VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

POHLED NA STAV JUNIT PRO TESTOVANOU INSTANCI ECLIPSE

JUNIT STATUS VIEW FOR TESTED ECLIPSE INSTANCE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MARTIN COUFAL**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ZBYNĚK KŘIVKA, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/18520/2016/xcoufa08

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Coufal Martin**
Obor: Informační technologie
Téma: **Pohled na stav JUnit pro testovanou instanci Eclipse**
JUnit Status View for Tested Eclipse Instance
Kategorie: Softwarové inženýrství

Pokyny:

1. Seznamte se s vývojem pro prostředí Eclipse včetně tvorby zásuvných modulů, pohledů a automatického testování uživatelského rozhraní (UI).
2. Analyzujte možnosti záznamu o průběhu testu pro zjištění důvodů neúspěchu testu a ladění testů se zaměřením na testy uživatelského rozhraní.
3. Dle pokynů vedoucího navrhněte modul(y) pro vylepšení ladění JUnit testů UI v Eclipse. Například vytvoření pohledu na průběh JUnit testů přímo do testované instance Eclipse během testování UI.
4. Návrh implementujte jako modul(y) pro Eclipse distribuovatelný(é) pomocí Eclipse Update Site.
5. Realizaci zhodnoťte a navrhněte nasazení na CI server Jenkins.

Literatura:

- Clayberg, E., Rubel, D.: *Eclipse Plug-ins*. Addison-Wesley, Third Edition, 2009, 878 s.
- Vogel, L.: *Eclipse 4 Application Development: The complete guide to Eclipse 4 RCP development* (*Volume* 1). Lars Vogel, 2012, 432 s.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Křivka Zbyněk, Ing., Ph.D.**, UIFS FIT VUT
Konzultant: Srna Pavol, Ing., RedHatCZ
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Cílem této práce je především návrh a implementace nástrojů umožňujících zobrazovat stav průběhu testů testované instance vývojového prostředí Eclipse. Řeší tak problém zobrazení výsledků právě probíhajících testů grafického uživatelského rozhraní. Práce také obsahuje popis struktury vývojového prostředí Eclipse IDE (*Integrated Development Environment*) a popis nástroje JUnit, který je k implementaci zmíněných nástrojů využit. Pomocí implementovaných nástrojů lze zobrazit informace o běhu testů, podobně jako je tomu u pohledu JUnit. Zároveň jsou tyto informace vždy viditelné a aplikace zobrazující tyto informace nevyžaduje aktivitu okna (*angl. focus*). Implementovaný nástroj umožňuje detailnější pohled na probíhající testy a je tak užitečný v případě vytváření a analýzy testů – lze snáze odhalit, v jakém místě by se mohla nacházet chyba.

Abstract

The aim of this paper is to design and implement tools which allow to display testing progress of an Eclipse integrated development environment instance. This resolves problem of viewing currently running graphical user interface test run results. This thesis also contains description of Eclipse IDE (*Integrated Development Environment*) and JUnit framework architecture, that is used to achieve this goal. Implemented applications allow to take a closer look at currently running tests, in a similar way to JUnit view. To achieve visibility, the information window is displayed always on top and doesn't require the focus. That is useful to locate an error both when creating a new test or analysing already implemented one.

Klíčová slova

zásuvný modul pro Eclipse, testování Eclipse IDE, rozšíření JUnit, výsledky JUnit, testování grafického uživatelského rozhraní Eclipse IDE

Keywords

Eclipse plug-in, Eclipse IDE testing, JUnit extension, JUnit results, Eclipse graphical user interface testing

Citace

COUFAL, Martin. *Pohled na stav JUnit pro testovanou instanci Eclipse*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

Pohled na stav JUnit pro testovanou instanci Eclipse

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D. Další informace mi poskytl Ing. Pavol Srna, zaměstnanec firmy Red Hat Czech s. r. o., zabývající se mimo jiné testováním vývojového prostředí Eclipse. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Coufal
17. května 2017

Poděkování

Chtěl bych poděkovat svému vedoucímu práce za metodické vedení, pozitivní přístup a konstruktivní návrhy při tvorbě této práce, Ing. Srnovi a Mgr. Wágnerovi za cenné rady ohledně návrhu a implementace výsledné aplikace.

Obsah

Kapitola 1

Úvod

V dnešní době se na testování klade velký důraz, a proto je potřeba se touto částí vývoje software zabývat detailně. Aplikace lze testovat různými způsoby, od jednoduchých manuálních testů po užití sofistikovaných nástrojů, které automaticky spouští vybrané sady testů. Pro tyto účely existuje mnoho nástrojů[1], které usnadňují programátorům práci. Cílem těchto nástrojů je redukovat množství napsaného kódu opakujícího se v testech pro podobné komponenty nebo jejich vlastnosti.

Co se týče programovacího jazyka Java, můžeme vybírat z velkého množství nástrojů[2] pro testování podle toho, jaký aspekt software chceme testovat. Pro testování Servler, Bean a Java tříd lze testovat pomocí nástrojů, jako jsou například *Servlets*, *JUnit*, *Arquillian*, *ServletUnit* nebo *Mock objects*. Pro testování grafického uživatelského rozhraní vytvořeného pomocí Swing lze použít například *UISpec4j*, *Abbot*, *Fest*, *QF-Test* a další. Pro funkcionální testování lze použít například *HTTPUnit*, *JWebUnit*, *TestNG* nebo *Selenium Webdriver*, zatímco pro výkonnostní testování lze použít například *Apache JMeter* [9].

Tato práce se zabývá popisem infrastruktury vývojového prostředí *Eclipse* (dále zkráceně Eclipse IDE) a jeho zásuvných modulů, s přihlédnutím k budoucímu použití pro vytvoření vlastního zásuvného modulu. Dále se zabývá popisem testovacího rámce JUnit a jeho architekturou. JUnit je snadno rozšiřitelný a je obsažen ve velkém množství testovacích nástrojů a vývojových prostředí včetně Eclipse IDE, kde ho lze použít při testování jeho grafického uživatelského rozhraní (dále zkráceně GUI). Klíčovou částí práce je návrh a implementace zásuvného modulu pro Eclipse IDE, který zpracovává data z testovacího rámce JUnit a posílá informace o průběhu testů do vytvořené klientské aplikace, kde je zobrazuje uživateli. Důvodem pro tvorbu tohoto projektu je potřeba programátora zjistit v jaké fázi se probíhající sada testů nachází, který test v dané chvíli běží a jak dopadly již proběhlé testy.

Podobným již existujícím nástrojem umožňující zobrazení informací o průběhu testů je *pohled JUnit*. Ten je bohužel při testování GUI překryt oknem s testovanou instancí a v případě přepnutí na okno s pohledem JUnit riskujeme ztrátu aktivity okna a selhání testů. Navíc při testování grafického uživatelského rozhraní hraje roli spousta dalších proměnných. Často jen změna komponenty nebo konfigurace platformy, na které testujeme, může způsobit selhání testů. Proto je návrh implementované aplikace přizpůsoben co nejmenšímu zásahu do GUI testované instance Eclipse IDE.

[1] https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks
[2] https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#Java

Kapitola 2

Vývojové prostředí Eclipse

Eclipse IDE je integrované vývojové prostředí poskytující podporu pro mnoho programovacích jazyků, jako jsou například Java, C, C++, JavaScript a PHP. Je založen na modulové architektuře, která umožňuje snadné rozšíření této platformy. Pro přidání nové funkcionality do vývojového prostředí stačí nainstalovat příslušný zásuvný modul (*angl. plug-in*). Projekt Eclipse je udržován radou správců *eclipse.org*, která vznikla z iniciativy společností *Borland*, *IBM*, *MERANT*, *QNX Software Systems*, *Rational Software*, *Red Hat*, *SuSE*, *TogetherSoft* a *Webgain*. Cílem eclipse.org je tvorba univerzálního rozšiřitelného integrovaného vývojového prostředí, které poskytuje nástroje pro integraci různých platforem a zároveň potřebné nástroje pro jejich tvorbu a rozšíření [6].

V následující kapitole je popsána struktura Eclipse IDE a částí, ze kterých se skládá. Dále je zde popsán systém zásuvných modulů Eclipse IDE a jejich částí.

2.1 Infrastruktura Eclipse IDE

Eclipse IDE není monolitické vývojové prostředí, ale spíše komplexní soubor zásuvných modulů. V základě je rozdělen do několika podsystémů, které jsou koncipovány jako jeden nebo více zásuvných modulů. Minimální množina zásuvných modulů, která je potřeba pro vývoj klientské aplikace, se nazývá *Eclipse RCP* (*Rich Client Platform*)[1]. V dnešní době se však Eclipse IDE často používá i pro vývoj serverových aplikací, tato infrastruktura se potom nazývá *EAF* (*Eclipse Aplication Framework*). Důležitou komponentou je jádro, které načítá jednotlivé zásuvné moduly. Toto jádro je implementováno na základě specifikace *OSGi Service Platform*, která definuje standard pro dynamické modulární systémy v Javě [3]. Tento standard je definován mezinárodním konsorciem *OSGi Alliance*[2]. Díky implementaci rámce *OSGi Equinox*[3] používá Eclipse IDE pro načítání jednotlivých zásuvných modulů návrhový vzor *lazy-loading*, který zajišťuje načítání jen nezbytných zásuvných modulů.

Zásuvné moduly se v základu dělí podle funkce do několika skupin [3]:

Core: skupina nízkoúrovňových zásuvných modulů zajišťujících základní funkce, jako jsou zpracování a rozšíření zásuvných modulů a zdrojových kódů.

SWT (*Standard Widget Toolkit*): knihovna nástrojů pro manipulaci s uživatelským rozhraním, která poskytuje API nezávislé na operačním systému.

[1] https://wiki.eclipse.org/Rich_Client_Platform
[2] https://www.osgi.org/
[3] http://www.eclipse.org/equinox/

JFace: knihovna přidávající další funkcionalitu jako nástavbu nad SWT.

GEF: rámec poskytující prostředí pro vývoj grafických editorů.

Workbench: skupina zásuvných modulů poskytujících funkcionalitu specifickou přímo pro Eclipse IDE, jako je manipulace s projekty nebo grafickými prvky (pohledy, perspektivami, atd.).

Team: skupina zásuvných modulů poskytujících podporu pro správu verzí v Eclipse IDE.

Help: poskytuje dokumentaci k jednotlivým prvkům Eclipse IDE.

JDT (*Java Development Tools*)[4]: přidává podporu pro vývoj Java aplikací a navíc do Eclipse IDE přidává perspektivy, pohledy, průvodce a další nástroje pro práci s Javou.

PDE (*Plug-in Development Enviroment*)[5]: skupina zásuvných modulů poskytujících různé nástroje (pohledy, editory, atd.) pro práci se zásuvnými moduly a jejich manifesty.

Mylyn: poskytuje rámec pro správu úkolů a životního cyklu aplikace (*Application Lifecycle Management*)

Na obrázku 2.1 jsou znázorněny základní komponenty platformy Eclipse. *Workbench* představuje obálku, která umožňuje uživateli orientaci ve vývojovém prostředí. Definuje *body rozšíření* (*angl. extension points*), kde je možno k zásuvnému modulu přidat další rozšiřující modul. Díky těmto bodům lze přidávat například komponenty grafického rozhraní, jako jsou pohledy, editory a menu. *Pracovní plocha* (*angl. Workspace*) definuje rozhraní pro programování aplikací (dále zkráceně API) za účelem vytváření a správy projektů, souborů a složek. Zde jsou projekty překládány a sestavovány. Pracovní plocha navíc obsahuje další informace k projektům, jako je například uživatelské nastavení. *Help* poskytuje body rozšíření sloužící pro zobrazení nápovědy nebo dokumentace. Modul *Team* definuje model pro vývoj aplikací v týmu, s podporou správy verzí aplikace a zdrojových kódů. Komponenta *Platform Runtime* spravuje informace týkající se právě běžícího prostředí Eclipse. Například dynamicky vyhledává a spravuje informace o zásuvných modulech a jejich bodech rozšíření. Také poskytuje informace ohledně správy procesů, argumentů příkazové řádky, adresářové struktury jednotlivých zdrojů a další [3].

