalej musí zaistiť, že testy grafického rozhrania sa spustia vtedy, keď editor bude pripravený.

[7] https://mochajs.org/

Kapitola 4

Úvod do programovacieho jazyka JavaScript

Programovací jazyk JavaScript za posledné roky prešiel mnohými zmenami. Posledným trendom desaťročia sa stala technológia NodeJS[1] spolu s technológiou Electron, ktoré umožňujú používať webové technológie v natívnych systémových aplikáciách. Tento trend postupne zvyšuje popularitu programovacieho jazyka mimo web ekosystém. V roku 2020 na základe prieskumu vykonaným portálom StackOverflow sa web technológie umiestnili na popredných priečkach [28]. Na webových technológiách sú postavené aj editory Visual Studio Code a Eclipse Che.

Táto kapitola sa v úvode bude zaoberať problematikou dedičnosti programovacieho jazyka JavaScript, problematikou asynchrónneho programovania v programovacom jazyku JavaScript. Pochopenie asynchrónneho programovacieho modulu v jazyku JavaScript je nesmierne dôležité pre túto prácu a pri testovaní rozšírení testovacím modulom.

4.1 Úvod do dedičnosti

Nasledujúca podkapitola sa bude zaoberať primárne dedičnosťou objektu a definíciou samotného objektu. Podkapitola čerpá zo štandardu ECMAScript [1] a zo stránky mdn [13].

JavaScript je programovací jazyk, ktorý implementuje štandard ECMAScript. ECMAScript je štandard objektovo orientovaného programovacieho jazyka, ktorý bol pôvodný navrhnutý ako webový skriptovací jazyk.

Štandard ECMAScript definuje objekt ako množinu vlastností, ktoré majú vlastné atribúty. Atribúty plnia úlohu metadát (napríklad atribút „Writeable“ určuje či je vlastnosť konštantná). Každá vlastnosť môže v sebe uchovávať primitívny dátový typ, referenciu na funkciu alebo ďalší objekt. Funkcia, ktorá je naviazaná na objekt sa označuje v ECMAScript štandarde ako metóda.

Objektovo orientované programovanie je v štandarde ECMAScript realizované pomocou objektových prototypov. Prototyp je špeciálna objektová vlastnosť, ktorá implicitne obsahuje referenciu na rodičovskú inštanciu objektu (vlastnosť prototype môže obsahovať akýkoľvek objekt alebo primitívny dátový typ). Ak je objekt prvý v dedičnej hierarchií, vo vlastnosti prototyp je uložená hodnota „null“. Hodnota „null“ označuje koniec reťaze prototypov. Reťaz prototypov sa používa pri vyhľadávaní vlastnosti objektu, ktorá sa nenachádza priamo v inštancii objektu, ale nachádza sa v ľubovoľnom predkovi. Ak inštancia objektu

[1] https://nodejs.dev/

obsahuje vlastnosť, ktorá je obsiahnutá aj v nejakom predkovi, daná vlastnosť je prepísaná novšou verziou.

Na základe fungovania mechanizmu rezolúcie vlastnosti objektu je možné rozširovať, alebo testovať menšie opravy chyb, bez potreby kompilovať zdrojové kódy od znova. V oboch prípadoch stačí prepísať alebo inicializovať novú vlastnosť v prototype.

4.2 Asynchrónne programovanie

Programovací jazyk JavaScript implicitne používa jedno vlákno. Napriek jednému vláknu programovací jazyk JavaScript dokáže efektívne manažovať asynchrónne neblokujúce operácie. [21]

JavaScript na realizáciu asynchrónneho vykonávania úloh používa techniku cyklus udalostí (anglicky sa táto technika nazýva „event loop“). Pri používaní techniky cyklus udalostí sa predpokladá, že úlohy nebudú trvať dlhý čas. Pôvodne bol programovací jazyk JavaScript určený na obsluhu elementov HTML. Avšak postupom času sa komplexnosť stránok a aplikácií začala zvyšovať spolu s časovou komplexnosťou úloh. Komplexnosť aplikácií taktiež spôsobila, že niektoré techniky programovania neboli udržateľné kvôli znižujúcej kvalite prehľadnosti kódu. [17]

Táto podkapitola sa bude zaoberať metódami asynchrónneho programovania v programovacom jazyku JavaScript, výhodami určitých metód a ich nevýhodami. Pre problematiku cyklu udalostí bola špeciálne vyhradená samostatná podkapitola. Keďže sa jedná o pomerne zložitejšiu tému. Túto podkapitolu je možné nájsť pod číslom 4.3.

Spätné volania funkcií

Technika spätného volania funkcií bola jediným spôsobom asynchrónneho programovania v programovacom jazyku JavaScript až do vydania špecifikácie ECMAScript 2015. Táto technika sa používa ešte dodnes a stále ma svoje uplatnenie. Text tejto podkapitoly čerpá zo stránky nodejs.dev/learn. [15]

Funkcie v programovacom jazyku JavaScript sú klasifikované ako objekty. Objekty je možné vložiť do funkcie ako vstupný argument. Vloženú funkciu ako objekt je možné zavolať v tele funkcii. Z tohto dôvodu sa tieto funkcie volajú „spätné volanie“, pretože ich volaná funkcia volá naspäť. Príklady spätného volania je možné vidieť v nasledujúcej ilustráciách s číslom 4.1 a 4.2.

```
/** Vykona matematicky vypocet a / b */
function divide(a, b, callback) {
    if (b === 0) {
        // zavolaj funkciu callback s chybou
        callback("Vyraz nie je mozne delit nulou.");
    }
    else {
        // zavolaj funkciu callback s vysledkom delenia
        callback(a / b);
    }
}
```

Obr. 4.1: Ukážka deklarácie metódy so spätným volaním volania

```
divide(10, 5, function (result) {
    // ak je vysledok string => nastala chyba
    if (typeof result === 'string') {
        console.error(`Nastala chyba pri deleni: ${result}`);
    }
    // delenie prebehlo v poriadku
    else {
        console.log(`10 / 5 = ${result}`);
    }
});
```

Obr. 4.2: Ukážka použitia metódy spätného volania

Metódu spätného volania sa používa pri kóde, ktorý sa volá pri nejakej udalosti. To môže byť napríklad ovládač tlačidla, ovládač tcp spojenia alebo aj plánovanie úlohy pomocou funkcie setTimeout. V minulosti sa tento spôsob používal aj pri asynchrónnom programovaní.

Nesprávne používanie spätného volania môže viesť ku horšej kvalite kódu. Programy v jazyku JavaScript môžu mať v niektorých prípadoch viacero asynchrónnych volaní, ktoré sa musia vykonať sekvenčne za sebou. V takomto prípade dochádza k hlbokému zanáraniu spätných volaní a znižovaním prehľadnosti kódu (tento jav sa v angličtine označuje ako „callback hell“ – peklo spätných volaní). Ukážku javu „callback hell“ je možné vidieť na ilustrácií číslo 4.3.

```
/**
 * Program na vypocet vyrazu 5 + 10 / 2 - 3 * 9.
 * Kazda metoda v tomto priklade prijima tri argumenty.
 * Prve dva argumenty su ciselne a pouzivaju sa na matematicku operaciu.
 * Posledny argument je funkcia callback, ktora sa zavola
 * s ciselnym vysledkom alebo chybovym retazcom.
 */
divide(10, 2, function (result) {
    // ak je vysledok string => nastala chyba
    if (typeof result === 'string')
        console.error(`Nastala chyba pri deleni: ${result}`);
    }
    // delenie prebehlo v poriadku
    else {
        add(5, result, function (result) {
            if (typeof result === 'string') {
                console.error(`Nastala chyba pri scitani: ${result}`);
            }
            else {
                multiply(-3, 9, function(multiply_result) {
                    if (typeof multiply_result === 'string') {
                        console.error(`Nastala chyba pri nasobeni: ${result}`);
                    }
                    else {
                        // result = 5 + 10 / 2, multiply_result = -3 * 9
                        add(result, multiply_result, function(result) {
                            if (typeof result === 'string') {
                                console.error(`Nastala chyba pri scitani: ${result}`);
                            }
                            else {
                                console.log(result);
                            }
                        });
                    }
                });
            }
        });
    }
});
```

Obr. 4.3: Ukážka javu „callback hell“. Deklarácia metódy „divide“ sa nachádza na ilustrácií s číslom 4.1.

Návrhový vzor Promise

Metóda spätného volania nie je kódovo udržiavateľná, a preto došlo ku vzniku objektu Promise. Táto podkapitola uvedie čitateľa do problematiky programovania s objektom Promise. Podkapitola čerpá z internetovej stránky nodejs.dev. [18]

Návrhový vzor Promise (preložené sľub) je konštrukcia, ktorej výsledok momentálne nie je známy. Táto konštrukcia bola oficiálne zavedená v špecifikácií ECMAScript 2015. Jedná sa o proxy objekt, ktorý v jazyku JavaScript má tri stavy: čakajúci, dokončený a ukončený s chybou. Konštruktor objektu Promise má jeden argument. Argumentom je funkcia, ktorá prijíma dva argumenty objektov funkcií, ktoré sa väčšinou nazývajú resolve a reject (v preklade vyrieš a zamietni). Keď kód zavolá ako prvú funkciu „vyrieš“, objekt Promise vráti hodnotu, ktorá bola uvedená vo funkcii. V opačnom prípade, keď sa zavolá prvá funkcia „zamietni“, objekt Promise vráti chybu s dôvodom, ktorý bol vložený do funkcie „zamietni“.

Krátko po vytvorení Promise objekt nadobúda stav čakajúci. Následne objekt čaká na prevzatie kontroly spúšťania od cyklus udalostí. Keď objekt prevezme kontrolu, spustí sa kód vo funkcií, ktorá bola vložená do konštruktoru. Podľa prvej funkcie, ktorá bola zavolaná sa vyhodnotí objekt Promise.

S vyhodnoteným objektom Promise je možné ďalej pracovať. Objekt Promise obsahuje tri metódy: then, catch, finally. Metóda then prijíma argument funkciu, ktorá sa zavolá po úspešnom vyhodnotení objektu Promise. Metódy then je možné reťaziť, čo eliminuje predošlý problém spätných volaní. Metóda catch ako aj metóda then prijíma ako argument funkciu. Táto funkcia sa zavolá pri neúspešnom ukončení a tiež je možné ju reťaziť. V poslednom prípade metóda finally, ktorá taktiež prijíma ako argument funkciu. Metóda finally sa zavolá vždy bez ohľadu či bol objekt Promise úspešne vyriešený alebo zamietnutý.

Návrhový vzor Promise vyriešil predošlý problém spätných volaní a používa sa vo väčšej miere v programovacom jazyku JavaScript aj dodnes. V najnovšej verzii JavaScript sa nepriamo používajú v async funkciách. Na nasledujúcej ilustrácii číslo 4.4 je možné vidieť ukážku prepísaného kódu so spätnými volaniami na návrhový vzor Promise.

```
/** Funkcie divide, add, subtract a multiply reprezentuje matematicke operatory
 * /, +, - a * ako v minulom priklade. Avsak v tomto priklade funkcie
 * prijimaju ciselne argumenty a vracaju objekt typu Promise.
 * Program na vypocet vyrazu 5 + 10 / 2 - 3 * 9.
 */

// vypocitaj lavu stranu vyrazu
const left = divide(10, 2)
    .then((result) => add(result, 5)) // skrateny zapis funkcie
    .catch((e) => console.error(e));

// vypocitaj pravu stranu vyrazu
const right = multiply(-3, 9).catch((e) => console.error(e));

// pockaj na vypocet vsetkych vyrazov
Promise.all([left, right])
    .then(([leftExpression, rightExpression]) => add(leftExpression, rightExpression))
    .then((result) => console.log(`Vysledok je ${result}`))
    .catch((e) => console.error(e));
```

Obr. 4.4: Ukážka riešenia problému javu „callback hell“ pomocou objektu Promise

Moderné asynchrónne programovanie s kľúčovými slovami async/await

V špecifikácií ECMAScript 2017 sa zaviedli nové kľúčové slová zamerané na asynchrónne programovanie: async a await (async ako skratka od slova asynchrónny a await v preklade čakať). Návrhový vzor Promise hoci vyriešil problémy s vnorením spätných volaní, avšak reťazené volanie Promise objektov malo tiež svoje problémy s prehľadnosťou kódu. Z tohto dôvodu vyšli nové kľúčové slová async (označuje asynchrónnu funkciu) a await, ktorý pasívne čaká na ukončenie objektu Promise úspechom alebo neúspechom (pri neúspechu nastane výnimka, ktorá musí byť ošetrená výrazom *catch*). Funkcie označené kľúčovým slovom async implicitne vracajú ako výsledok objekt Promise. Avšak pri príkaze return návratový typ nemusí byť nutne typu Promise. Návratový typ sa automaticky konvertuje. [16]

