Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Testování webových aplikací

Zavedení automatických testů ve firmě Polarion s. r. o.

Diplomová práce

Autor: Ludmila Kellerová
Studijní obor: Informační management, IM5

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Hradec Králové duben 2015

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracovala samostatně a s použitím uvedené literatury.

V Hradci Králové dne 25. 4. 2015 Ludmila Kellerová

Poděkování:

Děkuji panu doc. Ing. Filipovi Malému, Ph.D., vedoucímu diplomové práce, za metodické vedení, trpělivost a ochotu. Mé poděkování zasluhuje i B.Sc. Siniša Lopičić, Director of R&D z firmy Polarion Software s. r. o., za věcné připomínky a vstřícnost při konzultacích během zpracování praktické části.

Anotace:

Tato diplomová práce si bere za cíl přiblížit problematiku testování obecně s důrazem na automatizaci v oblasti webových aplikací. Teoretická část nabízí obecný úvod do testování prostřednictvím popisu V-Modelu a testů z něho vyplývajících. Dále jsou předloženy důsledky zavedení agilních metodik v rámci vývoje softwaru, jako jsou krátké sprinty a neustálé dodávky, z hlediska testovacího procesu. Testování webových aplikací již ze své podstaty iniciuje další klíčové body, na které je nutno se soustředit při exekuci výkonnostních a bezpečnostních testů.

Praktická část aplikuje problematiku testování webových aplikací na produktu Polarion ALM. Navíc se věnuje zachycení přerodu firemního testovacího přístupu od tradičního manuálního ke složitějšímu automatizovanému. Polarion zaznamenává pokrok v oblasti automatizace, výkonnostního a bezpečnostního testování, zejména díky požadavkům ze strany klientů týkajících se dostatečné kvality produktu. Firma poskytuje klasické příklady rostoucích problémů v oblasti testování i obecných výzev, kterým musí čelit při tvorbě webové aplikace.

Annotation:

Title: Web application testing

The aim of this diploma thesis is to describe overall testing with emphasis on test automation in web applications. Theoretical part provides introduction into overall testing space referencing V-Model and tests that implies. Taking look into agile methodology in development and how it impacts testing with short sprints and continues deliverables. Testing web applications brings by nature additional focal points in testing such as performance and security (penetration) testing.

Practical part references testing of Polarion ALM web application. In addition there is a focus on transition that company is taking from "traditional" manual testing into complex testing environments. Driven by customer demand for overall good quality of the product, Polarion is making strong contribution in space of test automation, performance testing and penetration testing. Polarion as a software product company provides classical example of growing pains in testing space and overall challenges they face building web applications.

Obsah

1 Úvod

Webové aplikace, jež jsou založeny na principu klient-server, jsou v dnešní době hojně využívány pro svůj široký potenciál. Snazší by snad bylo sepsat seznam odvětví, kde se ještě nepoužívají, než ta kde se uplatňují. Již samotné webové vyhledavače, které laická i odborná veřejnost spouští každodenně, jsou jejich ukázkovými exempláři.

Některé důvody k používání webových aplikací jsou snadno identifikovatelné, jako například, že pro použití více uživateli není nutná jejich mnohonásobná instalace, k jejich spuštění je potřeba pouze webový prohlížeč, který je v dnešní době distribuován téměř v každém operačním systému, popř. je zdarma ke stažení. Dále je významnou pozitivní vlastností jejich dostupnost odkudkoli, a jejich centralizované datové úložiště, což usnadňuje správu a manipulaci s uloženými daty.

V současné době se také klade důraz na snižování času odezvy u tohoto typu aplikací, používají se pokročilejší technologie pro jejich vývoj a existuje zvýšená snaha o zachování funkčnosti a služeb i v případě výpadku připojení k síti či jiným službám. Neopomíjenými faktory jsou i uživatelské rozhraní, přehlednost aplikace, ovladatelnost a interaktivita s uživatelem [51].

Ruku v ruce s tímto trendem narůstá i potřeba **zajistit kvalitu** těchto produktů. Posláním testerů či QA pracovníků je hledat chyby a chránit koncového zákazníka. Náplň práce testerů se tak změnila z původního prokazování správnosti softwaru, na hledání chyb v programu, případně prolomení daného kódu aplikace. Co se však nezměnilo, je ochrana koncového zákazníka a také samotného produktu [41].

S přechodem aplikací na vícevrstvou architekturu a prudkým vývojem v oblasti nových technologií se začaly výrazným způsobem projevovat výkonnostní problémy nově vznikajících systémů. Zpočátku bylo hledání úzkých míst realizováno pomocí manuálně prováděných testů za účasti většího počtu uživatelů. Tento přístup však vystřídaly nově vznikající podpůrné nástroje nejen pro **automatizované** zátěžové testy.

Postupem času, jak se testování stávalo díky svému rozsahu a rozdílnému přístupu jednotlivých řešitelů samostatnou disciplínou, tak zároveň vznikla potřeba sjednotit alespoň základní pojmosloví a kategorie, aby mohly být definovány procesy testování a později, aby bylo možné je začlenit do nadřazených, komplexnějších procesů, a to Quality Assurance, řízení projektu, změnové řízení, apod.

Některé vybrané základní pojmy jsou důležité pro každého, kdo se na procesu testování v nějaké roli podílí.

Akceptační kritéria určují podmínky, za kterých bude systém předán jednak odběrateli, jednak do produkčního prostředí.

Use Case je technika pro zdokumentování případného požadavku na nový systém, nebo změny na stávající systém. Každý use case, poskytuje jeden nebo více scénářů, které zaznamenávají, jak by systém měl spolupracovat s koncovým uživatelem, nebo jiným systémem k dosažení konkrétních hospodářských cílů. Use casy nepopisují žádné interní procesy systému, nebo jak bude systém implementován.

Test Targets vyplývají z uživatelských požadavků (Use Cases). Je třeba zaevidovat všechny oblasti, které je potřeba otestovat, i když je již v daný okamžik jasné, že některé z nich z časových důvodů otestovat nepůjdou, nebo technicky není možné je otestovat.

Testovací případ (**Test Case**) - obsahuje cíl plánovaného testu, precondition[1], post-condition[2] a potřebná data pro realizaci plánovaného testu. Obvykle je vazba Use Case – Test Case 1 : 1. Jeden Test Case může obsahovat několik testů (několik skupin kroků testů) [20].

Kroky testu/test - Nejmenší jednotka testu, která obsahuje popis činnosti testera včetně vazby na konkrétní testovací data a na datech závislého očekávaného chování aplikace. Test obvykle obsahuje několik testovacích kroků. Často se používá i pojem **skript** (dle IEEE829), ačkoliv ten spíše patří automatizovaným testům [24].

Testovací sada - je uživatelem definovaná posloupnost testů, nebo testovacích skriptů, vytvářejících ucelený testovací blok.

Neshoda – jde o identifikaci a zaevidování každého neočekávaného chování testovaného řešení, zjištěný v průběhu testů.

Chyba – neshodu je třeba vyhodnotit a rozhodnout o tom, zda jde o chybu aplikace, chybu v testu, nebo chybu v datech. Také může jít o dosud neuvažované chování aplikace a z neshody se může stát požadavek na změnu.

[1] Jedná se podmínky, které je nutné splnit před exekucí samotného testu.
[2] Zde je výčet všech podmínek, jejichž splnění je vyžadováno po exekuci daného testu.

Záznam o neshodě – neshodu je nutné podrobně popsat, tj. chování aplikace, čas, kdy k chybě došlo (kvůli dohledání chyby v logu), jaká vstupní data byla použita atd. Neshoda je řízena stavy, aby se na žádnou identifikovanou neshodu nezapomnělo a případná chyba nezůstala bez řešení. Pro záznam neshod jsou obvykle využívány tzv. bug-trackingové nástroje, kde je možné nastavit životní cyklus neshody dle potřeb daného projektu.

Funkční specifikace – Komplexní analytické zadání požadavku nebo skupiny požadavků. Obvykle jde o množinu analytických dokumentů, diagramů, algoritmů, které slouží jako zadání pro programátory. Pro podporu analytické práce existují nástroje obvykle využívající UML prokreslení diagramů.

Testovací skript – Obsahuje sekvenci činností „testera“ při testování aplikace zkonvertovanou do formy programového kódu. Používá se pro automatizované funkční testy. Pro automatizované testy existují nástroje, které usnadňují konverzi do programového kódu, umožňují nahrání testovacího skriptu a jeho další úpravy.

Zátěžový skript – je obdobou testovacího skriptu, ve speciální úpravě pro využití v módu spouštění paralelních uživatelů na jednom stroji. I pro tvorbu zátěžových skriptů existují podpůrné nástroje [20].

2 Cíl a metodika práce

Tato kapitola je věnována specifikaci cílů týkajících se teoretické i praktické části práce a bližšímu popisu zvolené metodiky použité pro vypracování praktické části.

2.1 Cíl práce

Práce si bere za cíl přiblížit problematiku automatizace testování webových aplikací, která propojuje teoretické i praktické přístupy.

Teoretická část se tak zaměřuje na definici testovacích artefaktů, které jsou společné pro manuální i automatické testy. Praktická část se soustředí na představení firmy Polarion Software z hlediska potenciálu zajišťování kvality a popis procesu zavádění automatizace ve firmě. Cílem zde je analyzovat problémy z praxe, se kterými se QA pracovníci firmy setkávají a pro které se snaží nacházet řešení, ať už je to za pomoci evaluace nových nástrojů na trhu, změnou testovacích procedur či celkovou změnou přístupu firmy k problematice automatizace. Záměrem je také poodhalit implementační postup při vytváření nových automatizovaných testů.

První část práce, kapitola 3, popisuje obecný přístup k testování, principy a definice typické pro oblast testování softwaru, zmiňuje jeho přínosy, a nahlíží na srovnání manuálních a automatických testů. Detailně se zabývá automatickými testy webových aplikací a typy testů, které je vhodné automatizovat. Zakončení první části je věnováno agilním metodikám vývoje softwaru s přihlédnutím k řízení kvality.

Druhá část je soustředěna na dělení testů z více hledisek. Kapitola 4 seznamuje s typy jednotlivých testů, nejpoužívanějšími přístupy zapojení testování do procesu vývoje. Jednou ze zmíněných metodik je V-model, je zde popsána aplikace testů v čase a také členění z hlediska znalostí testovaného systému. Opomenuty nezůstávají ani testovací nástroje z hlediska jejich dopadu na testovaný systém.

Třetí seznamuje čtenáře s analýzou testovacích postupů ve firmě Polarion Software s. r. o. Nejprve představuje firmu a popisuje její organizační strukturu a strategii. V dalších krocích nahlíží na již provedenou analýzu konkurenčních společností a jejich produktů. Polarion ALM, produkt, který firma vytváří, je charakterizován pomocí strategie na trhu a zároveň jeho technické specifikace. Práce zmiňuje podporované prohlížeče, jež jsou zásadní z pohledu testování. Dále se práce soustředí zejména na přiblížení firemního přístupu k testování a automatizaci, stejně jako seznamuje s již existujícími

testy ve firmě a se zvýšenou potřebou o zlepšení stavu automatizace. Nastiňuje také kroky, které firma směrem k automatizaci již provedla.

Poslední část je věnována detailnímu představení implementace automatizace ve firmě Polarion. Jsou zde uvedeny kroky, které museli vývojáři i QA pracovníci vykonat, aby dosáhli aktuálního stavu automatizace v oblastech zátěžového a bezpečnostního testování a testů uživatelského rozhraní, včetně předchozí evaluace testovacích nástrojů. Součástí všech kroků k řízení a zajišťování kvality je také sestavení plánu automatizace, který tato diplomová práce také prezentuje.

2.2 Metodika

V prvních fázích vzniku této práce byly potřebné informace získávány formou konzultací se zástupci firmy Polarion Software s.r.o. z řad vedoucích pracovníků. Jednalo se o obecné informace týkající se firmy i technické detaily a problémy, se kterými se firma potýkala v minulosti v rámci manuálního i automatického testování svého produktu.

Následovala analýza testovacích přístupů a testovací architektury. Na jejím základě, pak byl vytvořen dlouhodobý plán automatizace testů stávajících a tvorba testů nových. Součástí plánu bylo také studium evaluace vybraných testovacích nástrojů, které jsou v současné době na trhu dostupné, a ověření (Proof of Concept), zda je možno daný nástroj efektivně zapojit do firemní architektury.

Konečně jednou z použitých metod k získání relevantních informací byl samotný proces testování produktu a implementace testovacích scénářů. Získaná zpětná vazba byla jedním z měřítek, zda je vhodné vybraný nástroj zapojit do firemních testovacích procesů.

3 Testovací přístupy

Následující kapitola je věnována popisu základních konceptů, obsahuje definici testování softwaru, zmiňuje jeho přínosy, a nahlíží na srovnání manuálních a automatických testů. Ve větším detailu se soustředí na automatické testy webových aplikací a typy testů, které je vhodné automatizovat, dále nastiňuje agilní metodiky vývoje softwaru s přihlédnutím k testování.

3.1 Základní koncepty

Následující část je věnována popisu základních konceptů dle [41] a [36], které oblast zkoumání a ověřování kvality využívá.

Kvalita softwaru dle normy ISO/IEC 25010 je míra, do jaké softwarový produkt splňuje stanovené a implicitní potřeby, je-li používán za stanovených podmínek.

Zajišťování kvality softwaru (**Software Quality Assurance**), do kterého lze zahrnout i plánování, je zaměřeno na kvalitu procesů celého životního cyklu softwaru, což ve výsledku silně ovlivňuje kvalitu samotného produktu. Mezi hlavní aktivity se řadí definice, zavedení procesů (včetně norem, procedur, metrik a nástrojů) a následná kontrola jejich dodržování a hodnocení s cílem nalézt případná zlepšení. Zajišťování kvality se tak snaží primárně předcházet vzniku defektů.

Řízení kvality softwaru (**Software Quality Control**) je zaměřeno na výstupy z jednotlivých procesů (typickými produkty jsou dokumentace, kód a spustitelný produkt), u kterých ověřuje, zda odpovídají specifikacím a všem požadavkům. Řízení kvality tak využívá kromě samotného testování produktu také techniky statické analýzy, tzn. revize, inspekce či strukturované procházení. Řízení kvality je orientováno na nalézání defektů, jejich odstranění a následné ověřování správnosti provedených změn.

Verifikace a validace (označované také jako **V&V aktivity**) jsou velmi blízké, avšak nikoli zaměnitelné koncepty, které jsou na sobě závislé a měly by být prováděny již od počátku projektu.

Verifikace je proces, jehož cílem je ověření, že dílčí produkt vývoje softwaru náležitě odráží specifikované požadavky. Podle normy ISO/IEC 12207:2008 pak verifikační aktivity zahrnují:

- Verifikaci samotných požadavků, Ty musí být konzistentní, proveditelné, kompletní a testovatelné.
- Verifikaci návrhu. Ověření, že návrh systému je správně odvozen z požadavků, přičemž vazby je možné dohledat.
- Verifikaci zdrojového kódu. Kód odpovídá návrhu, je správný, testovatelný, v souladu se stanovenými standardy, umožňuje dohledat vazbu k jednotlivým požadavkům.
- Verifikaci integrace. Zde je cílem ověření, že jednotlivé moduly jsou správně a kompletně integrovány do systému jako celku.
- Verifikaci dokumentace. Ověření, že dokumentace je úplná, v souladu s požadavky, konzistentní.

Validace je proces potvrzení, že dílčí produkt vývoje softwaru je správný s ohledem na požadavky na jeho zamýšlené použití. Jinými slovy, že funguje dle očekávání zákazníka, přičemž z jeho pohledu je také daný produkt testován. Problémy nalezené během validačního testování indikují problém s požadavky, tedy že produkt může zcela odpovídat specifikacím, ale neodpovídá představě zákazníka. Nejvýraznějším příkladem validace je akceptační testování.

3.2 Definice testování softwaru

Tento proces lze definovat jako postup řízeného spouštění softwarového produktu s cílem zjistit, zda splňuje specifikované či implicitní potřeby uživatelů [41].

Kromě vyhledávání a následné detekce defektů zde dochází také k ověřování zamýšleného záměru testované aplikace. Tento přístup blíže popisuje Boris Beizer:

- „Úroveň 0: Mezi testováním a laděním (debugging) není rozdíl. Testování je chápáno jako aktivita pomáhající odstraňovat defekty, bez výrazného odlišení od ladění.
- Úroveň 1: Smyslem testování je prokázat, že software funguje. Testování má demonstrovat funkčnost softwaru a soulad se specifikacemi – zaměřeno na demonstraci.
- Úroveň 2: Smyslem testování je prokázat, že software nefunguje. Testování je způsobem, jak nalézat implementační defekty, nesoulad se specifikací – zaměřeno na destrukci.

- Úroveň 3: Smyslem testování není prokazování ničeho konkrétního, ale snížení rizika. Role testování je rozšířena: kromě kódu se zaměřuje i na požadavky a návrh a stává se součástí celého cyklu vývoje softwaru.
- Úroveň 4: Testování je duševní disciplína vedoucí k produkci softwaru s nízkým rizikem. Tato úroveň již chápe testování a ostatní aktivity řízení kvality jako aktivity preventivní a především se tak systematicky zaměřuje na předcházení vzniku defektů v požadavcích, návrhu a konečně i kódu“. [41]

3.2.1 Přínosy testování

Za hlavní přínos testování může být považováno zajištění kvality výsledného produktu na základě odhalování a opravy jeho chyb.

„Cílem softwarového testera je vyhledávat chyby, vyhledat je co nejdříve a zajistit jejich nápravu“. [36]

V této chvíli je vhodné zmínit, co je považováno za **chybu**. Ron Patton sestavuje definici chyby v pěti následujících bodech:

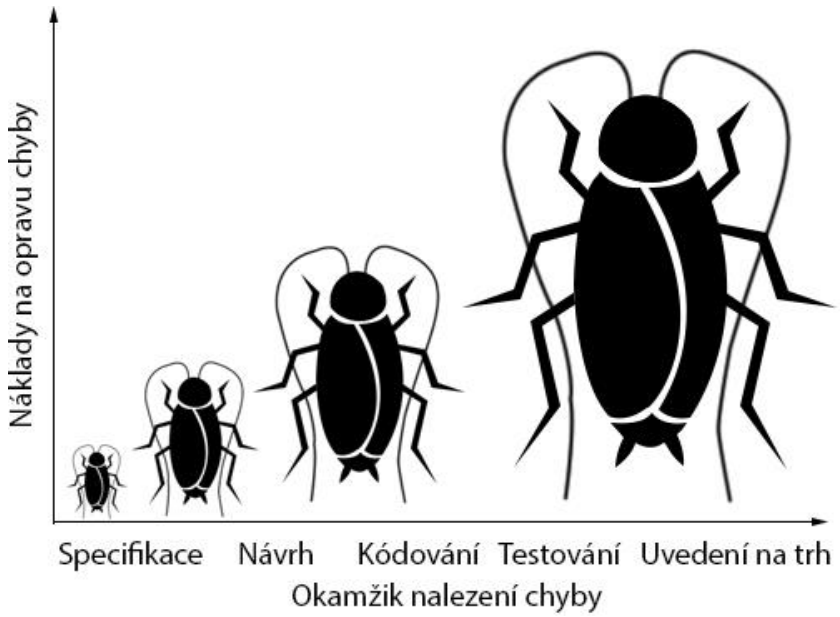
1. „Software nedělá něco, co by podle specifikace produktu dělat měl.
2. Software dělá něco, co by podle údajů specifikace produktu dělat neměl.
3. Software dělá něco, o čem se produktová specifikace nezmiňuje.
4. Software dělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat.
5. Software je obtížně srozumitelný, těžko se s ním pracuje, je pomalý nebo – podle názoru testera softwaru – jej koncový uživatel nebude považovat za správný“. [36]

Podle očekávaných či zjištěných projevů a následků defektu je možné ohodnotit jeho **závažnost** (**severity**) neboli míru negativního dopadu na systém či jeho část, a tím na poskytovanou službu. Jednotná škála klasifikace závažnosti defektů neexistuje, v praxi je často používaná následující čtyřstupňová škála závažnosti defektu:

1. Kritická: defekt ovlivňuje funkce kritické pro daný systém a znázorňuje jej správně používat. Může jít např. o riziko ztráty dat, havárii systému, vážné ohrožení bezpečnosti apod.