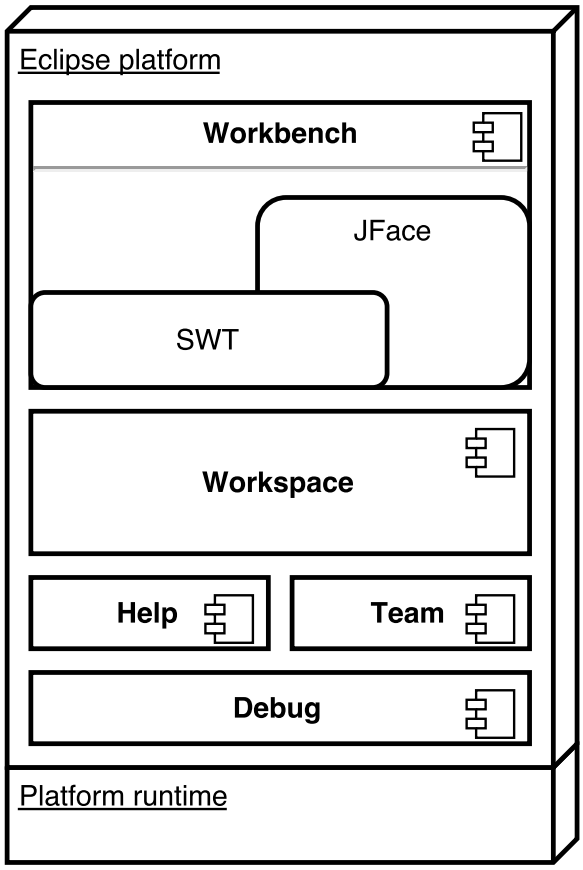
2.1.1 Workbench

Termín Workbench odkazuje na desktopové vývojové prostředí. Jeho cílem je integrace různých nástrojů a poskytnutí základního schématu pro tvorbu, správu a navigaci ve zdrojích pracovní plochy. Běžně se ve Workbench nachází lišta s hlavním menu, nástrojová lišta a několik pohledů, případně editorů [5]. Grafické prvky vývojového prostředí jsou však pro uživatele přizpůsobitelné. Běžný vzhled Eclipse Workbench je zobrazen na obrázku 2.2.

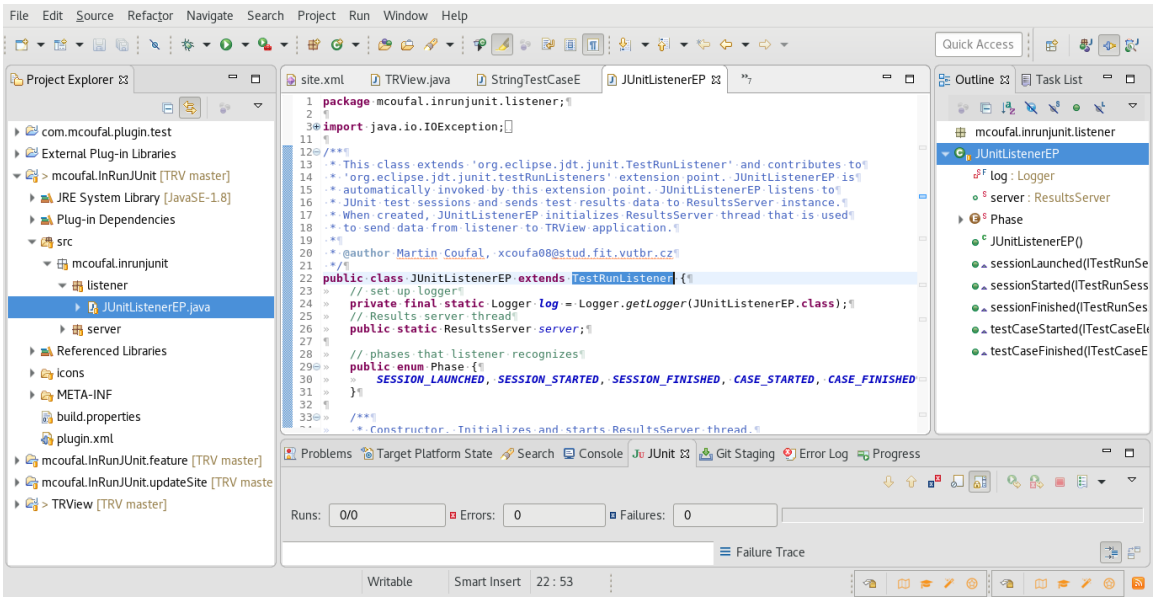
Pohledy

Pohled je jednou ze základních komponent tvořících grafické uživatelské rozhraní Eclipse IDE. Slouží k zobrazování informací uživateli a také k navigaci ve vývojovém prostředí. Každý pohled může mít svá menu a své nástrojové lišty.

[4] http://www.eclipse.org/jdt/
[5] http://www.eclipse.org/pde/



Obrázek 2.1: Architektura platformy Eclipse (inspirováno v [4]).



Obrázek 2.2: Běžný vzhled Eclipse Workbench.

Pro vytvoření nového pohledu je zapotřebí dvou kroků – vytvoření kategorie pohledu (pokud ho nechceme vytvořit v některé z již existujících kategorií) a deklarace nového pohledu. Obě dvě změny probíhají v manifestu zásuvného modulu. Přesto, že je možné tyto změny do manifestu dopsat ručně, vývojové prostředí Eclipse nabízí praktické nástroje pro editaci manifestů zásuvných modulů. Otevření některého z manifestů zásuvného modulu spustí editor zásuvného modulu. Nastavení rozšíření zásuvného modulu lze upravovat v záložce *Extensions*. Pro přidání kategorie pohledu i deklarace pohledu samotného je nutné nejdříve přidat správný bod rozšíření. Všechny pohledy se připojují na bod rozšíření `org.eclipse.ui.views`. K tomuto bodu rozšíření lze přidat novou kategorii pohledu nebo deklarovat nový pohled.

Běžně používanou třídou implementující toto rozhraní je `org.eclipse.ui.part.ViewPart`. Zdrojový kód s popisem chování daného pohledu se nachází ve třídě implementující rozhraní `org.eclipse.ui.IViewPart` [3].

Perspektivy

Perspektivy slouží jako nástroj pro seskupení relevantních komponent v aktivním okně Workbench. Komponenty se seskupují podle úkonu, který bude uživatel vykonávat, a zároveň podle programovacího jazyka, ve kterém uživatel projekt vytváří. Eclipse IDE umožňuje přidání nových perspektiv pomocí zásuvných modulů a bodů rozšíření. Pokud je to žádoucí, lze také při přidání nového zásuvného modulu některou z již existujících perspektiv pouze upravit.

Pro přidání nové perspektivy, podobně jako u přidání nového pohledu, je nutno do manifestu zásuvného modulu přidat správný bod rozšíření – `org.eclipse.ui.perspectives`. Rozložení komponent v dané perspektivě je definováno ve třídě implementující rozhraní `IPerspectiveFactory` [3]. Přidat nové rozšíření zásuvného modulu implementující novou nebo upravenou perspektivu lze jednoduše pomocí nástrojů v záložce *Extensions* u editoru zásuvného modulu. Chceme-li perspektivu pouze upravit, lze při tvorbě nové perspektivy vybrat některou z již existujících.

Editory

Editory slouží jako hlavní nástroj pro úpravu zdrojového kódu a jiných textových souborů. Eclipse IDE nabízí mnoho editorů, které je možno nainstalovat v rámci nějakého ze zásuvných modulů. Nejzákladnějším editorem je textový editor. Ten poskytuje pouze základní funkce pro práci s textem a neposkytuje možnost zvýraznění syntaxe nebo její kontrolu. I základní textový editor je však možno dále rozšířit pomocí dalších zásuvných modulů.

Editor lze přidat pomocí rozšiřujícího bodu `org.eclipse.ui.editors`. Na ten lze připojit vlastní třídu implementující rozhraní `org.eclipse.ui.IEditorPart` [3]. Proces vytváření nového editoru je stejný jako v předchozích dvou případech.

2.1.2 Standard Widget Toolkit

SWT představuje tenkou vrstvu nad nativním ovládáním uživatelského rozhraní platformy Eclipse. Poskytuje tak rozhraní pro pohodlnější interakci s uživatelským rozhraním, které zahrnuje většinu prvků nacházejících se v platformě Eclipse a tak lze pomocí SWT snadno vytvářet aplikace s grafickým uživatelským rozhraním [3]. Zároveň je na SWT postaven zásuvný modul *SWTBot*, sloužící pro testování uživatelského rozhraní.

2.2 Architektura zásuvných modulů v Eclipse IDE

Každý zásuvný modul v Eclipse IDE slouží buď jako knihovna pro dodatečné funkce jiným zásuvným modulům nebo pro rozšíření funkcionality platformy. Chování každého zásuvného modulu je popsáno v jeho zdrojovém kódu. Závislosti, body rozšíření a služby poskytované zásuvným modulem jsou popsány v manifestech zásuvného modulu – souborech `MANIFEST.MF` a `plugin.xml`. Načítání nových zásuvných modulů probíhá, až když je modul přímo vyžadován (dle návrhového vzoru lazy-loading). Na začátku jsou načteny pouze manifesty zásuvných modulů. Ty poskytují základní informace o zásuvném modulu a nemusí se tak načítat kompletní zásuvný modul. Díky tomuto modelu je Eclipse IDE i přes velké množství možných instalovaných zásuvných modulů kompaktnější a výrazně rychlejší při startu.

Zásuvný modul (ať už v podobě Java archivu JAR nebo v podobě adresáře projektu) se skládá z Javových tříd, manifestů zásuvného modulu (`MANIFEST.MF` a `plugin.xml`) a obrázkových souborů, které jsou typicky umístěny v adresářích pojmenovaných *icons* nebo *images*. Pokud je zásuvný modul ve formě adresáře, archiv JAR je uložen v některém z podadresářů. Název archivu JAR a jeho umístění je v takovém případě definováno v souboru `MANIFEST.MF`.

2.2.1 Model zásuvných modulů

Eclipse IDE při startu prohledá všechny adresáře se zásuvnými moduly a vytvoří vlastní model obsahující každý nalezený zásuvný modul. Tento model se vytváří pomocí manifestů zásuvných modulů tak, aby Elipse nemusel načítat celé zásuvné moduly a ušetřil tak čas a místo. Informace o jednotlivých nainstalovaných zásuvných modulech jsou uloženy v balíčcích (*angl. bundles*). Získáváním informací přes rozhraní těchto balíčků zajistíme, aby se zásuvné moduly nenačítaly, dokud nejsou opravdu zapotřebí.

Původně mělo Eclipse IDE vlastní mechanismus pro běh aplikace, ale to znemožnilo použití již vytvořených technologií v jiných oblastech jako Avalon[6] nebo JMX[7]. Proto byl nakonec nahrazen mechanismem pro běh aplikace založeném na technologii OSGi Alliance, která poskytuje model s detailní specifikací a podporuje dynamické chování [3].

2.2.2 Vytvoření zásuvného modulu

Pro vytvoření zásuvného modulu poskytuje platforma Eclipse zásuvný modul PDE, který poskytuje užitečné nástroje pro tvorbu, manipulaci a distribuci zásuvných modulů. Platforma Eclipse poskytuje mnoho variant konfigurací, jaké lze stáhnout a instalovat. Zásuvný modul PDE je součástí například Eclipse SDK, Eclipse Classic a Eclipse Standard. Eclipse PDE lze také zpětně doinstalovat například pomocí *Eclipse marketplace*[8].

V PDE má každý zásuvný modul svůj vlastní projekt. Tento projekt lze vytvořit pomocí průvodce (*angl. wizard*) nebo konvertovat již existující projekt na projekt se zásuvným modulem. Pro tvorbu nového projektu stačí použít průvodce *New Plug-in Project* a v něm vyplnit základní informace o projektu, zadat informace o zásuvném modulu[9] a dokončit průvodce [2].

[6] https://avalon.apache.org
[7] http://www.oracle.com/technetwork/articles/java/javamanagement-140525.html
[8] https://marketplace.eclipse.org/content/eclipse-pde-plug-development-environment
[9] Jednotlivé položky k vyplnění jsou blíže popsány v sekci 2.2.3

2.2.3 Manifesty zásuvného modulu

Manifesty zásuvného modulu jsou dva soubory – `MANIFEST.MF` a `plugin.xml`. První zmíněný obsahuje data nutná pro běh zásuvného modulu jako jsou *identifikátor*, *verze* a *závislosti* zásuvného modulu. Druhý obsahuje data ve formátu XML popisující případná rozšíření a body rozšíření.

Soubor `MANIFEST.MF`

V každém souboru `MANIFEST.MF` zásuvného modulu se nacházejí záznamy pro jméno, identifikátor, verzi, spouštěč a poskytovatele balíčku zásuvného modulu. Dále se zde mohou vyskytovat záznamy s *ClassPath*, exportovanými balíčky a závislostmi zásuvného modulu.

Jméno (*Bundle-Name*) a poskytovatel (*Bundle-Vendor*) jsou *human-readable*[10] řetězce, které nemusí být unikátní a lze je ukládat do zvláštního souboru `plugin.properties` za účelem internacionalizace.

Identifikátor (*Bundle-SymbolicName*) slouží k jednoznačné identifikaci daného balíčku. Většinou se jako identifikátor balíčku používá Javová konvence pro pojmenování balíčků: `com.<název společnosti>.<komponenta>[.<část komponenty>]`, kde část dané komponenty (například '`ui`' nebo '`core`') se uvádí v případě větších komponent, kde jsou jednotlivé balíčky rozděleny. U tohoto identifikátoru bývá ještě uvedena direktiva OSGi `singleton`, která udává, zda může být v platformě instalováno více verzí daného zásuvného modulu současně [2].

Verze (*Bundle-Version*) slouží k jednoznačné identifikaci verze daného balíčku. V případě shody identifikátorů je vždy vybrán balíček s novějším číslem verze. Toto číslo se skládá ze 3 číslic oddělených tečkami a v případě potřeby i alfanumerického řetězce použitelného pro užší specifikaci (například '`1.2.3.beta`'). První číslo označuje majoritní verzi produktu, druhé minoritní verzi produktu a třetí slouží k označení úrovně služeb[11].

