Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Implementace frameworku Allure do testů v Javě

Bakalářská práce

Autor: Daniel Bechný
Studijní obor: Informační management

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Hradec Králové Březen 2021

Prohlášení:

Prohlašuji, že jsem bakalářskou zpracoval samostatně a s použitím uvedené literatury.

vlastnoruční podpis

V Hradci Králové dne 13.4.2021 Daniel Bechný

Poděkování:

Děkuji vedoucímu bakalářské práce doc. Mgr. Tomáši Kozlovi, Ph.D. za metodické vedení práce a své rodině za podporu.

Anotace

Cílem této práce seznámit čtenáře s obecnou problematikou testování softwaru, jako nedílnou součástí vývoje softwaru.

Práce popisuje fungování automatických testovacích frameworků a nastiňuje jejich možnosti. Následně je v práci představena reprezentace výstupů automatizovaných testů do reportů, jenž jsou dobře čitelné běžným člověkem.

V práci je též ukázána implementace reportovacího frameworku Allure do již existujících testovacích projektů, které jsou vyvíjeny v programovacím jazyce Java a zabývají se testováním nástrojů na odhalování plagiátorství.

Annotation

Goal of this thesis is to introduce software testing domain in general. Author would like to demonstrate phase of testing among other software development phases. In this thesis will be described functions of automated test frameworks and introduced its possibilities. Then results of such automated tests will be represented into human readable form – reports.

In this thesis will be described Allure framework, which is reporting framework and its implementation into existing test projects developed in Java language. These testing projects are testing application for plagiarism checking.

Title: Implementation of Allure framework into Java tests

Obsah

Seznam obrázků

1 Úvod

Zvolené téma reprezentuje obecnou problematiku testování softwarů. V této práci budou představeny druhy testování, bude uskutečněno obecné představení postupů a cyklů vývoje softwarových aplikací a následně vhodné nástroje, které umožňující realizaci automatických testů.

Ačkoliv tomu tak nebylo vždy, dnes hraje testování v procesu vývoje software významnou roli, a kvalitní produkt se bez testerů v žádném případě nedokáže obejít. Testeři se poté nedokážou obejít bez nástroje, jenž je schopný zpracovat výsledky, vyprodukované testovací aplikací do lidsky přehledné a čitelné podoby. Právě to je jeden z důvodů nutnosti použití nástroje pro takovou reprezentaci výsledků – Allure report.

Autor je zaměstnán jako QA inženýr v mezinárodní společnosti, která se zabývá vývojem softwaru, jenž slouží k podpoře vzdělávání a výuky, zejména pak na vysokých školách. Práce tedy popisuje konkrétní důvody a konkrétní postup zavedení reportovacího nástroje Allure.

2 Cíl práce

Smyslem této práce je seznámení čtenáře s vývojovým procesem softwaru, zejména pak úkoly Quality Assurance (dále jen QA) týmů v rámci tohoto procesu. V práci budou představeny metodiky vývoje a jejich aplikace tak, aby došlo ke komplexnímu pochopení toho, jak nezbytnou fází je fáze testování softwaru. Bude popsán proces testování a reportování výsledků testů, evidování chyb a následné verifikace jejich oprav.

Tato práce si klade za cíl taktéž seznámení čtenáře s některými nástroji užívanými testery. Zejména je pak kladen důraz na reportovací framework od společnosti Qameta software – Allure report, důvody jeho potřeby a popsání jeho implementace do kódu testovací aplikace. Tato práce může též sloužit jako podrobný manuál k Allure reportu v jazyce Java.

3 Vývoj softwaru

3.1 Proces vývoje software

Cyklus vývoje softwaru se dělí do několika fází. V následující části práce budou tyto fáze představeny a podrobněji popsány.

Na tento cyklus se běžně aplikují tzv. metodiky vývoje softwaru. Ty nejdůležitější budou následně popsány.

3.2 Požadavky

Požadavky vycházející z konkrétních požadavků zákazníků, či z poptávky na trhu. Příkladem může být situace, kdy klient požaduje přidání nové funkčnosti do již existujícího informačního systému. Může se taktéž jednat o požadavek na vývoj kompletně nové aplikace.

3.3 Návrh

V této fázi je vhodné provést podrobnější analýzu požadavků definovaných v předchozím kroku. Softwarový architekt navrhne softwarovou aplikaci. Tím se rozumí zpracování návrhu obecného modelu funkcionality, návrh a specifikace zásadních funkčních požadavků systémů. Je vybrána vhodná platforma a technologie. Dále je zpracován návrh uživatelského rozhraní. Výsledek této činnosti je předán vývojářům, kteří se zabývají konkrétním kódováním systému.

3.4 Kódování

Jedná se o proces realizování návrhu obdrženého od softwarového architekta. Konkrétní implementace s využitím definovaných technologií. Zpracování zdrojového kódu, jeho dokumentace a verzování. Tato fáze je úzce provázána s testovací fází, jenž je hlavním předmětem této práce.

3.5 Fáze testování softwaru

V této fázi je produkt, nebo jeho část předána týmu testerů. Tato fáze je podrobněji definována a rozebrána níže. Tato část slouží k obecnému představení v rámci životního cyklu informačního systému.

Testovací tým v tuto chvíli využije jemu dostupné prostředky a technologie k otestování funkčnosti softwaru, ať už k testování logické funkcionality, tak k ověření robustnosti a stabilnosti aplikace. Je využíván manuální způsob testování nebo automatický (pomocí testovacích programů) nebo kombinace obojího zmíněného. Testování hraje klíčovou roli v ověřování správnosti implementace a ověřování bezchybnosti řešení. Jedná se o činnost, kdy se na jedné straně ověřuje správnost kódu, na straně druhé se cílí na nalezení, pokud možno co největšího počtu chyb. (O'Regan, 2017)

3.6 Zavádění softwaru

Zaváděním systému je míněna především jeho instalace, zavedení do provozu organizace, transformace původní datové základny tak, aby byla přístupná novému systému, poskytnutí manuálů a školení uživatelům. Při školení je nejlepším postupem nejprve školit vedoucí pracovníky a pokračovat zaměstnanci v provozu.

Tato etapa se nesmí v žádném případě podcenit, neboť jejím zanedbáním by mohla u budoucích uživatelů vzniknout averze vůči novému systému a tím neúspěch celého projektu.

Zavedení systému může být provedeno jedním z následujících způsobů (Šmíd, 2005):

Souběžná strategie – je založena na pokračujícím provozu původního systému + současný provoz nového systému. Provoz obou systémů trvá několik pracovních cyklů, dokud nový systém nepracuje spolehlivě a uživatelé s ním nejsou dostatečně seznámeni. Tato metoda je bezpečná, ale velice náročná pro zaměstnance, neboť musí provádět dvakrát totéž, což by mohlo vést k averzi vůči novému IS. Proto se na toto období najímají externí pracovníci.

Pilotní strategie – je založena na zavedení nového systému jen ve vybrané části podniku a po jeho ověření se systém zavede do celé organizace. Jako pilotní část se vybere taková, která je poměrně náročná a je možné na ní ověřit co nejvíce problémových oblastí.

Postupná strategie – využívá se zejména u velice složitých systémů, kde jsou složité vnitřní vazby. Nejprve se zavádějí primární části IS na kterých ostatní části závisí, po jejich ověření se podobným postupem zavádí ostatní části až po zavedení celého systému.

Nárazová strategie – spočívá v odstranění původního systému a zavedení kompletního nového systému. Tato strategie je velice riskantní, ale ušetří se při ní čas i pracovní síly.

4 Testování softwaru

Testování softwarového produktu slouží pro ladění kódu a zajištění všech požadovaných funkcionalit. Kvalita testování závisí na kvalitě a preciznosti testovacího týmů. Testeři chyby hledají a vytváří jejich dokumentaci, ta je poté použita k odstranění chyby příslušným vývojářem. V samotném testování existuje i riziko nenalezení chyby, protože není možné otestovat software úplně a komplexně, většinou je produkt tak složitý, že nelze otestovat všechny možnosti, existuje velké množství vstupů a výstupů. Při vývoji softwaru většinou existuje i mezní termín pro vytvoření, tento termín může být zadán zákazníkem nebo je-li produkt uváděn na trh, tak je mezní termín dán trhem. Ať už v případě produktu na trh, nebo zakázky od zákazníka je čas pro celý vývoj software předem daný a tím je daný i čas pro testování. Tester musí vytvořit optimální množství testů, aby byl software dostatečně otestovaný, ale aby testy nezabíraly přílišné množství času a neprodlužovaly jeho vývoj. Testy nikdy nedokazují, že chyby v produktu neexistují, někdy není úplně možné všechny chyby odstranit, a proto mají chyby svou prioritu, kterou určuje tester podle jeho vlastní úvahy. (Veselovský, 2014)

Proces testování je velmi úzce svázán s procesem vývoje produktu. Je klíčové navázat a udržovat komunikaci s vývojáři. Vhodně evidovat nalezené nedostatky, provázet je scénáři a důsledně verifikovat opravené chyby. Cyklus procesu testování členíme následovně (Zelinka, 2013):

- **Plánování testů**
    - Definice projektu, co kontrolovat a jak vyhodnocovat
    - Identifikace testovacích požadavků, strategie testování, zdrojů pro testy a příprava testovacího plánu.
- **Analýza a příprava testů**
    - Návrh jednotlivých testů (test cases), které ověří, zda jsou požadavky na systém splněny.
    - Identifikace a příprava potřebných testovacích dat.
- **Provedení a vyhodnocení testů**
    - Provedení testů a zaznamenání jejich výsledků.
    - Analýza výsledků a vyhodnocení, zda nedošlo k chybě.
- **Sledování defektů**

- Průběžná kontrola počtu defektů, jejich závažností a způsobu odstranění.

(Zelinka, 2013)

4.1 Quality Assurance a testování

Obecně se pojmem quality assurance (dále také QA) označuje testování, avšak nesprávně. QA a testování totiž není jedna a ta stejná věc, ačkoliv k sobě mají velmi blízko. Jsou podmnožinou obecnějšího konceptu známého jako quality management (řízení kvality, dále QM). QM zahrnuje veškeré aktivity, které vedou k organizaci činností souvisejících s kvalitou. Mezi jinými jsou zde zahrnuto právě QA a testování. QA se typicky zabývá sloučením procesů, které mají souvislost s kvalitou výsledného produktu tak, aby bylo možné přesvědčit pracovníky různých úrovní managementu, ale i zákazníky, o zachování, nebo nastolení vysoké kvality produktu. Při aplikaci takovýchto procesů je pak obecně kvalita výsledného produktu vyšší, než když aplikovány nejsou. Tento postup taktéž přispívá k předcházení chyb v softwaru (defect prevention).