Táto zmena výrazne zľahčila opravovanie kódu, keďže debug nástroje podporujú akciu vkroč do async funkcie. Taktiež táto zmena zvýšila prehľadnosť asynchrónneho kódu a zlepšila spracovávanie chýb v objektoch Promise. Ukážka modernej syntaxe async/await je demonštrovaná na obrázku číslo 4.5. [16]

```
/** Funkcie divide, add, subtract a multiply reprezentuju matematicke operatory
 * /, +, - a * ako v minulom priklade. Avsak v tomto priklade funkcie
 * prijimaju ciselne argumenty a vracaju objekt typu Promise.
 * Program na vypocet vyrazu 5 + 10 / 2 - 3 * 9.
 */

// Klucove slovo await je mozne pouzit len v async funkcii
async function main() {
    // Vypocitaj oba vyrazy paralelne.
    const middleDivide = divide(10, 2);
    const rightMultiply = multiply(-3, 9);

    // Pockaj na vypocet 10 / 2.
    const leftExpressions = add(5, await middleDivide);
    // Pockaj na vypocet vsetkych vyrazov a nasledne vypis vysledok.
    console.log(await add(await leftExpressions, await rightMultiply));
}

main();
```

Obr. 4.5: Ukážka prepísaného príkladu z ilustrácie číslo 4.4 na moderný štandard programovacieho jazyku JavaScript

4.3 Cyklus udalostí

V úvode podkapitoly o asynchrónnom programovaní (podkapitola číslo 4.2), bolo spomenuté, že JavaScript je asynchrónne orientovaný jazyk s jedným vláknom v jednom kontexte. Z tohto poznatku vyplýva, že v danom okamihu sa môže spúšťať iba jedna časť programu. Tento prístup má svoje výhody a nevýhody. Výhodou je intuitívnejšie písanie asynchrónneho kódu, pretože programátor prakticky riadi prepínanie kontextu v rámci programu (procesor má stále na starosti prepínanie kontextu, avšak v modeli programovacieho jazyku JavaScript nie je možné tento efekt spozorovať). Avšak technika cyklu udalostí prináša značnú nevýhodu. Pri nesprávnom chápaní cyklu udalostí sa môže stať, že programátor tento cyklus zablokuje. Zablokovanie cyklu spôsobí, že naplánované úlohy v ďalších čas-

tiach cykloch nebudú nikdy spustené. To môže mať vážne následky, ak nedôjde k včasnému spusteniu dôležitej úlohy. [17]

Táto podkapitola je zameraná na uvedenie čitateľa do problematiky plánovania úloh v prostredí NodeJS. Nasledujúci text podkapitol čerpá zo zdroja [17], pokiaľ nie je uvedené inak.

Zásobník volania funkcií

Volanie funkcií v programovacom jazyku JavaScript sa riadi podľa zásobníka volania. Zásobník volania je štruktúra, ktorá ukladá prvky a podporuje dve operácie. Prvou operáciu je vloženie prvku na vrch zásobníka. To pri vložení viacerých prvkov spôsobí, že prvý vložený prvok bude na dne zásobníka a posledný vložený prvok na vrchu zásobníka. Naopak, pri druhej operácií, ktorá je vyberanie prvkov zo zásobníka, sa vyberie prvý prvok z hornej časti zásobníku.

V nasledujúcej ukážke bude znázornené ako funguje zásobník volania. V ilustrácií číslo 4.6 sa nachádza zdrojový kód programu. V ilustrácií 4.7 sa nachádzajú stavy zásobníka pred a po volaní funkcií zdrojového kódu.

```
/**
 * Program na vypocet vyrazu 10 / 5. Program pred vypocitanim vyrazu
 * napise hodnoty operandov na prikazovom riadku.
 */
function divide(a, b) {
    console.log(`Dividing ${a} by ${b}`);
    return a / b;
}

function main() {
    const result = divide(10, 5);
    console.log(`10 / 5 = ${result}`);
}

main()
```

Obr. 4.6: Ukážka kódu, ktorý prejde rozborom volania zásobníku

Obr. 4.7: Stavy zásobníka počas vykonávania programu. Pri volaní novej funkcie sa funkcia pridá na vrch zásobníka. Pri návrate z funkcie je daná funkcia odobratá zo zásobníka.

Na základe ukážky je možné usúdiť, že daný model volania funkcií nemôže fungovať asynchrónne. Aby bolo možné podporovať asynchrónne volania v programovacom jazyku JavaScript, bol vytvorený poradovník správ (anglicky sa tento pojem označuje „message queue“).

Poradovník správ

Poradovník správ je štruktúra, ktorá obsahuje funkcie (správy), ktoré sú pripravené na spracovanie programom (napríklad pri stisku tlačidla elementu HTML). Správy sa spracovávajú v takom poradí, v akom boli vložené. Avšak je nutné podotknúť, že správy vytvorené objektom Promise v rámci jednej funkcie, respektíve modulu, majú väčšiu prioritu než obyčajné správy. Ku spracovávaniu správ dochádza, keď zásobník volaní je prázdny.

Do poradovníka správ je možné vkladať vlastné funkcie (správy) volaním jednej z uvedených funkcií:

- setTimeout – Funkcia setTimeout sa používa, keď programátor chce zavolať funkciu s oneskorením. Daná funkcia prijíma dva argumenty: objekt funkcie a časová hodnota minimálneho zdržania. Ak je hodnota minimálneho zdržania rovná nule, funkcia sa spustí v nasledujúcom cykle. [14]

- setImmediate – Funkcia setImmediate sa správa podobne ako v prípade volania funkcie setTimeout s nulovou hodnotou oneskorenia. Avšak použiť funkciu v tomto prípade je efektívnejšie, pretože sa interne nemusí kontrolovať čas oneskorenia. Rozdielov medzi setTimeout a setImmediate je viac, avšak detailný výklad pre túto prácu nie je potrebný. Konkrétnejšie informácie je možné nájsť v dokumentácií knižnice libuv[2]. [20]

- process.nextTick (výhradne v NodeJS) – Funkcia process.nextTick, ktorá sa nachádza iba v prostredí NodeJS, sa chová odlišne ako dve predošlé funkcie. Funkcia nextTick

[2]Dokumentácia libuv – http://docs.libuv.org/en/v1.x/design.html

na rozdiel od spomenutých funkcií spustí funkciu na konci aktuálneho cykly. Funkcia taktiež nemá nastavené limity početnosti spúšťania. Nesprávnym používaním funkcie nextTick je možné zablokovať cyklus udalostí. [19]

Spomenuté funkcie hrajú významnú rolu pri asynchrónnom programovaní v programovacom jazyku JavaScript. Ako bolo spomenuté v úvode tejto podkapitoly číslo 4.2, programátor má na starosti prepínanie kontextu. K prepínaniu kontextu dochádza, keď programátor zavolá jednu z funkcií zo zoznamu (pridá správu do poradovníka správ) a zaistí, že zásobník volaní bude prázdny. Po splnení oboch podmienok, interpreter začne spracovávať správy v poradovníku správ.

Technika prepínania kontextu sa využíva pri asynchrónnych úlohách, ktoré sú časovo náročné. Respektíve, ak by vykonávanie určitej úlohy zablokovalo cyklus udalostí na dlhší čas. Zablokovaný cyklus udalostí je kritická chyba, ktorej by sa mal programátor vyhýbať. V nasledujúcej ilustrácii číslo 4.8 je znázornený nesprávny spôsob asynchrónneho programovania a v ilustrácii pod číslom 4.9 je ukážka správneho asynchrónneho programovania.

```
let value = 5;

const valueChanged = new Promise((resolve) => {
    while(value === 5)
        /* no-op */;
    resolve();
});

new Promise((resolve) => {
    value = 10;
});

valueChanged.then(() => console.log(`value !== ${value}`));
```

Obr. 4.8: Ukážka nesprávnej manipulácie s cyklom udalosti. Najprv dôjde ku priradeniu hodnoty päť premennej *value*. Následne sa vytvoria objekty typu Promise. Na poslednom riadku dochádza k registrácií spätného volania pri úspešnom dokončení objektu Promise. V aktuálnej situácií je zásobník volania prázdny a poradovník správ obsahuje dve správy. Prvou správou je anonymná funkcia objektu *valueChanged*. Z poradovníka sa vyberie prvá správa a spustí sa príkaz *while(value === 5);*, ktorý spôsobí nekonečný cyklus. Dochádza ku blokácií cyklu udalostí, a teda k celkovej blokácií programu.

```
let value = 5;

function checkValue(resolve) {
    if(value === 5) {
        // Tento cyklus kontrola neprebehla uspesne, naplanuj volanie na dalsi cyklus.
        setImmediate(checkValue, resolve);
    }
    else {
        // Kontrola prebehla uspesne, objekt Promise je mozne vyriesit.
        resolve();
    }
}

const valueChanged = new Promise((resolve) => checkValue(resolve));

new Promise((resolve) => {
    value = 10;
    resolve();
});

valueChanged.then(() => console.log(`value !== ${value}`));
```

Obr. 4.9: Ukážka správnej manipulácie s cyklom udalostí. Najprv dôjde ku priradeniu hodnoty päť premennej *value*. Následne dochádza ku deklarácii funkcie *checkValue* a vytvoria sa objekty typu Promise. Na poslednom riadku dochádza k registrácií spätného volania pri úspešnom dokončení objektu Promise. V aktuálnej situácií je zásobník volania prázdny a poradovník správ obsahuje dve správy. Prvou správou je anonymná funkcia objektu *valueChanged*. Z poradovníka sa vyberie prvá správa a spustí sa príkaz *checkValue(resolve)*. Prvá podmienka funkcie *checkValue* sa vyhodnotí ako pravdivá a dochádza k preplánovaniu funkcie na ďalší cyklus. Funkcia v danom momente končí, zásobník volania je opäť prázdny a poradovník správ obsahuje 2 prvky. Avšak teraz sa na prvom miesta nachádza objekt Promise, ktorý má za úlohy zmeniť hodnotu premennej *value* na hodnotu desať. Dochádza ku zmene hodnoty spomínanej premennej a poradovník volania obsahuje len jednu správu. Posledná správa sa odstráni zo zoznamu a opäť sa vykoná kontrola premennej *value*. Tentokrát sa prvá podmienka nesplní a objekt Promise sa úspešne vyhodnotí. Vyhodnotený objekt Promise vloží novú správu do poradovníka, ktorá je spätným volaním na funkciu *then* na poslednom riadku.

V programovacom jazyku JavaScript je nesmierne dôležité vedieť správne manipulovať s cyklom udalostí. Nesprávnou prácou s cyklom udalostí môže dochádzať ku kritickým chybám, pri najlepšom dôjde len ku blokácií cyklu udalostí.

Súčasťou tejto podkapitoly boli aj dve ukážky práce s cyklom udalostí. V prvej ukážke číslo 4.8 došlo k uviaznutiu cyklu pri volaní príkazu *while*. Príkazy riadenia toku ako sú *while* a *for*, by mali zvýšiť ostražitosť každého programátora. Programovací jazyk JavaScript je implicitne jednovláknový, čo znamená, že spomínane príkazy riadenia toku programu majú potenciál zablokovať celé vlákno. Riešenie tohto problému je znázornené na ilustrácii číslo 4.9, na ktorej je problém vyriešený rekurzívnym volaním. Rekurzívne volanie je realizované prostredníctvom poradovníka správ.

Kapitola 5

Zhodnotenie súčasného stavu a návrh projektu

V aktuálnej dobe je možné testovať rozšírenia editora Che-Theia podobným spôsobom, akým sa testovali rozšírenia editora Visual Studio Code. Spôsob testovania rozšírení editora Visual Studio Code je detailnejšie vysvetlený v kapitole číslo 3.

Integračné testovanie sa v editore Che-Theia vykonáva nasledujúcim spôsobom. Najprv je potrebné vytvoriť rozšírenie špeciálne zamerané na spúšťanie testov. Testy musia byť spustené z rozšírenia, aby bolo možné používať aplikačné programátorské rozhranie rozšírení editora Che-Theia. Pre toto testovacie rozšírenie musí byť vytvorený súbor *meta.yml*. Následne súbor *devfile.yml* sa musí odkazovať na testované rozšírenie a na rozšírenie, ktoré spúšťa testy. Súbor *devfile.yml* sa musí v atribúte *projects* odkázať na repozitár so zdrojový kódom testov. [3]

Tento typ testovania rozšírení má podobné problémy ako integračné testovanie rozšírení editora Visual Studio Code. Integračné testy nekontrolujú funkcionalitu rozšírenia, ale len jeho integráciu s aplikačným programátorským rozhraním rozšírení daného editora.