Spouštěč (*Bundle-Activator*) je volitelná část manifestu, která umožňuje specifikovat třídu implementující rozhraní `BundleActivator` a poskytuje tak metody `start()` a `stop()`, které jsou přínosné pro správu životního cyklu balíčku.

Bundle-ClassPath je záznam využitý pro seznam balíčků a knihoven, které mají být do zásuvného modulu přidány. Modul tak může být instalován sám o sobě bez nutnosti stahování dalších balíčků nebo knihoven. Takto přidané balíčky však výrazně zvětšují velikost výsledného zásuvného modulu.

Záznam s exportovanými balíčky (*Export-Package*) je podmnožina z Bundle-ClassPath obsahující balíčky, které mají být viditelné pro ostatní balíčky zásuvných modulů.

Eclipse IDE vytváří pro každý načtený zásuvný modul novou instanci, která slouží k vyhledávání a načítání zásuvných modulů a používá *záznam závislostí* v manifestu k určení viditelnosti ostatních zásuvných modulů. Záznam závislostí (*Require-Bundle*) je seznam zásuvných modulů, které jsou pro daný zásuvný modul viditelné z hlediska vykonávání programu [3].

Soubor `plugin.xml`

V tomto manifestu zásuvného modulu mohou být uvedeny body rozšíření, na které je možno připojit nový zásuvný modul rozšiřující stávající funkcionalitu. Toto odloučení od rozšiřujících zásuvných modulů umožňuje existenci základního zásuvného modulu bez znalosti

[10] https://en.oxforddictionaries.com/definition/us/human-readable

[11] Více k číslování balíčků lze nalézt na https://wiki.eclipse.org/Version_Numbering.

```
<extension-point id="org.example.extensionpoints.point"
  name="Point"
  schema="schemas/scheme.exsd" />
```

Obrázek 2.3: Příklad zdrojového kódu s definicí bodu rozšíření.

```
<extension
  point="org.eclipse.ui.views">
  <category
    id="com.example.myview"
    name="Sample Category">
  </category>
  <view
    category="com.example.myview"
    class="com.example.myview.views.SampleView"
    icon="icons/sample.gif"
    id="com.example.myview.views.SampleView"
    name="Sample View">
  </view>
</extension>
```

Obrázek 2.4: Příklad zdrojového kódu s připojením na bod rozšíření `org.eclipse.ui.views`.

jakýchkoliv informací o ostatních zásuvných modulech. To zajišťuje snazší spolupráci při implementaci a znovupoužitelnost již implementovaných částí [3]. Bod rozšíření obvykle definuje identifikátor, jméno a schéma (viz obrázek 2.3). Schéma určuje, co musí rozšiřující zásuvný modul splnit pro správné rozšíření tohoto zásuvného modulu. Deklarace bodu rozšíření se vkládá do bloku s označením `<plugin>`.

Dále je zde možno definovat, který zásuvný modul chceme stávajícím modulem rozšířit – k tomu slouží rozšíření zásuvného modulu (viz obrázek 2.4). Obsah bloku `<extension point>` je dán bodem rozšíření, na který se připojujeme. V případě obrázku 2.4 jde o připojení na bod rozšíření `org.eclipse.ui.views`, který umožňuje přidání vlastního pohledu. K tomu je třeba specifikovat kategorii (v případě že vytváříme novou) a zároveň jednotlivé parametry implementovaného pohledu.

Kapitola 3

Testovací rámec JUnit

JUnit je jednoduchý nástroj pro psaní testů a testování aplikací. Jeho vývoj je založen na otevřeném zdrojovém kódu (*angl. open-source*), a proto lze najít množství dalších nástrojů, které rámec JUnit používají nebo z něj vycházejí. Cílem JUnit je poskytovat nástroj, který by umožňoval [1]:

Jednoduchost psaní testů: programátor nemusí psát zbytečně mnoho kódu, zbytek za něj vykoná rámec JUnit.

Snadné pochopení rámce JUnit: rámec by měl být pro Javu přirozený, tak aby se programátor naučil s rámcem pracovat co nejrychleji.

Rychlé provedení testů: spouštění jednotlivých testovacích případů by mělo probíhat bez zbytečných odkladů tak, aby šetřilo programátorům čas.

Izolované provedení testů: rámec by měl poskytovat izolaci jednotlivých testovacích případů, která zajišťuje dostatečnou stabilitu testovacích sad.

Skládat a provádět různé kombinace testů: lze seskupovat a spouštět různé testovací třídy a testovací sady.

Bohužel jsou některé z těchto podmínek v rozporu mezi sebou a tak nelze splnit všechny tyto požadavky v plném rozsahu. V této kapitole je popsán testovací rámec JUnit, jeho architektura, aplikační rozhraní, vybraná existující rozšíření a zásuvný modul pro Eclipse IDE, postavený na tomto rámci.

3.1 Architektura rámce JUnit

V roce 1999 publikoval Kent Beck svůj rámec pro jednotkové testování pro programovací jazyk Smalltalk – *SmalltalkUnit* (zkráceně SUnit). Idea tohoto rámce spočívá v nalezení ideální kombinace mezi jednoduchostí a užitečností. Později Erich Gamma přepsal SUnit do jazyka Java a vytvořil tak JUnit. Posléze tak začaly vznikat mnohé obdoby rámce JUnit podporující mnohé programovací jazyky [7]:

CppUnit: pro jazyk C++

CUnit: pro jazyk C

NUnit: pro jazyk .NET, včetně C#, VB.NET, J# a Managed C++

PyUnit: pro jayzk Python

vbUnit: pro jazyk Visula Basic

utPLSQL: pro jazyk PL/SQL od firmy Oracle

MinUnit: minimalistická verze pro jazyk C

3.1.1 Architektura rámců xUnit

Všechny rámce vycházející z původního návrhu SUnit mají podobnou architekturu. Hromadně se označují jako xUnit a drží se stejných základů. Klíčovými třídami v každém rámci jsou třídy `TestCase`, `TestRunner`, `TestFixture`, `TestSuite` a `TestResult` [7].

`TestCase`: je základní třídou reprezentující jednotkový test. Všechny jednotkové testy jsou odvozeny od této třídy a dědí její vlastnosti.

`TestRunner`: je třída spouštějící jednotlivé testy a zpracovávající jednotlivé výsledky. Existují dva základní typy runnerů – grafický a textový. Nejdůležitější metodou je metoda `run()`, která spouští runner a testy zadané jako parametr této metody.

`TestFixture`: je třída umožňující izolaci jednotlivých testovacích případů. Redukuje množství napsaného kódu a izoluje jednotlivé testovací případy tak, aby průběh jednoho nebyl závislý na výsledku druhého. V zásadě jde o poskytnutí metod, které připraví prostředí před během testovací sady nebo testovacího případu, a po proběhnutí testu zase vrátí prostředí do původního stavu a uvolní alokované zdroje.

`TestSuite`: je třída sloužící k seskupení množství jednotlivých testovacích případů i sad – umožňuje tak spouštět vybrané testovací případy a sady najednou.

`TestResult`: je třída sloužící pro získání výsledků o běhu testů. Instance této třídy se předává jako parametr metody `run()` (u tříd `TestCase` i `TestSuite`). Tato třída uchovává data jako jsou počet běhů testovacích případů, informace o chybách (běhových i testových) a informace o jednotlivých testovacích případech.

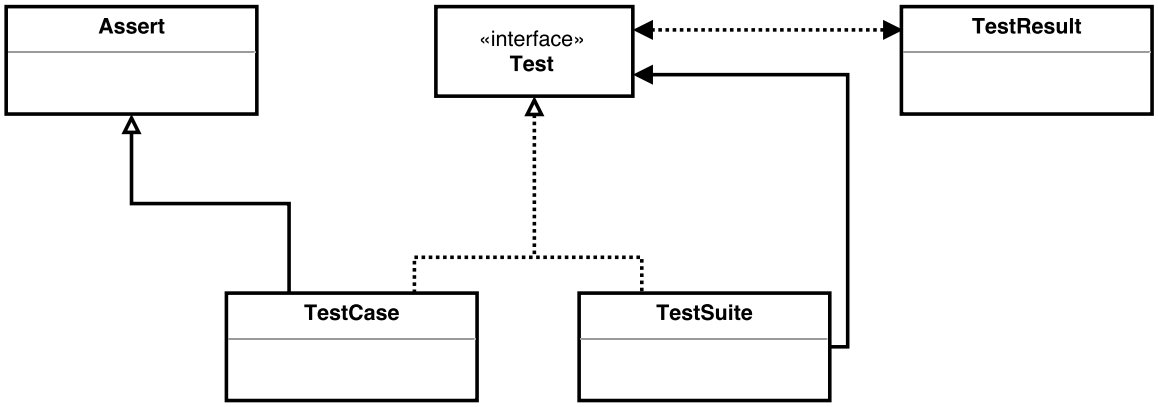
3.1.2 Základní prvky rámce JUnit

Architektura testovacího rámce (viz obrázek 3.1) JUnit se od xUnit liší jen v maličkostech. Obsahuje pět základních tříd, které vytváří aplikační rozhraní rámce: `Assert`, `Test`, `TestCase`, `TestSuite`, `TestResult`.

Třída `Assert` poskytuje kolekci metod sloužících pro ověření výsledků testovacích případů. Tyto metody jsou statické a většinou se volají přímo z daného testovacího případu. Slouží pro porovnání hodnot proměnných a umožňují zadat vlastní zprávu, která je v případě chybné hodnoty vypsána na standardní chybový výstup. V takovém případě je testovací případ ukončen a kód za metodou není vykonán.

Třída `Test` je rozhraní implementované všemi třídami, které se chovají jako testy. Definuje dvě základní metody – `countTestCases()` a `run()`. První zmíněná metoda vrací počet testovacích případů v dané třídě, druhá spustí testy a jejich výsledky ukládá do instance třídy `TestResult`, předaného jako parametr metody `run()`. Obyčejně toto rozhraní implementují třídy `TestCase` a `TestSuite`.

Třída `TestCase` představuje jeden testovací případ. Metoda `countTestCases()` tedy vrací číslo 1 a metoda `run()` spustí tři fáze: „set-up“, testovací metodu a „tear-down“. Díky



Obrázek 3.1: Architektura testovacího rámce JUnit. (převzato z knihy [1], strana 21)

těmto částem lze dosáhnout lepší izolovatelnosti provedení jednotlivých testovacích případů. Výsledek je zaznamenán do instance typu `TestResult` (parametrem metody `run()`).

Třída `TestSuite` je kolekcí dalších testů implementujících rozhraní `Test`. Může tak obsahovat jak jednoduché testovací případy typu `TestCase`, tak celé testovací sady `TestSuite` (popřípadě i jiné objekty implementující rozhraní `Test`). Většina IDE implicitně vytváří testovací sady a tak není potřeba je explicitně definovat.

Jak již bylo dříve zmíněno, třída `TestResult` slouží k uchování výsledků testů. Jedná se o data jako počet běhů testovacích případů, počet běhových chyb, počet testových chyb, detailnější informace o jednotlivých chybách a podobně. Na konci běhu testů instance třídy `TestResult` obsahuje kompletní seznam informací o proběhlých testech. JUnit také poskytuje užitečný nástroj, jakým lze hodnoty v instanci typu `TestResult` sledovat - *listener*. Jedná se o způsob sledování obsahu instance `TestResult`, kde v případě změny instance je daný listener o této události uvědoměn. Přidáním vlastního listeneru pomocí metody `addListener()` uvědomíme JUnit o skutečnosti, že má tomuto listeneru posílat data spojená s určitou instancí třídy `TestResult` [1].

3.2 Aplikační rozhraní rámce JUnit

V následující části se nachází stručný popis aplikačního rozhraní testovacího rámce JUnit, včetně vytváření vlastních testovacích sad, použití parametrizovaných tříd a definice pravidel.

3.2.1 Testovací třídy v rámci JUnit

Testovací třída v rámci JUnit je třída obsahující jednotlivé testovací případy. Konvencí pro pojmenování jednotlivých tříd je použití sufixu „Test“. Jednotlivé testovací případy jsou od ostatních metod odlišeny anotací `@Test`. Díky tomu rámec JUnit pozná jednotlivé testy, které má spustit. Jména metod jednotlivých testovacích případů by měla reflektovat jejich účel [10]. Seznam anotací, které lze v testovací třídě použít, je uveden v tabulce 3.1.

3.2.2 Testovací sady v rámci JUnit

Většina vývojových prostředí umí vytvářet testovací sady automaticky při spouštění (například z vytvořeného projektu). Vývojové prostředí najde všechny testy nacházející se

ANOTACE	VÝZNAM
@Test	Označí metodu jako testovací případ.
@Before	Metoda s touto anotací je zavolána před každým testovacím případem. Obyčejně se používá pro přípravu prostředí před během každého testovacího případu.
@After	Metoda s touto anotací je zavolána po provedení každého testovacího případu. Obyčejně se používá pro navrácení prostředí do stejného stavu jako před testem. Také ji lze použít k uvolnění alokovaných zdrojů zabírajících příliš mnoho paměti.
@BeforeClass	Metoda s touto anotací je zavolána pouze jednou před začátkem testovacích případů z dané testovací třídy. Obyčejně se používá pro přípravu prostředí společného pro všechny testovací případy dané testovací třídy.
@AfterClass	Metoda s touto anotací je zavolána pouze jednou před ukončením testovacích případů z dané testovací třídy. Obyčejně se používá pro navrácení prostředí do stejného stavu jako před zahájením testovací třídy.
@Ignore[("**Důvod**")]	Označí testovací případ jako ignorovaný – přeskakuje se. Do závorky za anotaci lze případně dopsat důvod proč má být testovací případ ignorován.
@Test (expected = Exception.class)	V případě, že testovacím případu nastane testová chyba stejného typu jako je uvedeno v anotaci, chyba se ignoruje.
@Test(timeout = 100)	Pokud je testovací případ vykonáván déle než je uvedeno v anotaci, je ukončen s chybou. Číslo se uvádí v jednotkách milisekund.