Řízení kvality tedy zahrnuje nejrůznější procesy včetně testovacích činností, podpory procesů k dosahování stanovených úrovní kvality. Testování je součástí celkového procesu vývoje a údržby. Jelikož se tedy QA zabývá správným provedením celého procesu, je procesní podporou i pro konkrétní a správné testování softwaru. (ISTQB, 2018)

4.2 ISTQB

ISTQB® (International Software Testing Qualifications Board) je nezisková organizace, která byla založena v listopadu roku 2002, a je registrována v Belgii. ISTQB® definovala schéma certifikačních procesů pro QA specialisty. Později se tyto certifikace staly celosvětově uznávaným dokladem při prokazování znalostí. (ISTQB, 2020)

4.3 CaSTB

Czech and Slovak Testing Board (CaSTB) je nezisková organizace založená v prosinci roku 2006. CaSTB je regionálním zástupcem ISTQB® pro Českou a Slovenskou republiku, jako

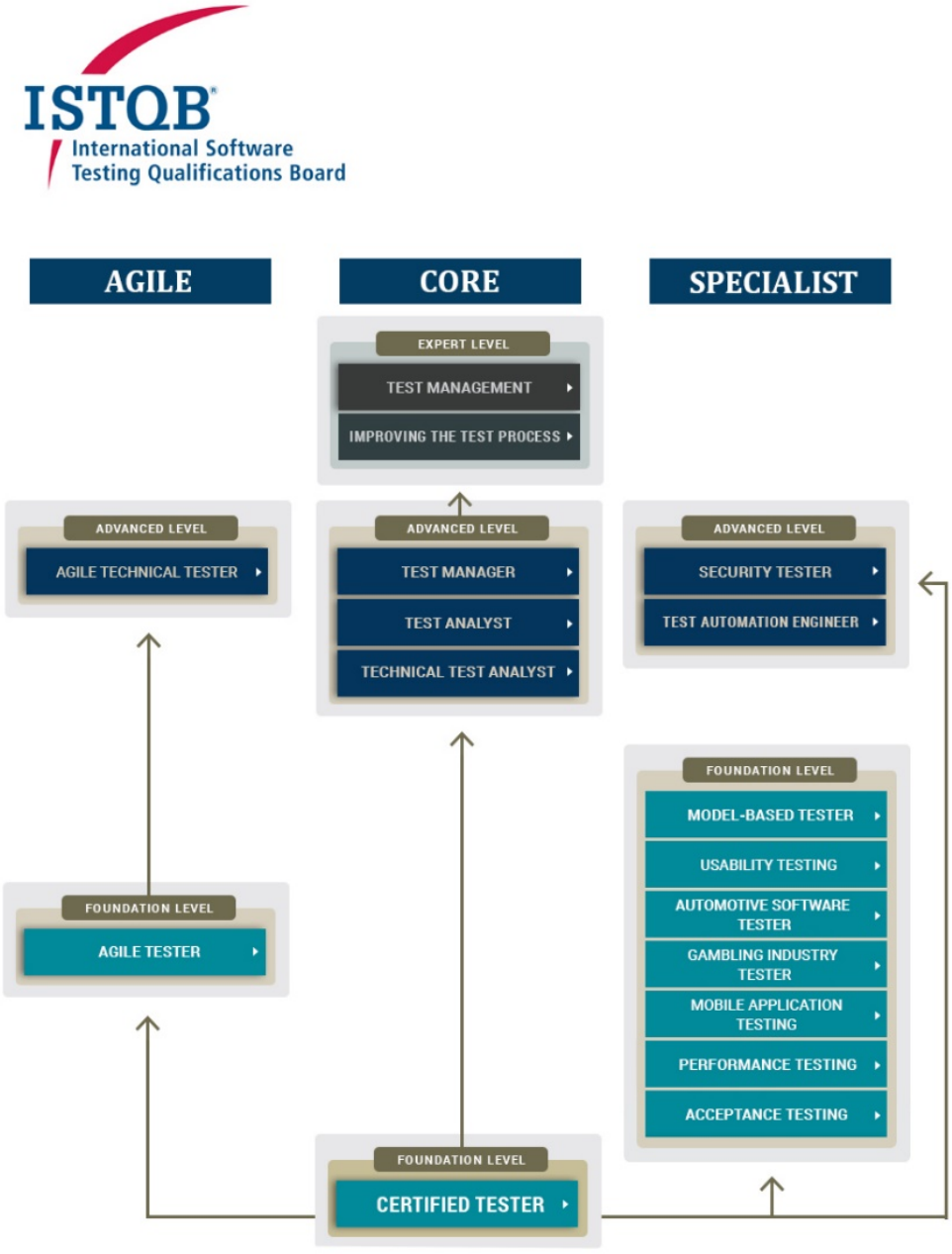
jeden z 45+ členských výborů ISTQB® po celém světě. Většina členů CaSTB je aktivně zapojena do dění v ISTQB® organizaci prostřednictvím pracovních skupin ISTQB®.

Členskou základnu CaSTB tvoří tým odborníků s bohatými zkušenostmi v oblasti testování softwaru, kteří dobrovolně investují svůj čas do rozvoje, údržby a propagace programu ISTQB® Certifikovaný tester v České a Slovenské republice.

CaSTB podporuje ISTQB® ve formě tvorby programu pro kvalifikaci testerů softwaru, zejména překladem Slovníku ISTQB® a Učebních osnov ISTQB® pro Základní stupeň do českého a slovenského jazyka. Další aktivitou je podpora lokálních poskytovatelů školení v podobě přípravy a poskytování hodnocení školících technik a materiálů v programu školení ISTQB®. Vytváříme také partnerství se společnostmi, které demonstrují svůj závazek ke znalosti oblasti testování softwaru svých zaměstnanců. (CaSTB 2013)

4.4 Úrovně certifikací dle ISTQB

ISTQB stanovuje 3 větve dostupných certifikací, všechny se odvíjejí od základní certifikace certifikovaného testera. Agilní, jádrová a větev specialistů je patrná v níže přiloženém obrázku pocházející z výukových skript společnosti ISTQB.



Obrázek 1: ISTQB certifikační úrovně (ISTQB, 2018)

4.5 Úrovně testů podle ISTQB

Úrovně testů jsou skupiny jednotlivých aktivit, které jsou hromadně organizovány a řízeny. Každá úroveň testu je instancí testovacího procesu skládajícího se z testovacích aktivit prováděných v kontextu daného vývojového stádia softwaru. Počínaje individuálními jednotkami, kompletními systémy konče. Jsou svázány se různými úrovněmi vývoje daného softwaru. (ISTQB, Foundation Level Syllabus 2018)

V praxi nejsou vždy prováděny všechny tyto úrovně. Implementace jednotlivých úrovní (druhů) testování se může lišit díky konkrétním potřebám projektu, často je časová, finanční nebo jiná tíseň, což se projeví i na kvalitách testování. Níže je uveden přehled základních úrovní testů tak. jak je popisuje ISTQB v sylabu k certifikaci úrovně „foundation“.

4.5.1 Testování komponent

Testy komponent (taktéž se velmi hojně užívají výrazy modulové – module, nebo jednotkové – unit testy) jsou zaměřeny na testování částí aplikace, které se dají samostatně testovat.
Předmětem unit testů je:

- Snižování rizik
- Verifikace funkčních i nefunkčních požadavků.
- Hledání nedostatků a chyb v komponentech.
- Zamezení chybám, aby pronikly do vyšších vrstev systému.

Zejména v inkrementálních a iterativních modelech vývoje softwaru (např. agile), kdy dochází k častým změnám kódu hrají unit testy významnou roli při verifikaci toho, že nové změny neovlivní již existující komponenty softwaru. Unit testy se většinou provádí v izolovaném prostředí od zbytku aplikace. Mohou pokrývat funkcionalitu (např. správnost výpočtů), nefunkční charakteristiky (např. únik paměti) a strukturální vlastnosti (testování rozhodnutí).

4.5.2 Integrační testy

Integrační testy se soustřeďují na interakci jednotlivými komponenty systému.
Předmětem integračním testům je:

- Snižování rizik
- Verifikace funkčních i nefunkčních požadavků
- Zvyšování důvěryhodnosti aplikačního rozhraní
- Hledání chyb, zejména v rozhraních jednotlivých komponentů systému
- Zamezení chybám, aby pronikly do vyšších vrstev systému

Podobně jako unit testy odhalují chyby, které by mohly způsobit nové změny softwaru, zejména pak v rozhraních. Mezi typicky odhalené nedostatky integračními testy patří nesprávná nebo chybějící data či jejich chybné kódování, špatná posloupnost, nebo načasování jednotlivých volání na rozhraní nebo chyby v komunikaci jednotlivých komponent.

4.5.3 Systémové testy

Jedná se o komplexní testování systému nebo produktu jako celku. Často jsou součástí tzv. end – to – end testy.
Předmětem systémových testů je:

- Snižování rizik
- Ověření funkcionálního i nefunkcionálního chování systému, zdali odpovídá návrhu
- Nalezení chyb
- Prevence pronikání defektů to vyšších úrovních testů nebo do produkce

Pro určité systémy může být také cílem testování ověření kvality dat. Podobně jako v případech testování komponent a integračního testování, automatizované regresní systémové testy podporují důvěru, že změny nenarušily integritu stávajících funkcionalit nebo schopností end-to-end činnosti systému. Systémové testování často poskytuje informace využívané zainteresovanými stranami při rozhodování o uvolnění systému do provozu. Systémové testování (resp. jeho detailní podoba) může být také vynuceno regulatorními požadavky a požadavky příslušných norem. (ISTQB, 2018)

4.5.4 Akceptační testy

Z anglického „acceptance tests“. Podobně jako systémové testy se zaměřují na celkové testování produktu (systému). Nejdůležitější úlohou akceptačních testů je vytvořit důvěru budoucích administrátorů a uživatelů, že budou schopni zajistit správný chod systému v provozním prostředí, a to za všech podmínek.
Předmětem akceptačních testů je:

- Testování z pohledu uživatele
- Provozní testování
- Smluvní a regulatorní testování
- Alfa a beta testy.

Akceptační testy poskytují informace o míře připravenosti systému k nasazení do produkčního prostředí. Případné nalezení velkého množství defektů za použití akceptačních testů může ohrozit projekt jako takový. V některých případech může být dojít ke zpoždění, případně zrušení či přepracování projektu.
Akceptační testy je možné dále dělit dle jejich forem.

4.5.4.1 Uživatelské akceptační testování

Uživatelské akceptační testování systému je běžně zaměřeno na ověřování vhodnosti systému k použití zamýšlenými uživateli v reálném nebo simulovaném provozním prostředí. Hlavním cílem je vybudovat důvěru uživatelů v to, že systém bude splňovat jejich potřeby a požadavky a umožní provádět podnikové procesy hladce s minimálními náklady a riziky. (ISTQB, 2018)

4.5.4.2 Provozní akceptační testování

Obvykle se jedná o testování provozním personálem nebo administrátory systému. Provádí se v testovacím produkčním prostředí. To je možno interpretovat jako testovací prostředí u zákazníka (odběratele daného produktu). Cílem je otestovat provozní náležitosti systému.

Dle ISTQB tyto testy mohou zahrnovat:

- testování zálohování a obnovy ze záloh,
- testování instalace, odinstalace a upgrade,
- testování obnovy po havárii,
- testování správy uživatelů,
- testování údržby systému,
- testování úloh načítání a migrace dat,
- kontroly zranitelnosti zabezpečení systému,
- testování výkonnosti.

4.5.4.3 Smluvní a regulatorní akceptační testování

Smluvní akceptační testování je vykonáváno vůči smluvním akceptačním kritériím pro provoz softwaru vyvinutého na zakázku. Akceptační kritéria by měla být definována v době, kdy zúčastněné strany odsouhlasí kontrakt. Smluvní akceptační testování nejčastěji provádějí uživatelé nebo nezávislí testeři. Regulatorní testování je vykonáváno vůči jakýmkoliv předpisům, které musí být dodrženy, jako například vládní, právní nebo bezpečnostní předpisy. Regulatorní akceptační testování nejčastěji provádějí uživatelé nebo nezávislí testeři, někdy za přítomnosti zástupců regulatorních orgánů. Hlavním cílem smluvního a regulatorního akceptačního testování je vybudovat důvěru v dodržení smluvních a regulatorních požadavků. (ISTQB, 2018)

4.5.4.4 Alfa a Beta testování

Alfa a beta testování obvykle využívají vývojáři komerčního krabicového softwaru (COTS, commercial off-the-shelf), kteří chtějí získat zpětnou vazbu od potenciálních nebo stávajících uživatelů, zákazníků a provozovatelů před uvedením softwarového produktu na trh. Alfa testování se provádí na pracovišti vývojové organizace, nikoliv však vývojovým týmem, ale potenciálními nebo stávajícími zákazníky a provozovateli systému nebo nezávislým testovacím týmem. Beta testování provádějí potenciální nebo stávající zákazníci nebo provozovatelé systému na svých vlastních pracovištích. Beta testování může následovat po alfa testování nebo mu alfa testování ani nemusí předcházet. Společným cílem alfa i beta testování je vybudování důvěry mezi potenciálními i stávajícími zákazníky a provozovateli, že mohou systém používat za normálních

každodenních podmínek, a ve svém provozním prostředí k dosahování svých cílů hladce s minimálními náklady a riziky. Dalším cílem může být nalezení defektů v souvislosti s podmínkami a prostředím, kde bude systém používán (zejména pokud tyto podmínky a prostředí mohl vývojový tým jen nesnadno replikovat). (ISTQB, 2018)

4.6 Funkcionální a Non-funkcionální testování

Výše jsou užity pojmy jako funkcionální a non-funkcionální testování. Pojem funkcionální testování představuje testování těch částí aplikace, případně takovým způsobem, aby došlo k otestování očekávaného chování systému. Tedy toho, které zpravidla bývá popsáno ve specifikaci byznysových požadavků nebo scénářích uživatelských úloh. Může vyžadovat účast experta na dané odvětví byznysu.