5.1 Zhodnotenie testovacieho nástroja VS Code Extension Tester

Testovací nástroj VS Code Extension Tester je naprogramovaný v programovacom jazyku TypeScript. Väčšina rozšírení pre editor Visual Studio Code je taktiež naprogramovaná v programovacom jazyku TypeScript, čo umožňuje bezproblémové zdieľanie zdrojové kódu rozšírení s testovacím kódom.

V nasledujúcich podkapitolách budú zhodnotené jednotlivé vlastnosti testovacieho nástroja VS Code Extension Tester.

Použiteľnosť nástroja

Testovací nástroj VS Code Extension Tester vykonáva v odvetví prípravy testovacieho prostredia excelentnú prácu. Nástroj dokáže automaticky stiahnuť akúkoľvek dostupnú verziu editora Visual Studio Code a prehliadačový ovládač pre internetový prehliadač Google Chrome. Pred spustením testov nástroj dokáže zabaliť rozšírenie do archívu, ktorý sa nainštaluje do testovacieho editora. Po skončení testov sa rozšírenie z editora odstráni.

Ďalšou silnou stránkou testovacieho nástroja je spôsob, akým distribuuje programové závislosti. Testovací modul VS Code Extension Tester sa skladá z troch hlavných modulov. Hlavný modul, ktorý obsahuje časť kódu, ktorá riadi ostatné moduly. Druhý modul obsahuje vyhľadávacie výrazy, ktoré sa väčšinou menia jedenkrát za mesiac. Tretí modul obsahuje v sebe implementáciu komponentov editora Visual Studio Code, ktoré sa menia v častých intervaloch. Tieto posledné dva moduly sú distribuované ako tranzitívne závislosti, čo ma za následok, že sa aktualizácie chýb nainštalujú bez intervencie používateľa modulu.

Spoľahlivosť testov

Testovací nástroj vykazuje vysokú mieru spoľahlivosti. Falošné výsledky testov spôsobené anomáliami grafického rozhrania editora sa vyskytujú zriedka. Jeho spoľahlivosť však klesá pri strojoch s obmedzenými systémovými prostriedkami ako sú napríklad stroje určené na priebežnú integráciu (v angličtine sa bežne používa skratka „CI“). Čím je stroj menej výkonný, tým častejšie dochádza ku anomáliám v grafickom rozhraní.

Daný problém by bolo možné vyriešiť za bežných podmienok implicitnými čakaniami knižnice Selenium. Knižnica Selenium WebDriver podporuje implicitné čakania, ale v tomto prípade tohto nástroja to nie možné. Jednak niektoré časti kódu nekontrolujú prítomnosť elementov HTML odporúčaným spôsobom. Tento nedostatok by sa dal vyriešiť úpravou kódu, ale druhý problém je závažnejší. Vývojári Selenium WebDriver neodporúčajú kombinovať implicitné a explicitné čakania, pretože sa tieto čakania navzájom ovplyvňujú. Kombinovanie rôznych druhov čakaní spôsobuje nečakané chovanie programu. [27] Explicitné čakania v nastrojí VS Code Extension Tester sú potrebné a nie je ich možné odstrániť zo zdrojového kódu.

5.2 Návrh testovacieho nástroja Theia Extension Tester

Najhlavnejšou prioritou tejto práce je vytvoriť nový testovací nástroj na testovanie rozšírení v editore Che-Theia. V budúcnosti by mal testovací nástroj podporovať všetky editory, ktoré sú postavené na platforme Eclipse Theia.

Hlavnou cieľovou skupinou testovacieho nástroja sú vývojári, ktorí chcú podporovať svoje rozšírenia editora Visual Studio Code aj v editore Che-Theia. Zdrojové kódy funkcionálnych testov musia byť kompatibilné s testami vytvorené pre nástroj VS Code Extension Tester. Nový testovací nástroj musí odraďovať ľudí od používania detekčných algoritmov testovaného editora (testy by nemali diskriminovať medzi editormi). Niektoré prípady detekcie testovaného editora môžu byť prijateľným prípadom využitia, preto túto detekciu bude možné vykonať (vhodným prípadom využitia je napríklad pri riešení problémov nekompatibilít medzi editormi vo vysoko úrovňovej knižnici).

Analýza použiteľnosti nástroja

Integrované programovacie prostredie Eclipse Che musí byť nasadené v prostredí Kubernetes. Prostredie Kubernetes vyžaduje značnú časť systémových prostriedkov a miesta na disku. Integrované programovacie prostredie Eclipse Che je taktiež možné nasadiť v rôznych módoch, ktoré je možné nájsť v podkapitole číslo 2.1. Prostredie Kubernetes má taktiež mnoho distribúcií. Z týchto poznatkov vyplýva, že nie je praktické a ani vhodné automatizovane

pripraviť testovacie prostredie, ako je to v prípade testovacieho nástroja VS Code Extension Tester.

V odvetví prípravy testovacích ovládačov (Selenium WebDriver) je možné zaistiť podobnú funkcionalitu ako testovací nástroj VS Code Extension Tester. V prípade testovacieho nástroja VS Code Extension Tester bolo nutné pripraviť jediný ovládač pre internetový prehliadač Google Chrome. Avšak v tejto práci bude nutné spravovať viacero testovacích ovládačov, pretože editor Che-Theia je primárne založený na webovom rozhraní, čo znamená, že rozhranie je možné používať na viacerých internetových prehliadačoch.

Integrované programovacie prostredie Eclipse Che môže od užívateľa vyžadovať autentifikáciu. Autentifikačné formuláre sa môžu v rôznych distribúciách prostredia Kubernetes od seba líšiť, alebo nemusia byť vôbec používané. Na základe tohto problému je nutné definovať spoločné aplikačné programovacie rozhranie pre autentifikáciu používateľov. V tejto práci je naprogramovaný iba autentifikačný mechanizmus pre prostredie OpenShift[1].

Po autentifikácii používateľa testovací nástroj musí vytvoriť pracovné prostredie určené v konfigurácii spúšťača. Pracovné prostredie je v konfigurácii uvedené formou adresy URL, ktorá sa odkazuje na súbor devfile.yml.

Analýza vhodnosti technológií a integrácie nového testeru

Testy napísané pre testovací nástroj VS Code Extension Tester používajú programovací jazyk TypeScript, knižnicu Selenium WebDriver a testovaciu knižnicu Mocha. Pre dôvody zachovania kompatibility a zavedených zvykov pri vyvíjaní rozšírení pre editor Visual Studio Code, budú v práci použité spomenuté technológie.

Testovací nástroj definuje a zároveň implementuje aplikačné programovacie rozhranie komponentov editora Visual Studio Code. Bohužiaľ v dobe písania tejto práce, čistá definícia bez implementácie aplikačného programovacieho rozhrania nebola dostupná. Z tohto dôvodu bolo nutné vytvoriť spoločný modul s definíciou aplikačného programovacieho rozhrania a upravený modul testovacieho nástroja VS Code Extension Tester.

Predpokladá sa, že pri vytváraní nového testovacieho nástroja bude nutné pozmeniť aplikačné programovacie rozhranie spoločného modulu. Ak sa prevedené zmeny v aplikačnom programovacom rozhraní osvedčia ako prínosné aj pre testovací nástroj VS Code Extension Tester, bude možné tieto zmeny integrovať do originálneho modulu.

Analýza rizík

Najväčším rizikom pre túto prácu predstavujú potenciálne nekompatibility medzi prostrediami. Pod prostredím sa myslí v editore Visual Studio Code operačný systém, v prípade editora Che-Theia internetový prehliadač. V testoch napísaných pre editor Visual Studio Code je možné používať systémové knižnice. Pri testovaní rozšírení Che-Theia nie je možné používať systémové knižnice, pretože kód rozšírenia nie je spustený na rovnakom systéme ako testovací kód. Pre tento prípad užitia bolo nutné pridať novú funkcionalitu do aplikačného programovacieho rozhrania.

Ďalšie riziko predstavuje nekompatibilita niektorých komponentov medzi editormi. Nekompatibilné operácie nad komponentmi, ktoré nie je možné opraviť, by mali vyvolať výnimku oznamujúcu túto skutočnosť. Ostatné nekompatibility musia byť opravené kódom, ktorý zaručí, že nastanú úkony ako pri editore Visual Studio Code, alebo definovaním novej funkcionality komponentu do aplikačného programovacieho rozhrania.

[1] https://www.openshift.com/

Posledným pozorovaným rizikom je menší výpočtový výkon inštancií editora Che-Theia v porovnaní s editorom Visual Studio Code. V podkapitole číslo 5.1 sa uvádzalo, aký negatívny dopad má nižší výpočtový výkon na stabilitu testov. Tento problém sa bude u integrovaného programovacieho prostredia Eclipse Che objavovať konzistentnejšie. Na základe týchto uvážení je nutné navrhnúť nejaký čakací mechanizmus, ktorý bude predstavovať hybrid medzi implicitnými a explicitnými čakaniami v knižnici Selenium WebDriver.

Rozsah implementovaného aplikačného programovacieho rozhrania

V prvej iterácii tohto projektu budú naprogramované najdôležitejšie komponenty, ktoré sú potrebné na úspešné vykonanie testov grafického rozhrania rozšírenia vscode-quarkus[2]. Rozšírenie vscode-quarkus bolo vybraté kvôli jeho funkcionalite, ktorá pokrýva najpoužívanejšie komponenty oboch editorov. Naprogramovanie všetkých komponent by bolo možné v časovom rozmedzí tejto práce, ale nebolo by možné zaručiť stabilitu komponentov.

Organizácia kódu

Kód je organizovaný do viacerých repozitárov, ktoré sú uvedené v nasledujúcom zozname:

- extension-tester-page-objects[3] – Modul poskytuje nástroju VS Code Extension Tester a nástroju tejto práce spoločné definície komponentov a ich aplikačného programátorského rozhrania. Tento modul bude taktiež slúžiť aj ako knižnica, ktorá bude obsahovať funkcie užitočné pre všetky testovacie nástroje.
- theia-extension-tester[4] – Modul obsahuje implementáciu aplikačného programovacieho rozhrania definované predošlým modulom *extension-tester-page-objects*, databázu vyhľadávacích výrazov, spúšťače testov, pomocné funkcie a pomocné objekty.
- vscode-extension-tester[5] – Modul obsahuje implementáciu novej funkcionality aplikačného programovacieho rozhrania, ktorá vznikla nekompatibilitou editorov a prostredí.

[2] https://github.com/redhat-developer/vscode-quarkus
[3] https://github.com/mlorinc/extension-tester-page-objects
[4] https://github.com/mlorinc/theia-extension-tester
[5] https://github.com/mlorinc/vscode-extension-tester

Kapitola 6

Implementácia

Nasledujúci obsah tejto kapitoly bude zameraný na uvedenie čitateľa do problematiky implementácie nástroja. Prvé dve podkapitoly sú zamerané na problematiku riadiaceho toku programu. Za podkapitolami o toku programu nasledujú podkapitoly zamerané na riešenie problémov z návrhu testovacieho nástroja. Posledné podkapitoly oboznámia čitateľa o špecifických významných komponentoch tejto práce.

6.1 Vstupný bod programu

Vstupný bod programu je časť programu, ktorá má na starosti inicializáciu prehliadačovho ovládača Selenium WebDriver, inicializáciu autentifikačného objektu pre integrované programovacie prostredie Eclipse Che a spúšťanie testov[1]. Autentifikačný objekt nie je povinný a môže byť vynechaný.

Používateľ testovacieho nástroja môže pred vytvorením ovládača Selenium WebDriver použiť knižnicu na stiahnutie ovládačov. Knižnica na sťahovanie ovládačov je súčasťou modulu extension-tester-page-objects. Momentálne knižnica podporuje len sťahovanie nasledujúcich ovládačov: chromedriver, geckodriver a operadriver. Ovládač internetového prehliadača Safari[2] nie je uvedený v tomto zozname, pretože ovládač nie je možné stiahnuť. Internetový prehliadač Microsoft Edge[3] nie je podporovaný, pretože testovací nástroj momentálne nepodporuje operačný systém Windows.

Autentifikačné objekty musia implementovať rozhranie Authenticator, ktoré obsahuje metódu authentificate. Metóda authentificate neprijíma žiadne parametre. Parametre autentifikačného objektu je možné nastaviť v konštruktore triedy, čo umožňuje vývojárom mať väčšiu voľnosť pri vytváraní autentifikačných objektov. Tento návrh bol ovplyvnený tým, že integrované programovacie prostredie Eclipse Che podporuje viacero autentifikačných mechanizmov, ktoré sa môžu medzi sebou podstatne líšiť.