Tabulka 3.1: Seznam anotací použitelných v testovací třídě JUnit. (inspirováno v [10])

hierarchicky pod spuštěných uzlem a vytvoří z nich testovací sadu. To ovšem nemusí vždy vyhovovat našemu použití a tak lze vytvářet vlastní testovací sady a seskupovat různé testovací třídy dle naší potřeby. Testovací sada se definuje pomocí anotace `@SuiteClasses`. Jako parametr anotace je zadán seznam jednotlivých testovacích tříd, které do testovací sady patří. Spuštěním testovací sady se spustí všechny testovací třídy a v nich uvedené testovací případy [10]. Do vytvořené testovací sady lze přidávat nejen třídy s testovacími případy, ale také třídy obsahující testovací sady. Před třídu lze navíc doplnit anotaci `@RunWith` specifikující runner, který bude při běhu testů použit. Pokud tato informace není uvedena, je použit výchozí runner.

3.2.3 Parametrizované testovací třídy

JUnit umožňuje vytvoření parametrizované třídy s vícehodnotovými parametry. Tato třída musí obsahovat pouze jednu metodu s anotací `@Test` a statickou metodu s anotací `@Parameters`. Statická metoda je využita k předávání různých hodnot parametrů. Parametry jsou globální proměnné označené anotací `@Parameter`. Testovací případ je poté spuštěn pro každou uvedenou hodnotu parametru [10].

3.2.4 Definice pravidel pro testovací případy

Pomocí anotace `@Rule` je možné upravit chování jednotlivých testovacích případů. Pomocí těchto pravidel[1] lze například vytvářet dočasné složky, externí zdroje nebo definovat očekávanou chybu. JUnit také umožňuje definici vlastních pravidel [10].

3.3 Rozšíření rámce JUnit

Pro pohodlnější testování a jednodušší psaní testovacích případů může být žádoucí si JUnit rozšířit o vlastní funkcionalitu. Použité třídy rámce JUnit jsou poměrně jednoduché a není problém je rozšířit. Pro většinu případů postačí vytvoření nové třídy rozšiřující třídu `TestCase`. Tím se docílí úpravy jednotlivých testovacích případů a může se tak vytvořit množství specializovaných tříd. Pokud tato změna není dostatečná, lze implementovat rozhraní `Test`, které poskytuje větší flexibilitu. Pomocí třídy `TextDecorator` lze obalit testovací případ nebo sadu do bloku a vykonat před nebo po tomto bloku vlastní kód. Dále lze vytvořit vlastní třídy se statickými metodami použitelnými pro vlastní ověření hodnot proměnných, podobně jako ve třídě `Assert` [1].

Dalších změn je možno dosáhnout pomocí tříd `TestResult` a `TestListener`. Pomocí rozšíření třídy `TestResult` je možné předat upravenou třídu jako parametr metody `run()`. Tento přístup umožňuje získaní dalších specifických informací o běhu testů. Nejjednodušším způsobem, jak získat informace o běžících testech, je vytvořit vlastní třídu implementující rozhraní `TestListener` a připojit ji k dané instanci třídy `TestResult` [1]. Tento způsob je použit při implementaci výsledné aplikace a detailní použití je popsáno v kapitole 4.

Markantnější úpravou je vytvoření vlastního runneru. Tato změna poskytuje rozsáhlou kontrolu nad běžícími testy a jejich vykonávání [1]. Pro spuštění vlastního runneru je však třeba tuto skutečnost ve zdrojových souborech testů explicitně definovat pomocí dříve zmíněné anotace `@RunWith`.

Samozřejmě již existují mnohá rozšíření a tak není vždy potřeba psát si vlastní. Tyto rozšíření se většinou zabývají specifickými aspekty dle účelu a architektury testované apli-

[1] Seznam těchto pravidel lze najít na https://github.com/junit-team/junit4/wiki/Rules.

kace. Mezi nejznámější patří *JUnitPerf*, *HtttpUnit*, *JWebUnit*, *Cactus*, *JFCUnit*, *JXUnit* a *Jester* [1].

3.4 Zásuvný modul Eclipse IDE rozšiřující rámec JUnit

Eclipse IDE využívá rámec JUnit pomocí přidání JAR archivu mezi závislosti projektu a zároveň poskytuje vlastní zásuvný modul `org.eclipse.jdt.junit`, který usnadňuje uživateli práci s rámcem JUnit v prostředí Eclipse IDE. Tento zásuvný modul je součástí JDT (viz sekce 2.1) a nabízí například průvodce pro tvorbu nových testovacích případů a sad, vlastní systémy pro spouštění testů nebo pohled JUnit.

Z hlediska implementace výsledné aplikace je důležitá část zásuvného modulu `org.eclipse.jdt.core`, která poskytuje aplikační rozhraní pro interakci Eclipse IDE s rámcem JUnit. Mimo jiné obsahuje třídu `TestRunListener` pro implementaci listeneru a několik rozhraní použitých pro předávání informací o průběhu testů. Jedná se především o rozhraní `ITestCaseElement`, `ITestElement`, `ITestRunSession`. Tato rozhraní jsou abstrakcí nad třídami rámce JUnit a v podstatě odpovídají dříve zmíněným třídám `Test`, `TestCase` a `TestSuite` (viz sekce 3.1.2).

Kapitola 4

Implementovaná aplikace TestRunView

Díky architektuře zásuvných modulů je snadné rozšířit Eclipse IDE o novou funkcionalitu a tak poskytuje četné množství prvků, které je vhodné otestovat. V rámci testování GUI Eclipse IDE se používají rozsáhlé testovací sady testující velké množství aspektů a možností. Automatické testy GUI jsou ale časově náročné a nejsou dokonalé. Vzniká tak potřeba kontroly nad sadou s probíhajícími testy. Průběh testů lze sledovat pomocí zásuvného modulu `org.eclipse.jdt.junit`, který poskytuje pohled JUnit zobrazující potřebné informace o probíhajících testech. Navíc umožňuje znovu spustit testovací sadu a také si uchovává historii běhů testů. Problémem tohoto pohledu je, že při spuštění testovací sady se otevírá nové okno s testovanou instancí Eclipse IDE. To způsobí překrytí pohledu JUnit a proto není možné sledovat tento pohled v průběhu testování. Také z hlediska časové náročnosti může být výhodnější testovat na vzdálených serverech a spouštět testy z terminálu. V takovém případě nelze pohled JUnit k zobrazení detailů o testování použít.

Implementovaná aplikace *TestRunView* (dále zkráceně TRV) poskytuje možnost zobrazení informací o probíhajícím testování GUI Eclipse IDE bez narušení průběhu testů. Mezi důležité informace, které tato aplikace zobrazuje, patří zobrazení průběhu a výsledků jednotlivých testovacích případů v rámci spuštěné testovací sady. V případě nějaké chyby v testovací sadě potom lze lépe prozkoumat, kde chyba nastala a jaká byla její příčina.

Tato kapitola detailně popisuje architekturu, implementaci, způsob testování a využití této aplikace.

4.1 Návrh architektury aplikace TestRunView

V rámci návrhu bylo třeba zvážit způsoby, jakými lze data o probíhajících testech získávat a jak lze tyto data zobrazovat. Ve společnosti *Red Hat* se používají pro testování rámce JUnit a *RedDeer*[1]. RedDeer je rámec pro testování zásuvných modulů Eclipse IDE a využívá k tomu rámec JUnit. Pomocí implementace vlastního runneru dostává kontrolu nad probíhajícími testy. To umožňuje definovat například pořadí spuštěných testů, vytvářet vlastní anotace, nebo definovat nové fáze probíhající v rámci testování.

Data tedy lze získávat z obou zásuvných modulů – JUnitu i RedDeeru. JUnit ovšem narozdíl od RedDeeru poskytuje možnost připojit se pomocí bodu rozšíření `org.eclipse.jdt.junit.testRunListeners` a tak umožňuje širší použití výsledné aplikace. Vy-

[1] https://github.com/jboss-reddeer

tvořený zásuvný modul není třeba explicitně přidávat v kódu rámce nebo testů, ale je automaticky spuštěn.

Možností, jak zobrazit informace zachycené z zásuvného modulu JUnit, je několik:

1. pomocí pohledu (nebo jiné komponenty Eclipse Workbench) přímo v instanci Eclipse IDE s běžícími testy
2. pomocí externí aplikace
3. pomocí nástrojů operačního systému (notifikace, terminál)

Při zobrazování dat je nutné, aby nedocházelo k narušování běhu testů a zároveň byly informace o běžících testech viditelné uživateli. V případě zobrazování informací o probíhajících testech přímo v testované instanci Eclipse IDE by bylo velmi problematické zajistit oba tyto případy. Zobrazení informaci pomocí nástrojů operačního systému by sice umožňovalo nenarušený běh testů, ale přehlednost zobrazených výsledků by byla pravděpodobně nižší. Proto je zobrazení výsledků implementováno pomocí externí SWT aplikace, která umožňuje nenarušený průběh testů a komfortní zobrazení výsledků.

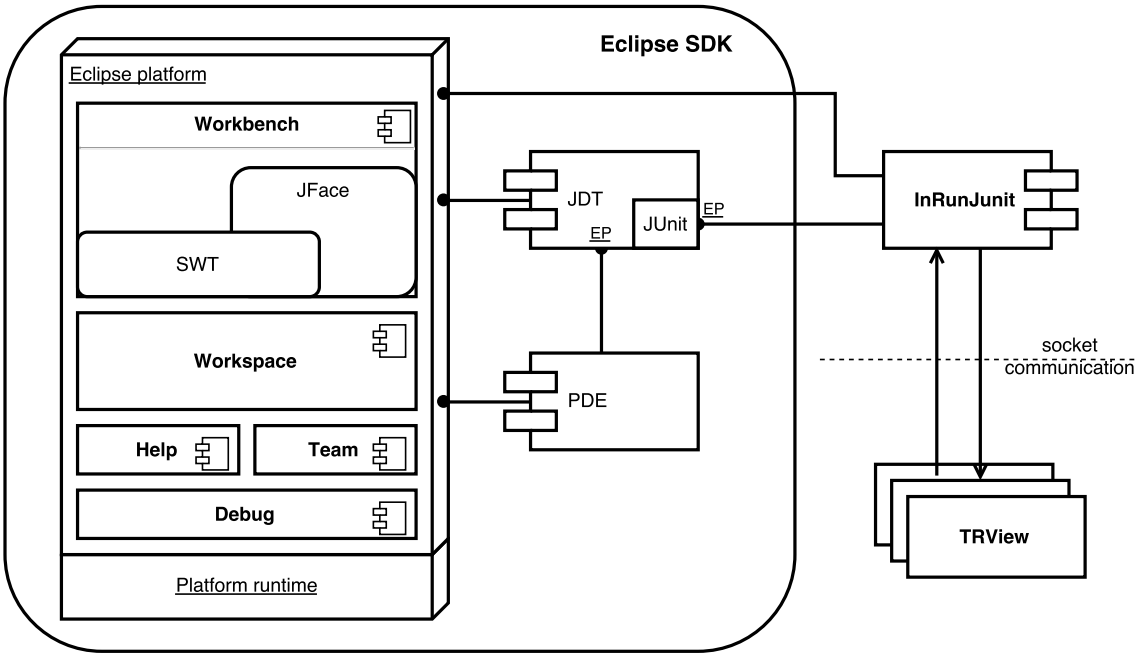
Velmi důležitou částí návrhu je způsob předávání dat mezi částí získávající informace o testech a částí, která tyto informace zobrazuje. To lze řešit například externím souborem nebo pomocí síťové komunikace typu klient-server. V aplikaci je z důvodu možnosti připojení klientské aplikace ke vzdálenému serveru, kde testy probíhají, zvolen způsob síťové komunikace typu klient-server.

Aplikace TRV se skládá ze dvou samostatných částí – zásuvného modulu InRunJUnit a SWT aplikace TRView (viz obrázek 4.1). InRunJUnit je nainstalován jako jeden ze zásuvných modulů platformy Eclipse a slouží k získání a zpracování výsledků ze zásuvného modulu JUnit. Zároveň slouží jako server, ke kterému lze připojovat klientské aplikace. TRView zpracovává data o probíhajících testech a zobrazuje je uživateli. Tato data přijímá pomocí soketů ze serveru vytvořeného v zásuvném modulu InRunJUnit. *JDT* (*Java Development Tools*) a *PDE* (*Plug-in Development Environment*) jsou zásuvné moduly, které rozšiřují funkcionalitu platformy Eclipse. JDT přidává nástroje pro práci s Javou a PDE přidává funkcionalitu potřebnou pro vývoj zásuvných modulů. Platforma Eclipse je popsána v kapitole 2.