Oproti tomu non-funkcionální testování má za úkol řešit otázku jakým způsobem se systém chová (jak dobře). Jedná se o testy použitelnosti systému, výkon či bezpečnost.

4.7 Míra pokrytí testů

Typicky spojována s non-funkcionálním testováním. Obecně platí, že se jedná o vyjádření, jak moc je daná komponenta nebo funkcionalita aplikace pokryta testy. Vyjadřuje se jako procento na základě popsané funkcionality daného prvku.

4.8 Konfirmační testování

Po opravě defektu může být software testován všemi testy, které v důsledku defektu selhaly, a je třeba je znovu provést na nové verzi softwaru. Software může být také testován zcela novými testy, které pokryjí změny potřebné k opravě defektu. Minimálně však musí být na nové verzi softwaru znovu provedeny kroky, které vedly k selháním způsobeným defektem. Účelem konfirmačního testu je potvrdit, že byl původní defekt úspěšně odstraněn. (ISTQB, 2018)

4.9 Regresní testování

Stává se, že změna provedená v části kódu, ať už v rámci opravy nebo jiného druhu změny, může náhodně ovlivnit chování jiných částí kódu. Může jít o dopad ve stejné komponentě, v jiných komponentách téhož systému nebo dokonce i v jiném systému nebo systémech. Změny mohou zahrnovat změny prostředí, jako je nová verze operačního systému nebo databázového systému. Takovým nežádoucím vedlejším účinkům se říká regrese. Regresní testy zahrnují testy určené ke zjištění těchto nechtěných vedlejších účinků. (ISTQB, 2018).

Regresní testování ověřuje, že se po změnách v kódu, či konfiguraci prostředí neprojevily chyby ve funkcionalitách či jiných ověřovaných kvalitách (např. výkonnost), které dříve fungovaly podle požadavků. Strategiím regresního testování je v řadě softwarových projektů věnována poměrně malá pozornost. Některé průzkumy ukazují, že náklady na regresní testování tvoří až 80 % celkových nákladů na testovací aktivity.

Nejvíce prostředků padá na opakované vykonávání regresních testovacích scénářů, na jejich průběžnou údržbu a na údržbu jimi využívaných testovacích dat. (Miroslav Bureš, 2016)

4.10 Principy testování

Podle organizace ISTQB byla formulována velká řada principů, které poskytují obecně definovaný postup, či scénář, dle kterého by měly být testovací sady navrženy. Není možné zohlednit všechny tyto principy. Ve skriptech ISTQB, respektive CASTBss jsou publikovány následující.

1) Testování ukazuje přítomnost defektů, nikoli jejich nepřítomnost. Testování snižuje pravděpodobnost, že v softwaru zůstaly neodhalené defekty, nicméně pokud nejsou žádné defekty nalezeny, není tím prokázána jeho správnost.
2) Kompletní testování není možné Testování všeho (všech kombinací vstupů a vstupních podmínek) není možné s výjimkou triviálních případů. Místo snahy o kompletní testování je lepší se zaměřit na analýzu rizik, vhodné techniky testování a prioritizaci.
3) Včasné testování šetří čas a peníze Pro včasné zjištění defektů by měly být statické i dynamické testovací aktivity zahájeny co nejdříve v životním cyklu

vývoje softwaru. Včasné testování je někdy označováno jako shift left (posun doleva). Testování v rané fázi vývoje softwaru pomáhá snížit nebo eliminovat budoucí náklady na opravu chyb v již zavedeném softwaru.

4) Shlukování defektů Většinu defektů zjištěných během testování před vydáním obsahuje obvykle malé množství modulů nebo jsou defekty v těchto modulech zodpovědné za většinu provozních poruch. Předpokládané a skutečně pozorované shluky defektů během testování nebo v provozu jsou významným přínosem pro analýzu rizik, která se využívá k vhodnému zaměření testovacích aktivit (jak zmiňuje Princip 2).
5) Vyvarování se pesticidnímu paradoxu Pokud se stejné testy opakují neustále dokola, neodhalí nakonec žádné nové defekty. Pro odhalení nových defektů může být nezbytné provést změny ve stávajících testech a testovacích datech, příp. je třeba vytvořit testy nové (testy již nejsou efektivní pro odhalování defektů, stejně jako pesticidy již po čase neúčinkují při ničení hmyzu). V některých případech (jako je například automatizované regresní testování) má však pesticidní paradox pozitivní přínos, kterým je relativně malý počet regresních defektů.
6) Testování je závislé na kontextu Testování se provádí odlišně v různých kontextech. Například řídicí průmyslový bezpečnostně kritický software se testuje jinak než mobilní aplikace pro e-commerce. Dalším příkladem je testování v agilním projektu, kde testování probíhá odlišně od testování na projektu s využitím sekvenčního životního cyklu vývoje softwaru (viz kapitola 2.1).
7) Nepřítomnost chyb je klam Některé organizace očekávají, že testeři dokážou provést všechny možné testy a najít všechny možné defekty, ale Principy 1 a 2 nám říkají, že to možné není. Dalším omylem (tj. mylnou představou) je očekávání, že pouhým nalezením a odstraněním velkého počtu defektů lze zajistit úspěch systému. Například i přes důkladné testování všech specifikovaných požadavků a odstranění všech zjištěných defektů by mohl přesto vzniknout obtížně použitelný systém, který by nesplňoval potřeby a očekávání uživatelů, příp. by nepřinesl takovou (vyšší) hodnotu v porovnání s jinými konkurenčními systémy. (ISTQB, 2018)

4.11 Management testů

Efektivní správa testů je důležitou součástí vývoje vysoce kvalitních softwarových, ale i hardwarových produktů. Prostřednictvím dobře naplánovaných a dobře řízených testovacích procesů se mohou týmy zaručit, že dodávají nejkvalitnější možné produkty, a přitom efektivně a maximálně využívají zdroje, jež jsou jim k dispozici.

Potřeba správy stále složitějších softwarových produktů vyústila v potřebu chytré správy testů napříč každým projektem.

Soutěž mezi dodavateli softwaru o vývoj co nejlepších produktů v co nejkratším čase vede ke vzrůstající potřebě vysoce rozvinutých procesů správy testů. Vzhledem k tomu, že testovací týmy spolupracují s vývojovými týmy na dodávání hotových výrobků v kratších termínech, pozornost se oprávněně upírá taktéž na správu testů.
(Test Management Systems Ltd, 2021)

4.12 Fáze testů

Proces testování se dělí do jednotlivých fází, které jsou podrobněji rozepsány v následujícím textu. (Test Management Systems Ltd, 2021)

4.12.1 Fáze plánování

Tato fáze zahrnuje vývoj celkového směru testovací fáze, včetně specifik proč, kdy a kde testovat. Testy se vytvářejí tehdy, pokud existuje konkrétní motivace k vytvoření daného testu. Měl by existovat konkrétní požadavek, který je ověřen.

4.12.2 Fáze vytvoření testu

Ve fázi vytváření jsou zachyceny kroky, které jsou nutné k dokončení daného testu, aby bylo možné odpovědět na otázku, jak bude test proveden. Stručně řečeno, během tohoto procesu dochází k definování obecných testovacích případů, které jsou poté rozděleny do podrobných kroků testu. Tyto kroky lze poté vyvinout jako ruční nebo automatizované testovací skripty.

4.12.3 Fáze provedení testu

Během fáze provádění jsou testovací případy spouštěny v logických sadách, které se obvykle označují jako testovací sada. Testy se provádějí proti známé konfiguraci testovaného softwaru, hardwaru nebo testovacímu prostředí.
Je důležité zaznamenat konfiguraci pro účely opětovného vytvoření stejných podmínek v pozdější fázi.

4.12.4 Fáze vyhodnocení výsledků

Monitoring testu nebo vyhodnocení testu je vyvození závěru z výsledků testů.

4.12.5 Hlášení výsledků

Hlášení neboli reporting. V této fázi jsou výsledky sděleny dalším, zúčastněným stranám. Cílem je zjistit aktuální stav testování projektu, ale taktéž poskytnout podrobnosti o celkové kvalitě aplikace nebo systému.

5 Technologie

Při automatizovaném testování softwaru se využívají nejrůznější nástroje. V následujícím textu budu stručně představeny ty technologie, které budou využity v následující případové studii o nasazení Allure.

5.1 Java

Java je objektově orientovaný programovací jazyk vyvinut Jamesem Goslingem a jeho kolegy ve společnosti Sun Microsystems v začátcích 90 let. Na rozdíl od konvenčních jazyků, které byly určeny k překladu do nativního (zdrojového) kódu nebo byly interpretovány ze zdrojového kódu za běhu, je Java určena ke kompilaci do bajtkódu, který je následně spuštěn pomocí JVM[1].

Jazyk jako takový se syntakticky inspiruje v jazycích C a C++, ale disponuje jednodušším objektovým modelem a méně nízko-úrovňovými možnostmi. Java je pouze vzdáleně příbuzná JavaScriptu, jelikož syntaxe obou jazyků vychází z jazyku C.

Java vznikla jako projekt s názvem Oak, který vytvořil James Gosling v červenci roku 1991. Goslingovým cílem bylo implementovat virtuální stroj a jazyk, který by měl povědomou syntaxi z jazyka C, ale zároveň by byl univerzálnější a jednodušší než C nebo C++. První veřejná implementace byla Java 1.0 v roce 1995.

Zavázala se příslibem „Napiš jednou, spusť kdekoliv“ s runtimy zdarma na populárních platformách. Bylo to bezpečné a zabezpečení bylo konfigurovatelné, což umožnilo omezit přístup k síti a souborům. Hlavní webové prohlížeče Javu brzy začlenily do svých standardních konfigurací v zabezpečené konfiguraci „appletu“. Nové verze pro velké a malé platformy (J2EE a J2ME) byly brzy navrženy s příchodem „Java 2“. Sun neoznámil žádné plány pro „Java 3“.

V roce 1997 společnost Sun oslovila normalizační orgán ISO / IEC JTC1 a později Ecma International, aby Javu formalizovala, ale brzy z procesu ustoupila.

Java zůstává de facto proprietárním standardem, který je řízen procesem Java Community Process. Sun poskytuje většinu svých implementací Javy zdarma, přičemž tržby generují

[1] Java Virtual Machine

specializované produkty, jako je Java Enterprise System. Sun rozlišuje mezi svou Software Development Kit (SDK) a Runtime Environment (JRE), což je podmnožina SDK, přičemž hlavní rozdíl spočívá v tom, že v JRE kompilátor není přítomen. (FREEJAVAGUIDE, 2013) V roce 2009 koupila firmu Sun společnost Oracle a tím i Javu.