6.2 Ovládač Selenium WebDriver

Ovládač Selenium WebDriver je spravovaný triedou BaseBrowser. Trieda BaseBrowser mimo spravovania ovládača Selenium WebDriver, má aj na starosti nastavovanie časových limitov na vyhľadávanie komponentov a nastavenie kontextu ovládača Selenium WebDriver.

[1] Nemusia to byť striktne testovacie programy.
[2] https://www.apple.com/safari/
[3] https://www.microsoft.com/sk-sk/edge

Objekt BaseBrowser v sebe uchováva premennú *_findElementTimeout*, ktorá určuje po akú dobu sa budú komponenty pokúšať vykonať nejakú operáciu. Podobný účel má aj premenná *_pageLoadTimeout*, ktorá sa používa pri prvotných fázach načítavania pracovného prostredia v integrovanom programovacom prostredí Eclipse Che. Ak užívateľ tieto premenné nedefinuje v konfigurácií, implicitne nadobúdajú hodnotu nula. Hodnota nula v časovači znamená, že sa daná akcia vykoná jedenkrát.

Proces inicializácie ovládača

Proces inicializácie ovládača je možné rozdeliť do viacero etáp. V prvej etape dochádza ku konfigurácii testovacieho nástroja. Konfigurácia testovacieho nástroja podporuje nasledujúce položky:

- Browser name – Meno podporovaného prehliadača[4].
- Distribution – Integrované programovacie prostredie Eclipse Che je voľne šíriteľný softvér, čo vývojárom dáva slobodu vytvoriť si vlastnú distribúciu. Aktuálne sú podporované distribúcie Eclipse Che a CodeReady Workspaces[5]. Medzi nepodporované, do budúcnosti rezervované distribúcie patria: Eclipse Theia browser a Eclipse Theia Electron. Tieto distribúcie sú špeciálne vyhradené pre editory Eclipse Theia. Na základe nastavenej hodnoty tejto premennej sa vytvorí inštancia príslušného objektu BaseBrowser, ktorá môže nadobúdať jeden z nasledujúcich typov: CheTheiaBrowser, TheiaBrowser a TheiaElectronBrowser.
- Browser location – Cesta ku binárnemu súboru internetového prehliadača.
- Driver location – Cesta ku binárnemu súboru prehliadačového ovládača.
- Clean session – Spustenie internetového prehliadača s vyčistenou pamäťou.
- Log level – Premenná určujúca, aké správy má ovládač ukladať do záznamov.
- Timeouts – Hodnoty časovačov.

V druhej etape dochádza ku vytvoreniu prehliadačového ovládača Selenium WebDriver. Prehliadačový ovládač je predvolene nastavený, aby prijímal neznáme certifikáty. V prípade všetkých podporovaných prehliadačov okrem prehliadača Safari je definovaný profil používateľa, aby sa pri každom testovaní nemuseli sťahovať súbory editora.

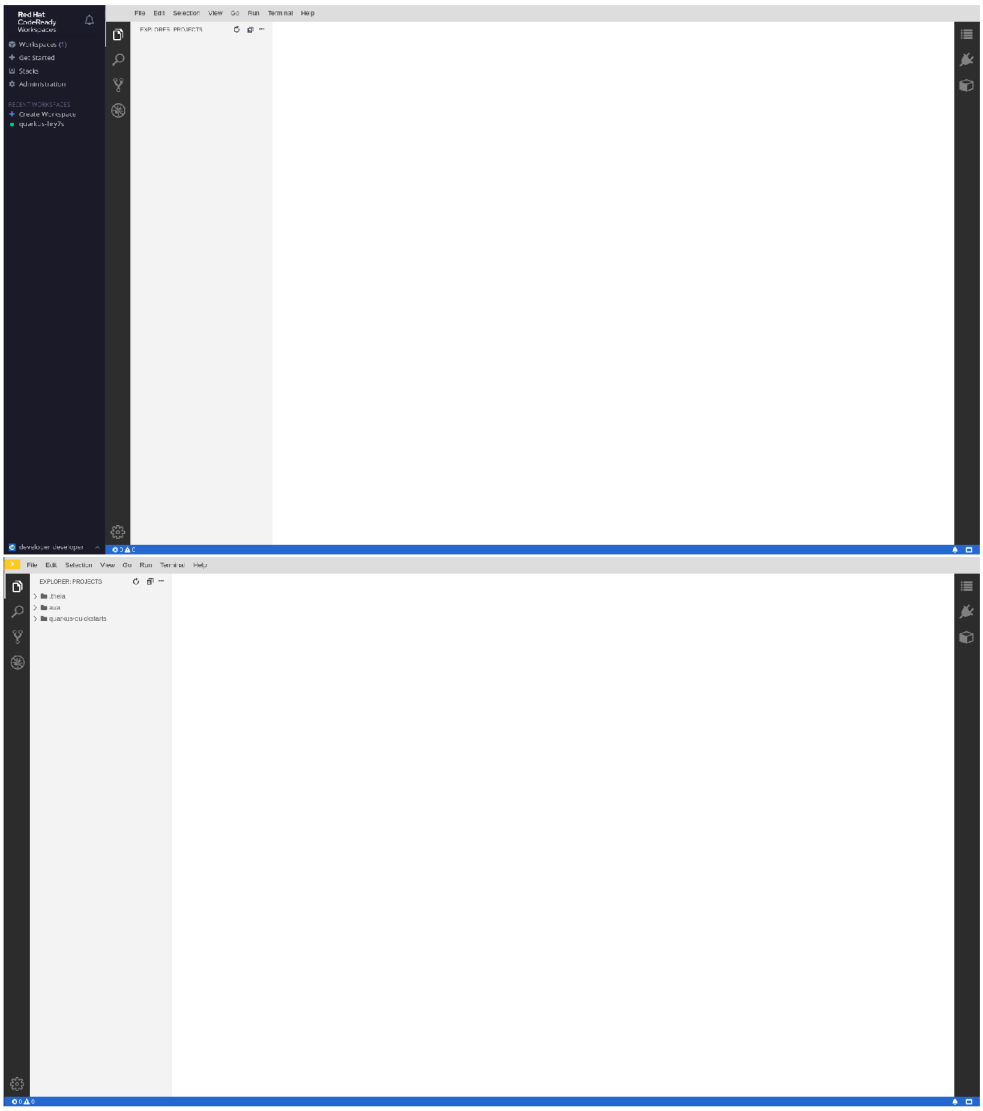
V tretej etape už je okno prehliadača otvorené. Avšak nie je predvolene otvorené v maximalizovanom režime, a preto je potrebné ho maximalizovať. Po maximalizovaní okna dôjdu ku načítaniu vyhľadávacích výrazov do počítačovej pamäte.

Príprava testovacieho prostredia

Príprava testovacieho prostredia sa vykonáva metódou waitForWorkbench, ktorá sa nachádza v triede BaseBrowser. Táto metóda nie je volaná inštanciou triedy BaseBrowser. Väčšinou sa táto metóda volá v spúšťačoch testov[6], pretože to dáva spúšťačom väčšiu kontrolu nad testovaním.

[4] https://www.selenium.dev/documentation/en/getting_started_with_webdriver/browsers/
[5] https://developers.redhat.com/products/codeready-workspaces/overview
[6] Spúšťač testov sa myslí objekt typu TestRunner, ktorý obsahuje jednu metódu *runTests*.

Úlohou tejto metódy je zaistiť, že editor Che-Theia je načítaný. Editor Che-Theia je v štruktúre HTML súčasťou elementu iframe, čo znemožňuje testovanie rozšírenia ovládačom Selenium WebDriver. Z tohto dôvodu metóda najprv čaká na spomenutý element iframe. Po nájdení elementu iframe metóda zmení kontext ovládača na kontext v elemente iframe. Akonáhle dôjde ku úspešnej zmene kontextu, metóda čaká na koniec načítavania editora Che-Theia. Po skončení načítavania, metóda čaká, pokiaľ žlté tlačidlo na hornej ľavej strane obrazovky sa posunie na ľavý koniec obrazovky. Ukážka pohybu tlačidla je možné vidieť na ilustrácii číslo 6.1.



Obr. 6.1: Na snímke obrazovky sa nachádza nepripravené pracovné prostredie. V tejto dobe žlté tlačidlo ešte nie je inicializované. V tomto stave pracovné prostredie nezostane dlho, ľavý panel sa začne zmenšovať. V tejto fáze by bolo možné spustiť testy, ale pohyb celého grafického rozhrania by negatívne ovplyvnilo testy, ktoré vykonávajú úkony na grafických komponentoch prostredníctvom počítačovej myši. Na dolnej snímke obrazovky je pracovné prostredie pripravené a stabilné.

6.3 Riešenie problémov spojené s výpočtovým výkonom

Ako bolo spomínané v podkapitole číslo 5.2, spoľahlivosť testov klesá neúmerne s výpočtovým výkonom. Tento problém sa zvykne väčšinou prejavovať chybami ako sú napríklad: StaleReferenceError (neplatný odkaz na element), chybovou hláškou „Unable to locate element“ (nebolo možné nájsť element) alebo chybovou hláškou „Element is not interactable“ (element nie je interaktívny).

V nasledujúcich podkapitolách bude vysvetlené ako boli spomenuté problémy vyriešené. Avšak riešenie týchto problémov vyžadovalo vytvorenie nových pomocných nástrojov, ktorým budú venované nasledujúce podkapitoly.

Trieda TimeoutPromise

Trieda TimeoutPromise vznikla na základe toho, že programovací jazyk JavaScript neobsahuje v sebe objekt Promise, ktorý by bolo možné časovo obmedziť. Vyvíjaný testovací nástroj intenzívne pracuje s úlohami, ktoré trvajú dlhší čas. Pri niektorých úlohách môže dôjsť ku vážnej chybe, ktorú nie je možné ľahko identifikovať. Z tohto dôvodu je vhodné úlohy časovo limitovať.

Trieda TimeoutPromise je objekt, ktorý rozširuje objekt Promise, ktorý bol detailnejšie vysvetlený v podkapitole číslo 4.2. Nová trieda pridáva novú funkcionalitu do objektu Promise ako je identifikátor objektu a vnútorný časovač.

Vnútorný časovač zamietne objekt Promise, ak sa do uvedenej doby časovača objekt Promise nevyrieši alebo nezamietne sám. Pri vypršaní časovača sa objekt Promise automaticky zamietne s chybovou hláškou, ktorá oznamuje vypršanie časovača. Ku chybovej hláške môže programátor doplniť vlastnú časť chybovej hlášky. Časovač nemôže byť nastavený na negatívnu hodnotu. Avšak časovač podporuje nulovú hodnotu a hodnotu *undefined*. Pri nulovom časovači sa objekt Promise zamietne, ak sa objekt Promise nevyrieši po jednom cykle udalostí, ktoré boli vysvetlené detailnejšie v pod kapitole číslo 4.3. V prípade časovača nastaveného na hodnotu *undefined*, objekt Promise nemá časové obmedzenie.

Funkcia repeat

Funkcia repeat plní rovnakú funkcionalitu ako funkcia wait z knižnice Selenium WebDriver. Funkcia wait prijíma ako prvý argument funkciu. Funkcia je opakovaná až pokiaľ nevráti výsledok, ktorý je v programovacom jazyku JavaScript interpretovaný ako pravdivý, alebo nevyprší časový limit. Interne funkcia wait používa na plánovanie ďalších cyklov funkciu setTimeout s predvolenou hodnotou dvesto milisekúnd (cyklus sa opakuje každých dvesto milisekúnd). Ďalšími parametrami funkcie wait sú: hodnota časovača a chybová hláška pri vypršaní časovača. [26]

V testovacom nástroji je niekedy nutné spúšťať funkciu opakovane s nulovým zameškaním, čo nie je optimálne (vysvetlené v podkapitole číslo 4.3). Funkcia wait neumožňuje prevádzkovať centrálne manažovanie konfigurácie časovačov. Funkcia wait pri časovači hodnoty nula opakuje funkciu pokým sa neukončí program [26]. Takéto správanie funkcie nie je v testovacom nástroji žiaduce, pretože pri nulovom časovači je praktické spustiť funkciu jedenkrát (užívateľ knižnice chce vykonať jednorazovú kontrolu).

Ďalším problémom funkcie wait je, že nie je možné použiť funkciu pri vyhľadávaní potomka rodičovského elementu v triede AbstractElement. Pôvodne sa používal mechanizmus čakania *driver.wait(until.elementLocated(...))*. Pri tomto mechanizme dochádzalo ku rekurzii a následne ku zlyhaniu celého programu.