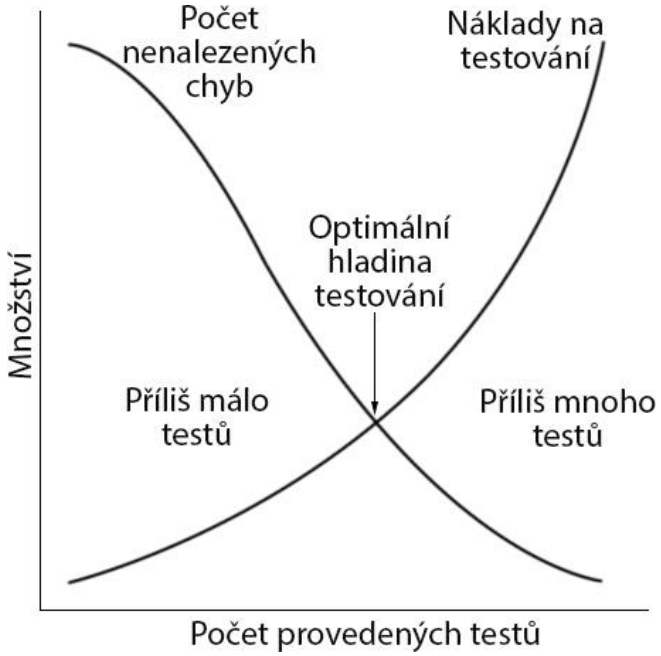
2. Vysoká: defekt ovlivňuje významné funkce systému, ovšem jeho přítomnost umožňuje (v omezené míře a s obtížemi) systém používat.
3. Střední: defekt neomezující významnou funkčnost systému, případně působící nadměrné obtíže uživatelům při jeho používání.
4. Nízká: defekt nijak neovlivňuje funkčnost systému. Typicky jde o kosmetické vady, jako chyby v textových popiscích, špatný typ písma, drobné nekonzistence v grafickém uživatelském rozhraní (GUI neboli Graphical User Interface) např. velikost, barva tlačítek či jiných ovládacích prvků.

V základních fázích vývoje je objevování a oprava chyb nejlevnější. Oprava v této fázi pohlcuje nejnižší náklady na čas i finance[36].



Obrázek 1: Náklady na opravu chyb v čase [36]

To nahrává domněnce, že čím více testů bude systém pokrývat, tím vyšší kvalitu systému lze zajistit. S ohledem na použitý čas a finanční prostředky, které firma může či chce obětovat, je nutné říci, že většině případů je nemožné otestovat každou část systému nebo vlastnost, protože jednoduše existuje buď příliš velký počet možných vstupů a výstupů nebo existuje příliš mnoho cest, které skrze software vedou. Testování je tedy efektivní a přínosné pouze do chvíle kdy je dosažena **optimální hladina testování** dané množiny testů či potažmo projektu[36].



Obrázek 2: Optimální hladina testování [36]

Na základě přístupu **Good Enough Quality**, který souvisí s tématikou pokrytí systému testy, lze provést výběr testů, které nebudou realizovány, s minimálním dopadem na konečnou kvalitu požadovaného produktu.

3.3 Manuální a automatické testy

Dle způsobu exekuce testů se testy člení na **manuální**, které tester provádí ručně, procházením jednotlivých kroků v rámci testovacího scénáře, a na testy **automatické**, které jsou implementovány pomocí zdrojového kódu.

Nevýhodou manuálních testů je jejich časová náročnost, nemožnost otestovat velké množství vstupních dat a také finanční náročnost v případě, kdy je testování nutné opakovat. Výhodou jsou nižší požadavky na lidské zdroje, zejména programátorské schopnosti testera a také fakt, že u některých typů testu automatizace použít nelze a je nutné provádět je ručně (např. test použitelnosti) [14].

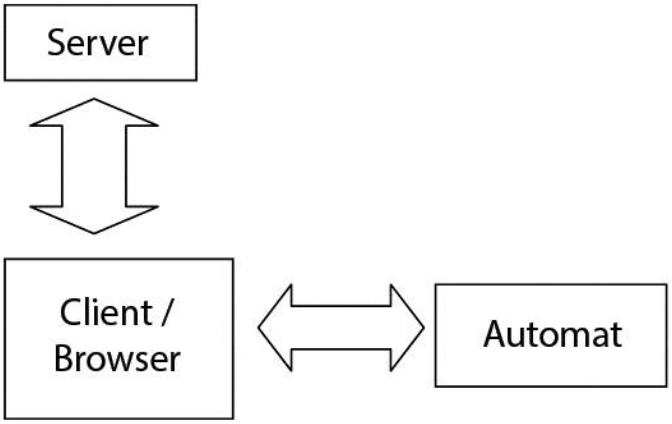
Automatické testy přesně pokrývají nedostatky manuálních testů, neboť jsou rychlé, efektivní, umožňují testování vstupních dat velkého objemu a jejich opakování je snazší a méně finančně náročné. Oproti testování uživatelem je automatický test vždy správný a přesný a také se vyznačuje neúnavností. Zpracování výsledků testů je jednodušší, protože mohou být automaticky zapisovány do logu. Některé typy testů také nelze provést jinak než automaticky (např. zátěžové testování). Naproti tomu mají tyto testy vyšší

vstupní náklady v podobě času stráveného nad tvorbou testovacích skriptů dle navržených scénářů [14][36].

Vstupní náklady automatizace jsou však návratnou investicí, protože vyvinuté testy či jejich části (vytvářející framework) mohou být znovupoužitelné, a uspořit náklady, které by byly věnovány do budoucích projektů [41].

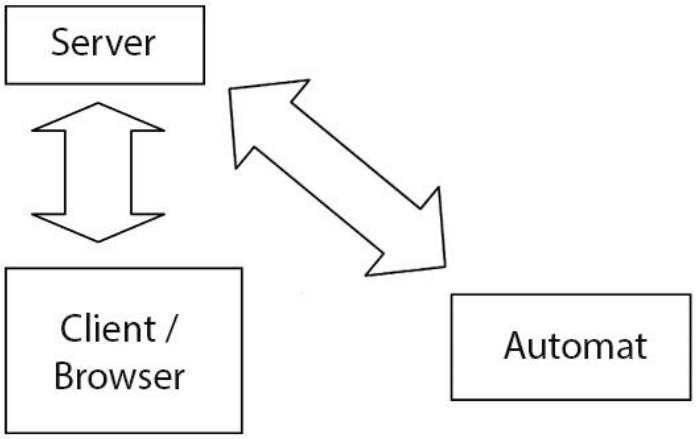
3.4 Automatické testy webových aplikací

Automatizované testy typu klient-server lze dělit na dva typy podle toho, k čemu má testovací automat přímý přístup. Prvním typem je **automat simulující uživatele**. V této architektuře je jeho přístup k serveru zprostředkovaný, a to pouze pomocí klienta. Prováděné operace mají stejný charakter, jako by test prováděl živý tester. Důležitou podmínkou je schopnost automatické aplikace porozumět obsahu zobrazených dat a informací. Lidský tester vidí dosažený výsledek hned. Je tedy nezbytné, aby automat správně identifikoval výstupní oblast a rozeznal získaný výsledek, který následně porovnává s výsledkem očekávaným. Nevýhodou toho přístupu je omezená životnost testů, protože jsou závislé na vzhledu stránky (z hlediska rozložení). Jakákoli změna tedy může vést k nutnosti přepracovat celý kód testu[9].



Obrázek 3: Schéma komunikace automatu při simulaci uživatele [9]

Druhým typem je simulace stroje, kdy dochází k obejití klienta a **automat přistupuje přímo k serveru**. Testována je tedy čistě funkcionalita serveru skrze http požadavky. Uživatelské rozhraní (GUI) je v tomto případě nedotčeno, a proto jsou tyto testy odolnější vůči všem změnám provedeným v rozhraní, což je značná výhoda. Naproti tomu však uživatelské rozhraní není v tomto případě testováno vůbec a případné činnosti prováděné na klientovi (jako může být validace vstupů) zůstávají neotestovány [9].



Obrázek 4: Schéma komunikace automatu při simulaci stroje [9]

3.4.1 Oblasti automatizace a využitelné typy testů

Jednou ze základních chyb je očekávat či vyžadovat od automatizace stoprocentní pokrytí testovacích případů. Nasazení automatizovaných testů by měla předcházet analýza, zda vůbec bude automatizace účelná [20][41].

Proč tedy automatizovat? V případě, že je jisté, že je nutno provést více kol testů, automatizace zkrátí čas na provedení následujícího opakování. Tím, že lze snadno parametrizovat skripty, automatizace přináší možnost protestovat různé datové varianty v krátkém časovém úseku. Kromě variant vstupních dat se nabízí možnost variací i vývojových prostředí – v případě webových aplikací se může jednat o kombinaci operačních systémů a webových prohlížečů.

V první fázi implementace kódu nově vznikající aplikace je typické využití automatizace pro **jednotkové testy** (Unit tests) a v dalších fázích projektu také pro **integrační testování**.

Jak již bylo řečeno, automatizované **funkční testování** je poměrně náročné na přípravu a údržbu, není tedy vhodné jej použít pro nově vyvíjené uživatelské rozhraní. Automatizace testů má smysl tam, kde se již příliš často nemění GUI, ale mění se například jen některé funkční vlastnosti aplikace, např. zpracování dat na aplikačním serveru. Často se využívá při **regresních testech** jako ověření, že realizovanou změnou nedošlo k poškození stávajících funkčností, které změna postihnout neměla.

Výjimkou je testování webových služeb (Web Services[3]), kdy je obtížné testovat je pomocí **testů černé skříňky** (Black-box testů)a má tedy smysl využít automatizovaný test [20].

Při simulaci zátěže generované stovkami až tisíci uživatelů se bez určité úrovně automatizace obejít nelze, proto se vhodné ji nasadit v případě **zátěžového testování** [41].

Bezpečnostní testy jsou za určitých okolností automatizovány, ale v takových případech jde spíše o ověřování odolnosti vůči známým způsobům napadení či jejich variantám, nikoli o inteligentní hledání slabin systému. Specialisté často automatizují jednotkové či funkční testy s cílem narušit bezpečnost systému.

V aplikacích, kde je obrazovkové flow řízeno datově, je automatizace problematická, a v těchto případech je třeba mít analýzu jednotlivých průchodů a množiny dat, pro které platí.

Jedním z důvodů proč automatizace není tak rozšířená je fakt, že zákazníci mají často přehnaná očekávání a nevhodným výběrem nástroje výsledek ještě více sníží (ne vždy se výběr lacinějšího, či open source nástroje vyplatí).

Dalším kritickým faktorem je výběr testovacích případů nevhodných pro automatizaci a s tím spojená spolehlivost testovacích skriptů, například při synchronizaci. Dalším problémem je údržba skriptů, kdy i malá změna testované aplikace může vést k jeho nefunkčnosti.

Kromě pracnosti s údržbou testovacích skriptů bývá problém s vyšší úrovní QA inženýrů, kteří znají manuální testování, ale navíc umí i programovat a jsou tedy i nákladnější (SDET neboli Software Development Engineer in Test) [20].

[3] Webová služba – jak ji definuje konsorcium W3C – je programová součást navržená tak, aby podporovala schopnosti spolupráce mezi zařízeními (machine-to-machine) v rámci jednotlivých interakcí probíhajících po síti. Rozhraní webové služby je strojově srozumitelné díky použití formátu WSDL (Web Service Description Language). Ostatní systémy komunikují s webovými službami pomocí SOAP zpráv. Tyto zprávy jsou typicky dopravovány skrze HTTP za použití XML serializace a dalších webových stadardů. Lze tedy říci, že se jedná o rozhraní, které rychle poskytuje přístup k dané funkcionalitě po síti. [25]

3.5 Agilní metodiky vývoje z pohledu testování

Agilní přístup, jak napovídá samotný název, znamená adaptabilitu, flexibilitu, či přizpůsobivost změnám, které velmi často přicházejí během vývoje, což bývá pro tradiční přístupy často problematické. Hodnoty a principy agilního vývoje byly v roce 2001 formulovány IT odborníky pracujícími s alternativními způsoby vývoje do tzv. Agilního manifestu [41]. Ten jako hodnoty uvádí následující:

- Jednotlivci a interakce před procesy a nástroji.
- Fungující software před vyčerpávající dokumentací.
- Spolupráce se zákazníkem před vyjednáváním o smlouvě.
- Reagování na změny před dodržováním plánu [6].

Oproti tradičnímu způsobu vývoje je tak agilní přístup silně orientovaný na jednotlivé členy týmu, jejich schopnosti a interakci. Do pozadí naopak ustupuje striktní dodržování procesů, používání daných nástrojů či vytváření jiné než nezbytně nutné dokumentace. Agilní vývoj je proto jen málo formální a klade především důraz na doručování funkčního softwaru, což slouží i jako metrika pokroku na projektu.

Agilní přístupy využívají kratších iterací, kdy jednotlivé funkční celky jsou dodávány v časovém rámci většinou dvou až čtyř týdnů. Požadavky se implementují podle zákazníkem určené priority. Zásadní je pak cena, časová náročnost odvozená z jejich odhadnuté obtížnosti. To tvoří omezení, za kterého má tým během jednotlivých iterací doručit co nejvíce implementovaných požadavků.

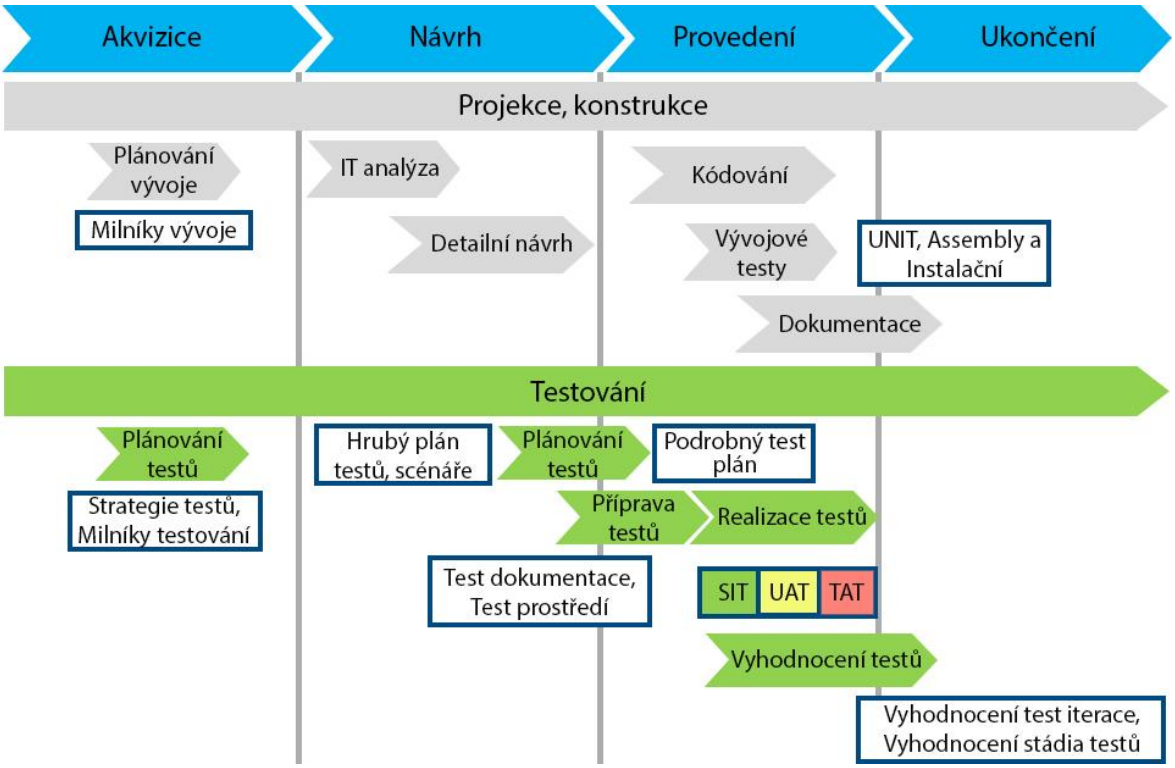
Mezi frameworky agilních přístupů patří například velmi známý a populární **SCRUM** [41].

3.5.1 Aktivity procesu testování v rámci obecného projektu

Proces testování si lze představit jako uzavřený kruh, který má počátek u evidence a správy test požadavků. Dále pokračuje přes plánování testů k samotné přípravě a exekuci funkčních testů, a to jak manuálních tak automatizovaných. Po dokončení funkčního testování přichází na řadu zátěžové testování, které je primárně prováděno uvnitř sítě zákazníka. Dle charakteru aplikace lze toto testování realizovat mimo síť a zohlednit tak práci bezpečnostních síťových prvků. Zde ovšem proces nekončí a z výsledků testování většinou vzejde další kolo testů, které má analogický průběh.

Při pohledu na procesy řízení projektu, lze identifikovat, že projekt má čtyři fáze:

- **Akviziční**, kdy je vymezen rozsah (scope) projektu, jsou definovány oblasti řešení (včetně strategie testování, tj. použití typů testů v průběhu projektu), jsou provedeny odhady pracnosti všech aktivit, jsou definovány hlavní milníky projektu.
- Ve fázi **Návrh** probíhá IT analýza, vzniká dokument IT-solution, který obvykle schvaluje zadavatel jako finální potvrzení pochopení požadavků na řešitelské straně. Po schválení IT solution jsou zahájeny práce na detailním návrhu, který slouží jako zadání pro vývojáře. Za oblast testování vzniká dokument Plán testů, kde jsou definovány oblasti, které budou testovány (Test Targets), je upřesněn odhad pracnosti, jsou definovány milníky testování.
- Ve fázi **Provedení** probíhá vlastní vývoj softwarového řešení, příprava testů, příprava testovacího prostředí, příprava testovacích dat a posléze i realizace jednotlivých stádií testů (vývojové, SIT, UAT a TAT testy).
- Ve fázi **Ukončení** projektu je řešení nasazeno do produkčního prostředí a předáno do rutinního provozu. Dále je pak kompletována dokumentace, akceptační protokoly, je vypracováno závěrečné vyhodnocení projektu a projekt je ukončen [20].



Obrázek 5: Aktivity procesu testování v rámci obecného projektu. Upraveno dle [20].

Pro testování je nutno znát jednoznačné, měřitelné (testovatelné) **požadavky** a jednoznačná kritéria jejich naplnění. Na uvedeném příkladu je vidět, že je nutné ověření již ve fázi definice uživatelských požadavků, protože netestovatelné požadavky jsou buď ignorovány, pak řešení nemusí splňovat představy uživatele, nebo je nutné v okamžiku zjištění nejednoznačných požadavků provést jejich revizi a upřesnění, což může v konečném důsledku způsobit zpoždění projektu.