4.1.1 Návrh architektury zásuvného modulu InRunJUnit

Integrace zásuvného modulu InRunJUnit do platformy Eclipse je znázorněna na obrázku 4.1. InRunJUnit je nainstalován jako jeden ze zásuvných modulů pro Eclipse a zároveň je připojen pomocí bodu rozšíření k zásuvnému modulu JUnit, který je součástí JDT. Zásuvný modul se skládá ze dvou částí – *listeneru* a serveru. Listener slouží pro zachycení výsledků ze zásuvného modulu JUnit. Opakem listeneru je potom *notifier*, jehož účelem je informovat listener o nových datech. Listener je pomocí bodu rozšíření poskytovaného zásuvným modulem JUnit zaregistrován mezi ostatní listenery. Poté, co se spustí JUnit za účelem testování, si JUnit zjistí, které zásuvné moduly jsou k bodu rozšíření připojeny, a automaticky je informuje o průběhu testů. Serverová část zásuvného modulu se stará o vytvoření serveru, komunikaci s klienty, vytvoření relevantních dat ve formě řetězců a jejich posílání všem připojeným klientům. Při posílání dat také specifikuje fázi testování[2], ze které data pochází.

[2]Může se jednat o zahájení (případně ukončení) testovací sady nebo testovacího případu.



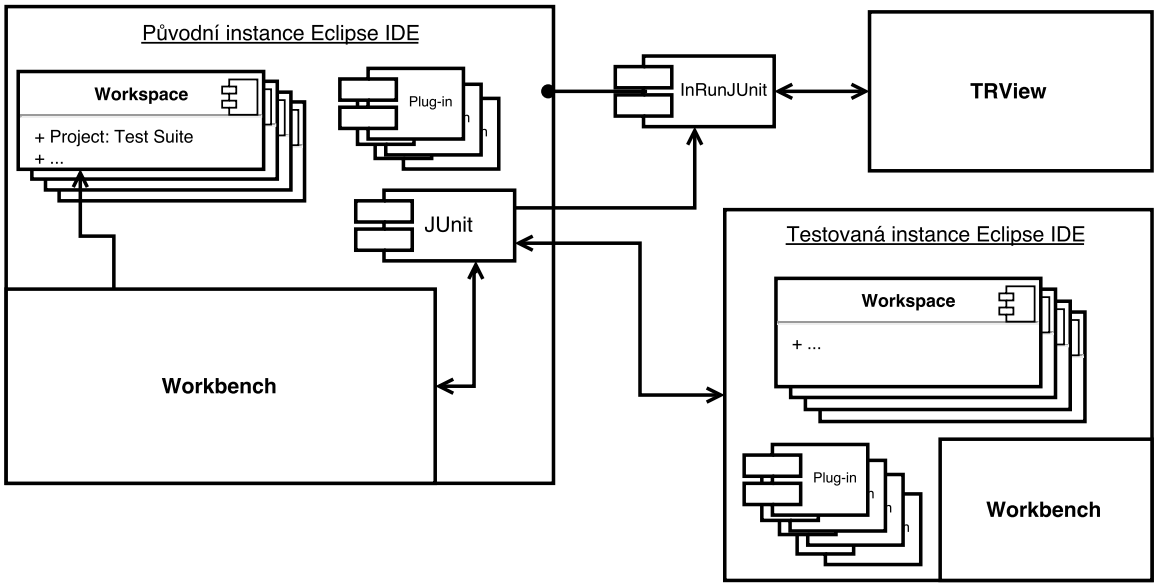
Obrázek 4.1: Znázornění architektury aplikace TRV.

4.1.2 Návrh architektury aplikace TRView

Aplikace TRView se také skládá ze dvou částí – klienta a obsluhy GUI. Klientská část umožňuje připojení k serveru a ukládání přijatých výsledků. Při přijetí dat ze serveru také inicializuje zpracování a zobrazení průběhu testů do GUI. Jedna část GUI se stará o vytvoření a obsluhu grafického uživatelského rozhraní, druhá se stará o zobrazení důležitých informací o průběhu testování. To zahrnuje:

- pole pro zadání adresy a portu serveru
- počet testovacích případů, které poběží
- počet testových chyb (*angl. failures*) v testech
- počet běhových chyb (*angl. errors*) v testech
- počet přeskočených testů
- stromovou strukturu jednotlivých testovacích případů s:
  - rozlišením, který test je aktivní.
  - rozlišením výsledku testovacího případu.
  - časem udávajícím trvání testovacího případu.
- zobrazení zásobníku volání (*angl. stack trace*) pro neúspěšné testy

V případě, že testy a klientská aplikace poběží na jednom systému (viz obrázek 4.2), je třeba vzít v úvahu problém aktivního okna. V původní instanci je z Eclipse Workbench pomocí zásuvného modulu JUnit spuštěn projekt obsahující zdrojové soubory testů. Zásuvný modul JUnit uvědomí zásuvný modul InRunJUnit o zahájení testů. Zároveň pomocí



Obrázek 4.2: Znázornění funkce aplikace TRV při spuštění testů z GUI Eclipse IDE.

runneru řídí a sleduje průběh testování. Je spuštěna nová instance Eclipse IDE, ve které probíhají testy. Tato instance používá vlastní pracovní plochu i ostatní zdroje. Při každé klíčové fázi testování uvědomí zásuvný modul JUnit modul InRunJUnit o změně. Zásuvný modul InRunJUnit mezitím vytvoří server, ustanoví spojení s případnými klienty aplikace TRView a průběžně jim posílá vybraná data.

Některé testy vyžadují pro nalezení a otestování komponent aktivitu daného okna. To může způsobit překrytí okna aplikace TRView a tak znemožnit zobrazení informací o probíhajících testech uživateli. Proto je okno aplikace TRView vytvořeno „vždy nahoře“ (*angl. always on top*). Okno je tak vždy viditelné, i když není právě aktivní. Přesto je třeba dát pozor, aby při průběhu testů bylo aktivní okno s testovanou instancí Eclipse IDE – například interakcí s aplikaci TRView by mohlo dojít ke ztrátě aktivity okna testované instance.

4.1.3 Integrace zásuvného modulu InRunJUnit a SWT aplikace TRView

Komunikace mezi zásuvným modulem InRunJUnit a aplikací TRView je velmi důležitou částí návrhu. Ovlivňuje spoustu klíčových aspektů výsledné aplikace, které je třeba zvážit:

- rychlost komunikace
- možnost komunikace po síti
- počet možných klientů (instancí aplikace TRView) připojených k zásuvnému modulu InRunJUnit
- oboustrannost komunikace

V aplikaci TRV je komunikace vyřešena pomocí klient-server modelu předávajícího informace pomocí soketů. Tento přístup poskytuje jednoduché a zároveň kompletní řešení uvedených problémů. Zpracováním dat v zásuvném modulu InRunJUnit lze posílat pouze relevantní data a tak minimalizovat objem posílaných dat a zrychlit komunikaci. Navíc tento model umožňuje připojení klientské aplikace po síti a řeší tak jednoduše problém

s aktivitou oken. Díky vícevláknové implementaci lze připojit více klientských aplikací a zároveň umožnit oboustrannou komunikaci mezi všemi klienty a serverem. Oboustranná komunikace slouží ke korektnímu ukončování jak serveru, tak klienta. V případě ukončení klienta je nutno oznámit serveru, že tomuto klientovi již nemá posílat data. V případě ukončení serveru je nutno oznámit všem klientům, že nemají čekat na další data. Oboustrannou komunikaci by však bylo možno využít i pro budoucí rozšíření, například pro kontrolu běhu testů z klientské aplikace.

4.2 Implementace aplikace TRV

Aplikace TRV je implementována v programovacím jazyce Java. Programovací jazyk Java byl vybrán na základě nutnosti implementace zásuvného modulu pro Eclipse IDE a následné komunikace s externí aplikací. Vývoj aplikace probíhá pomocí verze *JavaSE-1.8* a je doporučeno tuto verzi pro další vývoj nebo testování používat.

4.2.1 Implementace zásuvného modulu InRunJUnit

Součástí implementace zásuvného modulu je vytvoření manifestů zásuvného modulu, implementace listeneru a implementace serverové části. Více o funkci manifestů a architektuře zásuvných modulů je uvedeno v kapitole 2.

Manifesty zásuvného modulu InRunJUnit

Manifesty zásuvného modulu slouží jako zdroj základních informací pro platformu Eclipse. V souboru `MANIFEST.MF` jsou data poskytující základní informace o daném modulu (viz obrázek 4.3). Informace `Manifest-Version`, `Bundle-ManifestVersion`, `Bundle-Name`, `Bundle-SymbolicName`, `Bundle-Version` slouží pro popis manifestu a jednoznačnou identifikaci zásuvného modulu a z hlediska implementace nejsou příliš zajímavé. `Require-Bundle` obsahuje seznam všech ostatních balíčků, které jsou vyžadovány pro správné fungování zásuvného modulu. Za každým balíčkem může být ještě specifikována konkrétní verze balíčku, který je vyžadován. Balíčky uvedené na obrázku 4.3 poskytují některé metody a třídy, které byly použity k implementaci listeneru a serveru. `Bundle-RequiredExecutionEnvironment` specifikuje prostředí pro provádění aplikace – JavaSE verze 1.8.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: InRunJUnit
Bundle-SymbolicName: com.mcoufal.inrunjunit;singleton:=true
Bundle-Version: 1.0.0.qualifier
Require-Bundle: org.junit;bundle-version="4.12.0",
 org.eclipse.core.runtime;bundle-version="3.12.0",
 org.eclipse.debug.ui;bundle-version="3.11.201",
 org.eclipse.jdt.core;bundle-version="3.12.1",
 org.eclipse.jdt.junit.core
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
```

Obrázek 4.3: Zdrojový kód manifestu MANIFEST.MF obsahující základní informace o zásuvném modulu InRunJUnit.

Manifest `plugin.xml` obsahuje pouze informaci, že třída `com.mcoufal.inrunjunit.listener.JUnitListenerEP` implementuje potřebné rozhraní pro bod rozšíření `org.eclipse.jdt.junit.testRunListeners` (viz obrázek 4.4). Tato informace umožňuje, aby byl implementovaný listener načten a informován zásuvným modulem JUnit o průběhu testování. První dva řádky specifikují pouze verzi jazyka XML, ve kterém je manifest zapsán, a verzi Eclipse IDE, ve které byl zásuvný modul vyvíjen.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>

<plugin>
  <extension point="org.eclipse.jdt.junit.testRunListeners">
    <testRunListener class=
     "com.mcoufal.inrunjunit.listener.JUnitListenerEP"/>
  </extension>
</plugin>
```

Obrázek 4.4: Zdrojový kód manifestu plugin.xml zásuvného modulu InRunJunit.

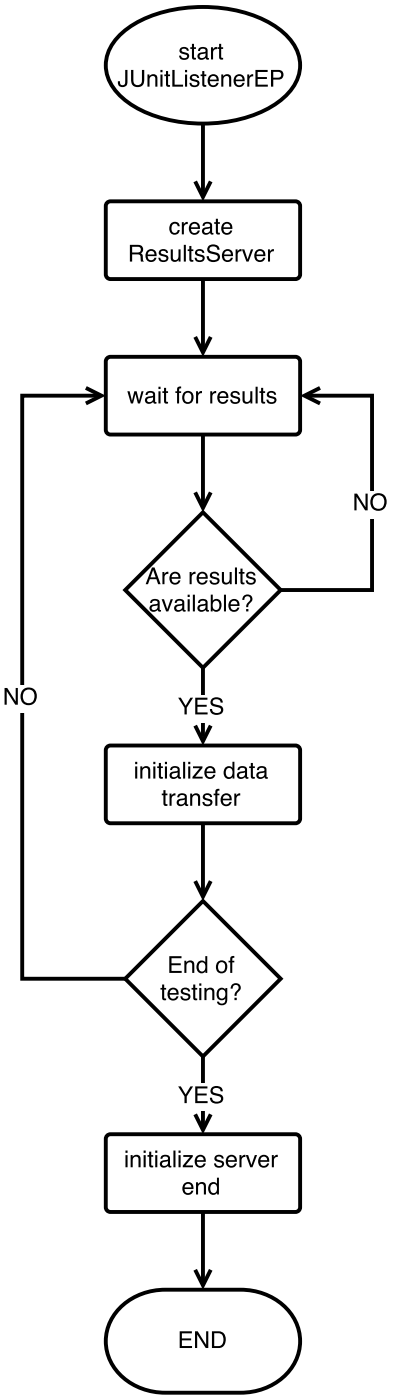
Implementace listeneru

V balíčku `listener` je obsažena pouze jedna třída – `org.mcoufal.inrunjunit.listener.JUnitListenerEP`. Funkce této třídy je znázorněna v diagramu na obrázku 4.5. Instance třídy JUnitListenerEP je automaticky vytvořena zásuvným modulem JUnit, a to díky deklaraci připojení k bodu rozšíření `org.eclipse.jdt.junit.testRunListeners`. Při inicializaci si tato instance vytváří nový server, který je obsluhován dalším vláknem. Metody listeneru jsou poté volány pokaždé, když začne příslušná fáze v testování. To umožní při zavolání této metody zpracovat výsledky přijaté jako parametr dané metody a inicializovat jejich poslání pomocí dříve vytvořeného serveru. Pokud je zavolána metoda spojená s koncem testovací sady, je po předání výsledků serveru zároveň zahájeno jeho ukončení. Pro úspěšné ukončení serveru je nejdříve nutno zrušit případné blokující operace (v tomto případě `ServerSocket.accept()`), a poté porušit podmínku cyklu, ve kterém server obsluhuje nově připojené klienty.