Na vyriešenie predošlých uvedených problémov vznikla funkcia repeat, ktorá nepoužíva funkciu setTimeout, ale funkciu setImmediate. Funkcia repeat, takisto ako trieda TimeoutPromise má vlastný identifikátor určený na identifikačné účely. Časovač funkcie repeat funguje na rovnakom princípe ako časovač triedy TimeoutPromise. Novinkou je parameter *count*, ktorý slúži ako alternatíva ku časovaču. Parameter *count* určuje koľko cyklov sa bude funkcia repeat pokúšať dostať správny výsledok. Za správny výsledok sa berie akýkoľvek výraz v programovacom jazyku JavaScript, ktorý sa vyhodnotí ako pravdivý. Pri správnom výsledku funkcia repeat vracia hodnotu získanú v poslednom cykle, obalenú vo vyriešenom objekte Promise. Ak sa po uplynutí časovača alebo počítadla nepodarí získať správny výsledok, funkcia vracia zamietnutý objekt TimeoutPromise.

Posledným parametrom je parameter *threshold*, ktorý určuje po akú dobu musí byť neprerušovane vrátený správny výsledok za sebou, aby sa funkcia repeat zastavila. Tento parameter sa v praxi používa napríklad u vstupných poliach, ktoré majú na starosti validáciu textu. Validácia textu sa v editora Che-Theia spúšťa pri každom zadanom znaku. V jednu dobu validácie môže cyklus vrátiť správny výsledok, ale ten sa môže napríklad zmeniť počas priebehu 150 milisekúnd. Tento jav by spôsobil falošný správny výsledok a nesprávnu interpretáciu výsledku testovania.

Zaujímavosťou funkcie repeat je fragment kódu (fragment kódu je možné vidieť na ilustrácii číslo 6.2), ktorý sa na prvý pohľad môže zdáť ako kontraproduktívny, ale je veľmi užitočný, pretože zachováva zásobník volaní. V podkapitole číslo 4.3 bolo vysvetlené ako funguje zásobník volaní. Aby sa mohla prvá iterácia funkcie repeat vykonať, zásobník volaní musí byť prázdny. Z tohto dôvodu, ak dôjde ku chybe v cykle, používateľ nebude mať k dispozícii výpis zo zásobníka volaní, čo robí opravovanie chýb v kóde komplikovanejšie.

```
let callStack: string | undefined = undefined;
try {
    // Vyvolaj vynimku.
    throw new Error();
}
catch (e) {
    // Ziskaj retazec zasbonika volania.
    callStack = (e as Error).stack;
    // Naformatuj retazec zasbonika volania na citatelnjesiu podobu.
    callStack = callStack?.split('\n\t').join('\t\n');
}

return new TimeoutPromise<T>((resolve, reject) => {
    // Naplanuj prvy cyklus
    setImmediate(closure, count, resolve, reject, callStack);
}, timeout, {
    onTimeout: () => run = false,
    id: options?.id,
    message: options?.message,
    callStack
});
```

Obr. 6.2: Ukážka riešenia na zachovanie výpisu zásobníka volaní pri používaní funkcie repeat.

Komponent AbstractElement

Trieda AbstractElement je rodičovskou triedou všetkých komponentov testovacích nástrojov. Trieda AbstractElement rozširuje triedu WebElement z knižnice Selenium WebDriver o novú funkcionalitu zameranú na stabilizáciu procesu testovania. Ďalšou úlohou triedy AbstractElement je uchovávanie statických pomocných premenných, ktoré ukladajú nasledujúce položky: konštantu klávesy Ctrl[7], verziu internetového prehliadača a vyhľadávacie výrazy.

Na základe spomínaného problému stability testov, vysvetlený v pod kapitole 5.2, bol vo veľkej miere upravený tento komponent, ktorý bol pôvodne prebratý zo zdroja [24]. Princíp fungovania komponentu bude vysvetlený v nasledujúcich podkapitolách.

Konštruktor

Konštruktor triedy AbstractElement prijíma dva argumenty. Prvý argument určuje element, ktorý bude táto trieda reprezentovať. Ak je prvý argument typu WebElement, daný element sa nemusí vyhľadávať v štruktúre HTML. Na druhej strane ak je argument vyhľadávacím výrazom, konštruktor triedy inicializuje vyhľadávanie elementu. Druhý argument je rodič elementu, a teda určuje kontext vyhľadávania. Pri argumente rodičovského elementu platia podobné pravidla ako v prvom argumente. Argument rodičovského elementu je voliteľný. Pri nedefinovanom argumente rodičovského elementu sa implicitne berie ako rodič element s menom tagu html.

Vyhľadávanie elementov

Pri vyhľadávaní prvkov sa ako prvý nájde rodičovský element. Pri vyhľadávaní prvkov sa pomerne často vyskytovali chyby typu StaleReferenceError alebo „Nebolo možné lokalizovať element ...“. Tieto typy chýb robia testy nestabilnými, pričom je pomerne jednoduché zotaviť sa z týchto chýb. Na zotavenie z chyby väčšinou stačí opakovane vyhľadať element. Jediným prípadom, kedy nie je možné zotaviť sa z chyby je, keď rodič nie je reprezentovaný vyhľadávacím výrazom (element je typu WebElement). V tomto prípade už nie je odkaz na element platný a nemá zmysel pokračovať vo vyhľadávaní.

Po nájdení rodičovského elementu je na rade vyhľadávanie jeho potomka. Pri potomkovi platia rovnaké pravidla ako u rodiča, ale v tomto prípade je garantované, že argument potomka je vyhľadávací výraz.

Metódy určené na stabilizáciu testov

Trieda AbstractElement implementuje aj nové metódy, ktoré sú: safeClick, safeDoubleClick a safeSendKeys. Spomenuté metódy sú stabilnejšími alternatívami ako ich základne verzie bez prefixu safe (metóda doubleClick neexistuje). Tieto metódy zaistia, že dôjde ku interakcii s komponentom v momente, keď sú viditeľné, aktivované a ich operácia by sa nechtiac neaplikovala na iný komponent.

[7] Operačné systémy môžu používať inú klávesu (napr.: zariadenia Apple používajú klávesu Meta)

6.4 Riešenie problémov nekompatibility testov

V návrhu tejto práce boli uvedené dva druhy nekompatibilít testov (podkapitola číslo 5.2). V prvom prípade sa jedná o nekompatibilitu medzi komponentmi editorov. Tento druh nekompatibility je momentálne väčšinou vyriešený formou vyvolania chybového stavu, pretože u väčšine prípadov sa jedná o neriešiteľné problémy[8]. Medzi riešiteľné problémy patrí problematika otvárania súborov, ktorá je vysvetlená v nasledujúcej podkapitole. V poslednom prípade sa jedná o problematiku používania systémových knižníc pri testovaní rozšírení.

Problematika otvárania súborov medzi editormi

Medzi riešiteľné problémy patrí problém otvárania súborov v editore. Akcia na zadanie cesty súboru môže byť vyvolaná užívateľom alebo rozšírením editora. V nástroji VS Code Extension Tester je to predvolene vykonané formou natívneho dialógu, ale tento spôsob je zastaralý. V modifikovanej verzii nástroja je z tohto dôvodu používaný komponent InputBox. V nástroji tejto práce je otváranie súborov realizované formou modálneho dialógu. Pri oboch prípadoch sa používajú objekty, ktoré implementujú rozhranie IOpenDialog. Spomenuté rozhranie bolo vytvorené na základe dodržania kompatibility medzi editormi.

Problematika používania systémových knižníc

Systémové knižnice nie je možné používať pri testovaní rozšírenia integrovaného programovacieho prostredia Eclipse Che, pretože rozšírenie a testovací nástroj sa nenachádzajú na jednom stroji[9]. Medzi problémové knižnice patria *fs* a *child_process*, ktoré sú súčasťou štandardnej knižnice NodeJS.

Riešenie problému používania systémovej knižnice *fs* je súčasťou tejto práce. Problém bol vyriešený zavedením novej funkcionality do komponentu DefaultTreeSection. Komponent DefaultTreeSection reprezentuje v editoroch súborový strom. Cez grafické rozhranie súborového stromu je možné otvárať, zmazať a vytvárať nové súbory. Implementácia komponentu DefaultTreeSection je podrobnejšie vysvetlená v podkapitole 6.8.

Problém používania knižnice *child_process* nie je aktuálne vyriešený v tejto práci, pretože to rozšírenie vscode-quarkus nevyžadovalo a daná knižnica sa v testoch nepoužíva často. V budúcnosti sa však tento problém bude riešiť pomocou komponentu TerminalView. Zmena funkcionality komponentu TerminalView pravdepodobne nebude potrebná.

Pri programovaní testovacieho nástroja sa prišlo na ďalší problém. Testy pre rozšírenia Visual Studio Code zvyknú otvárať v editore priečinky, aby mohli vygenerovať nové súbory prostredníctvom rozšírení. V editore Che-Theia je implicitne otvorený priečinok */projects*, v ktorom sa nachádzajú všetky projekty definované v súbore devfile.yml. Tento problém sa napokon vyriešil pridaním novej možnosti do spúšťača testov v testovacom nástroji VS Code Extension Tester. Novou možnosťou je *openFolder*, ktorý spúšťač pred spustením testov otvorí v editore Visual Studio Code. Táto možnosť má aj ďalšiu dobrú vlastnosť. Implicitne otvorený priečinok slúži ako ochrana proti nechcenému zmazaniu súboru komponentom DefaultTreeSection.

[8] Medzi neriešiteľný problém napríklad patrí prípad, keď komponent nemá kontextové okno.
[9] Virtuálne stroje sa rátajú ako samostatné stroje.

6.5 Komponent TheiaElement

Základným komponentom testovacích nástrojov je trieda AbstractElement, ktorá bola do detailov vysvetlená v podkapitole číslo 6.3. Základná funkcionalita triedy AbstractElement je dostačujúca, ale pre špecifické požiadavky pre editor Che-Theia bol vytvorený nový komponent TheiaElement, ktorý bude bližšie objasnený v nasledujúcich podkapitolách.

Triedu AbstractElement by bolo možné používať namiesto TheiaElement, ale pri tomto používaní by prestal správne fungovať mechanizmus stabilného vyhľadávania. Metódy triedy AbstractElement findElement a findElements vracajú ako výsledok prvky typu WebElement. Inštancie triedy WebElement neimplementujú mechanizmus stabilného vyhľadávania. Tento problém bol jeden z dôvodov prečo vznikla trieda TheiaElement. Ďalším dôvodom vzniku novej triedy TheiaElement je rozšírenie pôvodných vyhľadávacích výrazov o metadáta.

V nasledujúcich podkapitolách bude vysvetlené akým spôsobom bol zachovaný mechanizmus stabilného vyhľadávania, dôvod prečo boli metadáta pridané do vyhľadávacích výrazov a pravidlá pri práce s komponentmi.

Mechanizmus stabilného vyhľadávania

Ako bolo spomenuté v úvode tejto podkapitoly, metódy findElement a findElements vracajú objekty typu WebElement, ktoré neobsahujú mechanizmus stabilného vyhľadávania triedy AbstractElement. Mechanizmus stabilného vyhľadávania je možné obnoviť prepísaním metódy findElement, ktorej obsah a vysvetlenie princípu fungovania mechanizmu je možné nájsť na ilustrácii číslo 6.3.