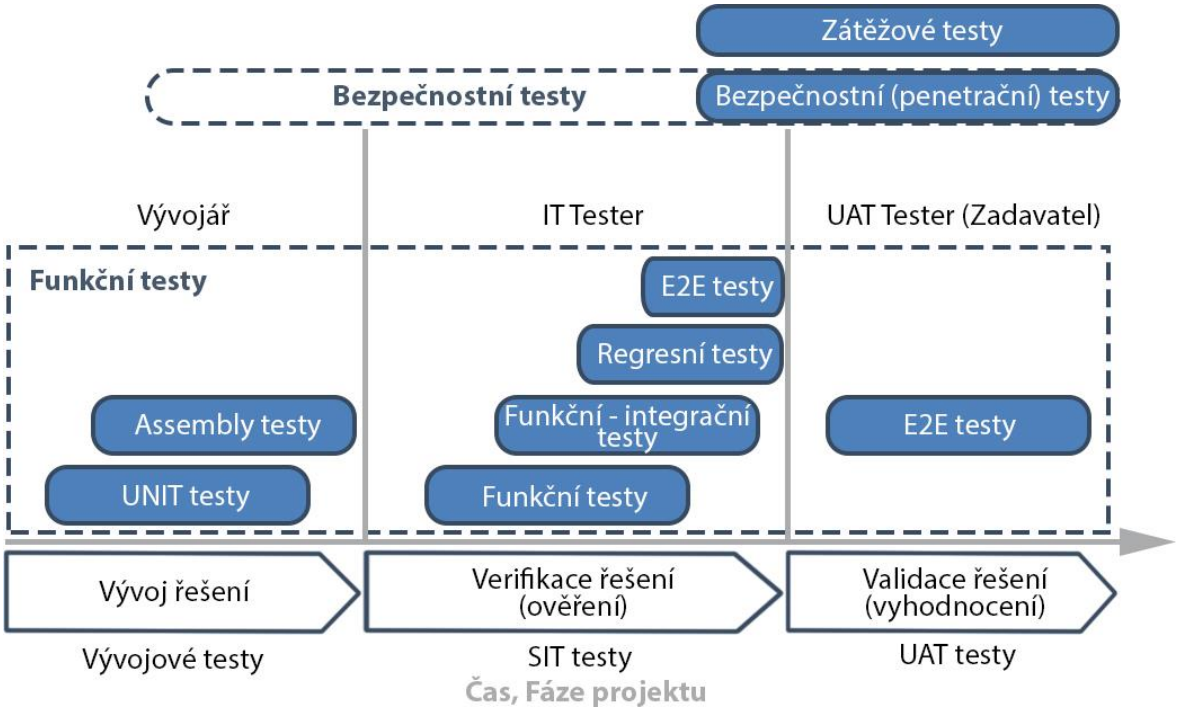
Kritéria pro schvalování řešení se nazývají akceptační. **Akceptační kritéria** by měla být nadefinována současně se zadáním požadavku.

Je nutné si uvědomit, že testováním je možné chyby odhalovat, ale ne garantovat bezchybnost aplikace, proto i akceptační kritéria připouštějí nasazení řešení do produkčního prostředí s identifikovanými chybami. Jistý Murphy říká: „Použije-li programátor příkaz o dvou písmenech, udělá v něm jednu chybu. Opraví-li programátor tuto chybu, pak obsahuje příkaz chyby dvě“ [20].

4 Fáze a typy testů

Bez ohledu na metodiky testování lze dělit testy z několika hledisek. Jedním z nich je aplikace testů v čase v rámci vývoje aplikace, druhým je zobrazení pomocí V-modelu. Následující části jsou věnovány členění testů z pohledu znalosti zdrojového kódu aplikace, která je podrobena testům, a dalším odvozeným typům testů pokrývajícím jiné než funkční vlastnosti aplikace.

4.1 Aplikace typů testů v čase



Obrázek 6: Typy testů v časové ose projektu. Upraveno dle [20].

Na obrázku 6 jsou zobrazeny projekty nejčastěji využívané typy testů a jejich rozložení v časové ose se zvýrazněním fází projektu (vývoj řešení, ověřování, vyhodnocení) a stádia testů (vývojové, SIT, UAT). Zátěžový a penetrační test má smysl realizovat až v době končících UAT testů, protože by měly ověřovat finální stav řešení před předáním do produkčního prostředí a obvykle bývají jejich výsledky součástí akceptačního protokolu. Kategorie bezpečnostních testů však může mít zastoupení již při revizi návrhu řešení, review kódu i funkčních testech (ověřování loginu).

V rámci SIT testů se postupuje od testování jednotlivých funkčností na jednotlivých aplikacích řešení přes testy integrace mezi aplikacemi a základního ověření E2E testy. V poslední iteraci testů by měly být regresními testy ověřeny vybrané funkčnosti, do kterých změna přímo nezasahovala, ale mohly být ovlivněny novým buildem.

Jedno z hledisek jaké testy probíhají je stádium testů – vývojové, SIT, UAT. Vývojové testy obvykle provádí přímo vývojář, SIT testy provádí specializovaný QA pracovník a UAT provádí zástupce zadavatele (často zkušení koncoví uživatelé).

Assembly, **funkční, integrační, zátěžové, bezpečnostní, regresní, instalační** a **smoke testy** budou blíže popsány u V-modelu.

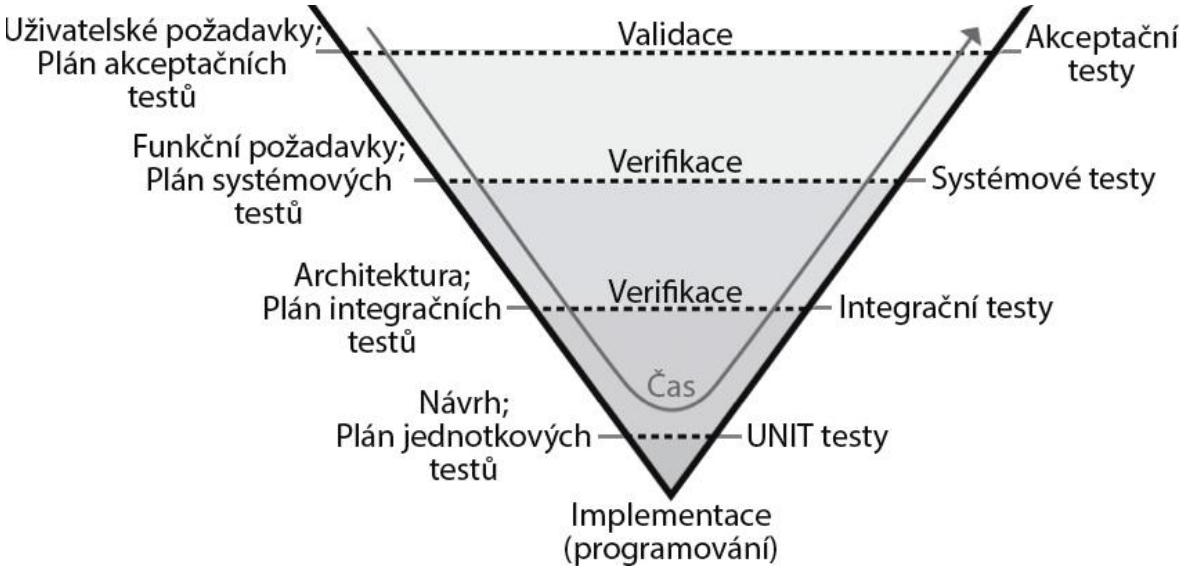
E2E testy – prakticky jde o manuální integrační testy, kde se ověřuje celý business proces vybraného produktu, obvykle se již neověřují dílčí funkčnosti, ale klade se důraz na realizovatelnost požadovaného produktu v celém jeho životním cyklu. Užívaný především v UAT testech.

Technologické testy - slouží k ověření, zda je vybraný testovací nástroj použitelný pro testování nad danou aplikací [15].

4.2 Typy testů z hlediska V-modelu

Poměrně známé zobrazení životního cyklu tvorby softwaru je tzv. V-model, kde je znázorněno, co je ve které fázi projektu testováno a proti jaké dokumentaci. Vývojové testy tak probíhají proti detailnímu návrhu, testy komponent pak přibírají jako podklad IT solution s návrhem řešení. Testování systému obvykle probíhá formou integračních testů, jako vstupní dokumentace jsou přibrány systémové požadavky. Akceptační testování probíhá proti uživatelským požadavkům, kde jsou formulovány produkty a jejich vlastnosti.

Z uvedeného je patrné, že dojde-li k vynechání některého ze vstupů (v reálu často nejsou v rámci projektu zpracovány uživatelské a systémové požadavky), nemůže být otestování úplné, nebo si musí testovací tým shánět vstupní informace pro přípravu testů jiným způsobem, např. konzultacemi se zadavatelem, což není vždy možné [20].



Obrázek 7: Dělení testovacích fází na základě V-modelu. Upraveno dle [20] a [35].

4.2.1 Testování programátorem

„Ihned po vytvoření programového kódu, je tento kód prověřen programátorem. V praxi jsou tyto testy označovány jako Assembly tests. Většinou si však programátor netestuje svoji část kódu, ale realizuje se tzv. test čtyř očí“. [22]

Vzhledem k tomu, že tyto testy jsou prováděny v počáteční fázi vývoje, případná oprava chyb je levnější jak z hlediska času tak finančních prostředků. Chyba, kterou objeví sám vývojář, lze opravit levněji, než tato stejná chyba objevená testerem v pozdějších etapách. Tento přístup lze uplatnit na všech typech projektů.

4.2.2 Testování jednotek

Fáze, která navazuje na testování programátorem. „U objektově orientovaného programování se jedná o testování jednotlivých tříd a metod“. [22]

Zda k této fázi v rámci testování dojde, je nutné rozhodnout již v návrhu projektu. „Testy jednotek se velmi špatně aplikují na již zaběhlých projektech. U již vytvořených aplikací se většinou musí provést kompletní refaktoring kódu či dokonce mnohem hlubší úpravy“. [22]

Unitní testy jsou tvořeny programovým kódem, je tudíž časté, že jejich tvorba je v rukou vývojářů. Ti je píší s využitím nástrojů na bázi frameworků, které jim usnadňují a urychlují vývoj testů.

Každý vývojář by si měl zaměřit i na to, jestli používá vhodné algoritmy, návrhové vzory a datové typy [15].

4.2.3 Integrační testování

V této fázi přistupuje k vývoji a následnému provádění testů testovací tým. Cílem těchto testů je ověřit bezporuchovost komunikace jednotlivých komponent, ze kterých se testovaná aplikace skládá. Proto jsou testy také nazývány „testy vnitřní integrace“. [22]

Na komponenty je nyní pohlíženo jako na celky, které již úspěšně prošly fází testování jednotek.

Snahou testerů může být verifikovat integraci i mezi komponentou a operačním systémem, hardwarem či rozhraním různých systémů.

Testy mohou být implementovány jako manuální, ale i automatizované.

Častým řešením v rámci vývoje menších projektů je vynechání této fáze testování, aniž by to mělo dopad na bezporuchovost aplikace. To platí pouze pod podmínkou, že musí být důkladně exekvovány testy z ostatních fází. Při uvážení, že lze snížit náklady včasným odhalením chyby, je tedy vhodné integrační testy z procesu testování nevynechat [22].

Představiteli této fáze jsou např. regresní testy, inkrementální testy integrace či zahořovací testy (Smoke tests).

Inkrementální test integrace se provádí po přidání dalšího modulu aplikace k již otestovaným částem [15].

Zahořovací test slouží ke kontrole, zda je aplikace připravena do další fáze systémových testů a že nedojde např. k pádu software po jeho spuštění. „Smoke testy se zaměřují pouze na hlavní funkce programu, které nebývají příliš často upravovány. Často se také kombinují s kategorií testů splněním. Jelikož je rozsah smoke testů menší než tomu bývá u ostatních kategorií a pokrývají pouze hlavní funkce, jsou velmi často automatizovány“. [22]

4.2.4 Systémové testování

Přístup slučující systémové testování s integrační fází (SIT neboli System integration tests) je také jedním z důvodů, proč lze samostatné integrační testy přeskočit.

Nyní je již aplikace verifikována v podobě funkčního celku a návrhy testů jsou sestaveny z pohledu zákazníka. Děje se tak vzhledem k tomu, že systémové testy jsou posledními provedenými testy před předáním aplikace zadavateli. Spouštěné testy jsou z kategorií jak funkčních tak nefunkčních [22].

Testování v rámci této fáze probíhá v několika kolech, kdy se opakuje postup „testování – objevení chyby – oprava“ [36]. Díky tomu, že systémové testy zastávají funkci výstupní kontroly, je nemožné je z procesu testování vynechat. Takové jednání by mělo za následek znehodnocení předchozích testovacích fází, které by ztratily význam, a také by nemohla být garantována požadovaná funkčnost a kvalita produktu.

Je tedy vhodné na tyto testy klást velký důraz, neboť nástin jejich budoucí podoby je obsažen již ve specifikaci SRS.

Představiteli této fáze jsou např. testy instalace (Installation tests), testy obnovitelnosti (Recovery tests), bezpečnosti (Security tests), zátěžové (Performance) testy, pod něž spadají zátěžové (Load) testy, Stress testy a kapacitní (Capacity) testy.

Test instalace prověřuje, jak probíhá instalace a odinstalování aplikace na dané platformě [22].

Účelem testu obnovitelnosti je „otestovat, jak rychle a zda vůbec se produkt vzpamatuje po pádu systému, HW chybě, výpadku proudu atd. Tento požadavek by měl být uveden v SRS dokumentu v sekci nefunkčních požadavků“. [15]

Test bezpečnosti odhaluje, jak a zda vůbec je systém chráněn před neautorizovaným přístupem, systém ukládání hesla, implementaci integritní ochrany. Dále „slouží k nalezení nežádoucího kódu, bezpečnostních chyb, zranitelností“. [15]

Při testu výkonnosti „systém odolává velkému počtu různých požadavků a sleduje se, jaká je jeho odezva, resp. jak je ovlivněn výkon aplikace, např. jak rychle je aplikace schopna na jednotlivé typy požadavků odpovídat. Tím lze vysledovat, které části systému je třeba věnovat větší pozornost a provést v ní příslušné optimalizace (Může to

být refaktorizace kódu, ale stejně tak i prosté vytvoření indexů nad tabulkou databáze)“. [15]

Při automatizaci testovací nástroj pro zátěžové testy do skriptu obvykle zaznamenává komunikaci mezi klientskou stanicí a serverem na úrovni příslušného komunikačního protokolu jako např. HTTP(S), SOAP, Corba, ODBC a podobně (dáno použitou technologií při vývoji aplikace). Do zátěžového skriptu je tedy nahráván "paket", který při práci uživatele opouští klientské PC a je směrován na aplikační server, či přímo databázi (v případě dvouvrstvých aplikací a těžkého klienta) [20].

Dalším testem, jehož cílem je verifikovat chování testovaného produktu v reálném prostředí, je Load test. Test spočívá v přihlášení vysokého počtu uživatelů k dané aplikaci, proto je vhodné celé testování zautomatizovat. Důležitým aspektem jsou použitá data pro tvorbu testovacích uživatelských účtů. Data by měla být pro každého uživatele jedinečná, aby se rozlišilo, zda při nadměrném zatížení aplikace stále zobrazuje správná data a zda jsou zobrazena pouze tomu uživateli, který k nim má mít přístup [12]. V rámci Load testu lze také uskutečnit vytrvalostní test (Endurance test), který je zaměřen na určování metrik týkajících se zátěže aplikace. Může se jednat o vypočtení času, po jehož uplynutí dojde k pádu programu a jiné [31].

Cílem Stress testu je „ověřit, zda při velké zátěži, která může být vygenerována automaticky např. provedením velkého počtu složitých dotazů, a nedostatku zdrojů, nedojde k chybě, která by se za normálního provozu neobjevila“. [15] Testovaná aplikace se může potýkat s omezeným množstvím přidělené paměti, nedostatečným prostorem na disku, nebo chybou serveru. Často identifikovanými druhy chyb u webových aplikací jsou problémy se synchronizací a prosakováním paměti (memory leaks). Tyto testy také napomáhají určit podmínky, za jakých dojde k pádu aplikace, identifikovat jakým způsobem k pádu dojde a zvýraznit ukazatele, které mohou před pádem varovat.

Kapacitní testy určují přesný maximální počet uživatelů či jimi provedených transakcí, které aplikace zvládne obsloužit, aniž by porušila požadavky na ni kladené ve specifikaci. Lze je také použít při plánování budoucího růstu aplikace, ať už se jedná o zvýšení objemu dat či rozšíření uživatelské základny [31].

4.2.5 Akceptační testování

Akceptační fázi (UAT neboli User acceptance tests) provádí testovací tým na straně zadavatele. Navazuje na úspěšný průchod aplikace všemi předchozími etapami a následné předání produktu zákazníkovi.

Podoba testů vychází z připravených scénářů, které byly sestaveny společnými silami zadavatele i dodavatele. Testování však již probíhá v testovacím prostředí u zadavatele. Objevené chyby jsou hlášeny zpět vývojovému týmu na straně dodavatele, přičemž je vhodné si stanovit formalizovaný postup reportingu chyb i oprav mezi oběma stranami.

Oprava chyb nalezených akceptačními testy by měla probíhat co nejrychleji, protože má vliv na konečné nasazení produktu u zákazníka, a to jak z hlediska časového tak kvalitativního. V konečném důsledku tato fáze velmi silně ovlivňuje úspěch či neúspěch celého projektu [22].

Mezi akceptační testy lze zařadit Alfa test, Beta test či Long term test.

Alfa test stojí svým provedením na rozhraní systémových a akceptačních testů, protože je spouštěn v prostředí vývoje, ale je vykonáván někým jiným než vývojovým týmem.

Beta test je již exekvován v prostředí u zákazníka a hlášení chyb probíhá smluvenou formalizovanou cestou.

Long term test odhaluje chyby, které se mohou vyskytnout až po delším užívání vyvinuté aplikace [15].

4.3 Kategorie testů

Testy, které se vyskytují v jednotlivých etapách vývoje software, lze typově dále dělit na základě různých dalších kritérií, než je jejich pouhá aplikace v čase. Důvodem tohoto dělení může být také fakt, že se použití těchto testů prolíná více fázemi vývoje aplikace či dochází k jejich opakování.

4.3.1 Funkční a nefunkční (non-funkční)

Typická ukázka testů, které se vyskytují ve více časových etapách, respektive v integračních testech, v SIT a v akceptační fázi.

Funkční testy ověřují splnění všech úkolů, pro které je aplikace vyvíjena. Testovány jsou všechny funkce aplikace, jež jsou obsaženy ve specifikaci SRS a ověřována je i skutečnost, zda byly implementovány v souladu se specifikací [27].

Nefunkční testy naproti tomu ověřují ostatní vlastnosti aplikace, které přímo nesouvisí s jejími funkcemi, ale mají dopad na použitelnost aplikace [22].

4.3.2 Testy splněním a selháním

Z pohledu testovacích dat dochází k dělení na testy splněním (test-to-pass) a selháním (test-to-fail), jinak také pozitivní a negativní. Tato kategorie testů se uplatňuje zejména v systémové fázi testování [22][36].

„U testů splněním jako vstupní hodnoty v aplikaci využíváme jen množinu dat, kterou musí aplikace vždy akceptovat“. [22] Následný výstup aplikace se pak musí shodovat s požadavky uvedenými ve specifikaci SRS.

Testy selháním naproti tomu vyžadují zadávání dat mimo akceptovatelnou množinu, kdy je cílem testera způsobit pád aplikace. Nutné je zde kontrolovat, zda aplikace nevrací hodnoty mimo množinu očekávaných výstupních dat, což by mělo probíhat opět v souladu s požadavky zákazníka uvedenými ve specifikaci SRS. Součástí výstupu jsou také chybová hlášení, jejichž kontrola by neměla být opomíjena a která by měla mít relevantní podobu a dostatečnou informační hodnotu pro uživatele.

Vhodné použití této kategorie testů je nejprve spouštět testy splněním (ve snaze zaručit, že aplikace splňuje požadavky na práci se standardními daty), a poté testy selháním [22].

4.3.3 Progresní a regresní testy

Vývoj software není rigidním procesem, ale reaguje dynamicky např. na změny požadavků po konzultaci se zadavatelem. Proto lze rozlišit další typy testů, jako jsou progresní a regresní testy.

„Progresní testy využíváme při kontrole nových funkcí nebo vlastností aplikace. K jejich správnému provedení je nutná dokumentace, která přesně popisuje nově implementované oblasti. Používáme je ve všech etapách testování [22].

„Regresní testy se využívají při opětovném testování funkcí a vlastností aplikace. Jejich smyslem je ověření, že provedené změny či implementace nových vlastností v aplikaci nemělo žádný vliv na stávající funkce a vlastnosti“.[22] Testují tedy zejména části aplikace, jež nebyly ve zdrojovém kódu změněny. Jelikož se tak děje např. po opravení chyb či vydání nové verze aplikace, je velmi prozíravé tyto testy automatizovat [22].

4.3.4 Statické a dynamické testy

Dělení v rámci této kategorie testů je závislé na nutnosti běhu dané aplikace při spuštění testů.