5.2 Jenkins

Pro spouštění testů je často využíván software s názvem Jenkins. Jenkins je soběstačný, open source server pro automatizaci. Může být využit pro automatizaci rozličných úkolů jako sestavení, testování, vydávání nebo nasazování softwaru. Jenkins může být instalován pomocí nativních systémových balíčků, v Docker kontejneru nebo jako samostatný software na jakémkoliv zařízení, kde je zároveň nainstalována Java (JRE). (Jenkins, 2020)

5.3 Framework

Během programování aplikace není potřeba začínat od nuly. Existuje široká škála nástrojů, která vývoj aplikace značně ulehčí a standardizují. Jedná se o software, který je vyvinut a používán vývojáři k vývoji aplikací. Vzhledem k tomu, že framework obvykle vytváří vícero zkušených softwarových inženýrů a programátorů jsou softwarové frameworky obvykle spolehlivé, robustní a účinné. Použití softwarového frameworku umožní programátorovi soustředit se na vyšší vrstvy právě vyvíjené aplikace, o nižší úrovně funkcionality se stará právě framework.

K dispozici je mnoho typů softwarových frameworků, které usnadňují vývoj aplikací napříč různými aplikačními doménami. (Singh, 2020)

5.4 Report framework

Jelikož většina automatických testů produkuje velké množství dat, v lidsky špatně čitelné podobě, je téměř nutností využít nástroje k zobrazení výsledků v podobě člověku dobře čitelné. O tuto část se stará komponenta, která se nazývá reportovací framework. Přečte výstup z testovacích programů a zhotoví zprávu o průběhu, chybách či problémech. Je zvykem automatické znázorňování spolehlivosti testů na grafech. Tyto nástroje značně usnadňují interpretaci výsledků majitelům produktu, manažerům či jiným oddělením,

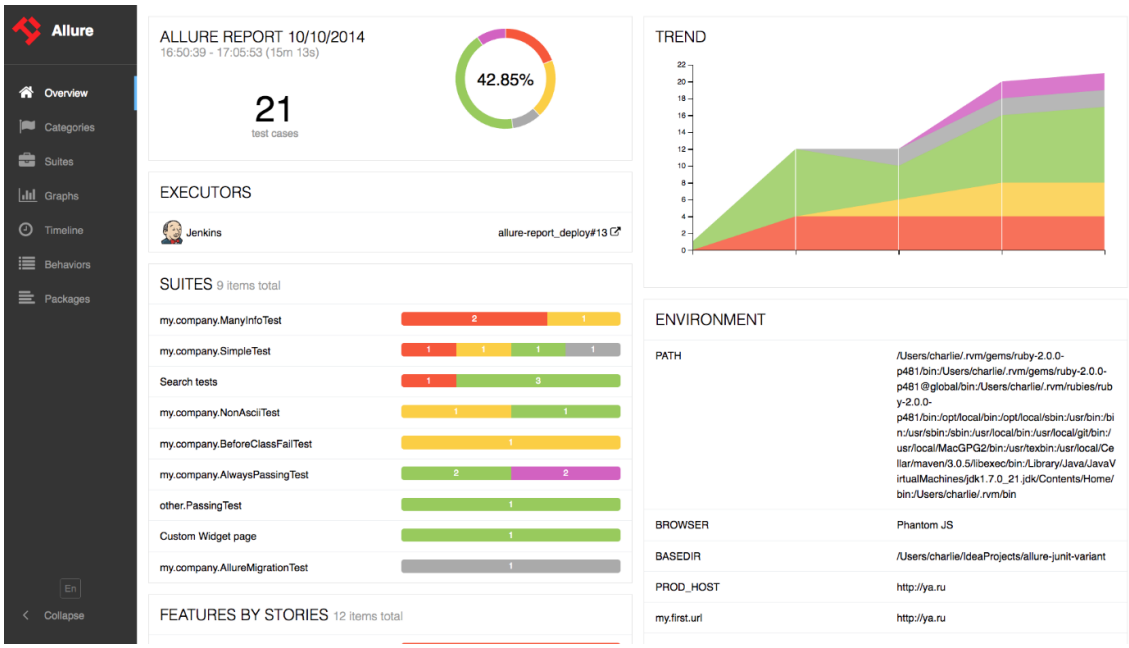
které mají potřebu znát metriky chování dané aplikace. Níže v práci bude prakticky ukázána implementace reportovacího frameworku do testovacího projektu v jazyce Java.

5.5 Allure Framework

Allure Framework je flexibilní, na výkon nenáročný, open source, multilinguální nástroj nejen k vytváření testovacích reportů. Allure poskytuje nejen stručnou reprezentaci toho co bylo testováno v úhledném webovém prostředí, ale umožňuje spolupracovat všem zúčastněným ve vývojovém procesu a tím pádem obdržet maximum vhodných informací z každodenního běhu testů.

QA inženýrům poskytuje možnosti rozdělit nalezené defekty na bugy (chyby aplikace) a chybné testy (chyby v testovací aplikaci). Poskytuje možnosti logování, zachycení jednotlivých kroků testu, přidávání příloh do reportů, časování, historie a integraci se systémy pro správu testů.

Pro manažery projektu a jeho vlastníky nabízí jasný a ucelený pohled na to, které funkce jsou pokryty, a kde se nahromadily chyby. Jak vypadá časová osa exekuce a spoustu dalších užitečných funkcí.

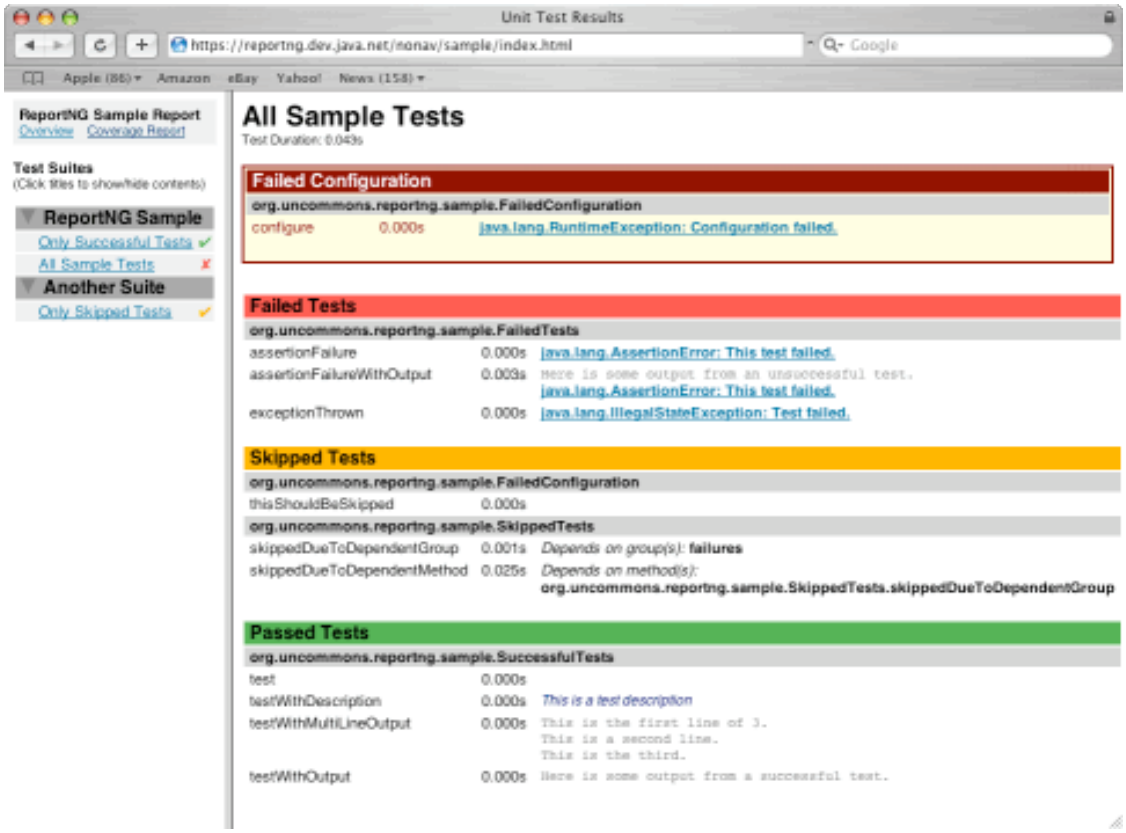


Obrázek 2: AllureReport ukázka (Qameta Software, 2019)

5.6 ReportNg

ReportNg je jednoduchý HTML reportovací plugin určený pro TestNG testovací framework. Je zamýšlen jako náhrada pro výchozí TestNG HTML reporty. Výchozí report je obsáhlý, ale ne dobře čitelný. ReportNG proto poskytuje jednoduchý a barevný souhrn výsledků testů. Výstup reportNg je možno kustomizovat pomocí vlastní CSS šablony. Od verze 0.9.0 obsahuje taktéž druhý reportovací nástroj, který produkuje výstup ve formátu XML. To umožňuje integraci s nástroji jako Hudson. (Dyer, 2013)

ReportNg přestalo být podporováno v roce 2013, přesto je v praxi stále hojně rozšířené.



Obrázek 3: ReportNg ukázka (TestNg)

6 Implementace Allure do projektu

V následující kapitole budou představeny cílové projekty, do kterých byl Allure implementován a popsáno obecné použití Allure.

6.1 Cílové projekty

V první řadě je nezbytné představit projekt, do nějž byl zvolený reportovací framework implementován. Jedná se o 2 různé projekty v jazyce Java, které dříve používaly výše zmíněné ReportNg. Projekty jsou sestavovány pomocí nástroje Apache Maven[2]. Jde o projekty s názvy Automation a Tests. Oba projekty, jak je patrné níže, využívají stejných či obdobných technologií, ačkoliv jsou na sobě nezávislé. Jako IDE[3] je použita IntelliJ IDEA od společnosti JetBrains. Spouštění testovacích suit zajišťuje Jenkins, ve většině případů je možné i lokální spuštění testovacích scénářů, tudíž je potřeba mít možnost sestavit report jak pro testovací běhy na Jenkinsu, tak při vygenerování výsledků na lokální disk.

6.1.1 Projekt Automation

Projekt, který je nazýván Automation zahrnuje nejrůznější sady UI[4] testů pokrývající aplikaci pro kontrolu plagiátorství. Tato aplikace je součástí komplexního informačního systému určeného pro univerzity, či instituce. UI testy pokrývají značnou část aplikace, jak úkony správce systému, tak instruktora či studenta. Zároveň nepřímo testují informační systém, do nějž je tato aplikace integrována. Zároveň testují pomocí logicky sestavených kroků také správnou funkčnost backendu, tedy serverů a logické vrstvuy zmíněného nástroje. Taktéž je nezbytné zajistit vytvoření a publikování snímku obrazovky v případě, že test odhalí chybu.

Roli testovacího frameworku plní TestNg, které, využívá nástroje na bázi Selenia[5]. Není zde využito čisté Selenium, nýbrž vlastní framework postavený na jeho základech

[2] Apache Maven je nástroj na softwarovou správu projektu a snazší pochopení projektu. Zakládá na principu „project object model“ (POM). Umožňuje správu sestavení, reportování a dokumentace projektu na základě informací centralizovaných v POM souboru. (Project, 2021)
[3] IDE – Integrated Development Environment, tedy integrované vývojové prostředí.
[4] UI – User Interface což v překladu znamená uživatelské rozhraní.
[5] https://www.selenium.dev

optimalizovaný pro specifické potřeby. Interakci se strojem, kde běží aplikace zajišťuje technologie Selenium Grid.

Projekt se skládá z několika modulů, které spolu logicky souvisí a tvoří jistou hierarchii. Jedná se o tyto části:

- base – Jak vyplývá již z názvu, jedná se o základní stavební kámen, který využívají následně další části. V tomto adresáři jsou specifikovány a definovány třídy a metody potřebné pro práci s webový informační systém jehož součástí je zmíněný software pro kontrolu plagiátorství.
- reportNg – V této je část, která řeší specifické potřeby pro vytváření a ukládání reportů.
- web-automation – Adresář s výše zmíněným nástrojem postaveným na Seleniu. Má za úkol interakci přímo s webovou aplikací a prohlížečem.
- webui-acceptance-test – Zde jsou testy samotné a jejich konfigurace. Dá se říct, že se jedná o nejvyšší úroveň. Tyto třídy, využívají nástroje a metody právě z výše zmíněných adresářů.