Metóda findElements v skutočnosti nepoužíva stabilný mechanizmus vyhľadávania elementov. V tejto metóde nie je vhodné používať spomínaný mechanizmus, pretože táto metóda ho priamo nevyžaduje. Tento mechanizmus vyžadujú nájdení potomkovia elementu. Obsah metódy a vysvetlenie princípu metódy je možné vidieť na ilustrácii číslo 6.4.

```
// Najdi element podla vyhladavacieho vyrazu
findElement(locator: Locator | TheiaLocator): WebElementPromise {
    // Prekonvertuj pripadny vyhladavaci vyraz typu TheiaLocator na typ Locator.
    const baseLocator = TheiaElement.handleLocator(locator) as Locator;
    // Ak je vyhladavaci vyraz typu TheiaLocator, uloz vyraz do premmenej.
    const loc: TheiaLocator | undefined = isTheiaLocator(locator) ? (locator as
    TheiaLocator) : undefined;
    // Vyhladavanie elementu. Ak sa element nenajde do casu uvedeneho v premmenej
    // findElementTimeout, tak sa vyvola vynimka.
    const element = this.getDriver()
        .wait(
            () => super.findElement(baseLocator).catch(() => undefined), SeleniumBrowser.
    instance.findElementTimeout,
            `Could not find element with locator "${baseLocator.toString()}" relative to "$
    {this.constructor.name}".`
        )
        // Prekonvertuj element typu WebElement na TheiaElement, aby nedoslo
        // ku strate mechanizmu stabilneho vyhladavania.
        .then((element) => new TheiaElement(element as WebElement, this, loc))

    // Prekonvertuj vysledok na format, ktory pouziva kniznica Selenium
    return new WebElementPromise(this.getDriver(), element as Promise<TheiaElement>);
}
```

Obr. 6.3: Implementácia prepísanej metódy findElement, tak aby bol zachovaný mechanizmus stabilného vyhľadávania.

```
// Najdi element podla vyhladavacieho vyrazu
findElement(locator: Locator | TheiaLocator): WebElementPromise {
    // Prekonvertuj pripadny vyhladavaci vyraz typu TheiaLocator na typ Locator.
    const baseLocator = TheiaElement.handleLocator(locator) as Locator;
    // Ak je vyhladavaci vyraz typu TheiaLocator, uloz vyraz do premmenej.
    const loc: TheiaLocator | undefined = isTheiaLocator(locator) ? (locator as
    TheiaLocator) : undefined;
    // Vyhladavanie elementu. Ak sa element nenajde do casu uvedeneho v premmenej
    // findElementTimeout, tak sa vyvola vynimka.
    const element = this.getDriver()
        .wait(
            () => super.findElement(baseLocator).catch(() => undefined), SeleniumBrowser.
    instance.findElementTimeout,
            `Could not find element with locator "${baseLocator.toString()}" relative to "$
    {this.constructor.name}".`
        )
        // Prekonvertuj element typu WebElement na TheiaElement, aby nedoslo
        // ku strate mechanizmu stabilneho vyhladavania.
        .then((element) => new TheiaElement(element as WebElement, this, loc))

    // Prekonvertuj vysledok na format, ktory pouziva kniznica Selenium
    return new WebElementPromise(this.getDriver(), element as Promise<TheiaElement>);
}
```

Obr. 6.4: Implementácia prepísanej metódy findElements, tak aby bol u potomkov zachovaný mechanizmus stabilného vyhľadávania.

Vyhľadávacie výrazy

Vyhľadávacie výrazy sú reprezentované objektom typu By, ktorý patrí knižnici Selenium WebDriver. Objekt By uchováva v sebe výraz, pomocou ktorého je možné nájsť elementy v štruktúre HTML. Štruktúra HTML sa u editorov časom mení, a preto je nutné tieto vyhľadávacie výrazy nejakým spôsobom spravovať. Vyhľadávacie výrazy sú spravované tým istým objektom, ktorý používa nástroj VS Code extension tester [25].

V tejto práci sa však rozhodlo pridať ku vyhľadávacím výrazom metadáta. Metadáta v sebe ukladajú konštruktor komponenty, závislosť komponenty a bežne používané atribúty. Prvé dva metadáta sú rezervované do budúcnosti, ak by sa začal realizovať projekt na automatické generovanie vyhľadávacích výrazov. V tejto práci nie sú ešte používané. Pod bežnými atribútmi sa myslia atribúty elementov, u ktorých sa predpokladá, že sa budú často meniť (ukážkový príklad atribútov komponentu je možné vidieť na ilustrácii číslo 6.5). Projekt sa riadi filozofiou, že samotné komponenty by mali ideálne používať vyhľadávacie výrazy iba v konštruktore. To avšak nie je možné pretože by bolo nutné definovať ešte viacej atribútov, čo by v kóde nebolo veľmi prehľadné.

```
node: {
    locator: By.className('theia-TreeNode'),
    properties: {
        focused: has('class', 'theia-mod-focus'),
        selected: has('class', 'theia-mod-selected'),
        expandable: has('class', 'theia-ExpandableTreeNode'),
        enabled: hasNot('class', 'theia-mod-busy')
    }
},
```

Obr. 6.5: Príklad vyhľadávacieho výrazu pre komponentu potomka stromu. Potomok sa vyhľadá na základe triedy „theia-TreeNode“ a má nasledujúce atribúty: aktívny, vybratý, rozšíriteľný a aktivovaný.

Pravidlá pri práci s komponentmi

Pri vytváraní alebo pracovaní s komponentmi je dôležité dodržiavať nasledujúce pravidlá, aby nedochádzalo k neočakávanému chovaniu. Komponenty s potomkami, ktoré je možné získať prostredníctvom aplikačného programovacieho rozhrania sa musia riadiť nasledujúcimi pravidlami:

- Ak sa vyhľadáva jeden potomok, vyhľadávacia metóda sa musí pokúšať nájsť potomok po dobu nastaveného časovača, ktorý je možný získať zavolaním funkcie *getTimeout()*.
- Ak sa zisťuje prítomnosť daného potomka v štruktúru HTML, musí byť použitá metóda findElements, aby nedochádzalo k zbytočnému blokovaniu programu.
- Ak je možné zmazať potomkov komponenty, daný komponent musí ošetriť prípady neplatných odkazov.
- Ak komponent má metódu, ktorá zaberá dlhší čas, daná metóda by mala používať funkciu repeat, ktorá bola vysvetlená v podkapitole 6.3. Ak funkciu repeat nie je možné použiť, je nutné napodobniť implementačné detaily časovača funkcie repeat, aby bola funkcionalita konzistentná.

6.6 Komponent ScrollableWidget

Komponent ScrollableWidget je objekt, ktorý reprezentuje grafický prvok s posuvníkom. Komponenty s posuvníkom slúžia na zobrazovanie ďalších komponentov, ktoré sú súčasťou dlhého zoznamu. Pôvodne sa na navigáciu v dlhých zoznamoch používali šípky na klávesnici, klávesa Page Down alebo klávesa Page Up. Tento prístup bol hoci jednoduchší na implementáciu, ale prinášal svoje nevýhody.

Značnou nevýhodou bola závislosť detekčného algoritmu konca a začiatku zoznamu na štruktúre HTML. Detekčný algoritmus má za úlohu detegovať koniec alebo začiatok daného zoznamu a prípadne prerušiť vyhľadávanie, ak nie je možné získať ďalšie komponenty zo zoznamu. Ďalšou nevýhodou ovládania komponentu pomocou spomenutých kláves je, že pri pomalých strojoch nie je možné garantovať, či došlo k plnému posunutiu zoznamu komponentov.

Komponent ScrollableWidget vznikol na základe získaných poznatkov o stromových štruktúrach komponentov z platformy Eclipse Theia. Stromové štruktúry z platformy Eclipse Theia používajú knižnicu React Virtualized[10], ktorá sa používa na optimálne zobrazovania veľkých zoznamov v štruktúre HTML. Optimalizácia je realizovaná na základe zobrazovania viditeľných komponentov alebo komponentov, ktoré sa vyskytujú blízko k viditeľným komponentom v zozname. Na základe týchto poznatkov je možné usúdiť, že metóda *findElements* z triedy *WebElement* nevráti všetky komponenty zo zoznamu, a teda nevráti správny výsledok. Z tohto dôvodu vznikol komponent ScrollableWidget.

Princíp fungovania komponentu

Komponent delí zoznam na stránky. Stránka je množina komponentov, ktoré sú aktuálne zobrazené v grafickom rozhraní. Komponent na základe aplikačného programovacieho rozhrania poskytuje používateľom komponentu mnoho funkcionalít, avšak medzi najdôležitejšie patria: zmena stránky, vyhľadávanie určitých komponentov, prechádzanie komponentov a prepnutie na prvú stránku.

V tejto práci sú podporované dva druhy vyhľadávania. Prvým vyhľadávaním je sekvenčné (metóda *findItemSequentially*) vyhľadávanie, ktoré vyhľadáva komponenty s časovou zložitosťou $O(N)$. Sekvenčné vyhľadávanie je vhodné použiť pri nezoradených zoznamoch.

Druhým vyhľadávacím algoritmom je binárne vyhľadávanie (metóda *findItemWithComparator*), ktoré vyhľadáva komponenty s časovou zložitosťou $O(\log N)$. Metóda vyžaduje pre správne fungovanie porovnávaciu funkciu, ktorá vracia číselnú hodnoty. Ak je vrátená negatívna hodnota, hľadaný komponent je menší než komponent zo zoznamu. Istý princíp platí aj pre kladné hodnoty, avšak v tomto prípade je hľadaný komponent väčší než komponent zo zoznamu. Ak je vrátená hodnota nulová, komponent bol úspešne lokalizovaný. Metóda každý cyklus vykonáva kontrolu platnosti vrátených hodnôt (nemôžu si hodnoty krajných komponentov odporovať).

V oboch vyhľadávacích algoritmoch sa pri neúspešnom vyhľadávaní vyvolá výnimka typu *ScrollItemNotFound*.

[10] https://github.com/bvaughn/react-virtualized

Princíp procesu zmeny stránok

Ako bolo spomenuté v úvode tejto podkapitoly, pôvodný algoritmus menenia stránok bol závislý na indíciách (v tomto prípade trieda elementu), ktoré sa nachádzali v štruktúre HTML. Tento spôsob menenia stránok nie je univerzálny a má problémy s detekciou plného vymenenia stránky.

Algoritmus založený na komponente posuvníka nemá problémy s detekciou plného vymenenia stránky. Veľkosť posuvníka zodpovedá veľkosti jednej stránky. Ak dôjde k posunutiu posuvníka o jeho veľkosť smerom nadol, v zozname sa objaví ďalšia stránka. Detegovať koniec procesu zmeny stránky je možné odvodiť podľa novej počiatočnej pozície posuvníka, ktorá by mala byť posunutá o veľkosť posuvníka. Ak sa posunutý posuvník nachádza na začiatku alebo konci zoznamu, algoritmus to berie ako koniec procesu zmeny stránky. Ak sa posuvník pred procesom zmeny stránky nachádza na začiatku alebo konci zoznamu a jeho posunutie by išlo za hranice rodičovského komponentu, algoritmus vyvolá výnimku typu Error.

6.7 Komponent príkazovej palety

Komponent príkazovej palety (v zdrojových súboroch trieda Input) je zaujímavý v tom, že nie je zobrazovaný bežným spôsobom. Príkazová paleta spolu s komponentom notifikácií sú zobrazované na obrazovke pevným spôsobom. Ich pozícia je konštantná a sú neustále prítomné v štruktúre HTML.

Hlavným problémom tejto komponenty bol, že pri zobrazení novej notifikácie, notifikácia prebrala kontrolu okna. Príkazovú paletu je možné zatvoriť stisnutím klávesy Escape. Ak bola v grafickom prostredí aktívna nejaká notifikácia, tak tento stisk klávesy Escape bol nechtiac ošetrený komponentom notifikácie namiesto príkazovej palety. Pôvodným riešením tohto problému bolo neustále opakovanie pokusov o zatvorenie príkazovej palety, ktoré používali funkcie kategórie safe. Toto riešenie malo vedľajší účinok, že ak sa komponent zatvoril až po kontrole otvorenosti komponentu, tak funkcia kategória safe zablokovala program, pretože čakala pokým príkazová paleta bude viditeľná. Nové riešenie používa alternatívnu funkciu bez stabilizačných mechanizmov. Tento problém sa vyskytuje aj u iných komponentov, ktoré majú funkcionalitu zatvorenia komponentu.

Vedľajšími problémami boli mazanie textu a nastavenie textu vstupného poľa. Niektoré rozšírenia vykonávajú kontrolu vstupov, ktoré sa kontrolujú po zmenách jedného znaku. Keďže integrované programovacie prostredie Eclipse Che nie je tak výkonné ako editor Visual Studio Code, vstupné pole sa niekedy nesprávne aktualizovalo a neobsahovalo očakávanú hodnotu. Z tohto dôvodu sa do vstupných polí text nepíše, ale kopíruje zo systémovej schránky. Na podobnom princípe fungoval aj problém s mazaním textu, ale v tomto prípade bol problém vyriešený funkciu repeat a časovačom, ktorý neustále vykonával mazanie textu.

Najzaujímavejšou častou komponentu príkazovej palety je implementácia možností príkazovej palety. Táto problematika bude vysvetlená v nasledujúcej podkapitole.