„Statické testy nevyžadují běh software. Využívají se tedy zejména v raných fázích životního cyklu software, kdy ještě není vytvořen funkční prototyp aplikace. Lze je použít ještě před začátkem psaní kódu na kontrolu specifikace požadavků a analýzu zdrojového kódu“. [22]

Naproti tomu dynamické testy vyžadují spustitelný prototyp aplikace. „Používají se v pozdějších fázích vývoje a jsou zaměřeny na provoz software“. [22]

4.3.5 Testování černé, šedé a bílé skříňky

Na základě znalostí testera o vnitřní struktuře dané aplikace lze testy rozřazovat do testování černé skříňky (black box testing), šedé skříňky (grey box) a bílé skříňky (white box).

Testování černé skříňky je vhodné ve chvíli, kdy jsou přesně definované vstupy a rozsahy možných hodnot. Tester provádí za použití testovacích scénářů, které mu byly poskytnuty. U některých typů testů si je vytváří sám i přesto, že nemá k dispozici žádnou dokumentaci ani zdrojové či binární kódy aplikace. Testy mohou být exekvovány jak manuálně, tak probíhat automaticky a jelikož stačí mít k dispozici daný program, či znát jeho umístění, může být aplikace testována i vzdáleně [10].

„Black box testu lze využít např. na zjištění problémů typu odepření služby (DoS) nebo aktuálních zranitelností již běžícího systému nebo aplikace“. [10]

Tento druh testů skýtá výhody v podobě malé náročnosti na lidské zdroje, kdy tester nemusí mít znalost programovacího jazyka, pokud test není automatizován. Dále se vyznačuje rychlostí, neboť lze v krátkém čase otestovat i rozsáhlejší systémy. Výhodou

pro zákazníka je transparentnost testu, protože jednotlivé scénáře jsou pro něj srozumitelné. Samotné sestavení scénářů je možné již ve chvíli, kdy je vyhotovena specifikace požadavků SRS. Podoba testů není závislá na použitém programovacím jazyku aplikace, operačním systému či hardware, tedy v případě změny zůstává testovací postup nezměněn. Vyskytuje se zde také menší riziko úniku know-how, které aplikace obsahuje, neboť – jak již bylo zmíněno – tester se nedostane do styku se zdrojovým kódem aplikace.

Nevýhodou tohoto přístupu je riziko nízké kvality kódu, protože zobrazení očekávaného výstupu po zadaném vstupu neimplikuje efektivnost aplikace a další podstatnou nevýhodou spojenou s programovým kódem je možná přítomnost kódu nežádoucího. U aplikace tak po provedení testů není zaručeno, že kromě očekávaného chování neprovádí i další akce, které mohou mít škodlivý charakter [10].

Testování šedé skříňky se provádí zvenčí podobně jako u černé skříňky, s tím rozdílem, že tester již zná některé vnitřní struktury dané aplikace. Jeho znalost však není tak důkladná jako v případě bílé skříňky[11].

„Grey box testu se obzvlášť používá, když se provádí integrační testování dvou modulů od dvou různých dodavatelů a je potřeba otestovat interfacy. Další velice častý případ užití je v případě testování vícevrstvé aplikace, kdy máme kontrolu nad vstupem, výstupem a máme přímý přístup do databáze. Můžeme tak porovnávat všechny tři hodnoty: vstup, hodnotu v databázi a výstup“. [11]

Tyto testy jsou vhodné i pro analýzu HTML formulářů, skriptů a práci s cookie soubory a lze takto provádět i penetrační testování webových aplikací.

Testovat šedou skříňku se vyplatí, pokud testerovi nejsou poskytnuty binární či zdrojové kódy záměrně, ale na druhou stranu je nežádoucí, aby ztrácel čas zjišťováním informací o architektuře aplikace či topologii sítě. Proto je mu tento typ informací k dispozici.

Výhodou je záměrně neintrusivní přístup, kdy je testování založeno pouze na znalosti funkční specifikace, rozhraní a architektuře aplikace. To nahrává skutečnosti, že testy budou sepsány vhodným způsobem jak pro manipulaci s daty tak pro užité komunikační protokoly.

Mezi nevýhodami lze najít podobný problém jako u testování černé skříňky, a sice nemožnost verifikovat přítomnost nevyžádaného kódu. Omezenost přístupu ke zdrojovým kódům připouští možnost, že datové toky nebudou pomocí testů plně pokryty [11].

Testování bílé skříňky již testerovi dovoluje znát celou vnitřní strukturu zkoumané aplikace. Je obeznámen s programovými strukturami, vlastní implementací systému včetně dostupné dokumentace, proto je testování někdy označováno jako „audit zdrojového kódu“ (code-review exercise). Tester buď má k dispozici přímo zdrojový kód, či ho získá pomocí dekompilace z binárního kódu.

Tento druh testů lze vhodně využít pro webové služby zejména z počátku vývoje aplikace, neboť analýza zdrojového kódu usnadňuje eliminaci chyb ještě před kompilací kódu [17].

“White box test můžeme využít ke zjištění, jak se systém chrání před neautorizovaným přístupem, jak je řízen přístup k jednotlivým částem aplikace a k datům, jak je implementována integritní ochrana nebo jak jsou ukládána hesla. White box test můžeme také použít k nalezení nežádoucího kódu, bezpečnostních chyb a zranitelností”. [17]

Odhalování kódu, který nemusel být do aplikace zanesen se zlým úmyslem, ale opomenut (ladění aplikace programátorem) je také největším přínosem tohoto přístupu v porovnání s testování černé a šedé skříňky. Pomocí těchto dvou zmíněných přístupů lze nežádoucí kód najít pouze v ojedinělých případech, a to když se kód náhodou prozradí sám, např. zabráním výrazně více místa na disku či v paměti, způsobením nárůstu I/O operací, zvýšením zátěže procesoru.

Naproti popsaným výhodám klade testování bílé skříňky vysoké požadavky na znalosti testerů, co se týče cílové aplikace i dalších nástrojů a programovacích jazyků. Dále lze zaznamenat vyšší nároky na použité vybavení software, např. analyzátory zdrojového kódu, debuggery [17]. V neposlední řadě též hrozí riziko odhalení know how nebo dokonce zcizení části kódu při testování externím pracovníkem či firmou [18].

Kombinací přístupů lze docílit např. nalezení zranitelností aplikace pomocí white box testu a ověření, zda lze na aplikaci touto cestou provést útok, za použití black box testu [11].

Testy se dají dále kombinovat se statickým a dynamickým přístupem. Je možno docílit 4 známých kombinací, kterými jsou testování:

- černé skříňky staticky (revize softwarové specifikace),
- černé skříňky dynamicky (test chování aplikace, prováděno v roli koncového uživatele, tzv. výzkumné testování),
- bílé skříňky staticky (zkoumání návrhu a programového kódu, tzv. strukturální analýza) a
- bílé skříňky dynamicky (úplná analýza dat a programového kódu z hlediska jednotek, modulů i celého systému, tzv. strukturální testování) [36].

4.3.6 Testy konfigurace a kompatibility

Tyto testy jsou prováděny kvůli existenci různých hardwarových i softwarových platforem a také širokým možnostem jejich nastavení. Je nutné otestovat danou aplikaci při různých konfiguracích. Pro kontrolu, zda testovaný program korektně spolupracuje s jiným softwarem, pak slouží testy kompatibility.

Důležitá hlediska, která by měla být zahrnuta v testech konfigurace, jsou zmíněné hardwarové platformy, webové prohlížeče a jejich verze, zásuvné moduly prohlížeče (tzv. plug-in), volby prohlížeče, rozlišení obrazovky, velikost textu a také rychlost připojení, potažmo odezva webových stránek [36].

Z pohledu kompatibility lze testovat zpětnou či dopřednou kompatibilitu, dodržování standardů (např. standardy definované organizací The World Wide Web Consortium) nebo použitelnost aplikace při sdílení dat (uložení/načtení souboru, export/import dat, a vyjímání, kopírování či vkládání dat) [48].

Kritéria, podle kterých jsou vytvářeny scénáře testů kompatibility, jsou stáří programů (a jejich verzí), popularita softwaru a četnost jeho užití či výrobce [36].

4.3.7 Testy lokalizace

Pokud je výsledná aplikace distribuována i v jiných zemích, než je její země původu, přistupuje se často k překladům celého programu do cizích jazyků popř. k přizpůsobení odlišnostem dané kultury. Samotný proces překladu je náročný jak po stránce jazykové, tak po stránce programové. Z hlediska vývoje se totiž jedná zejména o vyřazení

všech textových zpráv a popisků do zvláštních tzv. resource souborů, mimo kód aplikace, aby mohly být snáze přeloženy do cizího jazyka. Další důležité oblasti mohou být texty na tlačítkách, kde může po překladu dojít k jejich neúměrnému prodloužení či zkrácení, přístup k funkcím programu přes klávesové zkratky, tzv. hot keys, nebo abecední řazení. Proto by o provedení resp. neprovedení lokalizace mělo být rozhodnuto již v rámci návrhu aplikace či v jejím raném stádiu vývoje.

Testování takového programu klade vysoké jazykové nároky na testovací tým, proto není výjimkou, že testování lokalizace je v praxi prováděno externí specializovanou firmou [36].

4.3.8 Testy použitelnosti

Jak napovídá název, testy jsou zaměřeny na možnost funkční a efektivní komunikace softwarového produktu s člověkem v reálném životě. Jednou z možností, jak je taková interakce realizována je pomocí uživatelského rozhraní. V případě webových aplikací se jedná přímo o grafické uživatelské rozhraní, neboli GUI (Graphical User Interface).

Nejčastějšími chybami, které se mohou v rámci návrhu a vývoje rozhraní vyskytnout, jsou technologická omezení, nedostatek času, nevhodné provedení lokalizace či jen fakt, že rozhraní vytváří programátor, který není odborníkem na ergonomii dané aplikace.

Mezi cíle, na které je nutné se při tomto testování zaměřit, patří dodržování standardů nebo zásad, které se stávají rozšířením specifikace daného produktu (např. zachování vzhledu tzv. „look and feel“ operačního systému, na kterém aplikace bude spouštěna). Ovládání by mělo být intuitivní, tedy uživatel by měl být dostatečně informován při používání aplikace (např. obrázková tlačítka doplněna popisky s funkcemi či přítomnost nápovědy k aplikaci), ale nemělo by dojít k přesycení uživatele informacemi. Uživatelské rozhraní by mělo být konzistentní, tedy operace, které již uživatel zná a užívá je i v jiných aplikacích, by se měly provádět podobně (např. řazení odkazů v hlavním menu). Dalším důležitým cílem je flexibilita, tedy přizpůsobení se uživateli. Aplikace by měla být pro uživatele čitelná a jednotlivé kroky z pohledu uživatele logické i přes její vysokou složitost. Rozhraní by mělo být pohodlné, ulehčovat uživateli práci, nevtírat se a přiměřeně informovat uživatele o činnosti aplikace (např. pomocí stavových řádků). Uživatelské rozhraní by mělo splňovat kritérium správnosti, tedy s aplikací se

děje opravdu to, co je uživateli zobrazeno (spadá sem i např. použití WYSIWYG editoru). Dále je kladen důraz na užitečnost uživatelského rozhraní, při jejímž testování mohou být odhaleny v programu zbytečné funkce [36].

Volba vhodných testerů je v tomto případě zásadní. Není nejlepším řešením sáhnout do vlastních řad a přenechat testování použitelnosti pouze uživatelům ve firmě, kteří danou aplikaci znají a přesně vědí, jaký způsobem ji ovládat. Zahrnuti by proto měli být i uživatelé, kteří s aplikací ještě nepřišli do styku. Testeři mohou být vybíráni na základě různých kritérií, kterými jsou např. věk, vzdělání či zkušenosti. Úlohy, které je nutno otestovat, by neměly svým zadáním nijak navádět k řešení. Proto i zadání úloh by mělo být co nejjednodušší.

Samotný průběh testu spočívá v zaznamenávání postupu, jakým se tester snaží dostat ke splnění úlohy, a zároveň vlastní hodnocení testera, jak dobře se mu s aplikací pracuje, co očekává, či co mu nevyhovuje [16].

4.3.9 Penetrační testy

Tento druh testů (někdy označovaných jako ethical hacking či pentest) slouží jako metoda hodnocení bezpečnosti daného počítačového systému nebo sítě na základě simulace škodlivých útoků bez autorizovaného přístupu i od útočníků s určitým stupněm oprávnění [13][39].

Penetrační testy jsou implementovány metodou černé skříňky. Může jim však předcházet analýza zranitelností (vulnerability analysis), která v sobě zahrnuje manuální revizi programového kódu. Tento přístup klade vysoké nároky na čas i na znalost testera, proto je možné oba přístupy doplnit automatickými testy [39].

Odhalené zranitelnosti či slabá místa v softwaru mohou nepříznivě ovlivnit důvěrnost a integritu dat a také přístupnost webové aplikace. Na základě zjištění autorů v [39] je za nejčastější slabé místo v programu považována nedostatečná validace vstupů. Proto při následných testech černé skříňky je možné využít známých škodlivých postupů, jako je vsunutí škodlivého kódu ve formě SQL dotazu (SQL injection), či ve formě skriptu (Cross-site scripting, XSS).

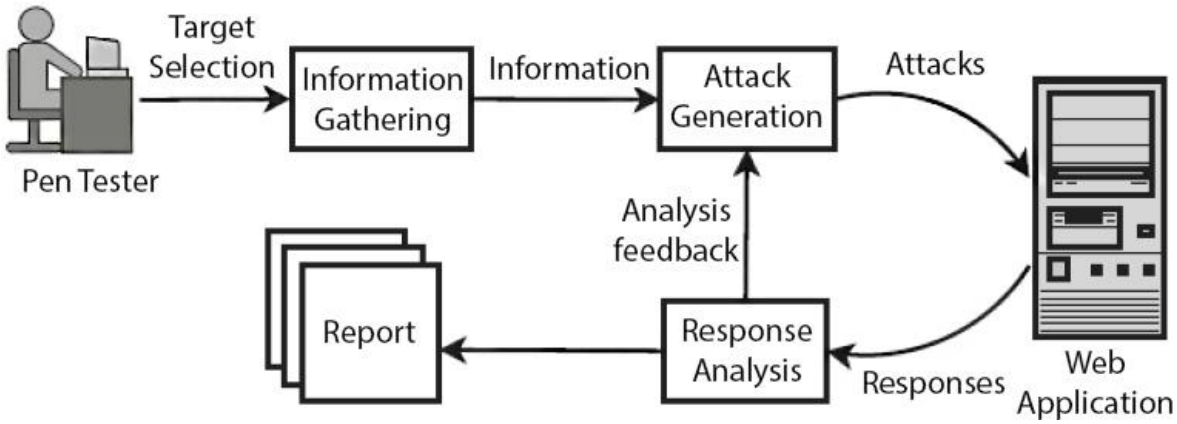
Pro identifikaci bezpečnostních slabin je autory v [39] využíván tzv. Tainted Mode Model (TMM) přístupu k datům, který staví na následujících předpokladech:

1. Všechna data přijata od klienta pomocí HTTP požadavků jsou nedůvěryhodná.
2. Všechna data, která jsou uložena lokálně z hlediska aplikace, jsou důvěryhodná.
3. Jakákoliv nedůvěryhodná data se mohou stát důvěryhodnými v případě, že projdou procesem sanitace.

Lze říci, že bezpečnostní slabina je jakékoli porušení pravidel, která z těchto předpokladů vyplývají. „Nedůvěryhodná data by neměla být použita v konstrukcích http odpovědí, což předchází útokům cross-site skriptů. Škodlivá data by neměla být ukládána v lokálním úložišti, což předchází možnosti použití takových dat v http odpovědích. Nedůvěryhodná data by neměla být použita v systémových voláních a při tvorbě příkazů používaných pro přístup k externím službám, jako přístup do databáze, mailu apod. Škodlivá data, která projdou ze vstupu, by neměla být použita při konstrukci příkazů, což zabraňuje skriptovacím útokům“. [39]

Proces sanitace může spočívat v odstranění meta znaků, které mají speciální význam. Jednou z metod ošetření potencionálně nebezpečných znaků je tzv. escaping [21]. Další metodou může být zkrácení vstupního řetězce, tzv. truncation.

Penetrační test se skládá z několika fází. První je identifikace zranitelnosti umožňující testerovi určit slabiny systému, na který útočí. Následuje zahájení útoku, jehož cílem je získání plného přístupu k danému systému. Zde může být proveden test odepření služby, tzv. DOS útok (Denial of Service), aby se ověřila stabilita systému. V případě zjištění systémové nestability lze její opravou zamezit velkým finančním ztrátám na straně dodavatele i zákazníka. Po ukončení testování je nutné podat podrobnou zprávu s výsledky, které mohou popisovat další odhalené slabiny systému a které je nutno opravit [39].



Obrázek 8: Průběh penetračního testu[39]

4.4 Nástroje využitelné pro automatizaci testování

Z teoretického hlediska lze automatizované testovací nástroje dělit na základě jejich přístupu k operačnímu prostředí systému na tzv. **neinvazivní** a **invazivní** nástroje.

„Jestliže určitý nástroj slouží pouze k monitorování a zkoumání softwaru, aniž by jej jakkoli modifikoval, považujeme jej za neinvazivní. Pokud ale nástroj jakkoli kód programu modifikuje, nebo pokud libovolným způsobem manipuluje s jeho operačním prostředím, jedná se o invazivní nástroj. Různé nástroje vykazují přitom různou míru invazivnosti, přičemž testeři se obvykle snaží využívat co nejvíce neinvazivní, aby tak snížili možnost ovlivnění výsledku testů samotným nástrojem“. [36]

K mnoha dostupným komerčním nástrojům na trhu existují freeware alternativy. Obecně platí, že freeware systémy mají prostor především při tvorbě jednoduchých aplikací, složitější komerční nástroje mají smysl ve velkých firmách, kde probíhá větší množství změnových řízení na velkých integračních systémech.

Bugtracking (neboli hlášení defektů) je nejrozšířenějším podprocesem procesu testování, který je řešen pomocí podpůrných nástrojů. Výběr mezi nástroji je tedy široký a každý z nástrojů má svá plus a mínus [23].

5 Firma Polarion Software s. r. o.

Společnost Polarion Software s.r.o. fungující již 10 let (od roku 2004) se neustále snaží naplňovat svou strategii, a to přispívat k rozvoji firem z hlediska správy, údržby a samotného vývoje software za pomoci unifikovaného řešení pro řízení životního cyklu aplikací (Application Lifecycle Management).

Nabízené řešení, webová aplikace Polarion ALM, které firma vyvíjí, pokrývá jak vývoj software (správa požadavků, návrh softwarové architektury, programování a testování) tak i údržbu softwaru, řízení změn, projektové řízení a správu dodávek systému.

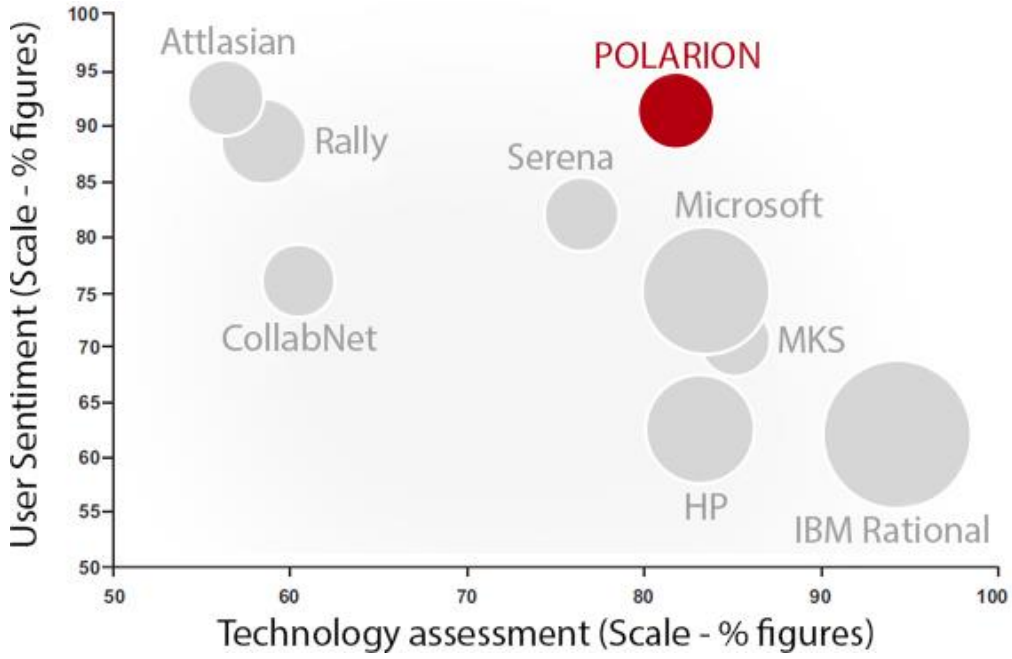
5.1 Firemní strategie

Firemní strategie je silně ovlivněna strategií modrého oceánu[4], tedy snahou o vytváření si nových bezkonkurenčních trhů. Jelikož modrý oceán lze vytvořit i rozšířením rudého, proto lze v tomto odvětví nalézt částečné konkurenty.