Každý z těchto zmíněných modulů má vlastní pom.xml soubor. Ty jsou pak závislé na centrálním pom nesoucí název „parent“. Tyto soubory definují závislosti na knihovnách pro Maven.

6.1.2 Projekt Tests

Tento projekt obsahuje API[6] testy všeho druhu. Jelikož zde neexistuje GUI[7] odpadá nutnost využití Selenia a Gridu. Testy jsou napsány pomocí vlastního řešení v jazyce Java, opět s využitím TestNg.

Jelikož testy nemají GUI, není zde nutno řešit screenshoty obrazovky. Jako přílohy se ovšem v některých testech přidávají soubory ve formátu HTML, či JSON.

[6] Application Programming interface – Aplikační rozhraní
[7] Graphic User Interface – Grafické uživatelské rozhraní

6.2 Použití Allure obecně

6.2.1 Instalace Allure commandline

Při běhu testů pomocí Apache Mavenu jsou pomocí doplňku Maven Surefire generovány textové a XML soubory se záznamy, které se vygenerují v rámci běhu definované suity, nebo určeného testu. Z těchto dat je již možné vygenerovat hotový report.

K tomu slouží nástroj zvaný Allure commandline. Abychom vyhověli požadavku sestavovat reporty z tzv. *lokálních běhů* (tedy běhů testů, které byly spuštěny z lokálního zařízení, nejčastěji pomocí funkce „run“ v IDE IDEA), je nutné nainstalovat Allure na lokální zařízení.

Allure je dostupný na všechny zásadní platformy. Rozumíme tím Windows, Linux a Mac OS X. Instalace se drobně liší, ale na každé z výše zmíněných platforem je možné využít služeb některého z programů pro správu instalací.

6.2.2 Instalace na Windows

Instalace na operační systém (dále OS) pomocí manažera doplňků Scoop[8]. Po otevření Windows PoweShell je nutné zadat příkaz:

```
scoop install allure
```

Tímto dojde k instalaci Allure na počítač. V případě, že je nutné Allure aktualizovat, učiníme tak pomocí příkazu:

```
\bin\checkver.ps1 allure -ux
```

v instalační složce Scoopu. Tím dojde k zjištění, zda-li je dostupná novější verze. Pokud ano, je třeba následovat příkazem:

```
scoop update allure
```

který zajistí instalaci nové verze. (Qameta Software, 2019)

6.2.3 Instalace na Linux Debian

Na systémy Linux typu Debian (i Ubuntu) spouštíme následující posloupnost příkazů v terminálu:

[8] https://scoop.sh

```
sudo apt-add-repository ppa:qameta/allure
sudo apt-get update
sudo apt-get install allure
```

(Qameta Software, 2019)

6.2.4 Instalace na Mac OS X

Instalace na Mac OS X je prováděna skrze manažera Homebrew[9] tímto příkazem:

```
1. brew install allure
```

(Qameta Software, 2019)

Ověření správné instalace se pak provádí příkazem:

```
allure --version
```

na který jako odpověď následuje číslo aktuálně nainstalované verze Allure.

6.2.5 Vygenerování reportu

Poté co je Allure správně nainstalován, a na zařízení se nacházejí vygenerované výsledky testů, je možné sestavit a publikovat report. Do konzole je nutné zadat příkaz:

```
allure serve /cesta/k/vysledkum
```

Jak je patrné, klíčové slovo *serve* zajistí vygenerování reportu a jako parametr potřebuje zadat cestu ke složce, kde se výsledky nacházejí. Ta může být libovolná. Tento příkaz vygeneruje report do dočasné složky z dat, které jsme předložili v parametru příkazu. Poté dojde k vytvoření instance Jetty serveru, na které bude publikován Allure report a zároveň dojde k otevření výchozího prohlížeče s adresou této instance.

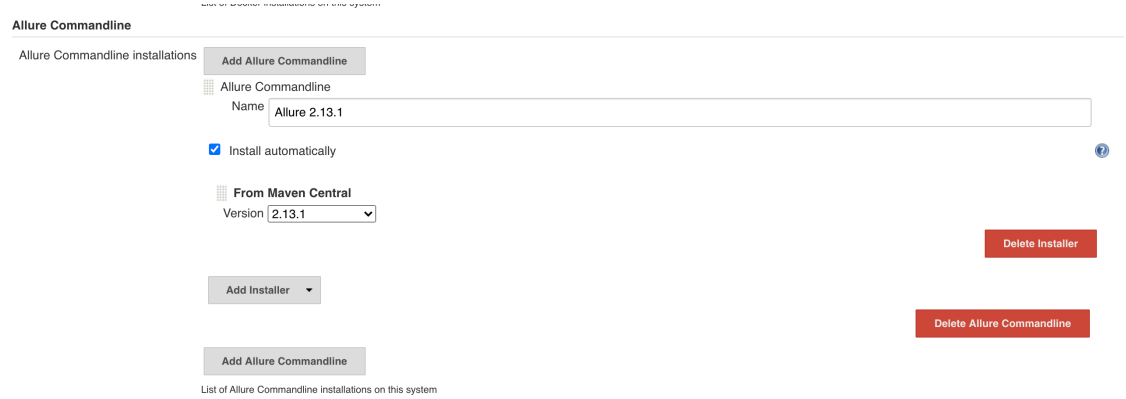
6.3 Konkrétní implementace do zmíněných projektů

6.3.1 Použití na Jenkinsu

V tuto chvíli je Allure instalován na lokálním zařízení, jak ale bylo zmíněno výše, většina exekucí testů probíhá na serveru pomocí nástroje Jenkins.

[9] https://brew.sh

Pro fungování Allure reportu na této platformě je nejdříve nutné do Jenkinsu nainstalovat plugin Allure[10]. Toto lze provést skrze manažera doplňků, který je k nalezení v nastavení Jenkisu. Poté je vhodné otevřít globální nastavení nástrojů a zde vyhledat „Allure commandline“. Následně přidat Allure Commandline a vybrat verzi. Plugin je propojen s Apache Maven repozitáři, tudíž není nutné danou verzi nikterak stahovat a nahrávat manuálně. Následně se tyto kroky potvrdí tlačítkem „Apply“.



Obrázek 4: Allure commandline v Jenkins (Vlastní zpracování)

V tento moment je nakonfigurována potřebná infrastruktura. Ve zmíněných projektech jsou použity tzv. skriptované „pipeliny“ napsány v jazyce Groovy. Do takto definovaných pipeline je nyní nutno přidat krok, který povede k vygenerování a publikování reportů.

V souboru obsahujícím Jenkins pipeline definujeme metodu, která se stará o vygenerování a publikování report následujícím způsobem:

```
1. def reportAllure(Configuration configuration) {
2.   String testName = configuration.getTestName()
3.   allure jdk: '', report:
4.    'automation/webui-acceptance-tests/reports/allureReports-'+
   configuration.getTestName(),
5.           results: [[path:'automation/webui-acceptance-tests/target/'+
   configuration.getTestName()+ '/allure-results']]
6.    publishHTML (target: [
7.           allowMissing         : false,
8.           alwaysLinkToLastBuild: true,
9.           keepAll              : true,
10.          reportDir            : 'automation/webui-acceptance-
   tests/reports/allureReports-'+ configuration.getTestName(),
```

[10] https://plugins.jenkins.io/allure-jenkins-plugin/

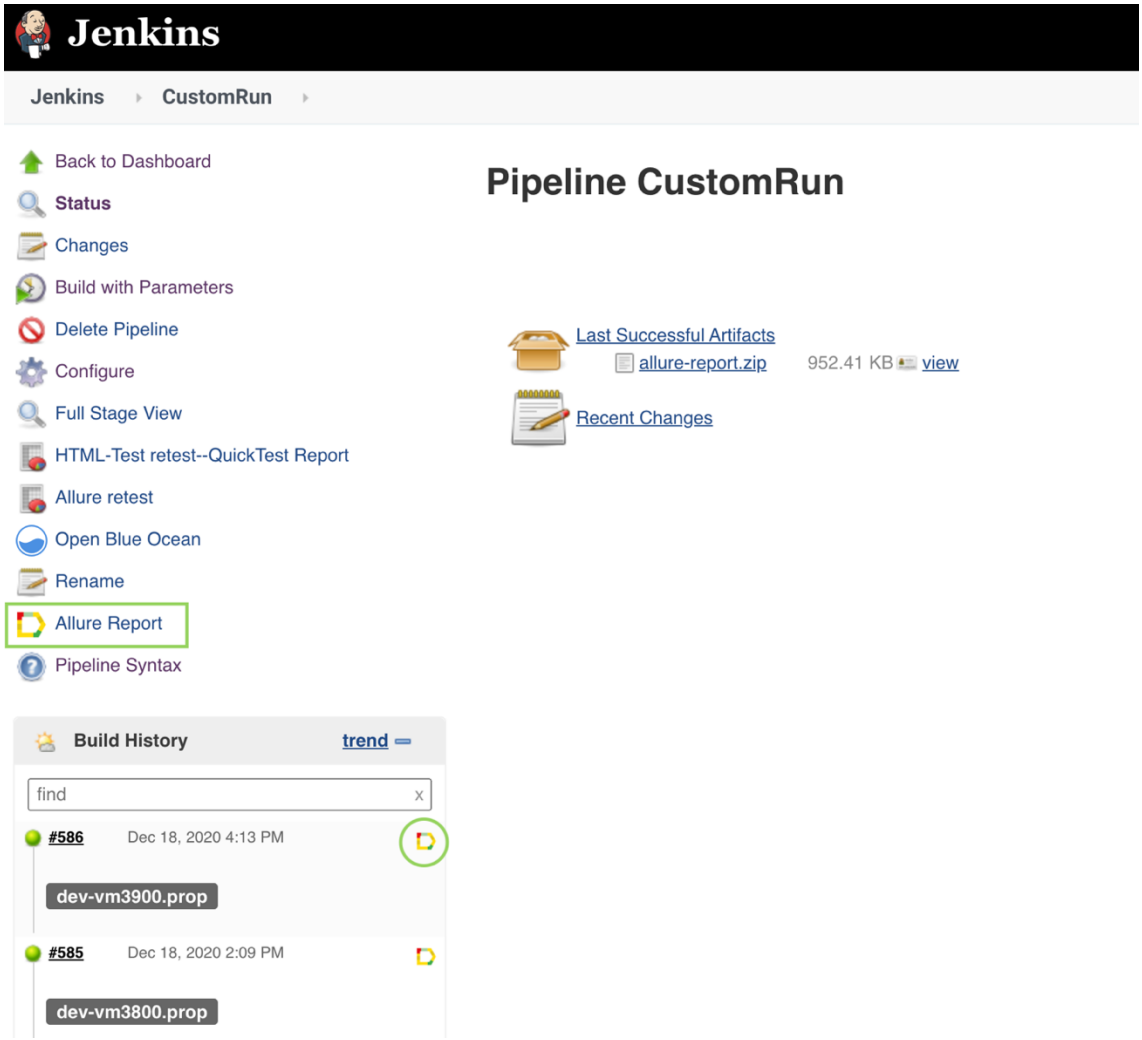
```
11.          reportFiles          : 'index.html',
12.          reportName           : 'Allure ' + configuration.getTestName() ])
```

Kód k vygenerování samotného Allure reportu začíná na řádku č. 3. parametry příkazu *allure* jsou:

- jdk – ve výchozím nastavení je prázdný a znamená to, že je využívána systémová jdk.
- report – jedná se o cestu kam budou vygenerovány soubory reportu.
- results – cesta ke složce odkud budou brány výsledky pro vygenerování reportu.