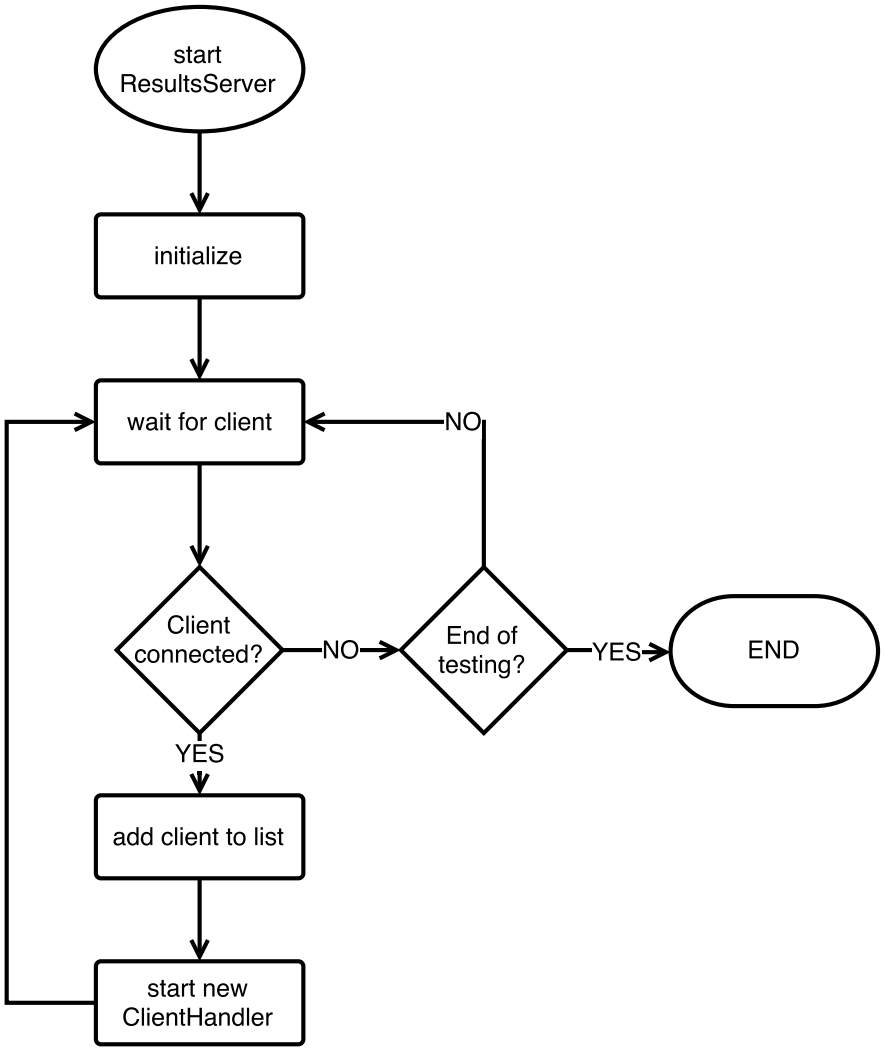
Nejdůležitějším krokem implementace třídy `JUnitListenerEP` je rozšíření třídy `org.eclipse.jdt.junit.TestRunListener`. Přepsáním (*angl. override*) metod této třídy je dosaženo kontroly nad jednotlivými metodami, které rámec JUnit volá a tak informuje listener o průběhu testování. Více o způsobu získávání dat pomocí listeneru je uvedeno v sekci 3.3.

Implementace serverové části

Hlavní třídou definující chování serveru je třída `ResultsServer` (viz obrázek 4.6). Tato třída má na starosti vytvoření serveru a správu klientů. Po vytvoření serveru na portu číslo 7357 se čeká na připojení klienta. Na každého připojeného klienta si ukládá odkaz a zároveň vytváří nové vlákno, které má na starosti obsluhu daného klienta. Tato obsluha je implementována ve třídě `ClientHandler`. Díky vytvořenému seznamu odkazů na jednotlivé klienty umožňuje třída `JUnitListenerEP` inicializovat posílání dat jednotlivým klientům a zároveň pomocí více vláken umožňuje obsluhu více klientů najednou. V případě zachycení dat spojených s ukončením testovací sady je po poslání dat server ukončen. Pro názornost



Obrázek 4.5: Diagram znázorňující funkci implementovaného listeneru `JUnitListenerEP` rozšiřujícího třídu `org.eclipse.jdt.junit.testRunListeners`.

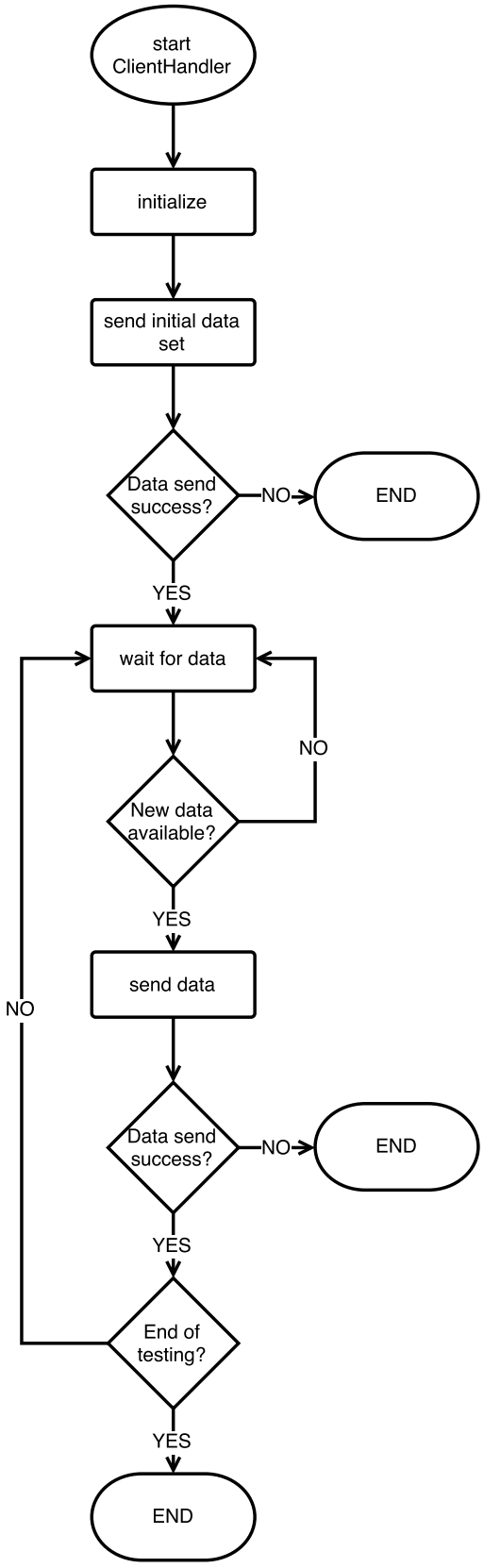


Obrázek 4.6: Diagram znázorňující funkci implementovaného serveru `ResultsServer` zajišťujícího komunikaci mezi zásuvným modulem InRunJUnit a aplikací TRView.

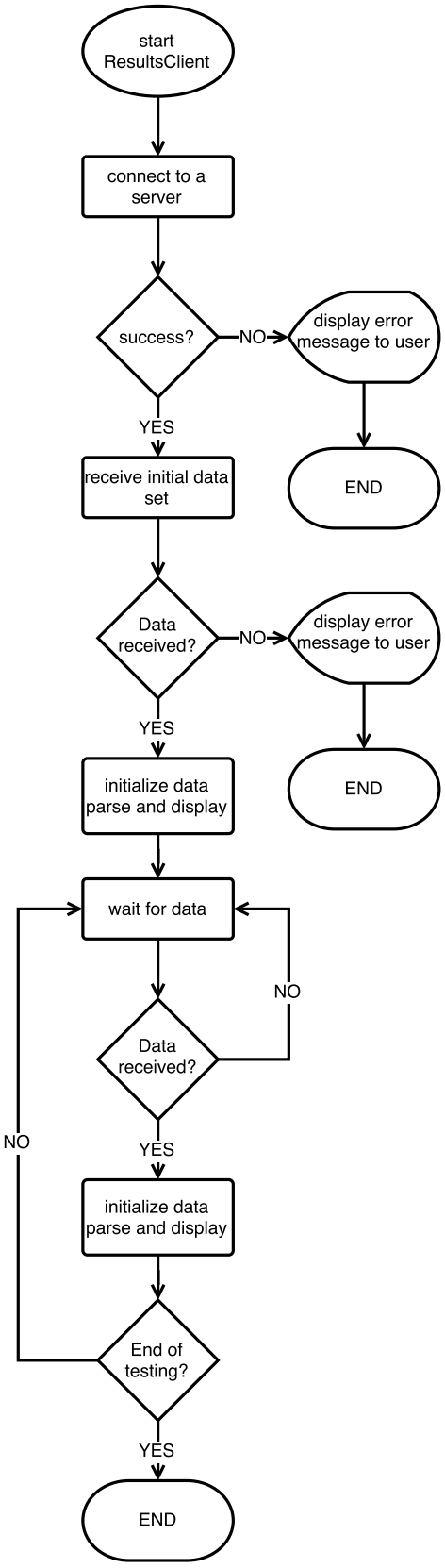
je v diagramu na obrázku ukončení serveru zaznačeno, ve skutečnosti však ukončení serveru probíhá asynchronně.

Hlavním účelem třídy `ClientHandler` je obsluha klienta a přenos dat. Chování této třídy je znázorněno na obrázku 4.7. Jakmile dojde k inicializaci a úspěšnému spojení s klientem, posílá se počáteční soubor dat. Tento soubor obsahuje všechna data zachycená pomocí listeneru do doby, než došlo k připojení klienta. Nedochází tak ke ztrátě dat v případě, že se klient připojí až v průběhu testů. Dále se již posílají jen nově zachycená data (odesílání těchto dat inicializuje `JUnitListenerEP` prostřednictvím instance `ResultsServer`). Díky tomuto modelu komunikace nedochází k posílání irelevantních dat a zbytečnému zatížení síťové komunikace. Při zachycení dat z fáze ukončení testovací sady dojde k poslání dat klientovi a poté se `ClientHandler` ukončí.V případě ukončení uživatelem z grafického uživatelského rozhraní nejdříve dojde k výjimce při posílání dat a poté se `ClientHandler` ukončí.

Dále serverová část obsahuje pouze třídy použité pro zpracování a posílání dat. Cílem těchto tříd je uložení potřebných informací do serializovatelné formy. Dosáhneme tak zmen-



Obrázek 4.7: Diagram znázorňující funkci obsluhy klienta implementovanou v třídě `ClientHandler`.



Obrázek 4.8: Diagram znázorňující funkci obsluhy klienta implementovanou v třídě `ResultsClient`.

šení objemu posílaných a jednoduššího zpracování dat v klientské aplikaci. Hlavní třídou pro manipulaci s výsledky je třída `ResultsData`. Ta obsahuje vždy instanci jednoho objektu a fázi, ve které byla data objektu zachycena. Díky fázi potom klient pozná, jak s objektem nakládat při zpracování a zobrazení dat. Ostatní třídy se starají pouze o uložení dat jednotlivých objektů do řetězcové nebo číselné podoby. Jedná se o třídy `StringDescription`, `StringResult`, `StringTestCaseElement`, `StringTestElement`, `StringTestRunSession` a `TestRunSessionParser`.

4.2.2 Implementace aplikace TRView

Implementace aplikace TRView je rozdělena do dvou balíčků – `view` a `client`. Balíček `view` zajišťuje vytvoření a obsluhu grafického uživatelského rozhraní a zároveň poskytuje metody pro zpracování a zobrazení dat získaných pomocí balíčku `client`. Balíček `client` se stará o komunikaci se serverem a zpracování přijatých dat.