5.1.1 Konkurenční společnosti a produkty

Na základě reportu z roku 2012 provedeného společností Ovum lze mezi hlavní konkurenty zařadit Attlasian, IBM Rational Doors a HP Quality Center [3]. Následující graf ukazuje úspěšnost produktu na základě míry spokojenosti zákazníka (svislá osa grafu) a technologického posudku (vodorovná osa). Velikost jednotlivých bublin znázorňuje tržní dopad (market impact) jednotlivých produktů.

[4] Smyslem strategie modrého oceánu je přitáhnout zcela novou skupinu zákazníků a přestat ve vzájemném konkurenčním boji. Modré oceány jsou charakteristické dosud nevyužitým tržním prostorem, vytvářením poptávky a příležitostmi k vysoce ziskovému růstu. Mohou existovat opravdu mimo hranice existujících odvětví, většina z nich je vytvářena uvnitř rudých oceánů tím, že se hranice existujících odvětví rozšiřují.
Základ strategie modrého oceánu představuje tzv. hodnotová inovace spočívající ve snaze vyřadit konkurenty ze hry tím, že nakupujícím i své firmě poskytne skokový přírůstek hodnoty, a otevře si tak svrchovaný a nedotknutelný tržní prostor.[1]



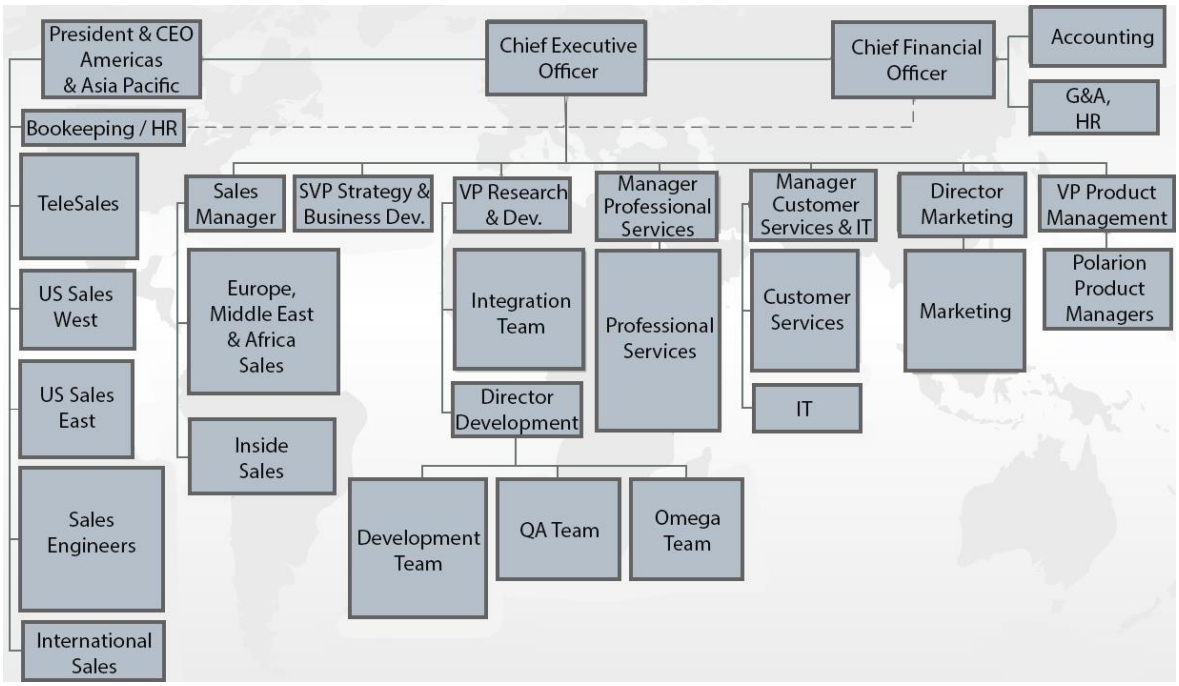
Obrázek 9: Porovnání konkurenčních produktů na základě reportu Software Lifecycle Management 2011/2012[3]

5.2 Organizační struktura

V současnosti má firma z geografického i organizačního pohledu dvě hlavní vedoucí střediska, jedno pro americkou část firmy, konkrétně v Kalifornii, a druhé pro část evropskou v německém Stuttgartu.

Evropské pobočky jsou soustředěny zejména na vývoj samotného produktu, nachází se zde oddělení R&D, IT, zákaznická podpora (Support), Marketing, zatímco americká oddělení se věnují produktovému managementu, strategickým integracím, a jsou tedy centrem B2B operací.

Struktura organizace je funkční, přičemž zaměstnanci s podobnými úkoly, schopnostmi či aktivitami jsou zařazeni do jedné skupiny, což je znázorněno v následujícím obrázku [49]. Detailní schéma struktury včetně jmen zaměstnanců je v Příloze č. 1.



Obrázek 10: Organizační struktura Polarion Software 2015

Pobočka v České republice, konkrétně v Praze, zastřešuje celý R&D (neboli Research and Development) a Customer Services&IT. Tedy v Praze operuje celkem pět hlavních týmů:

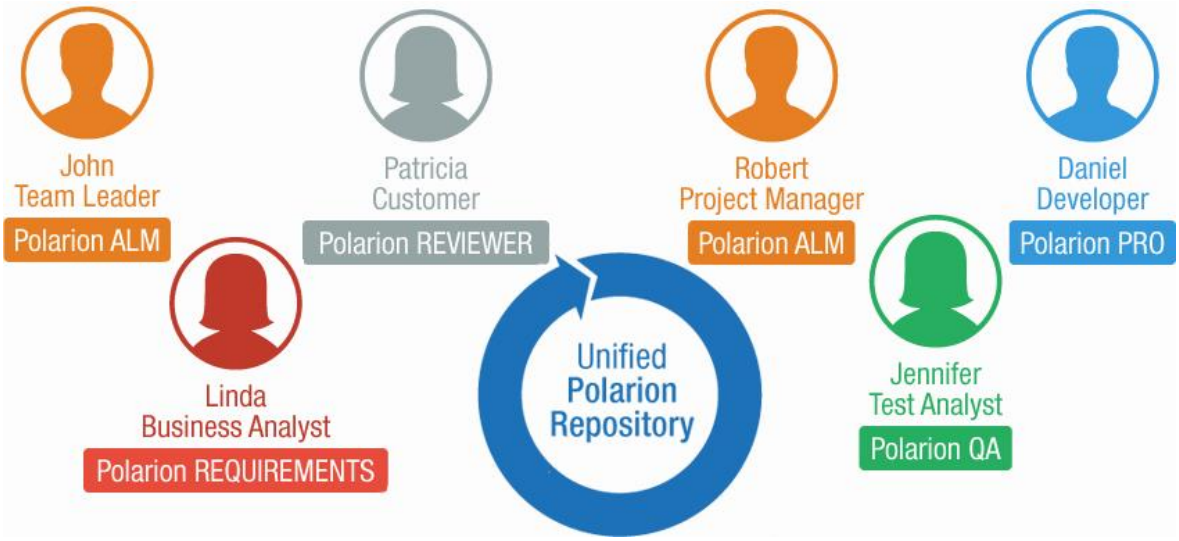
- Dev Team – tým vývojářů, projektových manažerů a Scrum Master
- QA – tým testerů,
- Omega Team – tým vývojářů s vlastním QA pracovníkem,
- Customer Services – tým školených pracovníků zákaznické podpory,
- IT – tým zajišťující veškerý interní SW i HW,

z čehož se dále sdružují tři vývojové týmy – vždy v kombinaci projektový manažer, tři vývojáři a jeden QA pracovník. Tyto tři týmy zajišťují vývoj a řízení kvality na nových i stávajících vlastnostech produktu Polarion ALM, zatímco Omega Team je soustředěn na vývoj tzv. záplat (neboli patches) pro starší verze produktu a zároveň na kompatibilitu Polarionu jako aplikace s dalšími konkurenčními produkty, které trh nabízí. V případě, že nový zákazník používal jiný produkt pro řízení požadavků, plánování, testování či dalších aktivit v rámci vývoje, firma Polarion (konkrétně právě Team Omega) mu nabízí vývoj programové extenze pro usnadnění převodu všech jeho dat do nového systému Polarion.

V současnosti (březen 2015) je odštěpen nový automatizační tým zahrnující pouze projektového manažera a QA pracovníka, jež se soustředí na automatizaci stávajících manuálních a vývoj nových testovacích scénářů.

5.3 Nástroj Polarion ALM

Polarion ALM je komplexní řešení, které firma nabízí společně s různými typy licencí. Po instalaci Polarion ALM se na základě licence zpřístupní uživateli příslušné součásti programu. V současnosti mezi nabízené licence patří ALM (kompletní řešení pokrývající všechny oblasti), REQUIREMENTS (obsahuje součásti týkající se správy požadavků), QA (zaměřeno na oblast test managementu) a dále licence PRO a REVIEWER [29].



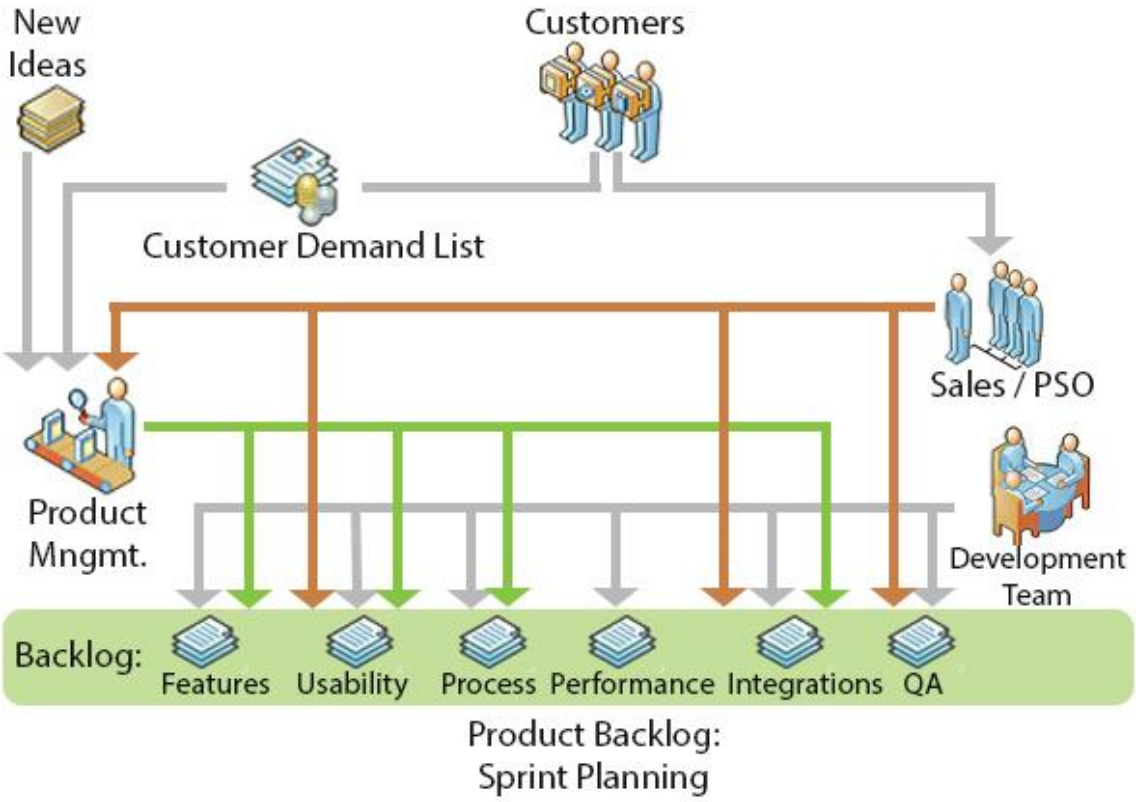
Obrázek 11: Přehled nabízených licencí k produktům Polarion[29]

Díky zapojení Product Lifecycle Management (PLM) a Application Lifecycle Management (ALM) procesů je aplikace úspěšně používaná i v odvětvích jako jsou strojírenství, automobilový a farmaceutický průmysl, vývoj hardware, elektronický a softwarový vývoj.

Obrázek 12: Oblasti správy, které jsou pokryty Polarion ALM řešením[2]

Polarion ALM se snaží být cross-platformní podnikovou aplikací, kterou je jednoduché nasadit a přizpůsobit potřebám zákazníka, dovolující efektivnější správu požadavků, řízení kvality a správu testování. Stejně jako jednotný ALM pokrývající široké spektrum průmyslových odvětví – tzn. letectví, automobily, lékařské přístroje a systémové inženýrství. Firemní jednotné řešení může být zavedeno ve zlomku času, který vyžaduje nasazení konkurenčních řešení. Polarion dovoluje svým zákazníkům neutrácet čas a finance za drahá řešení či zapojování konzultantů do procesu integrace s dalšími produkty.

Firma se snaží rozšiřovat své produktové portfolio na základě interních požadavků, pocházejících z oddělení v rámci celé firmy a externích pocházejících od zákazníků. Kromě toho usiluje o využití osvědčených a spolehlivých komponent, jako Subversion, Apache, Maven, Lucene a Wiki, a snížit tak náklady na vývoj[2].



Obrázek 13: Vstupy do produktového backlogu - proces zadávání požadavků na rozšíření produktového portfolia. Upraveno dle [19].

5.3.1 Produktová strategie

Aplikace Polarion ALM byla vyvinuta na těchto strategických principech:

- „Kvalita, škálovatelnost a výkon,
- intuitivní rozhraní vyvinuté na základě uživatelských zkušeností,
- správa dat velkého objemu (Big Data management), trasovatelnost (traceability), vyhledávání a podávání zpráv (reporting),
- strategické integrace a rozšíření nejvíce požadované firemními zákazníky,
- investice do partnerských technologií, které jsou podkladem pro produkci cenných vylepšení pro firemní zákazníky,
- integrace mezi PLM dodavateli a produktem Polarion ALM vedoucí k ALM-PLM synergii,
- expanze firmou patentovaných technologií včetně Automated Workflow™, LiveDoc™, LivePlan™ a LiveBranch™". [2]

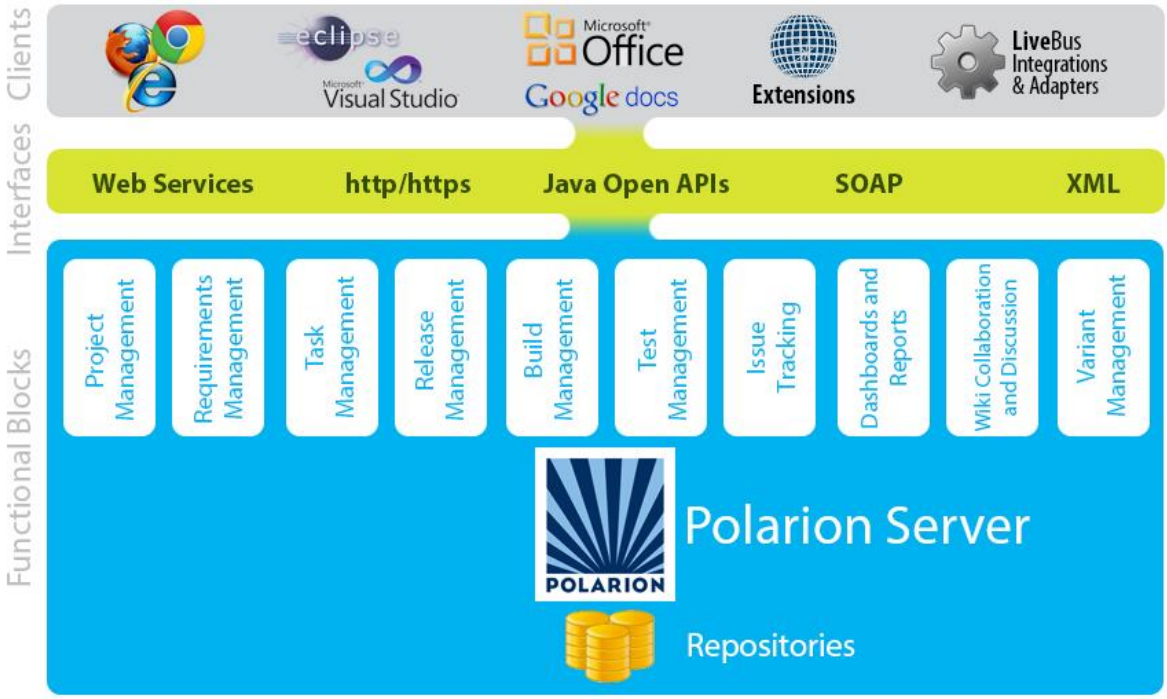
5.3.2 Technická specifikace produktu

Polarion ALM má podobu webové aplikace, nepoužívá žádné ActiveX prvky/komponenty a pro její běh nejsou vyžadovány žádné další instalace. Části Java komponent esenciální pro běh aplikace jsou již zakomponovány v instalačním balíčku.

Jak již bylo řečeno, produkt je vyvíjen s důrazem na nízké požadavky na svoji údržbu. Instalace je rychlá a uživatelsky přívětivá a samotná aplikace nezatěžuje koncový systém ani při odinstalaci.

Podporovaným prohlížečům je věnována následující podkapitola.

Architektura aplikace je třívrstvá. Spodní vrstva je tvořena za použití technologií jako SVN, H2 databáze. Střední vrstva využívá technologie jako Java 2SE, lze zde najít webové služby a Polarion API, které obě otevírají prostor pro integrace a další rozšíření. Nejvyšší vrstva obsahuje GWT, HTML/JS. Toho je využíváno jak k malým vizuálním úpravám tak např. k přepsání nativního chování prohlížeče a jeho defaultní implementace, neboť součásti aplikace nazývané Wiki pages a nyní i nové Rich Pages jsou schopny dynamicky stahovat data a pracovat s nimi (organizovat je skrze dotazy).



Obrázek 14: Architektura aplikace Polarion ALM[19]

Aplikace je vhodná pro instalaci jak na jednom fyzickém či virtuálním stroji tak pro nasazení v clusteru.

5.3.3 Podporované prohlížeče

Firmy vyvíjející webové aplikace se potýkají také s problémy souvisejícími s podporou jednotlivých internetových prohlížečů, případně zpětnou kompatibilitou.

Výhodou je, že výrobci prohlížečů se pokouší o zjednodušení, např. Google Chrome. O podobný přístup se plánovaně snaží i Microsoft s Internet Explorerem [43]. „Aby pomohl rychlejšímu šíření nových verzí, od poloviny ledna 2016 bude na Windows podporovat vždy jen poslední dostupnou verzi IE pro daný operační systém. Ostatní nebudou dostávat bezpečnostní a další záplaty“. [44]

Podporované prohlížeče produktu Polarion ALM jsou Microsoft Internet Explorer (aktuálně ve verzích 9, 10 a 11), Google Chrome (vždy nejnovější verze) a Mozilla Firefox (verze 24 a novější), které jsou dostupné na operačních systémech Windows, Linux a OS X.

5.4 Firemní přístup k vývoji a řízení kvality

Polarion pro vývoj své aplikace používá agilní metodiky - SCRUM a Kanban.