Dále je v této metodě publikován již vytvořený Allure report pomocí doplňku HTML Publisher[11]. Výše představená metoda je následně volána v průběhu pipeline. Tímto dojde k publikování reportu na serveru Jenkins, viz. přiložený obrázek.

[11] https://plugins.jenkins.io/htmlpublisher



Obrázek 5: Allure v Jenkins (Vlastní zpracování)

6.3.2 Maven pom.xml

Jak bylo zmíněno výše, celý projekt, do kterého je Allure implementován je sestavován pomocí nástroje Maven. Je proto nutné přidat Allure závislost do souboru pom.xml.

V tomto případě v pom.xml, které je označováno jako parent, jsou Allure knihovny definovány v sekci „dependencyManagement“, pod elementem „dependencies“. Zkráceně parent pom poté vypadá takto:

...

```
<allure.version>2.13.7</allure.version>
```

...

```
<dependencyManagement>

  <dependencies>

        <dependency>

          <groupId>io.qameta.allure</groupId>

          <artifactId>allure-java-commons</artifactId>

          <version>${allure.version}</version>

        </dependency>

        <dependency>

          <groupId>io.qameta.allure</groupId>

          <artifactId>allure-testng</artifactId>

          <version>${allure.version}</version>

        </dependency>

  </dependencies>

</dependencyManagement>
```

Pro jednodušší správu je verze artefaktů definována proměnou. Změny verzí jsou pak jednodušší, jelikož stačí přepsat hodnotu na jediném místě.
V pom.xml se jménem webuitests je závislost definována následovně:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>io.qameta.allure</groupId>
      <artifactId>allure-testng</artifactId>
    </dependency>
  </dependencies>
</dependencyManagement>
```

V ostatních pom.xml se reference na Allure nevyskytují.

6.3.3 Přidávání příloh do reportu

V tomto projektu se nevyužívá funkcionality kroků (steps), které Allure Framework nabízí. Za tvorbu logů je zodpovědná třída Reporter.java z TestNg, výstupem je kolekce List typu String. Tímto způsobem vytvořené logy je nutné připojit k Allure reportu, pro tento účel se zavedla v modulu web-automation třída AllureUtils.java. Obsahuje jedinou metodu, jejíž funkce bude do detailu rozebrána níže.

```
import com.blackboard.webautomation.common.Log;
import io.qameta.allure.Allure;

import java.util.List;

public class AllureUtils {
  /**
   * Method parameter usually collects output from Reporter, converts to
String, and attach to Allure Report as HTML file.
   * @param outputList
   */   public static void allureAddAttachmentHTML(List<String> outputList)
{
    try {
      String result = String.join("", outputList);
      Allure.addAttachment("Logs", "text/html", result, "html");
    } catch (Exception e) {
      Log.logError("Exception: " + e.toString());
    }
  }
}
```

Jak se zmiňuje již v dokumentaci, metoda v parametru obdrží List typů String, spojí jednotlivé elementy a uloží do datového typu String. Následuje volání metody z knihoven Allure s názvem *addAttachment()*. Ta v parametrech obdrží název, typ přílohy, obsah, kterým je v tomto případě výše popsaný String. Na posledním místě
stojí koncovka souboru.

Jelikož Reporter z TestNg generuje log ve formátu HTML, i do Allure připojujeme přílohu jako HTML soubor. Tato funkcionalita je obalená try/catch blokem a odchytává se případná výjimka.

Obdobným způsobem je řešeno i přidávání screenshotu v rámci složitější metody takeScreenshot() ve třídě WebSession.java.

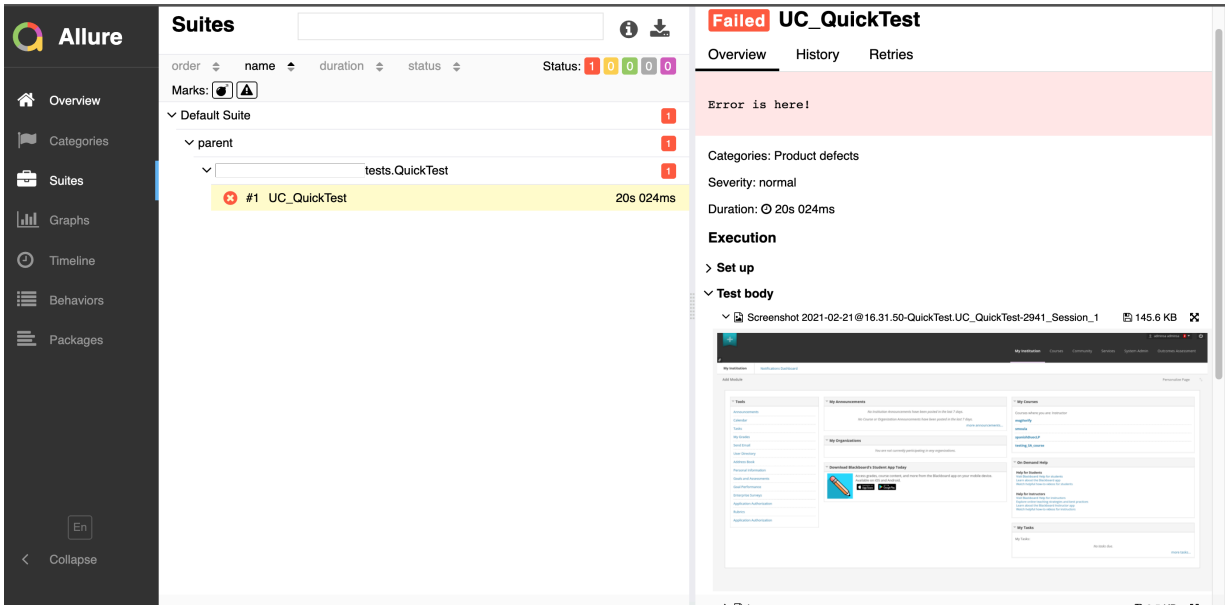
```
/**
 * Takes screenshot of browser content and return absolute path of screenshot.
 * @param name name of screenshot
 * @return absolute path of screenshot or empty string, if there is no screenshot
 */ public String takeScreenshot(String name) {
  String screenshotPath = "";

  try {
    Log.logAction("Current Url: " + webDriver.getCurrentUrl());
    Log.logAction("Taking screenshot: " + name);
    String testcaseFolder = Globals.getCurrentIsoDateTime() + "-" + testcaseName;
    File scrFile = ((TakesScreenshot) webDriver).getScreenshotAs(OutputType.FILE);
    Allure.addAttachment("Screenshot " + name, new FileInputStream(scrFile));
    String filePath = "/screenshots/" + testcaseFolder + "/" + name + ".png";
    FileUtils.copyFile(scrFile, new File("./reports/" + System.getProperty("param.suiteReportDir") + "/html" + filePath));
    Log.logHTML("--> Screenshot: <a href=\"./" + filePath + "\">" + name + "</a> <br>");

    screenshotPath = scrFile.getAbsolutePath();

  } catch (UnhandledAlertException uae) {
    Log.logError("Unhandled Alert Exception!");
    Log.logPermanently(uae.toString());
    alertDismiss();
    if (isAlertPresent()) {
      Log.logTrace(Arrays.toString(uae.getStackTrace()));
      Assert.fail("Cannot dismiss Alert window.");
    }
    takeScreenshot(name);
  } catch (WebDriverException wde) {
    Log.logError("Cannot take screenshot!");
    Log.logPermanently(wde.toString());
    Log.logTrace(Arrays.toString(wde.getStackTrace()));
  } catch (IOException e) {
```

```
    Log.logError("Cannot write screenshot into file system! Exception: " +
e.toString());
    Log.logTrace(Arrays.toString(e.getStackTrace()));
  }
  return screenshotPath;
}
```

Konkrétně na níže uvedeném řádku je volána metoda addAttachment(), která je přetěžována. Tentokrát se volá s jinými parametry. Konkrétně s parametrem typu String obsahující název screenshotu, a poté se přidává screenshot, pomocí otevřením Java FileInputStream.

```
Allure.addAttachment("Screenshot"+ name, new FileInputStream(scrFile));
```

Takto přidaný obrázek se následně v reportu zobrazí jako příloha.



Obrázek 6: Screenshot reportu (Vlastní zpracování)

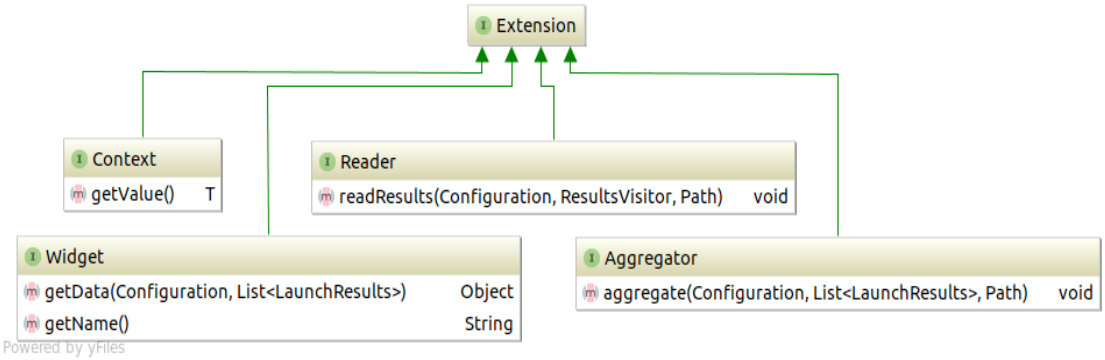
6.4 Další komponenty Allure

Allure Framework nabízí sadu dalších komponent a nástrojů, jenž mohou pozitivně ovlivnit čitelnost reportu z testů dané aplikace. Níže budou představené ty vybrané z nich.

6.4.1 Pluginy

Allure Framework nabízí veřejné API, které slouží ke konfiguraci existujících komponent, nebo vytvoření nových pluginů. Jelikož je Allure navržen tak, aby byl velmi dobře přizpůsobitelný konkrétním požadavkům vývojových týmů různě po světě, vývojářům a testerům, kteří chtějí využívat jiné metriky a mají vytvořený svůj vlastní a unikátní přístup k testování nebo jejich produkty vyžadují odlišné přístupy než ty obecné. Pro tyto případy má Allure systém pluginů, který umožňuje přizpůsobení podoby reportu.

Pro upřesnění se veškerá základní funkcionalita skládá z pluginů a je vnitřně implementována jako plugin. (Qameta Software, 2019)



Obrázek 7: Návrh systému pluginů (Qameta Software 2019)

Výše uvedený vývojový diagram obsahuje model tříd na nejvyšší úrovní pluginového systému Allure. Front-endová implementace je psána v jazyce Javascript za použití frameworku Backbone.JS.

Obecně je možné říct, že každý plugin Allure reportu se tedy bude skládat ze dvou částí. První částí jsou Java třídy, které budou zpracovávat data a vygenerují nějaké výsledky do složky s reportem. Druhá část je zmíněný Backbone.JS tyto soubory převezme a vytvoří jejich reprezentaci na front-endu reportu (widget, tab).

6.4.2 Nestabilní testy

V angličtině je tato funkce Allure nazvána jako „flaky tests“. Do českého jazyka je možné ho přeložit jako nestabilní testy.

V dokumentaci k frameworku se píše následující:

„V reálné praxi nejsou vždy všechny vaše testy stabilní a vždy jen zelené nebo červené. Mohou začít tzv. blikat – tedy čas od času selhávat bez zjevné příčiny. Pokud se těchto testů nechceme zbavit (i přes toto chování obsahují cenné informace, nebo budeme test dále sledovat a vyhodnocovat.), nabízí Allure možnost speciálně takový test označit. Výsledkem toho je, že se v reportu budou zobrazovat jako nestabilní.“