Implementace balíčku `view`

Balíček `view` obsahuje dvě třídy – `TRView` a `ResultsParser`. Třída `TRView` tvoří základ aplikace – inicializuje a vytváří grafické uživatelské rozhraní aplikace a definuje chování pro akce, které nastaly v GUI (akce provedené uživatelem pomocí myši nebo klávesnice). Pokud uživatel zadá IP adresu a port serveru, na kterém běží testy, stiskem tlačítka `CONNECT` se resetuje GUI, ukončí se instance stávajícího klienta `ResultsClient` (pokud byl již nějaký vytvořen) a vytvoří se nové vlákno zajišťující jeho funkci. V případě označení některého z testovacích případů se zobrazí výpis zásobníku volání. Pokud uživatel ukončí aplikaci, zahájí se ukončení klienta `ResultsClient` a aplikace se ukončí. Pro úspěšné ukončení klienta je nejdříve nutno zrušit případné blokující operace (v tomto případě `Socket.getInputStream()`), a poté porušit podmínku cyklu ve kterém klient přijímá data ze serveru. Chování implementované v třídě TRView je znázorněno na obrázku 4.9.
Grafické uživatelské rozhraní se skládá z:

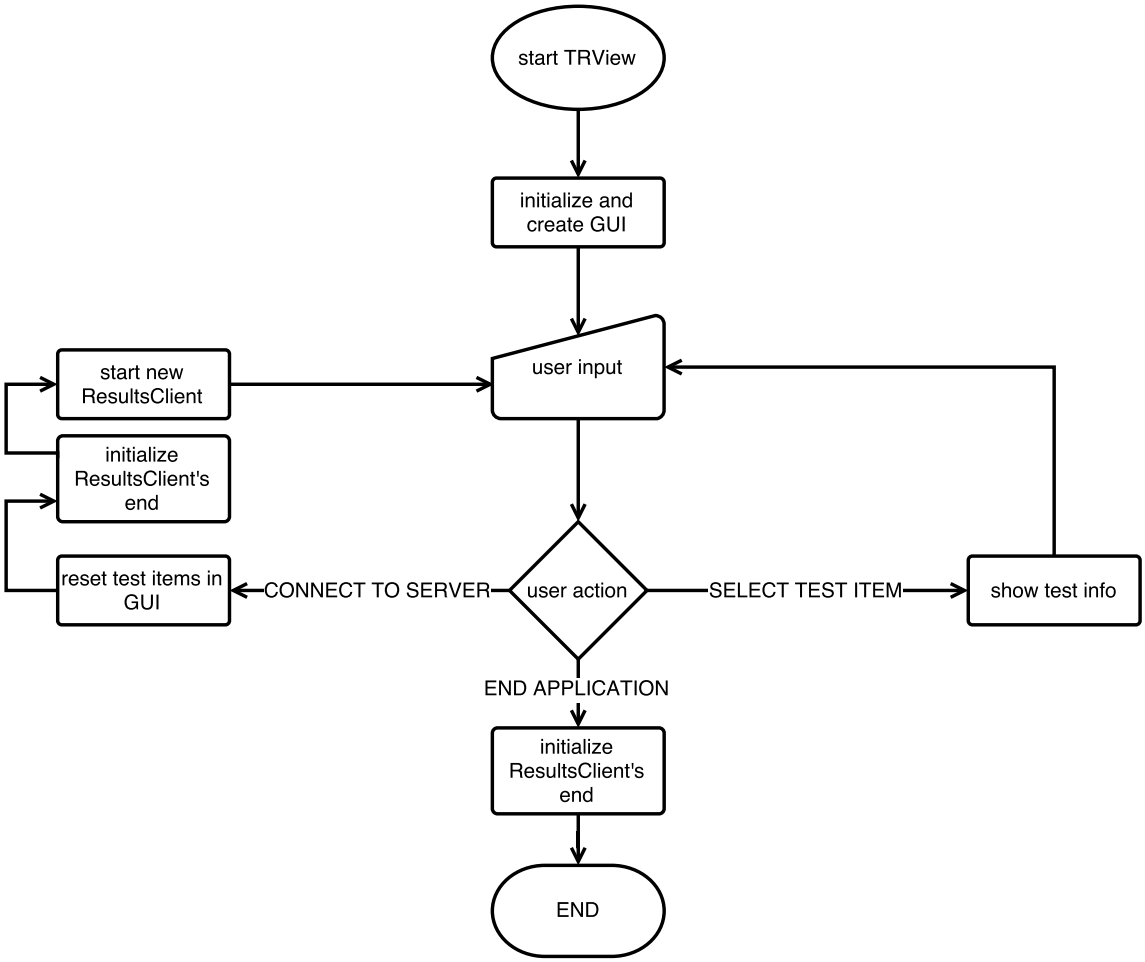
Menu: obsahuje jednoduchou nabídku pro snadné ovládání aplikace. Nabídka obsahuje možnosti `Connect...`, `Re-connect`, `Disconnect` a `Exit`, sloužící pro připojení a opětovné přípojení k serveru, odpojení od serveru a ukončení aplikace. Dále obsahuje jednoduché nastavení s možností vypínat a zapínat automatické posouvání na probíhající testovací případ ve stromové struktuře. Nakonec umožňuje nastavení rozbalování zobrazené stromové struktury.

Popisků: slouží k identifikaci jednotlivých komponent (říkají tak uživateli, jaký je význam dat dané komponenty)

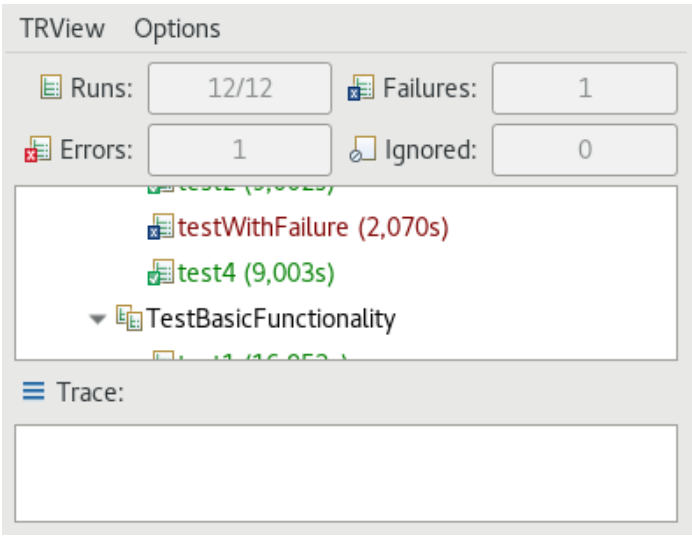
Textových polí: slouží pro zobrazení informací (počet chyb, běhů testů, atd.) nebo pro zadávání informací uživatelem (IP adresa serveru, port serveru).

Stromové struktury: nejdůležitější část GUI aplikace. Do této struktury se zobrazuje průběh testů jednotlivých testovacích případů. Pro každý uzel ve stromu je zobrazena příslušná ikona (viz tabulka 4.1) informující o jeho stavu.

Zásobníku volání (*angl. stack trace*): textový záznam zobrazující výpis metod, které byly volány než došlo u vybraného testovacího případu k chybě. Pokud žádná chyba nenastala, textové pole obsahuje pouze řetězec informující uživatele o tom, že je zásobník volání prázdný.



Obrázek 4.9: Diagram znázorňující funkci třídy TRView.



Obrázek 4.10: Grafické uživatelské rozhraní implementované aplikace TRView.

Ikona	Význam
	Uzel obsahující testovací případy. Může se jednat o testovací sadu, balíček nebo testovací třídu.
	Testovací sada právě běží.
	Testovací sada proběhla v pořádku.
	Testovací sada proběhla, nastalo v ní jedna nebo více běhových chyb.
	Testovací sada proběhla, nastalo v ní jedna nebo více testových chyb.
	Uzel představující jeden testovací případ.
	Testovací případ právě běží.
	Testovací případ proběhl v pořádku.
	Testovací případ skončil s běhovou chybou.
	Testovací případ skončil s testovou chybou.
	Testovací případ neproběhl – byl přeskočen.
	Testovací případ skončil , stav je neznámý a předpokládá se chyba.

Tabulka 4.1: Seznam ikon použitých v grafickém uživatelském rozhraní aplikace TRView.

Třída `ResultsParser` obsahuje metody, které na vstupu zpracovávají instance třídy `ResultsData` a zobrazují je do GUI. Díky fázi uvedené v objektu `ResultsData` lze poznat, v jaké fázi testování byla data zachycena a podle toho data zpracovat a zobrazit. Jednotlivé fáze jsou definovány v třídě `JUnitListenerEP` zásuvného modulu InRunJUnit.

Implementace balíčku `client`

Balíček `client` obsahuje pouze třídu `ResultsClient`. Funkce implementovaná touto třídou je znázorněna na obrázku 4.8. Instance třídy `ResultsClient` vzniká po zadání detailů připojení a stisku tlačítka `CONNECT` uživatelem. Instance se poté zkusí připojit k serveru na zadané IP adrese a portu. Pokud se připojení nezdaří, je uživateli vypsáno oznámení o chybě. Pokud proběhne připojení úspěšně, dochází k přijetí a zpracování počátečního souboru dat. Spojení mezi klientem a serverem je postaveno na spojované službě TCP a tak by nemělo docházet ke ztrátě dat. Pokud ovšem dojde k závažnějšímu problému na síti, je uživateli zobrazeno oznámení o chybě a `ResultsClient` je ukončen. Poté klient čeká na další data. Jakmile data obdrží, zahájí jejich zpracování a zobrazení pomocí třídy `ResultsParser`. Jelikož upravujeme komponenty vytvořené jiným vláknem, je zapotřebí zajistit, aby byl kód vykonán vláknem obsluhujícím GUI a nedocházelo tak k chybě typu *Invalid Thread Access*. Proto je tato operace prováděna pomocí synchronizace vláken (viz obrázek 4.11). SWT poskytuje metodu `syncExec()`, která pozastaví průběh současného vlákna, a argument této metody předá vláknu obsluhujícího GUI. Jakmile je to možné, vlákno obsluhující GUI vykoná kód daný argumentem metody a dále pokračuje ve své práci. Po vykonání kódu je vlákno, ze kterého byla metoda `syncExec()` volána, zase spuštěno [8].

```
TRView.getDisplay().syncExec(new Runnable() {
        @Override
        public void run() {
                ResultsParser.parseAndDisplay(receivedData);
        }
});
```

Obrázek 4.11: Zdrojový kód znázorňující práci s komponentami GUI z vlákna klienta `ResultsClient`. Metoda `parseAndDisplay(ResultsData)` zpracovává data a mění dle těchto dat GUI. Díky metodě `syncExec()` je kód této metody vykonán vláknem obsluhujícím GUI, a tak nedochází k chybě typu *Invalid Thread Access*.

V případě zachycení dat s fází konce testovací sady se stejným způsobem zahájí zpracování a zobrazení dat a poté se klient ukončí. Aktivní zůstává pouze vlákno obsluhující GUI.

4.3 Testování

Testování projektu proběhlo prozatím pouze manuálně – byl vytvořen jeden projekt obsahující dvě testovací sady. Tyto sady byly poté manuálně spouštěny a v jejich průběhu byla ověřována funkčnost nástroje interakcí s implementovaným nástrojem. Testovací případy v každé sadě se zaměřují na jiné aspekty aplikace.

První testovací sada je zaměřená pouze na obecnou funkčnost nástroje. Obsahuje několik testovacích případů s různými anotacemi a testuje tak zobrazení jednotlivých uzlů v okně aplikace TRView. Při spuštění testů pouze části testovací sady (například balíček nebo testovací třídu) očekává přizpůsobení stromové struktury v okně TRView.

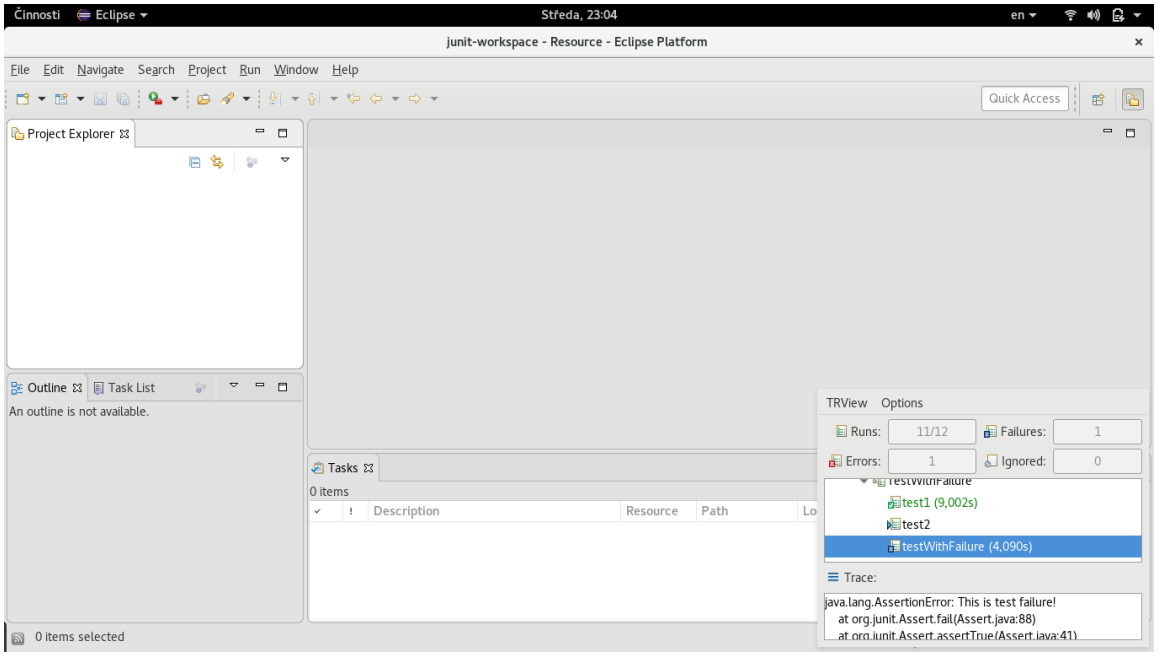
Druhá testovací sada je zaměřena na testování aktivity okna. Pomocí zásuvného modulu SWTBot se v testech hledají jednotlivé prvky aplikace. Pokud SWTBot prvek nenajde, došlo pravděpodobně k narušení aktivity okna. Při manuálním testování je třeba rozlišovat, zda narušení způsobila aplikace TRView nebo tester.