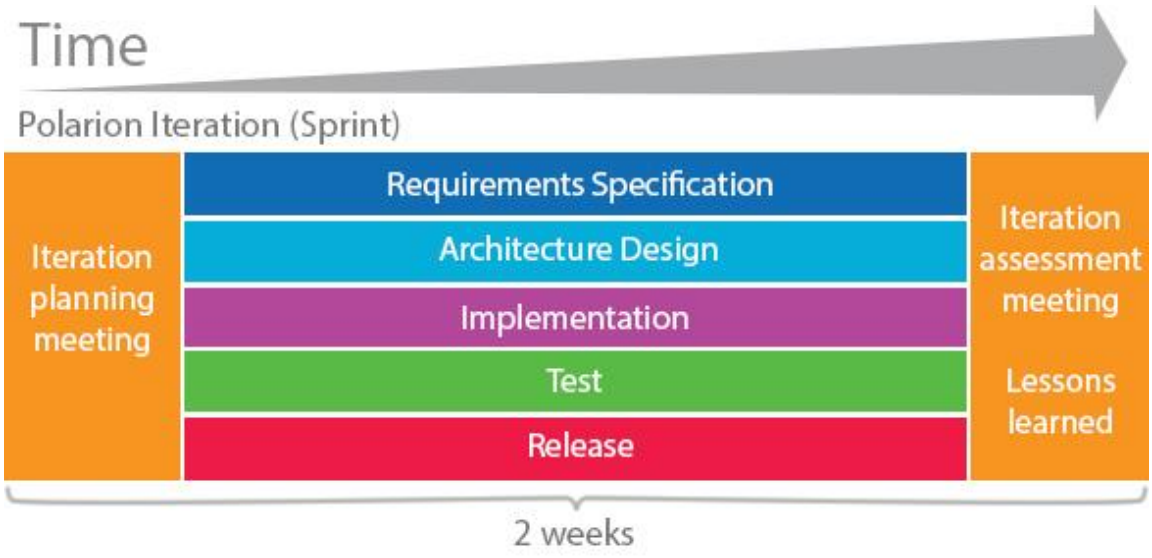
Vydání nové verze produktu (release) probíhá čtyřikrát ročně, vždy po 3 měsících (tzv. milníky neboli milestones). První 3 releasy jsou servisní a čtvrtý release je zároveň vypuštění verze (buildu) obohacené o nové kompletní součásti produktu mezi zákazníky.

Pro plánování prací pro následující rok se využívá hrubý odhad provedených prací a stráveného času, který je však založen na znalostech a zkušenostech experta pro danou část aplikace (SWAG neboli Scientific Wild-Ass Guess).

V rámci jednotlivých milníků se využívá metody T-shirt Size Matrix, která spočívá v distribuci jednotlivých uživatelských příběhů (user story) do skupin dle složitosti, na základě relativního srovnání.

Každý nový uživatelský příběh začíná vymezením v týmu, který se bude podílet na jeho implementaci. Jednotlivé způsoby užití jsou specifikovány s pomocí metodiky vývoje řízeného požadavky na chování (BDD neboli Behavior driven development). V této chvíli je využíváno absolutního odhadu plánovaných prací a stráveného času, který je produktem diskuze mezi jednotlivými členy týmu. Tento přístup umožňuje všem zainteresovaným stranám vyjádřit své obavy a připomínky a ovlivnit tak celkový dediko-

vaný čas. Součástí je samozřejmě i čas věnovaný testování, případným opravám a finálnímu otestování. Z hlediska metodiky SCRUM, kterou si firma přizpůsobila svým potřebám a očekáváním, se toto plánování týká dvoutýdenních sprintů.



Obrázek 15: Sprint [19]

Pro pokrytí všech potřeb pro vývoj nových verzí aplikace používá firma právě Polarion ALM v clusteru, tedy svůj vlastní produkt. Díky tomu dochází k testování všemi pracovníky již během vývoje, tzv. dogfooding.

Výhody, které plynou z použití vlastní aplikace, jsou úzce propojeny i s výhodami agilních metodik. Jedná se například o možnost pozorovat zdraví projektu a velmi rychle odhalit jakékoli nedostatky či rizika spojená s každým uživatelským příběhem či epikou (epic). Samotné uživatelské příběhy nabývají na reálnosti, díky čemuž lépe pokrývají skutečné případy užití koncovým zákazníkem. Pracovníci také mohou rychleji reagovat na změny a specifikovat očekávání průběžně. Zvyšuje se tím informovanost managementu, v jaké části projektu se vývojové týmy právě nacházejí [19].

5.4.1 Dopad firemního přístupu na testování

Tento přístup generuje zvýšený tlak na QA pracovníky v daném týmu, neboť veškeré testování musí být dokončeno v rámci jednoho sprintu (2 týdny). Testování by tedy mělo být cílené a efektivní a mělo by přijít ve chvíli, kdy je uživatelský příběh plně implementován.

V případě nedodržení toho přístupu dochází ke zvýšené potřebě regresního testování, neboť jakákoli následující změna zdrojového kódu může přinést neočekávané komplikace v podobě nově zanesených chyb.

Rozdělení implementované funkcionality do jednotlivých uživatelských příběhů s sebou také nese vyšší důraz na následně prováděné integrační testování, neboť části kódu v jednotlivých příbězích se navzájem doplňují.

Jak již bylo řečeno, SCRUM ovlivňuje též složení jednotlivých vývojových týmů. Každý tým má svého projektového manažera, vývojáře (SDE neboli Software Development Engineer) a své QA pracovníky. V současné době jeden z týmů zaměstnává junior testera, jehož čas je plně dedikován automatizaci stávajících a vývoji nových testů (SDET neboli Software Development Engineer in Test).

5.4.2 Software Development Engineer in Test

Pozice SDET má dominantní zastoupení zejména ve firmách jako Microsoft, Google, Amazon či Skype a je to v dnešní době de facto standard pro pozici zajišťování kvality software.

Náplní práce SDET pracovníka je tvorba softwaru, který slouží k testování jiného softwaru, tedy automatizovaných testů. Samozřejmostí je, že musí splňovat požadavky kladené na každého QA pracovníka, jako je tvorba testovacích plánů, exekuce a reporting. Výhodou je pokročilá znalost programovacích jazyků, funkčních a nonfunkčních testovacích přístupů. Práce zahrnuje také kooperaci s vývojáři a QA vedoucími pracovníky pro zajištění dlouhodobé strategie a plnění plánu. Neustálá snaha o nacházení příležitostí ke zvýšení pokrytí testovaného kódu testy a zvyšování úrovně automatizace.

V Polarionu je tato pozice významná z hlediska zavádění automatizace ve firmě. SDET se nyní nachází v samostatném automatizačním týmu a jeho plánem je analýza nových nástrojů v oblasti UI testování s možností jejich aplikace na produkt. Dále by měl položit základy automatizovaných funkčních i non-funkčních testů, pokrývajících hlavní produktové součásti (Document-like Editor, Rich Pages či Live Plans). Následně se pak angažovat v nově založených odvětvích, které by firemní automatizace měla zastřešovat, a to (již zmiňované) UI testy, bezpečnostní (security) a zátěžové (performance) testy.

5.5 Důvody pro automatizaci

Aktuální situace ve firmě vykazuje potřebu zavedení automatizace ve větší míře, než byla využívána doposud. Jedním z důvodů je nedostatek vyškolených lidských zdrojů v oblasti manuálního testování, neboť zaškolovací proces, i přes některé výrazné změny, není otázkou dnů ale měsíců, zejména díky komplexitě produktu.

Samotné testování součástí produktu (tzv. features) probíhá během celé doby v rámci sprintů, díky krátkým cyklům mezi vývojáři a testery a také díky dogfoodingu na firemním produkčním prostředí.

Před každým releasem prochází aplikace kompletním instalačním, regresním a testem updatu, při kterém se QA tým snaží pokrýt více kombinací operačních systémů a prohlížečů. Tento proces se tedy opakuje čtyřikrát ročně.

V případě vydání nové verze prohlížeče je exekvován souhrnný UI test, který by měl odhalit případné regrese v zobrazování aplikace ve vybraném prohlížeči.

Automatizace těchto manuálně prováděných testů by tyto procesy urychlila a také doplnila testové portfolio o zatím zcela chybějící bezpečnostní testování. Dále by se soustředila na dynamickou analýzu již existujícího automatizovaného zátěžového testování.

5.5.1 Pokrytí aplikace testy

S přihlédnutím k architektuře Polarion ALM probíhají testy jak manuální tak automatické v různých vrstvách aplikace.

Na nejnižší vrstvě jsou exekvovány jednotkové (unit) testy, které se týkají zejména základních scénářů funkčního testování a jejichž celý cyklus je automatizován. Od vytvoření buildu ve firemním produkčním prostředí, přes jeho instalaci, otestování a import výsledků zpět do produkčního prostředí.

Střední vrstva aplikace je pokryta statickou analýzou zátěžového testování. Statická proto, neboť jde o měření v předpřipraveném prostředí, které se nemění (počet projektů, uživatelů, dokumentů, atd.) Na základě opakování testovacího scénáře, jehož čas je měřen a následně zprůměrován. Není zde tedy zajištěna simulace více uživatelů, kteří by pracovali paralelně.

Testy v nejnižší a střední vrstvě jsou vytvářeny a udržovány vývojáři.

QA tým zajišťuje UI, funkční a testování použitelnosti v nejvyšší vrstvě aplikace. Tato exekuce je ryze manuální dle připravených a udržovaných scénářů.

6 Implementace automatizace ve firmě

Zapojení SDETa do procesu automatizace probíhalo velmi pozvolna. Vzhledem k již zmíněnému problému s QA pracovníky bylo rozhodnuto, že SDET bude v rámci svého zaučení pomáhat s regulérní prací QA první 3 měsíce. Poté mělo dojít k jeho osamostatnění a alokace jeho času čistě na automatizaci měla proběhnout během půl roku.

Tento prvotní plán selhal kvůli nedostatečné kvalitě nově zaměstnaných QA pracovníků. SDET tedy vykonával regulérní QA práci celý rok a souhrnně lze říci, že automatizaci věnoval 10% času (cca 2,5 sprintu). I přesto dokázal převzít již dosažené poznatky z oblasti UI a zátěžového testování, získané při předchozích ověřováních vhodných technologií a využít je v dalších krocích směrem k automatizaci.

Díky zaměstnání nových QA posil bylo možno sestavit plán automatizace pro rok 2015.

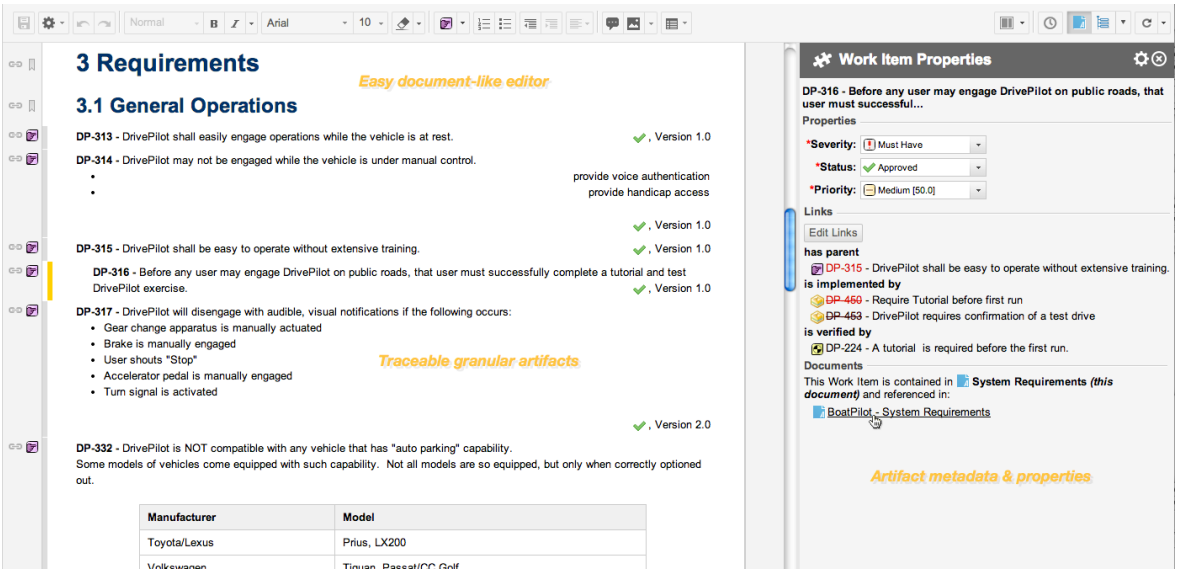
6.1 Transfer získaných znalostí v oblastech zátěžového a UI testování

Se všemi níže zmíněnými informacemi a znalostmi byl SDET obeznámen, aby z nich mohl později vycházet při tvorbě nových testů.

6.1.1 Zátěžové testování

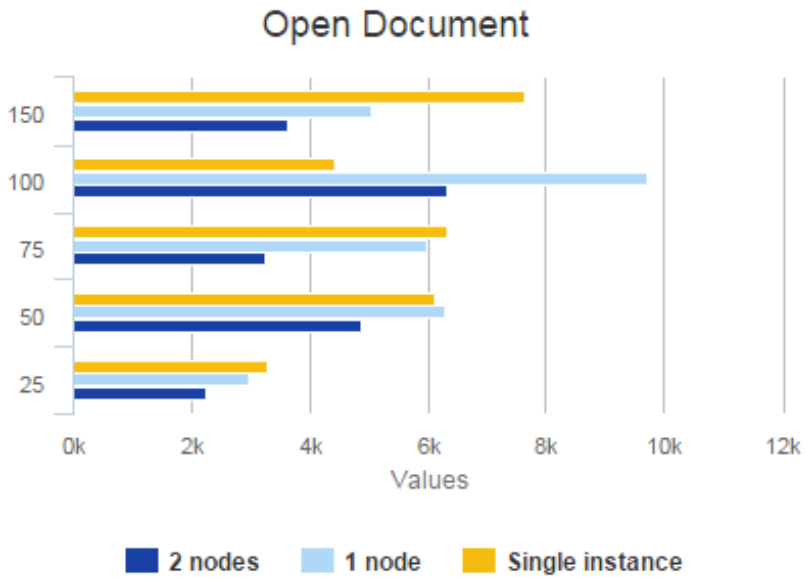
V oblasti zátěžového testování byla již v minulosti provedena analýza použitelnosti nástroje jMeter v podobě stress testu a definovány případy užití a množiny testovaných hodnot. Jednalo se zejména o využití nástroje pro měření odezvy aplikace Polarion ALM v clusteru při načítání velkého objemu dat a zobrazování uživateli [4].

Testy vytvořené v nástroji jMeter pokrývají hlavní součásti aplikace. Pro ukázku je možno zmínit Document-like editor, který je schopen zobrazovat požadavky, defekty či testovací případy v textové podobě včetně formátování. Vzhledově prostředí připomíná Microsoft Office Word obohacený o další funkční rozšíření.



Obrázek 16: Prostředí Document-like editoru[30]

Ilustrační test pokrývá poměrně jednoduchý případ užití, a sice otevření existujícího dokumentu. Testování probíhalo pro jednotlivé dávky 25, 50, 75, 100 a 150 uživatelů v prostředí clusteru při zapojení stroje se samostatně běžící instalací (single instance) a dvou uzlů (nody). Časy odezvy v následujícím grafu jsou uvedeny v tisících milisekund.



Obrázek 17:Součást reportu z analýzy nástroje jMeter při použití na produktu Polarion ALM

V dalším kroku došlo k seznámení se s aplikací jProfiler, který je ve firmě využíván zejména vývojáři k mapování jednotlivých požadavků klienta na server při běhu aplikace a odhalování kritických míst při výkonnostním testování [47]. Statická analýza,

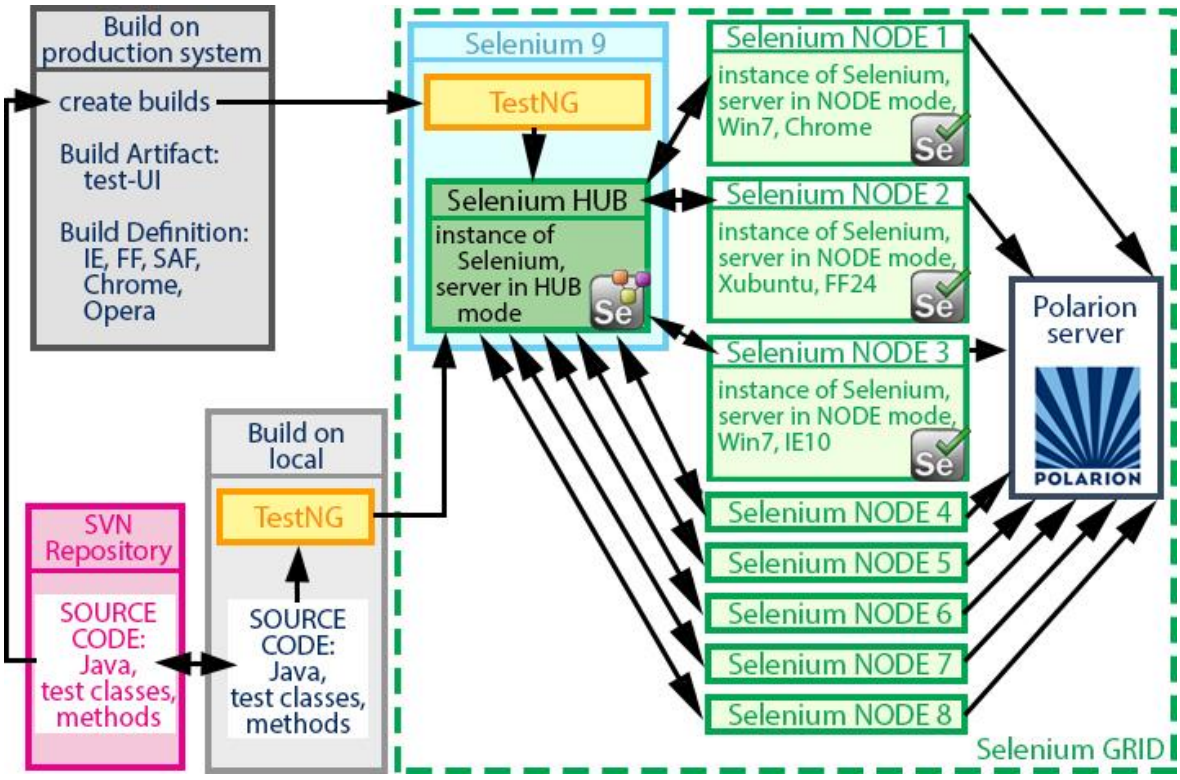
která obsahuje jUnit testy, je v případě problematického scénáře podrobena také profilování za pomoci tohoto nástroje. Z implementačního hlediska musí jednotlivé testy splňovat podmínku trvání alespoň 100 milisekund kvůli zachování konzistence výsledků, čemuž se podřizuje opakování scénáře vícekrát za sebou v rámci jednoho testovacího skriptu. Následuje změření jednotlivých časů a jejich zprůměrování. Dosažené hodnoty jsou importovány do firemního produkčního prostředí a výsledky měření zobrazeny v přehledném reportu. Ten je podroben zkoumání v rámci každého sprintu. Případné regrese jsou dále eskalovány a řešeny.

6.1.2 Testování uživatelského rozhraní

Oblast automatického UI testování je ve firmě časově nejmladší. Myšlenka zapojit Selenium GRID a nástroj TestNG do produkčního prostředí naplňovala cíl exekvovat testovací skripty paralelně a na různých prostředích (kombinace prohlížečů a operačních systémů) [42][46].

Implementace však ztroskotala na nepřehlednosti importovaných výsledků a neudržovatelnosti kódu. Samotné testovací skripty narážely na rozšířený problém v oblasti testování uživatelského rozhraní, a to přidělování dynamických identifikátorů jednotlivým elementům webové aplikace. Nebylo je tedy možné použít k jednoznačné identifikaci ovládacích prvků aplikace, které byly součástí webové stránky. Jediná možnost, jež se nabízela, se týkala využití kombinace vlastností atributů, např. name, odkazu či CSS třídy.

Vyvstala otázka, zda pokračovat se Seleniem nebo se soustředit na hledání vhodného robustnějšího řešení, které by se vypořádalo s dynamickými identifikátory tzv. out of the box, tedy defaultně hned po instalaci bez složitého nastavování či hledání cest, jak problém obejít.