```
@Flaky
public void nestabilniTest() {
     ...
}
```

Při použití kódu výše, se v reportu zobrazí tato ikona.

Obrázek 8: ikona nestabilních testů (Qameta Software 2019)

6.4.3 Informace o prostředí

Testy samozřejmě při běhu musí znát prostředí, ve kterém jsou spouštěny. Tato informace nicméně není vždy patrná v reportu. Allure framework umí tuto informaci propagovat až do UI reportu. Před generováním reportu je nutné přidat do složky allure-results, kde jsou zdrojové soubory pro generování reportu, soubor *environment.properties* nebo *environment.xml*.

Tyto soubory budou obsahovat 3 parametry a těmi jsou „Browser“, který udává název použitého prohlížeče, „Browser.Version“ s aktuální verzí tohoto prohlížeče a „Stand“, který v sobě nese informaci o prostředí (Produkce, Development...).

6.4.4 Kategorie

Allure obsahuje v základní konfiguraci 2 kategorie chyb u testů. Jsou to Product defects a Test defects.

Product defects označují testy s chybou. Jedná se tedy o testy, které spadnou v důsledku

chyby testované aplikace. Test defects se liší tím, že se chyba nachází s největší pravděpodobností v testu samotném a nemělo by tedy jít o chybu testované aplikace.

Allure umožňuje stanovit a přidat další vlastní kategorie. Toho docílíme pomocí vlastního souboru *categories.json*, opět ve složce „allure-results“.

6.5 Java anotace

Allure umožňuje využití těchto konstrukcí z jazyka Java k efektivnějšímu využití možností reportu.

1. **@DisplayName**
    a. Tato anotace umožňuje nastavit vlastní zobrazované jméno konkrétním testům.

2. **@Description**
    a. Slouží k popisu jednotlivých metod testů. Popis se přehledně zobrazuje ve výsledném reportu
3. **@Steps**
    a. Kroky jsou jednotlivé části daného testovacího scénáře. Mohou být znovu použity i v jiných testovacích scénářích, parametrizovány a musí mít název. Mohou přímo v anotaci obsahovat parametry, které budou danou metodou využity.

    Předpokládejme následující třídu. Testy následně využívají tyto třídní proměnné k loginu. V některých případech toto může být nepraktické a je jednodušší volat přímo daný objekt, který tuto informaci ponese. To ilustruje metoda s anotací @Step jenž přímo přistupuje k těmto parametrům.

```
public class User {

    private String name;
    private String password;
    ...
```

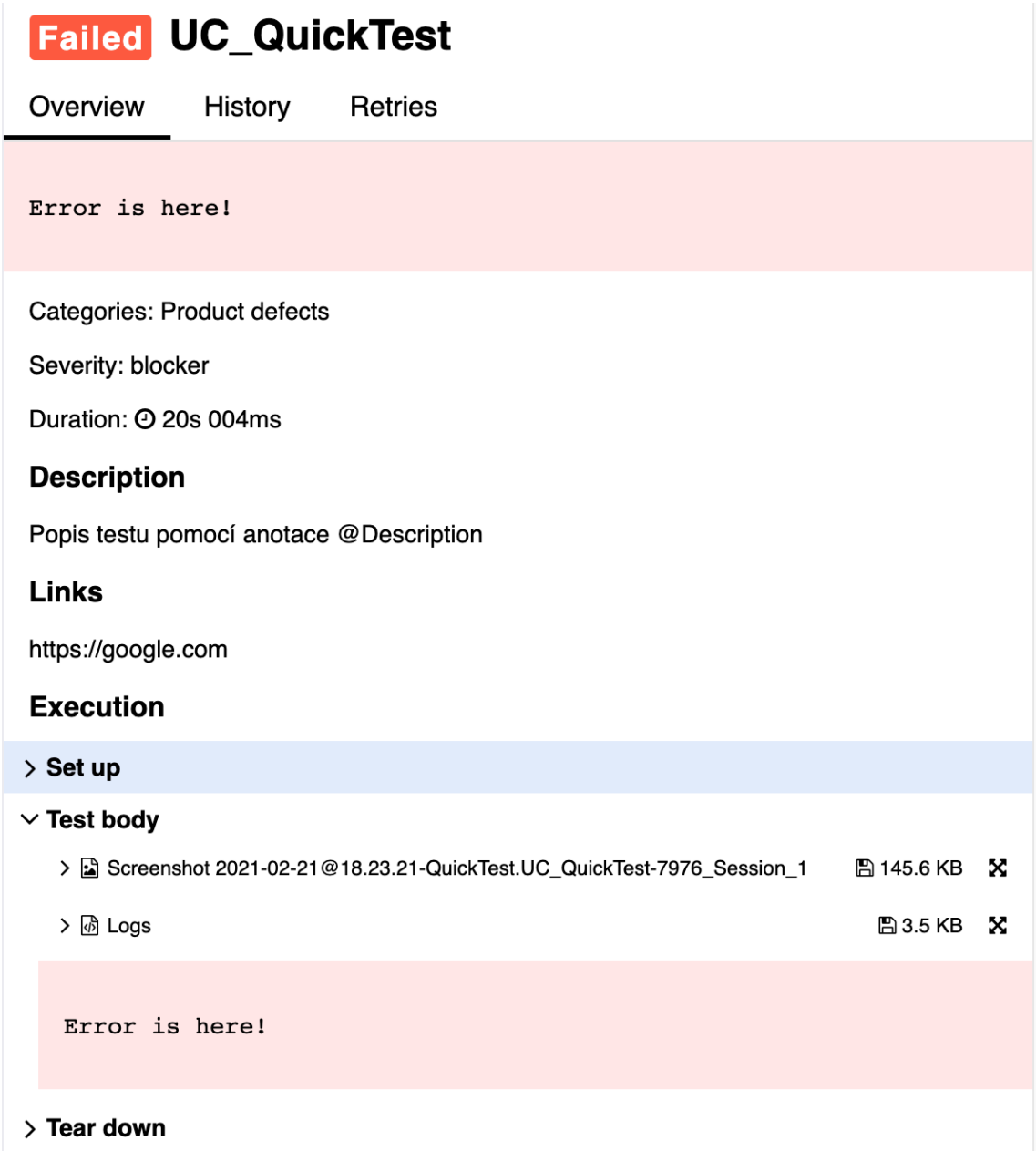
```
}



@Step("Type {user.name} / {user.password}.")
public void loginWith(User user) {
     ...
}
```

(Qameta Software, 2019)

4. **@Attachments**
    a. Anotace sloužící k přidání přílohy do reportu. V případě implementace výše je toto řešeno pomocí pomocných metod třídy Allure.
5. **@Link**
    a. Takto anotované testy je možno provázat přímo k různým TMS[12] nebo nástrojů pro evidence chyb (bug tracker system).
6. **@Severity**
    a. Slouží pro určení míry důležitosti konkrétního testu.
    Allure rozlišuje 5 úrovní. Od nejnižší jsou to: trivial, minor, normal, critical, blocker.

[12] Test Management System – nástroj pro správu testů.

Ukázka použití některých výše zmíněných anotací:



Obrázek 9: Výstup Allure za pomoci anotací (Vlastní zpracování)

Na snímku obrazovky lze vidět kategorii – zde je test automaticky zařazen do kategorie product defects, jelikož došlo k selhání testu na základě vyhodnocení stanovené podmínky, nikoliv kvůli nestabilitě aplikace jako takové.

Níže je možno vidět „severitu“, tedy závažnost nebo lépe důležitost tohoto testu.

Description, tedy popis, je taktéž připomojen pomocí anotace. Toto místo je ideální k zanesení testovacího scénáře do reportu.

V pasáži Links je odkazováno na vyhledávač Google.com, nicméně v případě reálného využití se zde bude nacházet odkaz ku příkladu na TMS systém.

Dále jsou patrné již zmíněné přílohy jako screenshot a soubor obsahující logy testu.

6.6 Dostupnost Allure pro ostatní technologie

Framework Allure je nejvíce používán v jazyce Java. Existují však i jeho implementace do ostatních významných jazyků.

Python

Allure zde využívá testovací framework Pytest[13], instalace je provedena pomocí správce balíků v příkazové řádce Pip[14].

JavaScript

Pro jazyk JavaScript je Allure dostupný hned pro několik frameworků:

Allure reporter je integrován do „lehkého" frameworku Jasmine. Tato technologie však postrádá většinu možností, o kterých se hovořilo výše. Neexistují tam např. parametry, nebo možnost napojení na TMS systémy.

CucumberJs již nabízí většinu těchto možností za pomocí tzv. labelů. Dále je Allure integrován do Mocha frameworku, kde je opět dostupná většina zmíněných vlastností. V tomto případě se nevyužívá anotací, jako v Javě, ale volají se metody přímo na objektu Allure.

Př.