Možnosti príkazovej palety

Možnosť príkazovej palety je komponent, ktorý používateľovi editora navrhuje potenciálne príkazy. Nedávno spustené príkazy sa zobrazujú na začiatku zoznamu príkazovej palety, ak je vstup príkazovej palety prázdny. Ostatný príkazy sú zoradené lexikografickým spôso-

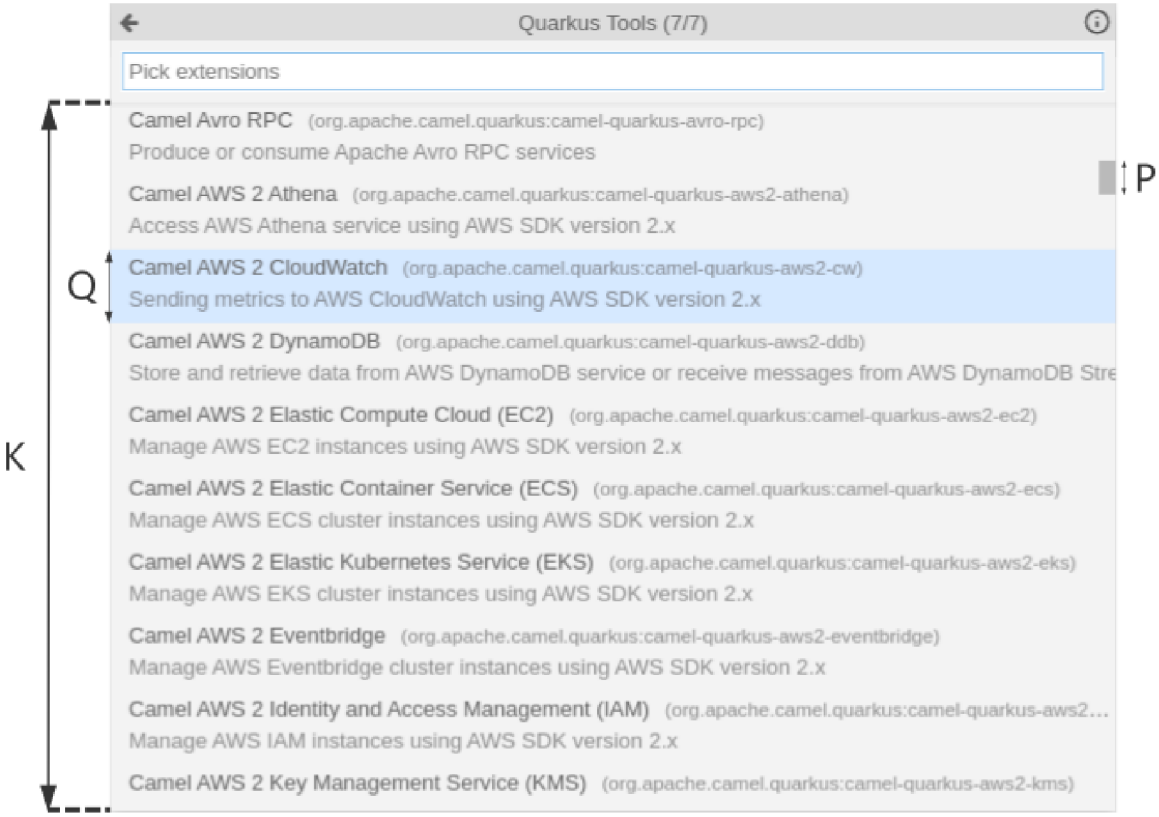
bom. Možnosti príkazovej palety je možné nájsť podľa mena možnosti alebo indexu možnosti. Index možnosti je indexovaný od čísla jeden.

Pôvodne bolo vyhľadávanie naprogramované sekvenčným vyhľadávaním pomocou kláves Page Up a Page Down. Problémom tohto algoritmu bolo, že pri prvom stisku klávesy Page Down sa stránka zoznamu nezmenila a bolo potrebné túto chybu ošetriť. Druhým problémom bolo, že sa musela aktívna možnosť kontrolovať, či sa daná možnosť nenachádza na konci zoznamu.

Po naprogramovaní komponenty ScrollableWidget došlo ku zmene algoritmu. Po zmene sa zo sekvenčného vyhľadávania prešlo na binárne vyhľadávanie. Sekvenčné vyhľadávanie sa ešte stále používa na začiatku algoritmu, z dôvodu, že nedávnom použité možnosti sa nachádzajú na začiatku zoznamu. Ďalšou zmenou, ktorá sa týka vyhľadávania na základe indexu, priniesla väčšie optimalizácie, ktoré budú popísane v nasledujúcej podkapitole.

Optimalizácia vyhľadávania možnosti pomocou indexu

Indexy možností príkazovej palety sú vzostupne zoradené. Na základe znalosti o komponente posuvníka a vedomosti o tom, že možnosti príkazovej palety sú rovnako veľké, je možné proces vyhľadávania optimalizovať. V nasledujúcom texte tejto podkapitoly sa budú používať označenia premien z nasledujúcej ilustrácie číslo 6.6.



Obr. 6.6: Rozmery kľúčových komponentov v príkazovej palete.

Veľkosť posuvníka (P) reprezentuje jednu stránku zo zoznamu komponentov. Na základe veľkosti možnosti (Q), veľkosti jednej stránky (K) a počtu komponentov (I_h[11] $- I_p$[12]) je možné približne predpovedať, kde sa bude hľadaný komponent nachádzať. Výpočet relatívnej vzdialenosti v jednotkách pixelov od posuvníka, na ktorej sa pravdepodobne nachádza komponent, je možné vypočítať na základe vzťahu:

$$V = \frac{Q}{K} \cdot P(I_h - I_p) \tag{6.1}$$

Na základe získanej hodnoty dôjde k posunu posuvníka. Zoznam možností nie je perfektne zarovnaný na úroveň všetkých možností a nejaká možnosť nemusí byť vykreslená v plnom rozsahu. Z tohto dôvodu sa po posunutí posuvníka vykoná binárne vyhľadávanie, aby sa prípadne skrytá možnosť dodatočne našla.

Výhodou nového algoritmu je jeho časová zložitosť. Pôvodný prerobený algoritmus binárneho vyhľadávania mal časovú zložitosť $O(\log N)$, nový algoritmus sa priblížil k časovej zložitosti $O(1)$. Nová časová odlišnosť má potenciál vylepšiť algoritmy v testoch na kontrolu duplicitných možností v príkazovej palete.

6.8 Komponent DefaultTreeSection

Komponent DefaultTreeSection patrí medzi posuvníkové komponenty, ktoré používajú posuvníkový komponent ScrollableWidget. Úlohou komponentu DefaultTreeSection je umožniť vývojárovi testov prístupovať ku súborovému systému. Tento komponent pôvodne nemal veľký význam, pretože vývojári testov pre rozšírenia Visual Studio Code mohli pristupovať k súborovému systému prostredníctvom štandardných knižníc prostredia NodeJS.

Funkcionalitu posuvníkového komponentu DefaultTreeSection by sa dalo rozdeliť na dve časti. Prvou časťou je samotný komponent, ktorý riadi operácie so súbormi. Druhou časťou je pomocný komponent FileTreeWidget, ktorý reprezentuje samotný súborový strom. Hoci rodičovský komponent TreeWidget podporuje vyhľadávanie položiek na základe cesty, tento algoritmus nie je optimalizovaný na vyhľadávanie v zoradených zoznamoch.

Súbory v strome súborov nie sú v skutočnosti reprezentované stromovou štruktúrou. V štruktúre HTML sú reprezentované ako postupnosť elementov. Hĺbku elementu je možné odvodiť od počtu elementov s triedou „indent“. Avšak pri komponente FileTreeWidget nie je potrebné odvodzovať hĺbku stromu, pretože každý element v sebe uchováva absolútnu cestu súboru a informáciu o aký súborový objekt[13] sa jedná. Na základe týchto dát a nasledujúcich pravidiel je možné vytvoriť porovnávaciu funkciu, ktorú bude možné použiť v binárnom vyhľadávaní. Poradie súborov sa riadi nasledujúcimi pravidlami:

- Ak majú súborové objekty rovnakého rodiča a sú rovnakého typu, porovnajú sa na základne lexikografického usporiadania.
- Ak majú súborové objekty rovnakého rodiča a prvý súborový objekt je typu súbor a druhý súborový objekt je typu priečinok, súbor je väčší (väčšie objekty sú nižšie v strome)
- Ak je jeden z objektov priečinok a zároveň rodičom druhého objektu, tak rodičovský priečinok je menší.

[11] Index hľadaného komponentu.
[12] Index prvého komponentu v zozname.
[13] Ako súborový objekt sa myslí súbor alebo priečinok.

- Ak oba objekty zdieľajú nejaký priečinok, ale nie sú priamymi potomkami, tak sa porovnávajú priamy potomkovia zdieľaného priečinku.
- Ak sa porovnávajú koreňový adresár, tak je v každom prípade koreňový adresár menši okrem prípadu, ak druhý porovnávaný objekt je taktiež koreňový adresár.

Problémy pri práci s komponentom DefaultTreeSection

Práca so súborovým stromom na rozdiel od ostatných komponentov je o mnoho komplikovanejšia, pretože dochádza ku mnoho zmenám ako sú napríklad: zmazanie súboru alebo vytvorenie súboru. Pri zmazaní súboru sa môže stať, že iná časť kódu používa referenciu komponentu na tento súbor. Neplatná referencia komponentu na súbor môže spôsobiť neočakávaný pád programu.

Ďalším problémom bola funkcionalita editoru, ktorá automaticky otvárala priečinky, ktoré mali jediného potomka. Problémom tejto funkcionality je, že otváraním priečinkov mení štruktúru zoznamu. Momentálnym riešením je aktívne čakanie na skončenie celého úkonu.

Posledným problémom pri práci so súborovým stromom je podobný ako bol spomenutý pri príkazovej palete. Komponenty zoznamu nie sú perfektne zarovnané a koncové komponenty môžu byť napríklad viditeľné iba na 10 % svojej výšky. Malá viditeľnosť komponentu môže spôsobiť, že sa akcia kliknutia myši na komponent nevykoná správne. Z tohto dôvodu sú do zoznamu zahrnuté iba tie komponenty, ktoré sú viditeľné aspoň na 55 % svojej výšky.

Kapitola 7

Testovanie

Testovací nástroj bol podrobený testom, ktoré sa skladali z troch fáz. Každá fáza mala za úlohu otestovať testovací nástroj na základe kritérií danej fázy. Testovacie fázy a ich priebeh bude vysvetlený v nasledujúcich podkapitolách.

7.1 Overenie funkčnosti komponentov

Prvou fázou bolo funkcionálne testovanie vytvorených komponentov. Funkcionálne testy majú za úlohu overiť základnú funkčnosť komponentov. Komponenty sa ideálne testovali, po ich vytvorení však niektoré komponenty boli závislé od ostatných komponentov, čo znemožnilo ich testovanie po vytvorení.

Funkcionálne testy ako aj ostatné testy iných fáz sú oddelené od modulu samotných testovacích nástrojov. Týmto oddelením od testovacieho modulu sa testovala ich integrácia s balíkovým manažérom npm. Pri testovaní funkcionality komponentov sa v zdrojových súboroch testov používa knižnica Mocha, aby sa čo najviac simulovalo reálne použitie testovacieho nástroja v praxi.

7.2 Integrácia testovacieho modulu s testami pre rozšírenia editora Visual Studio Code

Testovací nástroj vytvorený v tejto práci by mal byť čo najviac kompatibilný s testovacím nástrojom VS Code Extension Tester. Aby sa overila kompatibilita v čo najväčšom rozsahu tejto práce, na testovanie sa vybralo existujúce rozšírenie vscode-quarkus pre editor Visual Studio Code. Spomenuté rozšírenie bolo vybraté z toho dôvodu, pretože malo najviac unikátnych scenárov a pokrývalo veľký počet komponentov.

Avšak testy tohto rozšírenia používali problémové systémové knižnice, ktoré boli spomenuté v podkapitole číslo 5.2. Testy taktiež neboli aktuálne s novými funkcionalitami testovacieho nástroja VS Code Extension Tester, čo v niektorých prípadoch viedlo k tomu, že testy používali vlastnú implementáciu komponentov, čo zabraňovalo k integrácii nového testovacieho nástroja. Posledným problémom boli stabilizačné problémy, keď niektoré funkcie sa spoliehali na dostatočnú rýchlosť editora (v určitej miere je tento problém riešený nástrojom, ale niektoré funkcie nie je možné ošetriť). Problémy s rýchlosťou editora boli vysvetlené v podkapitole číslo 5.2.

Na základe týchto problémov bolo nutné pozmeniť časti zdrojového kódu. Použitá systémová knižnica *fs* bola nahradená komponentami DefaultTreeSection a Editor. Kompo-

nent Editor bol použitý na získanie textového obsahu vygenerovaných súborov. Stabilizačné problémy boli vyriešené pridaním aktívneho čakania a v prípade komponentu sprievodcu generovania projektu bola pozmenená podmienka na detegovanie ďalšieho alebo predošlého kroku.

Do pôvodných scenárov boli pridané nové scenáre testovania funkcionality notifikácií rozšírenia. Na základe tohto rozšírenia testov o testy notifikácií sa zvýšilo pokrytie testov o komponenty notifikačného centra a dolnej lišty na zobrazovanie stavu editora.