Obrázek 18: Schéma napojení Selenium GRIDu k produkčnímu prostředí. Vypracováno na základě interních materiálů společnosti Polarion.

6.2 Tvorba nových testů

Pro další kroky v oblasti automatizovaného testování muselo padnout rozhodnutí, zda je vhodné pokračovat v používání již známých testovacích nástrojů, či zda je na místě věnovat čas analýze nových produktů na trhu.

V případě jMeteru a jProfileru nedošlo k větším změnám, v případě Selenia ano, neboť bylo nutné změnit zcela přístup k testování UI. Díky partnerství firmy Polarion s firmou Ranorex vznikl nápad vyzkoušet právě jejich nástroj pro UI testování, což s sebou neslo také výhodu nízkých pořizovacích nákladů.

6.2.1 jMeter a jProfiler

SDET se v rámci vývoje nových testů pro jMeter soustředil zejména na editaci jednotlivých komponent v Polarionu. Vzhledem k tomu, že existující testy se týkaly pouze zobrazování dat, byly nové testy větší výzvou.

Účelem těchto testů bylo zejména seznámení s nástrojem a získání dalších znalostí z oblasti reportingu. Nešlo tedy natolik o samotné výsledky jako spíše o důkaz, že lze jMeter efektivně používat i nadále. Za zmínku stojí zejména parametrizace jednotlivých volání – tedy RPC (Remote Procedure Calling) – která je v jMeteru poněkud obnažena.

Lze upravit přímo v editoru aplikace a je tak náchylná k chybám, které se mohou zanést překlepem [8].

Oba nástroje pak byly použity pro analýzu defektu ze strany zákazníka, neboť schopnost systému obsloužit více uživatelů a zvláště její míra je oblíbeným námětem dotazů od zákazníků. Jedinou výraznou nevýhodou, která se objevila během měření, je invazivnost jProfileru. I přesto, že je pouze vzdáleně připojen k testovanému JVM (Java Virtual Machine) stroji, zatěžuje neúměrně stroj, kde měření probíhá a tím tento probíhající proces zpomaluje.

6.2.2 Ranorex

Důvody pro přechod od Selenia k Ranorexu jsou již zpracovány velmi přehledně na webových stránkách nástroje [38].

Ranorex se neustále snaží zlepšovat podporu pro lokalizaci webových elementů (DOM mapping). Aktuálně verze 5.3.1 nabízí robustnější řešení „out of the box“ [40]. Z hlediska firmy Polarion je nepostradatelná i možnost vytvářet nové testy bez nutnosti úprav v kódu. Lze tedy nahrát test pouhým provedením případu užití a poté test parametrizovat přes Ranorex GUI. Samozřejmě lze test také vytvořit znovu použitím už přeprogramovaných komponent (modulů), které připraví SDET, neboť zatím lidské zdroje nabízejí pouze manuální testery, kteří s programováním většinou nemají zkušenosti.

Výraznou výhodou při evaluaci nástroje je snadná předem zmapovaná cesta Ranorex specialistů pro testování webových služeb[52].

SDET měl zatím možnost v oblasti UI dojít nejdále. V procesu evaluace nástroje Ranorex absolvoval dvě online školení a po studiu dalších dostupných online materiálů byl schopen pokrýt alespoň části základních programových celků Polarion ALM automatickými testy [5]. Výsledky testů prošly transformací XSLT a importem do Polarion produkčního prostředí za pomoci připravených skriptů. Zde došlo k automatickému vytvoření záznamů o běhu testů (test runs) a v případě pádu testu i k vytvoření defektu, který může být dále eskalován a řešen v produkčním prostředí.

Řešení části problému s dynamickými identifikátory, který byl zásadní překážkou při vývoji testů pro Selenium, je díky rozsáhlému Ranorex fóru a uživatelskému manuálu

velice jednoduché [37][50]. Jedná se o snížení váhy atributu, kterou využívá inteligentní algoritmus pro lokalizaci prvku na webové stránce a je také možné na základě regulárního výrazu dynamicky vytvářené identifikátory jednoduše odfiltrovat. Dochází tak automaticky k vytvoření cesty k prvku v DOM struktuře, která se nazývá RanoreX-Path a chová se velmi podobně jako xPath [53].

Druhou částí problému je samotný kód, který generuje Polarion ALM. Zaměření jednotlivých elementů (ovládacích prvků Polarionu) v DOM struktuře zabírá 80-90 % času, který je stráven na sestavení jednoho testu v Ranorexu. Pro další možnost rozvoje bylo nutné najít rychlejší řešení tohoto problému. Po poradě se senior developery, kteří mají zkušenosti s nově vytvářeným UI, hlubším studiu xPath možností a již popsaných změnách nastavení v Ranorexu došlo ke snížení času na ¼ původního objemu. I přesto je zachována kvalita mapovaných cest, pro zajištění opakovatelnosti testu ve všech podporovaných prohlížečích. Optimální řešení tedy spočívá v možnosti, kdy bude zajištěno, že atribut jednoznačně identifikuje daný webový element, či že při změně ID ovládacích prvků stránky či změně struktury bude test fungovat bez nutnosti jeho dalších úprav.

Jelikož je část UI generována pomocí GWT je nutno zapojit do procesu i developery, aby byl zajištěn správný přístup k mapování elementů ve webové stránce [32].

Dalším problém, který se v budoucnosti vyskytne, což je známo již nyní, je spojen s používáním Jetspeed [26]. Vývojáři chtějí od jeho používání upustit, což může mít neblahý vliv na již existující UI testy, které bude nutno přepracovat.

Získané poznatky lze shrnout do tvrzení, že ne všechny UI prvky mohou mít optimální cesty pro svou lokalizaci a být automatizovány pro testovací scénář, protože jim chybí potřebné atributy. Všechny nové komponenty, které jsou vyvíjeny, by měly tyto atributy (tags) obsahovat aby tak napomáhaly rychlejší identifikaci.

SDET se musí zúčastnit návrhu implementace úplně na začátku, aby mohl na základě scénářů určit, které elementy bude nutno identifikovat automatizovaně. Vývoj aplikace a implementace samotného kódu musí být více ovlivněny z pohledu testera. To vyžaduje znalost všech plánovaných uživatelských příběhů v předstihu. Tudíž tester je

schopen říct si o potřebnou spolupráci při vývoji částí kódu, které budou dále automatizovaně testovány, což ovlivňuje a klade požadavky i na jeho přípravu scénářů obsažených v plánu testování předem.

6.2.3 Řízení bezpečnosti

Zákaznická základna Polarionu obsahuje firmy z oblasti regulačního průmyslu, jako jsou farmaceutické, zdravotnické či organizace vyskytující se v automobilovém průmyslu. Díky tomu jsou kladeny zvýšené požadavky na bezpečnost procesů, výsledků a nástrojů, které používají. Tyto zákaznické firmy jsou podrobovány neustálým auditům ze strany vlády a kontrolních regulačních orgánů. Proto se i Polarion ALM musí podřizovat požadavkům kladeným v rámci bezpečnostních auditů.

Vedení firmy se rozhodlo věnovat této oblasti více času a prostředků. Proto nyní probíhá výběrové řízení pro bezpečnostního analytika a firma je v procesu rozhodování, do kterého nástroje pro provádění auditu či penetračního testování by měla dedikované prostředky investovat. V souvislosti s nákupem licence je také ochotna sponzorovat certifikaci zaměstnanců, kteří s nástrojem přijdou do styku nebo se v dané oblasti angažují více.

Podnikání kroků v oblasti bezpečnostního testování započalo sestavením požadavků, jako je potřeba častých auditů prováděných nástroji třetích stran, a později přešlo do identifikace procesu penetračního testování.

Vzhledem k tomu, že proces evaluace jednotlivých nástrojů stále probíhá, je možno říci, že doposud byl automatizační tým soustředěn na distribuci Kali Linux, která obsahuje stovky jednotlivých nástrojů [28]. Díky popularitě této distribuce se Polarion rozhodl podrobit ji analýze vhodnosti nasazení do produkčního prostředí.

Dalším nástrojem, na který se soustředí pozornost automatizačního týmu je nástroj ZAP (The Zed Attack Proxy). ZAP poskytuje automatické skenování a pomáhá tak odhalit bezpečností mezery, což je vhodné zejména pro posudky firemních auditů [34]. Pro simulaci útoku pomocí SQL injection firma podrobuje evaluaci také nástroj SQLMap[45].

Všechny zmíněné nástroje jsou zatím pouze ve fázi výběru a není rozhodnuto, pro který z nich se Polarion rozhodne pro své další kroky v oblasti bezpečnostního testování. I

přesto však tato kombinace poskytuje velice dobré pokrytí pro analýzu stavu firemního produktu a také pro penetrační testování.

Pro zajištění dostatečného pokrytí testů a procedur firma využívá také testovacího průvodce od OWASP [33].

Významným krokem je audit provedený s pomocí nástroje ZAP, který poskytl velice obohacující zpětnou vazbu. Po prvním spuštění byl schopen detekovat více než 30 chyb zabezpečení na různých bezpečnostních úrovních, což je viditelné v následujícím shrnutí.

ZAP Scanning Report

Summary of Alerts

Risk Level	Number of Alerts
High	2
Medium	2
Low	27
Informational	10

Obrázek 19: Shrnutí výsledků auditu provedeného na produktu Polarion ALM pomocí nástroje ZAP

Dopady, které tato evaluace má na procesy ve firmě, nejsou rozhodně zanedbatelné. Vzrůstá potřeba změnit programovací standardy a zahrnout osvědčené postupy ohledně řízení bezpečnosti a minimalizace rizik ve vývoji. Některé kroky již byly podniknuty, zejména neustálá validace komponent (součásti produktové distribuce v podobě JAR knihoven) a jejich bezpečnostních slabin. Vývojové postupy k zabránění injekci nebezpečného kódu nevyjímaje.

Další release produktu (verze Polarion ALM 3.9.1) je možno zacílit na zajištění bezpečnosti z pohledu penetračních testů a také interního auditu.

6.3 Plán automatizace 2015

Následující plánované kroky vedou k rozdělení pozornosti automatizačního týmu do tří hlavních podskupin, a to:

- testování uživatelského rozhraní,
- zajištění bezpečnosti a odhalování rizik pomocí penetračních testů,
- a rozšíření analýzy zátěžového testování.

V návaznosti na předchozí úspěšnou implementaci Ranorex testů je plánováno pokračovat v pokrývání uživatelského rozhraní automatizovanými testy. S přihlédnutím k existujícím problémům a již dříve plánovaným změnám v UI je dalším krokem vytvoření knihovny objektů (repository). Ta bude obsahovat optimalizované cesty v DOM struktuře ke všem ovládacím prvkům aplikace Polarion ALM a bude přístupná i ostatním QA pracovníkům, kteří budou moci bez většího úsilí, za pomoci nástroje Ranorex, vytvářet snadno nové automatizované testy. Jejich následná parametrizace může zatím zůstat v rukou SDETa či dalších pracovníků v automatizačním týmu.

Pro exekuci Ranorex testů budou vyhrazeny další virtuální stroje, každý pro jeden testovaný prohlížeč.

Evaluace jednotlivých nástrojů pro bezpečností testování je zatím stále v průběhu, i přesto už lze zaznamenat její přínosy. Není pochyb, že firma plánuje investovat do jejího dalšího rozšiřování, což se týká jak školení pro členy automatizačního týmu tak do softwarového vybavení. Bezpečnostní testování je také často dotazovanou oblastí od stávajících i potenciálních zákazníků a je součástí auditů.

Plánované investice v oblasti zátěžového testování přináší zejména rozšíření stávajících případů užití o nové, více se dotýkající defektů pocházejících od zákazníků. To znamená větší důraz na zkoumání regresí s pomocí stávajících nástrojů (jMeter a jProfiler).

Aby byla zajištěna snadná kontrola plnění předsevzatého plánu, je také nutné zavést dostatečné metriky pro toto ověření.

Již samotné zadávání scénářů a případů užití aplikace, které využívá BDD přístupu by mělo být plně automatizováno za pomoci nástroje JBehave [53]. V několika málo krocích bude možno transformovat zadávané požadavky do koncových testovacích případů.

7 Shrnutí výsledků

Polarion jako firma vyrábějící softwarové produkty prošla díky trvalému růstu transformací od malého ke střednímu podniku, stejně jako se změnilo portfolio jejích zákazníků ze středních na korporátní klienty. Zásadní zlepšování kvality bylo poháněno shora dolů právě zákazníky, kteří požadovali snadno použitelný systém, který je vysoce výkonný a jeho používání nezvyšuje bezpečnostní rizika.

Vedení firmy zákaznické potřeby vyslyšelo a tento proces vyústil ve změnu přístupů v oblasti zajišťování kvality a samotného způsobu, jakým produkt firma dodává. Jako každá změna, musela i tato začít u zaměstnanců a u procesů. Skrze systematický proces začínající vytyčením si požadavků, pokračující hledáním cest k jejich naplnění a konečně změnou samotných procesů požadavky zákazníků naplnit. Polarion tak prošel změnami ve všech následujících oblastech na základě zachovávání stejných principů.

Polarion se zprvu soustředil na regrese, neboť se vyskytl znatelný přírůstek v počtu záplat (patches), které kvůli nim musely být naimplementovány. Dále pak na nástroje, které jsou schopny otestovat tyto uživatelské případy založené na objevených regresích. Ranorex a Selenium byly hlavními uchazeči a po dokončení evaluačního procesu se firma rozhodla pro profesionální nástroj s dostatečnou podporou, který je vytvořen partnerskou firmou. Firma věnovala čas porovnání obou nástrojů a vyzkoušela běh ukázkových testů v připraveném prostředí. Výběr nástroje byl následován rozšířením procesů ve snaze podpořit běh automatizovaných testů a konečně zvýšením povědomí ve firmě o prospěšnosti a výhodách, které plynou z času věnovaného nástroji Ranorex.

Díky růstu firmy se zvýšil počet jejich klientů, ale také počet používaných licencí, a tím pádem konkurenčních uživatelů v rámci jedné instalace. Bylo prokazatelné, že je nutno věnovat čas i v oblasti zátěžového testování. Opět započalo rozhodování podpořené poznatky získanými z evaluace testovacích nástrojů. Nepopiratelnou potřebou bylo také najmout zkušenější pracovníky v oblasti zátěžového testování. Jelikož hodnocení aplikace pod zátěží a snaha o její zrychlování odezvy je součástí vývoje softwarového produktu, jsou i zákaznické požadavky a standardy v tomto odvětví neustále se měnící.

Bezpečnost byla poslední ze zmíněných zákaznických potřeb. Polarion je stále v rané fázi evaluace zkoumaných produktů. Proces evaluace se ani v oblasti bezpečnostního

testování nemění. Firma se snaží z více aplikací vybrat ten správný produkt, který by pokryl všechny její potřeby. S přihlédnutím k jednotlivým požadavkům firmy v této oblasti byly zatím vyzkoušeny nástroje pro bezpečností audit, penetrační testování a nástroj pro cross-site skripty. Plánem je pokračovat v šetření jak problematiky bezpečnosti, tak evaluaci dalších nástrojů, na něž jsou vyhrazeny prostředky z firemního rozpočtu.

V praktické části práce bylo možné se setkat s jednotlivými příklady firemního přerodu v oblasti zajišťování kvality poháněnými požadavky od zákazníků. Proces popisuje jak přistoupit k systematické analýze požadavků na nový testovací nástroj a jeho zavádění do produkčního prostředí.

Automatizace v procesu řízení kvality je trendem, který většina firem vyvíjejících software následuje. Složitost samotného softwarového řešení a jeho závislost na komponentách ho často činí velmi těžko či téměř netestovatelným manuálními testy. Požadavek na použití sofistikovaného testovacího nástroje, který je schopen simulovat využití softwaru a měřit výkonnostní dopady je v dnešní době nutností.

8 Závěr

Agilní metodika obecně vyvíjí tlak na QA pracovníky. Zejména díky nutnosti zabezpečení kvality při rychlých a opakujících se dodávkách každý sprint. Tudíž i proces manuálního testování je nutno opakovat každý sprint. Jeho průběh křivky nákladů na testování však není lineární, ale více šikmý, blížící se exponenciálnímu průběhu. Děje se tak proto, že jednotlivé test suity se neustále rozšiřují o nové testy, stejně jako se testovaná aplikace zvětšuje o nové součásti.

QA pracovníci vyhledávají automatizaci ve snaze zamezení výše popsaného efektu a jejich cílem také snížení rizika lidského faktoru, které s sebou opakování činností nese. Jak známo, při opakované aktivitě dochází k únavě člověka a zvýšení počtu přehlédnutých chyb.

SCRUM metodika, kterou se firma Polarion snaží aplikovat pro vývoj produktu, vyžaduje mít na konci každého sprintu produkt ve stavu, jež je schopen vypuštění mezi zákazníky. Zde nastává rozkol, neboť není v lidských silách během jednoho sprintu exekvovat testy, které firma provádí každý release, a zároveň vyvinout a otestovat nové součásti. Základní instalační, regresní a test updatu svou náročností a rozsáhlostí nepřipouští možnost manuálního vykonání na konci každého sprintu, je li uváženo, že v tom samém sprintu dochází k manuálnímu testování nově přidaných součástí.

Firma proto rozšířila své strategické cíle, s čímž se pojí i její rozhodnutí investovat prostředky, tedy čas, finance a lidské zdroje do automatizace procesu testování.

Automatizace je nedílnou součástí pro dosažení stavu testovatelných sprintů a produktu schopného releasu. Firma se proto soustředí na pokrytí regresních a epických (E2E) testů prováděných každý sprint, aby došlo k zajištění požadované kvality produktu, jak předepisuje SCRUM.

Vysoké vstupní náklady při zavádění automatických testů do produkčního prostředí, mohou být přirovnány k průběhu logaritmické funkce. Naproti tomu manuální testování v té chvíli vyžaduje náklady nižší. Nesporné přínosy automatizace jsou viditelné ve chvíli, kdy se testovací základna rozrůstá o další testy a následně když dochází ke snížení nákladů pro opakující se procesy.

Praktickým dopadem tohoto přístupu je fakt, že Polarion zavádí Ranorex, a už od prvních kroků firma změnila svůj přístup k evaluaci tohoto nástroje. Naproti klasickému průběhu evaluace se snažila zapojit jej okamžitě do produkčního prostředí. Kritériem bylo zhodnotit, zda Ranorex obstojí ve firemním prostředí, které je značně přizpůsobené potřebám firmy. Tento detailní přístup pomohl firmě efektivně posoudit Ranorex po všech stránkách.

QA pracovníci strávili několik sprintů psaním testů, které jsou přímo přizpůsobené pro produkční prostředí i v konkurenčních nástrojích s cílem nalezení benefitů, které jejich používání firmě přinese.

Ranorex zvítězil díky velmi pokročilému DOM mappingu, přirozené podpoře GWT a konečně jako firemní partner nabízející plnohodnotnou podporu při vývoji a správě testovacích scénářů.

V oblasti automatického testování se Polarion také významně snaží pokrýt oblast zabezpečení. I přesto, že dle teorie spadá testování bezpečnosti do kategorie non-funkčních testů, v praktickém životě, pokud se jedná o webové aplikace, lze vidět, jak moc důležité penetrační testování v rámci celkového pokrytí aplikace testy je.

Tento zesílený důraz na penetrační testování se stává více evidentní ve všech webových aplikacích a na všech úrovních – ať už se jedná o intranet, extranet nebo internet. Zákazníci se obávají o bezpečnost dat a informací a vlády se stávají více aktivními v regulační politice, kterou firmy i vytvářené produkty musí respektovat.

I přes fakt, že Polarion jako firma je nováčkem, zatím nezatíženým zkušeností v této oblasti, zajištění bezpečnosti se stává starostí číslo jedna ve firemním testovacím přístupu. Změny v zákaznických požadavcích směřující ke zvýšení zabezpečení webových aplikací silně ovlivňují i změny firemního přístupu v oblasti bezpečnostního testování.

Pro firmu se tak stává tato oblast nejen položkou na seznamu non-funkčních testů, ale samostatnou kapitolou.