```
allure.description(description)
```

Ruby

V jazyce Ruby nabízí Allure integraci do frameworku s názvem Cucumber[15]. Jedná se o řešení, které je využíváno nejen začátečníky, jelikož se jedná o velmi snadno čitelný framework, který je založen na přirozených větách. Opět je zde dostupná většina funkcí.

Groovy

[13] https://docs.pytest.org/en/stable/
[14] https://pypi.org/project/pip/
[15] https://cucumber.io

Integruje se do frameworku Spock.

PHP

Scala

Adaptér na technologii ScalaTest umožňuje vygenerovat Allure report za použití tohoto frameworku v jazyce Scala. Bez možností podobným anotacím v Javě.

.NET

Allure dává k dispozici integraci s platformou .NET od společnosti Microsoft a technologií MSTest. Opět chybí většina doplňkových možností.

(Qameta Software, 2019)

7 Důvody pro změnu frameworku

7.1 Záměr

Záměrem autora v následující části práce bude porovnat možnosti dvou v teoretické části zmíněných reportovacích frameworků. Na jedné straně zastaralé ReportNg a na straně druhé stále udržovaný a vyvíjený Allure Framework. Autor má záměr porovnat jednotlivé funkce a vhodně zvolit framework k doporučení pro implementaci do projektu.

7.2 Důvod změny reportu

S postupem času vyvstala u řešeného projektu potřeba vhodně oznamovat dosažené výsledky, či spolehlivost na vyšší úrovně společnosti. Do doby před implementací nového frameworku existovala potřeba zpracovávání výsledků testů pouze odborníky z QA oddělení. Dnes je stěžejní informovat i vlastníky produktu a manažery, zejména o tzv. KPI[16]. Ti tak mohou sledovat dopad změn, které implementuje vývojový tým. U změn, jež jsou jednoduššího charakteru takto odpadne nutnost interpretace výsledků členem QA týmu.

S takovou změnou potřeby bylo rozhodnuto o prozkoumání možností implementace nového reportovacího frameworku, jenž by usnadnil práci s výsledky testů a také jejich pochopení i lidem ne přímo zainteresovaným do vývoje softwaru po technické stránce. Bylo rozhodnuto, že dojde k přezkoumání alternativ, složitosti implementace, přidané hodnoty konkrétního řešení a následně vyjádření užitku. Po této analýze vyšlo najevo, že je vhodné implementovat pokročilejší reportovací framework k prezentaci výsledků.

Největší důraz byl kladen na poměr efektivita – cena.
Toto posuzování bylo zcela v kompetenci QA inženýrů, kteří po rozhodnutí představili zvolené řešení týmu a zdůvodnili.

[16] Key Performance Indexes – Klíčové ukazatele výkonu

7.3 Očekávání

Již před provedením analýzy dostupných možných řešení se předpokládalo, že implementace nového reportovacího frameworku značným způsobem zefektivní každodenní proces vyhodnocování výsledků testů. Vzhledem ke komplexnosti testovacích projektů a komplexnosti aplikace, na kterou jsou nasazeny, je stále nutné manuální vyhodnocení člověkem se znalostí infrastruktury a logiky aplikace jako takové. Není proto možné sledovat pouze numerickou interpretaci průběhů testů, které skončily chybně. Naskýtá se tedy příležitost zefektivnit proces, který je vykonáván na denní bázi a není možné jej dobře automatizovat. To je možnost ušetřit značné množství času QA inženýrů a zároveň zjednodušit interpretaci výsledků a tím pádem i zjednodušit rekognici defektů v aplikaci.

Vzniká možnost využít moderních funkcí reportovacích frameworků jako jsou, screenshoty nebo video průběhu, datum a různé možnosti seřazení výsledků.
Existuje předpoklad, že implementace reportovacího frameworku zjednoduší práci QA týmu a zároveň zlepší interpretaci výsledků, čitelnou nikoliv pouze pro QA odborníky, ale i laiky z řad vedení společnosti a produktu. V neposlední řadě se předpokládá také rozšíření pole působnosti stávajícího QA týmu, jeho ještě bližší seznámení se se zdrojovým kódem projektu, který se ve společnosti předává již spoustu let a možnosti vlastního přizpůsobení.

7.4 Výběr konkrétního frameworku

QA tým stál před analýzou a následným výběrem konkrétního nástroje, jenž uspokojí všechny potřeby a zároveň nedojde ke zvýšení nákladů na testy. Bylo posouzeno několik frameworků, porovnány jejich možnosti, výhody oproti aktuálnímu ReportNG a taktéž jednoduchost a flexibilnost implementace do stávajícího kódu tak, aby stávající kód bylo možné využít do maximální možné míry. Jinými slovy mělo jít o nahrazení technologie bez nutnosti měnit signifikantní část testů.

8 Komparace s ostatními frameworky

8.1.1 Robot framework

Je open source framework pro automatizaci testů a robotických procesů. Má aktivní podporu a je používán ve spoustě softwarových společností.

Díky otevřenému kódu je možné jej rozšířit a integrovat s téměř jakýmkoliv jiným nástrojem za účelem výkonných a flexibilních řešení. Disponuje jednoduchou syntaxí a využíváním lidsky dobře čitelných klíčových slov. Nabízí rozšiřitelnost pomocí knihoven v Pythonu nebo Javě. Kolem frameworku existuje široký ekosystém skládající se z knihoven a nástrojů, které vyvinuli uživatelé, nebo společnosti jako samostatné projekty. (Robot Framework Foundation, 2021)
Robot framework nebyl vybrán z důvodu nutnosti implementovat značně složitější změny do stávajícího kódu. Zároveň není příliš užitečné využít pouze reportu z tohoto frameworku, ale je mnohem efektivnější jej implementovat jako celek.

8.1.2 Extent Reports

Extent reports je framework určen přímo jako reportér testů. Jeho implementace je závislá na použití jeho listenerů a to by znamenalo v daném projektu předělání téměř celého jádra testů, což by s sebou přinášelo značnou časovou náročnost. Nutno podotknout, že bez valné přidané hodnoty. Dále je zde menší komunita, a Extent Reports nabízejí tzv. pro verzi, jež narozdíl od verze zdarma obsahuje několik funkcí, které bezplatná verze nikoliv.

Mezi nimi jmenujme například informaci o prostředí, navázání testů na tickety nebo označení tzv. flaky testů. Z těchto důvodů se Extent Reports ukázal jako nevhodný.

8.2 Shrnutí

Vzhledem k výše zmíněným požadavkům, vyšel z komparace nejlépe Allure report. Ten nabízí velmi dobře zpracovanou dokumentaci, otevřený kód, širokou komunitu, u které je možné nalézt podporu při řešení nejrůznějších problémů, aktivní správu autorů včetně poměrně rychlých oprav bugů a taktéž vývoj produktu a držení kroku novými verzemi ostatních nástrojů.

Framework je poskytován zdarma, neobsahuje umístěnou reklamu. Jeho navržení umožňuje velmi dobré uzpůsobení jeho metod pro konkrétní potřeby (toto platí v jazyce Java). Zároveň se jedná o velmi rychlý framework, který je schopen exportovat reporty do HTML formátu. Toto bylo stěžejní, kvůli specifickému navržení testovacích sad, kdy u UI testů není možné generovat report pro každou sadu zvlášť, ale je nutné generovat report pro celou logicky související sadu. Tento report je pak na Jenkinsu publikován pomocí HTML publisheru. Integrace do Jenkinsu a Mavenu je u tohoto frameworku samozřejmostí.

Nejen díky výše popsaným vlastnostem, ale i díky velmi dobře navrženému vzhledu a promyšleným funkcím představuje Allure velmi moderní řešení pro reportování testů.

Taktéž manažeři a laici konstatovali velmi dobrou přehlednost a zpracování informací o testech. Ocenili zejména grafy, nebo časovou osu reportu. Dále také ocenili možnost rychlého přepínání na informace o předchozích běžích testů přímo v rámci reportu. Zároveň bylo oceňováno právě použití individuálně stylovaných logů, které jsou velmi umně přiloženy k danému testu. Zvlášť v případě správně proběhlých textů tyto logy není vůbec nutné rozbalovat, tudíž sbalené nezabírají místo na monitoru a neznepřehledňují prostředí reportu.

Další výhodou je stejné sbalení přiloženého screenshotu, ve starších řešeních byl publikován pouze odkaz a každý obrázek se otevíral v novém okně, nikoliv v náhledu jako u Allure. I odkaz na obrázek však byl v Allure zachován. Důmyslné grafické navržení tak umožňuje vyjít vstříc všem uživatelům reportu, i těm, kteří z nějakého důvodu preferují starší řešení screenshotů.

Jednodušší čitelnost reportu přinesla možnost vyhodnocovat některé druhy testů i vývojářům a je možné konstatovat, že Allure tak přispěl k agilnímu vývoji v týmu. Jednoznačně zvýšil zastupitelnost, a to právě na poli vyhodnocování výsledků testů.

9 Závěr

Jednoznačně je možno doporučit přechod z velmi rozšířeného, ačkoliv zastaralého reportovacího frameworku Report Ng na jiný, open-source, případně payware reportovací framework.

Přes některé překážky, které skýtá implementace obdobného řešení do rozsáhlých projektů, je výsledek a přínos pro střední až velkou společnost nedocenitelný. Při tvorbě testovacího projektu na novou aplikaci, úplně od začátku je vhodný reportovací framework již nutností.

V neposlední řadě je velkým bonusem možnost vývoje vlastního pluginu a jeho jednoduchém zobrazení v reportu.

Závěrem je nutno konstatovat, že implementace nového reportovacího systému byla velmi prospěšným krokem právě z důvodů popsaných v této práci. Allure report se ukázal být velmi vhodný jakožto řešení, jenž má být implementováno do rozsáhlejších, složitějších a zejména již starších projektů. Může sloužit jako jeden z bodů modernizace takovýchto testovacích projektů, jakožto velmi dobře viditelný bod modernizace.

Jak je popsáno v této práci, Allure je možno integrovat do vícero jazyků a testovacích technologií, nicméně integrace do projektu v Javě je téměř nativní. Framework takto nabízí nejvíce ze svých funkcí. Soudě dle diskusních fór komunity a množství dotazů na záležitosti týkající se právě jazyka Java soudím, že se jedná i o technologii s Allurem nejpoužívanější.

Allure je na tolik dobře navržené a komplexní řešení, že se velmi dobře hodí taktéž k integraci do projektů, kde bude plnit roli reportovacího frameworku již od začátku vývoje takovýchto projektů. V takovém případě nebude potřeba v blízké době uvažovat o změně reportovacího nástroje, jelikož Allure report obsáhne vše potřebné. Vzhledem k výše zmíněným vlastnostem a možností poměrně nenáročné integrace do již hotových projektů, jde vskutku o velmi univerzální nástroj s širokým rozsahem.

Jelikož je možné report vygenerovat do podoby HTML a následně publikovat jako webovou stránku (např. v S3 bucketu AWS[17]), jde i o vhodný nástroj na prezentaci výsledků ať už vedení společnosti a týmu, tak i koncovým zákazníkům, pro které bude takto vypadající report velmi snadno čitelný.

Implementaci moderního řešení na reportování výsledků testů obecně, ale i použití právě reportu Allure je možno jednoznačně doporučit.

[17] AWS – Amazon Web Services, jedná se o cloudové služby společnosti Amazon

10 Seznam použité literatury

Šmíd, Vladimír. 2005. Životní cyklus informačního systému. *mis-zivcyk.* [Online] Fakulta informatiky Masarykovy univerzity, 12. 12 2005. [Citace: 29. 5 2020.] https://www.fi.muni.cz/~smid/mis-zivcyk.htm.

CaSTB. 2013. O nás CaSTB. *O nás CaSTB.* [Online] CaSTB, 2013. [Citace: 20. 7 2020.] https://castb.org/cz/o-nas/.

Dyer, Daniel W. 2013. ReportNg. *ReportNG.* [Online] 2013. https://reportng.uncommons.org.

FREEJAVAGUIDE. 2013. History of Java programming language. [Online] 2013. [Citace: 21. 2 2021.] https://www.freejavaguide.com/history.html.

ISTQB. 2020. About ISTQB. *About ISTQB.* [Online] ISTQB, 2020. [Citace: 20. 7 2020.] https://www.istqb.org/about-us.html.

—. 2018. *Foundation Level Syllabus.* [Document] místo neznámé : International Software Testing Qualifications Board, 2018.

Jenkins. 2020. Jenkins doc. *Jenkins dokumentace.* [Online] Jenkins, 2020. https://www.jenkins.io/doc/.

Miroslav Bureš, a kolektiv, Miroslav Renda, Michal Doležel. 2016. *Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu.* místo neznámé : Grada Publishing, 2016. 8024755947, 9788024755946.

O'Regan, Gerard. 2017. Software Testing. *Concise Guide to Software Engineering.* s.l. : Springer, Cham, 2017.

Project, Apache Maven. 2021. Apache Maven . *Apache Maven Project.* [Online] Apache, 2021. https://maven.apache.org.

Qameta Software. 2019. [Online] 19. 9 2019. [Citace: 21. 12 2020.] https://docs.qameta.io/allure/#_installing_a_commandline.

—. 2019. Allure Framework documentation. *Allure documentation.* [Online] Allure, 19. Září 2019. https://docs.qameta.io/allure/.

Robot Framework Foundation. 2021. Robot Framework Introduction. [Online] 2021. [Citace: 28. February 2021.] https://robotframework.org#introduction.

Singh, Vijay. 2020. Hackr. [Online] 9. Duben 2020. https://hackr.io/blog/what-is-frameworks.

Test Management Systems Ltd. 2021. Test management. [Online] 2021. [Citace: 24. 2 2021.] http://www.testmanagement.com/about-test-management/.

Veselovský, Eduard. 2014. *Přehled nástrojů pro automatické testování aplikací.* [Bakalářská práce] Plzeň : Západočeská Univerzita v Plzni, 2014.

Zelinka, Bořek. 2013. *Testování softwaru.* [Document] Praha : Unicorn Systems, 2013.

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Daniel Bechný**
Studium:	I1800632
Studijní program:	B6209 Systémové inženýrství a informatika
Studijní obor:	Informační management
Název bakalářské práce:	**Implementace frameworku Allure do testů v Javě**
Název bakalářské práce AJ:	Implementation of Allure framework into Java tests

Cíl, metody, literatura, předpoklady:

Cíl práce: Cílem je prozkoumat problematiku prezentace výsledků testování a ukázat implementaci pokročilých reportovacích technologií do již existujícího projektu.

Osnova:

1. Úvod
2. Popis vývoje softwaru
3. Rešerše existujících testovacích a reportovacích frameworků
4. Zvolení konkrétního frameworku pro reportování výsledků.
5. Popis procesu implementace vybraného frameworku.
6. Zpracování výsledků
7. Závěr

Garantující pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	doc. Mgr. Tomáš Kozel, Ph.D.
Datum zadání závěrečné práce:	15.10.2020