7.3 Testovanie s potenciálnymi používateľmi

Testovanie s potenciálnymi používateľmi sa konalo v intervaloch približne jedenkrát do mesiaca a začalo sa v januári. Každé testovanie prebehlo v rovnakom formáte s deviatimi ľuďmi, ktorý sa venovali jednej z činnosti: vývoj Eclipse Che IDE, testovanie rozšírení pre editor Visual Studio Code, testovanie rozšírení špecificky pre editor Che-Theia alebo v jednom prípade sa jednalo o autora testovacieho nástroja VS Code Extension Tester. Formát testovania sa väčšinou riadil nasledujúcim protokolom:

- Potenciálni užívatelia boli oboznámení s princípom testovacieho dema a so zdrojovým kódom dôležitých časti dema.
- Autor práce vytvoril demo na oboch editoroch Visual Studio Code a Che-Theia (CodeReady Workspaces). Ideálne naživo, alternatívne s nahrávkou dema.
- Autor práce poukázal na aktuálne problémy v editore a zdrojovom kóde (integračné problémy).
- Na konci prebehla diskusia ohľadom dema, aktuálneho fungovania testovacieho nástroja a budúcnosti testovacieho nástroja.

Na základe nových získaných poznatkov a spätnej väzby z dema testovacieho nástroja sa určili ďalšie ciele na ďalší mesiac. Tento druh testovania sa ukončil na konci apríla.

Kapitola 8

Záver

Cieľom tejto práce bolo vytvoriť testovací modul grafických používateľských rozhraní pre vývojové prostredie Eclipse Che, ktorý bude kompatibilný s testovacím modulom VS Code Extension Tester. Výsledkom tejto práce je testovací modul, ktorý je možný použiť pri testovaní rozšírení vývojového prostredia Eclipse Che, ale aj pri testovaní samotného editora Che-Theia. Avšak testy napísané primárne pre editor Che-Theia sú kompatibilne s testovacím nástrojom VS Code Extension Tester, to nemusí platiť v opačnom prípade. Ak testy používajú systémové knižnice, tieto testy je nutné najprv upraviť, aby neboli závislé na systémových knižniciach.

Najhlavnejším prínosom nového testovacie nástroja je zníženie časových nárokov na migráciu testov z nástroja VS Code Extension Tester a nový mechanizmus vyhľadávania komponentov. Nový mechanizmus vyhľadávania komponentov výrazne zvýšil stabilitu testov, najmä u strojoch s obmedzeným výpočtovým výkonom. Tento prínos je hlavne dôležitý pri testovaní na serveroch určené na priebežnú integráciu. K lepšej stabilite testov prispieva aj nový prístup implementácie komponentov a manipulácie s komponentami.

Pri vývoji testovacieho modulu došlo ku optimalizáciám určitých komponentov testovacieho nástroja. Najvýraznejšia optimalizácia bola vykonaná v komponente príkazovej palety, ktorý pôvodne vyhľadával indexy príkazových návrhov s lineárnou časovou zložitosťou. Po optimalizácií sa podarilo časovú zložitosť znížiť na konštantnú časovou zložitosťou. Ku optimalizáciám došlo aj u ostatných komponentov, ktoré narábajú so zoradenými zoznamami. U týchto zoznamov sa podarilo znížiť časovú zložitosť na logaritmickú.

Počas integrácie a stabilizácie testov rozšírenia vscode-quarkus sa zistilo, že testovacie moduly nie sú dostatočne efektívne pri testovaní funkcionality rozšírení s premenlivým stavom. Testovacie moduly poskytujú testerom prístup ku komponentom editora, ale samotné testovanie stavu komponentov nie je praktické, pretože stav komponentu sa môže napríklad zmeniť pomalšie, než sa očakáva. Do budúcnosti by bolo vhodné navrhnúť a naprogramovať rozširujúci zásuvný modul pre testovaciu knižnicu Chai[1], ktorý by priniesol viacej deklaratívny prístup do testovania rozšírení.

V najbližšej budúcnosti bude najväčšou prioritou integrovať novú funkcionalitu z upraveného testovacieho modulu[2] pre editor Visual Studio Code do originálneho testovacieho modulu. Následne bude potrebné finalizovať finálne aplikačné programovacie rozhranie medzi oboma editormi.

[1] https://www.chaijs.com/
[2] https://github.com/mlorinc/vscode-extension-tester

Literatúra

[1] Ecma International. *ECMAScript® 2020 Language Specification* [online]. [cit. 2021-04-20]. Dostupné z: https://262.ecma-international.org/11.0/.

[2] Finnigan, K. *Why Kubernetes native instead of cloud native? - Red Hat Developer* [online]. [cit. 2021-04-20]. Dostupné z: https://developers.redhat.com/blog/2020/04/08/why-kubernetes-native-instead-of-cloud-native/.

[3] Florent Benoit, V. S. a Williams, E. *Ability to run tests of a vscode extension on top of che-theia* [online]. [cit. 2021-05-03]. Dostupné z: https://github.com/eclipse/che/issues/18219.

[4] Florent Benoit, D. M. a Dahlquist, N. *Bez názvu* [online]. [cit. 2021-04-20]. Dostupné z: https://che-incubator.github.io/vscode-theia-comparator/status.html.

[5] Helming, J. a Koegel, M. *Eclipse Theia extensions vs. plugins vs. Che-Theia plugins* [online]. [cit. 2021-04-20]. Dostupné z: https://eclipsesource.com/blogs/2019/10/10/eclipse-theia-extensions-vs-plugins-vs-che-theia-plugins/.

[6] Helming, J. a Koegel, M. *How to create/develop an Eclipse Theia IDE extension* [online]. [cit. 2021-04-20]. Dostupné z: https://eclipsesource.com/blogs/2019/11/21/how-to-create-develop-an-eclipse-theia-ide-extension/.

[7] Helming, J. a Koegel, M. *How to create/develop an Eclipse Theia IDE plugin* [online]. [cit. 2021-04-20]. Dostupné z: https://eclipsesource.com/blogs/2020/05/04/how-to-create-develop-an-eclipse-theia-ide-plugin/.

[8] Helming, J. a Koegel, M. *Using VS Code extensions in Eclipse Theia and Che* [online]. [cit. 2021-04-20]. Dostupné z: https://eclipsesource.com/blogs/2019/10/31/using-vs-code-extensions-in-eclipse-theia-and-che/.

[9] Microsoft Corporation. *Code Navigation in Visual Studio Code* [online]. [cit. 2021-04-20]. Dostupné z: https://code.visualstudio.com/docs/editor/editingevolved.

[10] Microsoft Corporation. *Extension Anatomy* [online]. [cit. 2021-04-20]. Dostupné z: https://code.visualstudio.com/api/get-started/extension-anatomy.

[11] Microsoft Corporation. *Monaco Editor* [online]. [cit. 2021-04-20]. Dostupné z: https://microsoft.github.io/monaco-editor/.

[12] Microsoft Corporation. *Testing Extensions* [online]. [cit. 2021-05-05]. Dostupné z: https://code.visualstudio.com/api/working-with-extensions/testing-extension.

[13] Mozilla Corporation. *Inheritance and the prototype chain - JavaScript | MDN* [online]. [cit. 2021-04-20]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain.

[14] Node.js contributors. *Discover JavaScript Timers* [online]. [cit. 2021-04-20]. Dostupné z: https://nodejs.dev/learn/discover-javascript-timers.

[15] Node.js contributors. *JavaScript Asynchronous Programming and Callbacks* [online]. [cit. 2021-04-20]. Dostupné z: https://nodejs.dev/learn/javascript-asynchronous-programming-and-callbacks.

[16] Node.js contributors. *Modern Asynchronous JavaScript with Async and Await* [online]. [cit. 2021-04-20]. Dostupné z: https://nodejs.dev/learn/modern-asynchronous-javascript-with-async-and-await.

[17] Node.js contributors. *The Node.js Event Loop* [online]. [cit. 2021-04-20]. Dostupné z: https://nodejs.dev/learn/the-nodejs-event-loop.

[18] Node.js contributors. *Understanding JavaScript Promises* [online]. [cit. 2021-04-20]. Dostupné z: https://nodejs.dev/learn/understanding-javascript-promises.

[19] Node.js contributors. *Understanding process.nextTick()* [online]. [cit. 2021-04-20]. Dostupné z: https://nodejs.dev/learn/understanding-process-nexttick.

[20] Node.js contributors. *Understanding setImmediate()* [online]. [cit. 2021-04-20]. Dostupné z: https://nodejs.dev/learn/understanding-setimmediate.

[21] Node.js Website WG contributors. *The Node.js Event Loop, Timers, and process.nextTick() | Node.js* [online]. [cit. 2021-04-20]. Dostupné z: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/.

[22] OpenJS Foundation. *Electron | Build cross-platform desktop apps with JavaScript, HTML, and CSS.* [online]. [cit. 2021-04-29]. Dostupné z: https://www.electronjs.org/.

[23] OpenJS Foundation. *Visual Studio Code | Apps | Electron* [online]. [cit. 2021-01-28]. Dostupné z: https://www.electronjs.org/apps/visual-studio-code.

[24] Richter, J. *Vscode-extension-tester/AbstractElement.ts at master · redhat-developer/vscode-extension-tester · GitHub* [online]. [cit. 2021-05-03]. Dostupné z: https://github.com/redhat-developer/vscode-extension-tester/blob/master/page-objects/src/components/AbstractElement.ts.

[25] Richter, J. *Vscode-extension-tester/loader.ts at master · redhat-developer/vscode-extension-tester · GitHub* [online]. [cit. 2021-05-03]. Dostupné z: https://github.com/redhat-developer/vscode-extension-tester/blob/master/page-objects/src/locators/loader.ts.

[26] Selenium. *Selenium/webdriver.js at trunk · SeleniumHQ/selenium · GitHub* [online]. [cit. 2021-05-03]. Dostupné z: https://github.com/SeleniumHQ/selenium/blob/8d80348b75aff4b712aa8101da979bbe21e12bd4/javascript/node/selenium-webdriver/lib/webdriver.js#L807.

[27] Selenium. *Waits* [online]. [cit. 2021-05-03]. Dostupné z: https://www.selenium.dev/documentation/en/webdriver/waits/#implicit-wait.

[28] Stack Exchange, Inc.. *Stack Overflow Developer Survey 2020* [online]. [cit. 2021-04-20]. Dostupné z: https://insights.stackoverflow.com/survey/2020#overview.

[29] The Eclipse Foundation. *Authoring Theia Extensions* [online]. [cit. 2021-04-20]. Dostupné z: https://theia-ide.org/docs/authoring_extensions/.

[30] The Eclipse Foundation. *Authoring Theia Plug-ins* [online]. [cit. 2021-04-20]. Dostupné z: https://theia-ide.org/docs/authoring_plugins/.

[31] The Eclipse Foundation. *Che architecture overview :: Eclipse Che Documentation* [online]. [cit. 2021-04-20]. Dostupné z: https://www.eclipse.org/che/docs/che-7/overview/che-architecture/.

[32] The Eclipse Foundation. *GitHub - eclipse-che/che-theia* [online]. [cit. 2021-04-20]. Dostupné z: https://github.com/eclipse-che/che-theia.

[33] The Eclipse Foundation. *Introduction to Eclipse Che :: Eclipse Che Documentation* [online]. [cit. 2021-04-20]. Dostupné z: https://www.eclipse.org/che/docs/che-7/overview/introduction-to-eclipse-che/.

[34] The Eclipse Foundation. *Theia - Cloud and Desktop IDE Platform* [online]. [cit. 2021-04-20]. Dostupné z: https://theia-ide.org/.

[35] The Eclipse Foundation. *Theia/monaco-workspace.ts at master · eclipse-theia/theia · GitHub* [online]. [cit. 2021-04-20]. Dostupné z: https://github.com/eclipse-theia/theia/blob/master/packages/monaco/src/browser/monaco-workspace.ts.

[36] The Eclipse Foundation. *Understanding Che workspace controller :: Eclipse Che Documentation* [online]. [cit. 2021-04-20]. Dostupné z: https://www.eclipse.org/che/docs/che-7/administration-guide/che-workspace-controller/.

[37] The Eclipse Foundation. *Understanding Che workspaces architecture :: Eclipse Che Documentation* [online]. [cit. 2021-04-20]. Dostupné z: https://www.eclipse.org/che/docs/che-7/administration-guide/che-workspaces-architecture/.

[38] The Eclipse Foundation. *What is a Che-Theia plug-in* [online]. [cit. 2021-05-06]. Dostupné z: https://www.eclipse.org/che/docs/che-7/end-user-guide/what-is-a-che-theia-plug-in/.