Nástroj byl testován na operačních systémech Linux (distribuce Fedora a RHEL). K testování byla použita verze Neon (4.6) platformy Eclipse.

4.4 Praktické využití

Využití aplikace TRV spočívá především v použití při vývoji nebo opravách testů pro GUI platformy Eclipse. Spuštěním sady testů se otevře okno testované instance Eclipse IDE, kde probíhají jednotlivé testovací případy dané testovací sady. Bez pohledu JUnit však nelze jednoduše zjistit, který testovací případ právě běží a jak předchozí testovací případy dopadly. Na obrázku 4.12 je snímek obrazovky zachycující funkci aplikace TRV.

Pro funkcionalitu aplikace TRV je nutno mít v Eclipse nainstalovány zásuvné moduly JUnit a InRunJUnit. Na obrázku 4.2 je znázorněno spuštění testovací sady z instance Eclipse IDE. Eclipse SDK představuje soubor všech komponent a zásuvných modulů, které Eclipse poskytuje jako minimální balíček pro vývoj aplikací. K tomuto minimálnímu balíčku lze instalovat další zásuvné moduly. V tomto případě se jedná o rámec JUnit a zásuvný modul InRunJunit. Uživatel vidí pouze Workbench, který mu umožňuje pracovat s jednotlivými projekty uloženými v pracovní ploše (Workspace). Spuštěním projektu s testovací sadou



Obrázek 4.12: Snímek obrazovky pořízený při testování GUI Eclipse IDE zobrazující stav běhu testů nástrojem TRV.

se uvědomí zásuvný modul JUnit a zvolený runner. Runner se stará o průběh testů a zároveň získává informace o průběhu testů. Tyto informace posílá zásuvnému modulu JUnit, který je dále zpracovává. Díky připojení pomocí bodů rozšíření může zásuvný modul JUnit automaticky informovat zásuvný modul InRunJUnit o průběhu testů. InRunJUnit tyto informace pomocí soketů posílá aplikaci TRView, která je zobrazuje na obrazovku. Tato komunikace může probíhat buď po síti (testy tak poběží na jednom stroji a aplikace TRView na druhém) nebo lze komunikovat pomocí *localhost* adresy na jednom stroji.

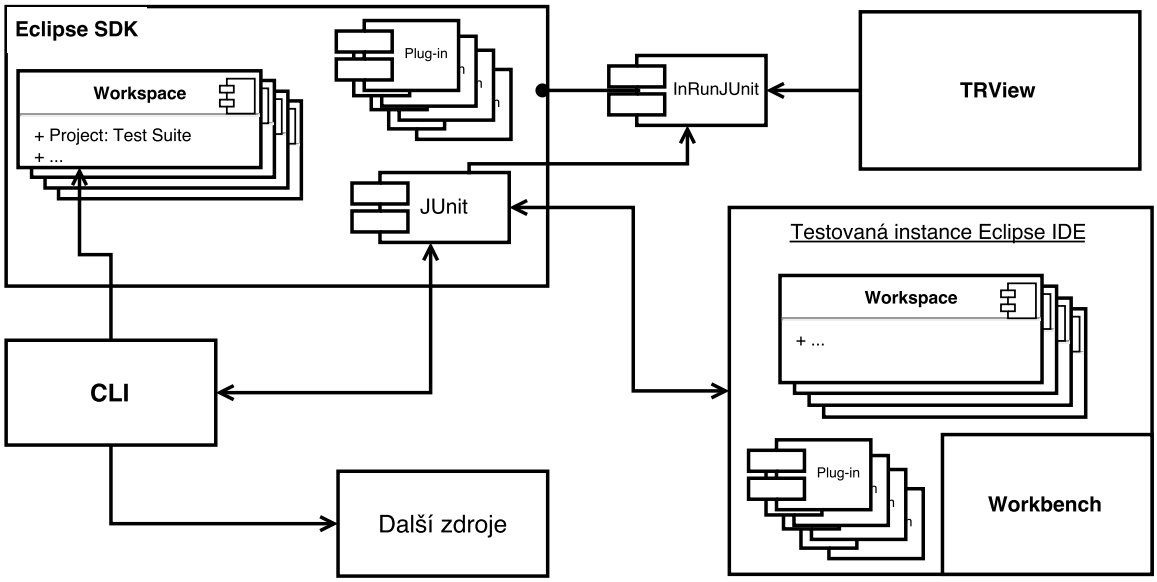
4.4.1 Instalace aplikace TRV

Instalace je odlišná pro obě části nástroje TRV. Zásuvný modul InRunJUnit lze instalovat pomocí Eclipse Update site a aplikaci TRView stačí pouze spustit pomocí JAR archivu.

Instalace zásuvného modulu InRunJUnit

Pro instalaci zásuvného modulu InRunJUnit je zapotřebí vývojové prostředí Eclipse s JDT. Doporučena je verze Neon (4.6) a vyšší. Instalaci lze provést pomocí *Eclipse Update site* buď z online repositáře nebo lokálně. Pro instalaci zásuvného modulu InRunJUnit je zatím vytvořen pouze lokální Update site. Z lokálního zdroje lze nainstalovat zásuvný modul pomocí `Help -> Install New Software... -> Add... -> Local...` , kde se zadá cesta k Eclipse Update site projektu. Ten je dostupný na paměťovém médiu přiloženém k této práci nebo na webové službě GitHub[3].

[3] https://github.com/mcoufal/TRV/tree/master/mcoufal.InRunJUnit.updateSite



Obrázek 4.13: Znázornění funkce aplikace TRV při spuštění testů z terminálu.

Instalace aplikace TRView

Pro spuštění aplikace TRView je zapotřebí pouze stažení JAR archivu a jeho následovné spuštění pomocí javy. Tento archiv je dostupný na paměťovém médiu přiloženém k této práci nebo na webové službě GitHub[4]. Archiv TRView.jar se nachází ve složce *TRV/TRView* a je možné ho spustit pomocí příkazu '`java -jar TRView.jar`'.

4.4.2 Použití při testování Eclipse IDE v Red Hat

Testování Eclipse IDE probíhá z velké části automatizovaně. Testovací sady jsou spouštěny oproti jednotlivým verzím operačních systémů Linux, Windows a macOS, s různými verzemi OpenJDK nebo OracleJDK. Tyto testy jsou spouštěny buď manuálně, nebo po splnění nastavených podmínek pomocí serveru *Jenkins*[5]. K průběhu takto spuštěných testů bohužel chybí pořádná zpětná vazba – lze se orientovat pouze podle výpisů v terminálu, které jsou nepřehledné a navíc jsou zahlceny dalšími pro testy irelevantními daty. Testy jsou spuštěny příkazem `java`[6] a nevzniká tak druhé okno s GUI Eclipse IDE. Tento způsob spuštění zachycuje diagram na obrázku 4.13. Je spuštěna pouze jedna instance Eclipse IDE s parametry definujícími mimo jiné umístění pracovní plochy, identifikaci balíčku s implementovanými testy a nastavení parametrů pro cílovou platformu, na které testy poběží. Zásuvný modul InRunJUnit funguje na pozadí v rámci aplikace řídící spuštěné testy.

4.5 Další rozšíření

Díky obousměrné komunikaci mezi klientskou a serverovou částí má aplikace TRV potenciál na mnohá rozšíření. Inspirací pro rozšíření by mohl být dříve zmíněný pohled JUnit, který

[4] https://github.com/mcoufal/TRV/tree/master/TRView
[5] https://jenkins.io/
[6] Příklad spuštění z konzole lze najít na https://wiki.eclipse.org/SWTBot/Automate_test_execution.

poskytuje možnosti konfigurace zobrazení, historii spuštěných běhů testů a možnost testy zastavit nebo znovu spustit.

Plánovaným rozšířením je mód *debug*, který způsobí pozastavení výkonu testů, než se připojí alespoň jeden klient aplikace TRView. Zajistí se tak uživateli dostatek času na připojení a možnost sledování průběhu testů již od spuštění.

Další užitečnou funkcionalitou by bylo například pozastavení průběhu testů. Uživatel by tak mohl mezitím lépe prozkoumat příčinu neúspěšného testu nebo uvést testovanou instanci do korektního stavu (aby nedošlo ke zbytečnému selhání následujících testovacích případů).

Uživatelsky přívětivým rozšířením by byla flexibilnější konfigurace uživatelského rozhraní. Ta by mohla zahrnovat například změnu velikostí polí zobrazujících strom testovacích případů a zásobníku volání nebo také možnost ukládání nastavení.

Kapitola 5

Závěr

Tato práce obsahuje návrh a implementaci nástrojů poskytujících informace o probíhajících testech uživatelského rozhraní platformy Eclipse. Tyto nástroje jsou navrženy pro funkci s testovacím rámcem JUnit, obsaženým v Eclipse JDT (*Java Development Tools*). V první části je popsána architektura platformy Eclipse, se zaměřením na strukturu a tvorbu zásuvných modulů. Druhá část se zabývá architekturou testovacího rámce JUnit, jeho aplikačním rozhraním, možnostmi rozšíření a jeho integrací ve vývojovém prostředí Eclipse.

Hlavní část práce představuje autorem implementovaný nástroj TRV, skládající se ze dvou částí – InRunJUnit a TRView. Díky těmto nástrojům lze sledovat průběh testů uživatelského rozhraní, bez problémů s aktivitou okna. Uživateli je zobrazena stromová struktura s vyznačenými stavy jednotlivých testovacích případů. Zároveň poskytuje uživateli přehled o počtu běhů testů, testových chyb, běhových chyb a ignorovaných testovacích případů. V případě chybného testovacího případu zobrazuje i výpis posledních volání na zásobníku (*angl. stack trace*).

Nástroj TRV byl předán firmě Red Hat, kde bude diskutováno a navrženo jeho budoucí použití. Primární nasazení by mělo spočívat ve spolupráci se serverem Jenkins. Dále je zde možnost integrace nástroje TRV do testovacího rámce RedDeer, který již poskytuje některé nástroje pro CI (*angl. continuous integration*).

Literatura

[1] BECK, K.: *JUnit Pocket Guide*. O'Reilly media, Inc., 2004, ISBN 978-0-596-00743-0.

[2] BLEWITT, A.: *Eclipse 4 Plug-in Development by Example*. Packt Publishing, 2013, ISBN 978-1-78216-032-8.

[3] CLAYBERG, E.; RUBEL, D.: *Eclipse Plug-ins*. Addison-Wesley Pearson Education, 2009, ISBN 978-0-321-55346-1.

[4] *Eclipse documentation: Platform Plug-in Developer Guide*. 2013, [Online; navštíveno 9. 5. 2017].
URL https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Farch.htm

[5] *Eclipse documentation: Workbench User Guide*. 2013, [Online; navštíveno 10. 4. 2017].
URL http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Fconcepts%2Fconcepts-2.htm

[6] *About the Eclipse Foundation*. [Online; navštíveno 9. 4. 2017].
URL https://eclipse.org/org

[7] HAMILL, P.: *Unit Test Frameworks*. O'Reilly media, Inc., 2004, ISBN 978-0-596-00689-1.

[8] RÜDIGER, H.: *How to Safely Use SWT's Display asyncExec*. Září 2014, [Online; navštíveno 16. 4. 2017].
URL
http://www.codeaffine.com/2014/09/15/safely-use-swt-display-asyncexec

[9] *Top 25 Tools for Automated Testing of JAVA Applications*. Duben 2017, [Online; navštíveno 20. 4. 2017].
URL http://www.softwaretestinghelp.com/java-testing-tools/

[10] VOGEL, L.: *Unit testing with JUnit – Tutorial*. [Online; navštíveno 20. 4. 2017].
URL http://www.vogella.com/tutorials/JUnit/article.html

Přílohy

Příloha A

Obsah přiloženého paměťového média

- **TRV/:** kořenový adresář
  - **com.mcoufal.plugin.test:** Adresář se zdrojovými soubory napodobujícími činnost automatizovaných testů GUI Eclipse IDE.
  - **mcoufal.InRunJUnit:** Adresář obsahující zdrojové soubory implementovaného zásuvného modulu InRunJUnit.
  - **mcoufal.InRunJUnit.feature:** Adresář obsahující feature projekt se zásuvným modulem InRunJUnit.
  - **mcoufal.InRunJUnit.updateSite:** Adresář obsahující update site projekt se zásuvným modulem InRunJUnit.
  - **README.md:** Soubor popisující základní použití a instalaci vytvořeného nástroje TRV.
  - **technical-report:** Adresář obsahující zdrojové soubory použité pro tvorbu této práce.
  - **TRView:** Adresář obsahující zdrojové soubory aplikace TRView a spustitelný JAR archiv `TRView.jar`.
  - **xcoufa08-Pohled-na-stav-JUnit-pro-testovanou-instanci-eclipse.pdf:** Soubor ve formátu PDF obsahující elektronickou verzi této práce.