Každá změna pracovního procesu nebo nástroje musí být poháněna lidmi. Toto je kritický faktor, který rozhoduje nejen o rychlosti zavedení změny, ale také o úspěšnosti, respektive neúspěšnosti transformace. To je obvykle skryto v teoretické oblasti, která zdůrazňuje procesy a nástroje, ale stává se evidentní při praktické implementaci. Aby

transformace mohla být považována za úspěšnou, musí s sebou nést prokazatelné přínosy viditelné kolegům, managementu i zákazníkům. Každá z těchto skupin totiž zaujímá odlišný postoj k nahlížení na přinášenou hodnotu.

Pro management je benefitem celková spokojenost zákazníka, jež je řízena náklady věnované do QA oblasti. A také automatizace procesů, která by je měla urychlovat a nikoli zpomalovat či jinak znesnadňovat.

Zaměstnanci firmy Polarion byli zprvu skeptičtí k zavedení automatizace díky nedostatku důvěry, že takový nástroj jako Ranorex může dosáhnout efektivního pokrytí aplikace testy. To samozřejmě vyžaduje dedikovaný čas na tvorbu Ranorex testů v produkčním prostředí, což zvyšuje důvěru a odkrývá přínosy tohoto přístupu.

Konečně zákazníci, kteří požadují dostatečně otestovaný, intuitivní a bezpečný software. Automatizace použitá při jeho vytváření umožňuje zákazníkům soustředit se na používání dané aplikace, která jim pomáhá řídit jejich business procesy, místo toho aby investovali své zdroje do její údržby.

Polarion musel čelit obrovské složitosti v průběhu tohoto procesu transformace z manuálních na automatizované testovací procesy. To vše bylo řízeno zákazníky, kteří soustředili své požadavky do oblastí zájmu podle toho, co je pro ně samotné důležité.

Metodické přístupy poskytly dostatečnou základnu pro porozumění procesům a nastínily směr růstu v této oblasti. Praktické dodaly více smyslu a detailněji zachytily jak věci v produkčním prostředí fungují.

Bylo možné vidět, jak se firemní priority změnily na základě zákaznické zpětné vazby a jak Polarion dosáhl uspokojení na základě splnění zákaznických potřeb.

Složitost dnešních technologií činí proces transformace mezi nimi více bolestivým, protože zahrnuje zákazníky, zaměstnance i management, kteří v přeměně spatřují odlišné přínosy.

9 Seznam použité literatury

[1] 8 Key Points of Blue Ocean Strategy. *Blue Ocean Strategy*. [online]. 2015 [cit. 2015-04-20]. URL: http://www.blueoceanstrategy.com/8-key-points-of-blue-ocean-strategy/

[2] About Polarion Software. *Polarion Software*. [online]. 2015 [cit. 2015-04-20]. URL: http://polarion.com/company/index.php

[3] Analyst report: Why Ovum recommends Polarion. *Polarion Software*. [online]. 2012 [cit. 2015-04-20]. URL: http://www.polarion.com/specials/ovum/index.php

[4] Apache Software Foundation. Apache JMeter™. *The Apache Software Foundation*. [online]. 2015 [cit. 2015-04-20]. URL: http://jmeter.apache.org/

[5] Automated Testing Webinars. *Ranorex GmbH*. [online]. 2015 [cit. 2015-04-25]. URL: http://www.ranorex.com/automated-testing-webinars.html

[6] BECK, K. a kol. Manifest Agilního vývoje software. *Manifesto for Agile Software Development.* [online]. 2001 [cit. 2013-02-08]. URL: http://agilemanifesto.org/iso/cs

[7] BOROVCOVÁ, A. Základy testování. *poeta.cz*. [online]. 2008 [cit. 2013-02-08]. URL: http://www.poeta.cz/Zaklady_testovani.pdf

[8] BOX, D. a kol. Simple Object Access Protocol (SOAP) 1.1. *W3C*. [online]. 2000 [cit. 2015-04-25]. URL: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383532

[9] ČERMÁK, M. Automatizované testy aplikací typu klient-server. *CleverAndSmart*. [online]. 18.1.2011 [cit. 2013-02-08]. URL: http://www.cleverandsmart.cz/automatizovane-testy-aplikaci-typu-klient-server/

[10] ČERMÁK, M. Black box test. *CleverAndSmart*. [online]. 30.10.2010 [cit. 2013-04-08]. URL: http://www.cleverandsmart.cz/black-box-test/

[11] ČERMÁK, M. Grey box test. *CleverAndSmart*. [online]. 30.10.2010 [cit. 2013-04-08]. URL: http://www.cleverandsmart.cz/grey-box-test/

[12] ČERMÁK, M. Load test. *CleverAndSmart*. [online]. 30.10.2011 [cit. 2013-04-10]. URL: http://www.cleverandsmart.cz/load-test/

[13] ČERMÁK, M. Penetrační testování jako součást životního cyklu vývoje SW. *CleverAndSmart*. [online]. 25.01.2012 [cit. 2014-04-10]. URL: http://www.cleverandsmart.cz/usability-test/

[14] ČERMÁK, M. Testování SW. *CleverAndSmart*. [online]. 4.7.2009 [cit. 2013-02-08]. URL: http://www.cleverandsmart.cz/testovani-sw/

[15] ČERMÁK, M. Typy testování. *CleverAndSmart*. [online]. 26.10.2011 [cit. 2013-04-10]. URL: http://www.cleverandsmart.cz/typy-testu/

[16] ČERMÁK, M. Usability test. *CleverAndSmart*. [online]. 28.11.2010 [cit. 2014-04-10]. URL: http://www.cleverandsmart.cz/usability-test/

[17] ČERMÁK, M. White box test. *CleverAndSmart*. [online]. 30.10.2010 [cit. 2013-04-08]. URL: http://www.cleverandsmart.cz/white-box-test/

[18] ČERMÁK, M. Způsoby testování. *CleverAndSmart*. [online]. 06.11.2010 [cit. 2014-04-10]. URL: http://www.cleverandsmart.cz/zpusoby-testovani/

[19] Entin, N. eBook: Polarion Goes SCRUM. *Polarion Extensions*. [online]. 2015 [cit. 2015-04-20]. URL: http://extensions.polarion.com/extensions/45-ebook-polarion-goes-scrum

[20] Funkční testování. CoolPeople – kurzy. [online]. 2010 [cit. 2013-02-08]. URL: http://kurzy.coolpeople.cz/?class=sys_attach_new_class&action=AttachDownload&fileid=17387

[21] GRUDL, D. Escapování – definitivní příručka. *phpFashion*. [online]. 19.5.2009 [cit. 2014-04-18]. URL: http://phpfashion.com/escapovani-definitivni-prirucka

[22] HLAVA, T. Druhy, typy a kategorie testů. *Testování softwaru*. [online]. 21.8.2011 [cit. 2014-04-08]. URL: http://testovanisoftwaru.cz/category/druhy-typy-a-kategorie-testu/

[23] HLAVA, T. Nástroje pro testování softwaru. *Testování softwaru*. [online]. 2011 [cit. 2014-04-08]. URL: http://testovanisoftwaru.cz/nastroje/

[24] IEEE829. *829-2008 - IEEE Standard for Software and System Test Documentation.* 2008. URL: http://ieeexplore.ieee.org/servlet/opac?punumber=4578271

[25] Introduction to Web Services. *W3Cschools*. [online]. 2015 [cit. 2015-04-25]. URL: http://www.w3schools.com/webservices/ws_intro.asp

[26] Jetspeed Overview. *Jetspeed*. [online]. 2008 [cit. 2015-04-25]. URL: http://www.gwtproject.org/overview.html?csw=1

[27] JORGENSEN, P. *Software testing: a craftsman's approach*. 3rd ed. Boca Raton: Auerbach Publications, c2008, xxiii, 416 p. ISBN 0849374758.

[28] Kali Linux. *Kali Linux*. [online]. 2015 [cit. 2015-04-25]. URL: https://www.kali.org/

[29] Licensing Information. *Polarion Software*. [online]. 2015 [cit. 2015-04-20]. URL: http://www.polarion.com/products/alm/licensing.php

[30] LiveDoc Enhancements. *Polarion Software*. [online]. 2014 [cit. 2015-04-20]. URL: http://www.polarion.com/2014/LiveDoc-Enhancements-Document-like-editing-Specifications.php

[31] MEIER, J.D. a kol. Performance Testing Guidance for Web Applications. *Microsoft Developer Network*. [online]. 2007 [cit. 2014-04-08]. URL: https://msdn.microsoft.com/en-us/library/bb924375.aspx

[32] Overview. *GWT Open Source Project*. [online]. 11.5.2013 [cit. 2015-04-25]. URL: http://www.gwtproject.org/overview.html?csw=1

[33] Open Web Application Security Project – Testing Guide 4.0. *OWASP*. [online]. 2015 [cit. 2015-04-25]. URL: https://www.owasp.org/images/5/52/OWASP_Testing_Guide_v4.pdf

[34] OWASP Zed Attack Proxy Project. OWASP. [online]. 22.4.2015 [cit. 2015-04-25]. URL: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

[35] PACLÍK, V. Typy testů. *Bugtracker testování*. [online]. 2012 [cit. 2013-04-15]. URL: http://www.bugtracker.cz/index.php/metodika/11-typy-testu.html

[36] PATTON, R. *Testování softwaru*. Vyd. 1. Praha: Computer Press, 2002, xiv, 313 s. Programování. ISBN 80-7226-636-5.

[37] Ranorex Forum. *Ranorex GmbH*. [online]. 2015 [cit. 2015-04-25]. URL: http://www.ranorex.com/forum/

[38] Ranorex vs. Selenium. *Ranorex GmbH*. [online]. 2015 [cit. 2015-04-25]. URL: http://www.ranorex.com/ranorex-vs-selenium.html

[39] RAO, S. a KUMAR, N. Web application vulnerability detection using dynamic analysis with peneteration testing. *International Journal of Computer Science and Security*, 2012, 6.2: 1.

[40] Release Notes. *Ranorex GmbH*. [online]. 20.4.2015 [cit. 2015-04-25]. URL: http://www.ranorex.com/free-trial/release-notes.html#c8635

[41] ROUDENSKÝ, P. a HAVLÍČKOVÁ, A. *Řízení kvality softwaru: průvodce testováním.* 1. vyd. Brno: Computer Press, 2013, 208 s. ISBN 978-80-251-3816-8.

[42] Selenium-Grid. *SeleniumHQ*. [online]. 22.4.2015 [cit. 2015-04-25]. URL: http://www.seleniumhq.org/docs/07_selenium_grid.jsp

[43] Stay up-to-date with Internet Explorer. *Microsoft Developer Network Blogs*. [online]. 7.8.2014 [cit. 2015-04-20]. URL: http://blogs.msdn.com/b/ie/archive/2014/08/07/stay-up-to-date-with-internet-explorer.aspx

[44] Stop starému IE. Microsoft bude podporovat jen nejnovější verze. *Živě.cz*. [online]. 8.8.2014 [cit. 2015-04-20]. URL: http://www.zive.cz/bleskovky/stop-staremu-ie-microsoft-bude-podporovat-jen-nejnovejsi-verze/sc-4-a-174895/default.aspx

[45] SQLMap®. *SQLMap: automatic SQL injection and database takeover tool*. [online]. 2015 [cit. 2015-04-25]. URL: http://sqlmap.org/

[46] TestNG. *TestNG.org*. [online]. 2015 [cit. 2015-04-25]. URL: http://testng.org/doc/documentation-main.html

[47] The Award-Winning All-in-One Java Profiler. *eJ-technologies*. [online]. 2014 [cit. 2015-04-20]. URL: https://www.ej-technologies.com/products/jprofiler/overview.html

[48] The World Wide Web Consortium. Standards. *W3C*. [online]. 2015 [cit. 2015-04-18]. URL: http://www.w3.org/standards/

[49] Typy organizačních struktur a jejich členění. *BusinessInfo.cz*. [online]. 17.12.2010 [cit. 2015-04-20]. URL: http://www.businessinfo.cz/cs/clanky/typy-organizacnich-struktur-cleneni-2840.html#!&chapter=4

[50] User Guide. *Ranorex GmbH*. [online]. 2015 [cit. 2015-04-25]. URL: http://www.ranorex.com/support/user-guide-20.html

[51] VOBORNÍK, P. *Počítačové testovací systémy*. Sborník příspěvků z konference Alternativní metody výuky 2011. Praha : Univerzita Karlova, 28. 4. 2011. ISBN 978-80-7435-104-4.

[52] WALTER, T. How to Test Web Services with Ranorex. *Ranorex Blog*. [online]. 21.1.2014 [cit. 2015-04-25]. URL: http://www.ranorex.com/blog/how-to-test-web-services-with-ranorex

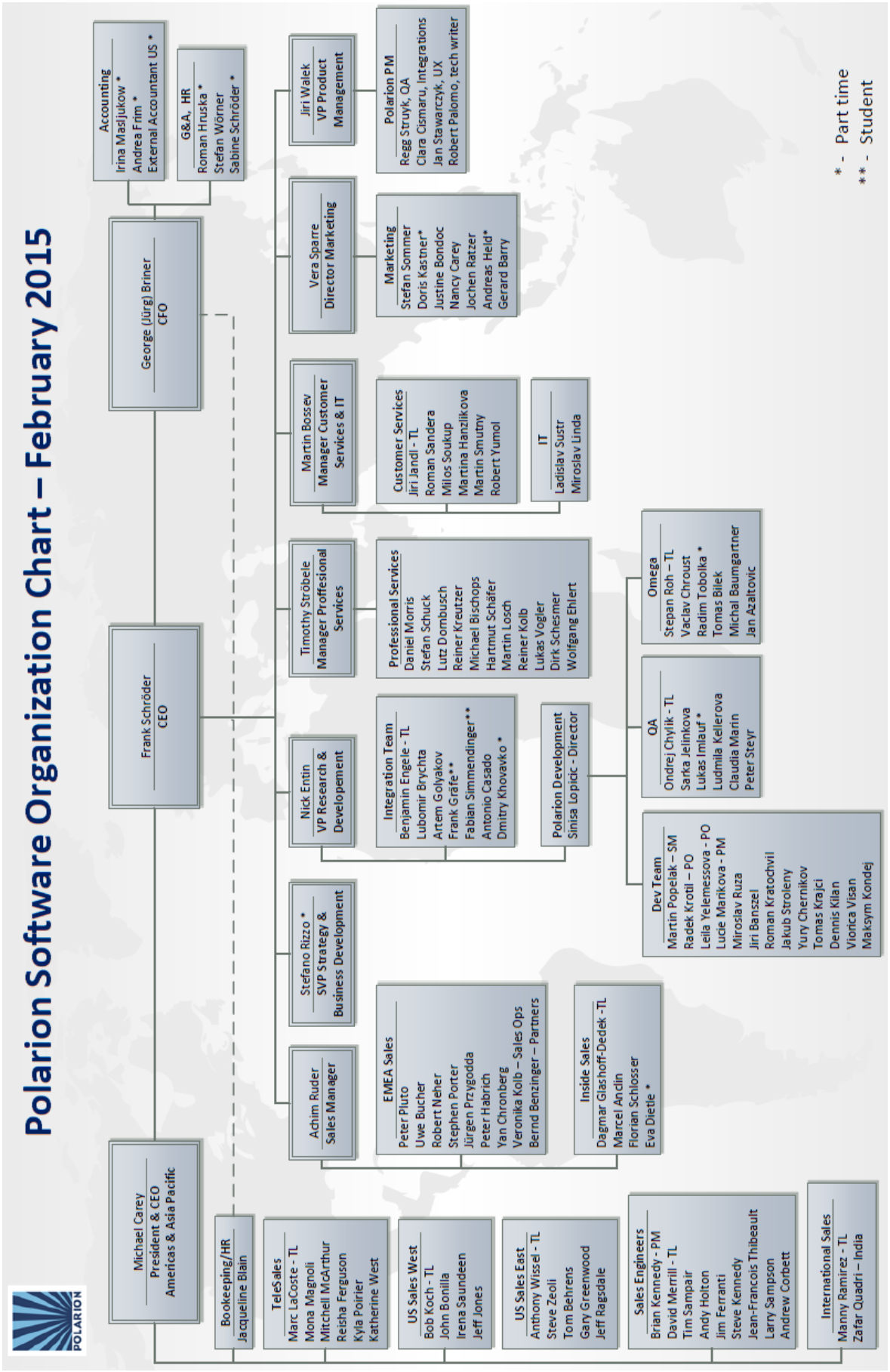
[53] What is JBehave. *JBehave*. [online]. 2014 [cit. 2015-04-25]. URL: http://jbehave.org/

[54] XPath Tutorial. W3Schools. [online]. 2015 [cit. 2015-04-18]. URL: http://www.w3schools.com/xpath/default.asp

10 Přílohy

Příloha č. 1: Organizační schéma firmy Polarion Software s. r. o. včetně jmen zaměstnanců, které odráží stav k únoru roku 2015.

Příloha č. 2: Zadání k závěrečné práci


Polarion Software Organization Chart – February 2015
Accounting
Irina Masljukow *
Andrea Frim *
External Accountant US *
G&A, HR
Roman Hruska *
Stefan Wörner
Sabine Schröder *
George (Jürg) Briner
CFO
Frank Schröder
CEO
Michael Carey
President & CEO
Americas & Asia Pacific
Jiri Walek
VP Product Management
Polarion PM
Regg Struyk, QA
Clara Cismaru, Integrations
Jan Stawarczyk, UX
Robert Palomo, tech writer
Vera Sparre
Director Marketing
Marketing
Stefan Sommer
Doris Kastner*
Justine Bondoc
Nancy Carey
Jochen Ratzer
Andreas Held*
Gerard Barry
Martin Bossev
Manager Customer Services & IT
Customer Services
Jiri Jandl - TL
Roman Sandera
Milos Soukup
Martina Hanzlikova
Martin Smutny
Robert Yumol
IT
Ladislav Sustr
Miroslav Linda
Timothy Ströbele
Manager Proffesional Services
Professional Services
Daniel Morris
Stefan Schuck
Lutz Dombusch
Reiner Kreutzer
Michael Bischops
Hartmut Schäfer
Martin Losch
Reiner Kolb
Lukas Vogler
Dirk Schesmer
Wolfgang Ehlert
Nick Entin
VP Research & Developement
Integration Team
Benjamin Engele - TL
Lubomir Brychta
Artem Golyakov
Frank Gräfe**
Fabian Simmendinger**
Antonio Casado
Dmitry Khovavko *
Polarion Development
Sinisa Lopicic - Director
Omega
Stepan Roh – TL
Vaclav Chroust
Radim Tobolka *
Tomas Bilek
Michal Baumgartner
Jan Azaltovic
QA
Ondrej Chylik - TL
Sarka Jelinkova
Lukas Imlauf *
Ludmila Kellerova
Claudia Marin
Peter Steyr
Dev Team
Martin Popelak – SM
Radek Krotil – PO
Leila Yelemessova - PO
Lucie Manikova - PM
Miroslav Ruza
Jiri Banszel
Roman Kratochvil
Jakub Stroleny
Yury Chernikov
Tomas Krajci
Dennis Kilan
Viorica Visan
Maksym Kondej
Stefano Rizzo *
SVP Strategy & Business Development
Achim Ruder
Sales Manager
EMEA Sales
Peter Pluto
Uwe Bucher
Robert Neher
Stephen Porter
Jürgen Przygodda
Peter Habrich
Yan Chronberg
Veronika Kolb – Sales Ops
Bernd Benzinger – Partners
Inside Sales
Dagmar Glashoff-Dedek -TL
Marcel Anclin
Florian Schlosser
Eva Dietle *
Bookeeping/HR
Jacqueline Blain
TeleSales
Marc LaCoste - TL
Mona Magnoli
Mitchell McArthur
Reisha Ferguson
Kyla Poirier
Katherine West
US Sales West
Bob Koch - TL
John Bonilla
Irena Saundeen
Jeff Jones
US Sales East
Anthony Wissel - TL
Steve Zeoli
Tom Behrens
Gary Greenwood
Jeff Ragsdale
Sales Engineers
Brian Kennedy - PM
David Merrill - TL
Tim Sampair
Andy Holton
Jim Ferranti
Steve Kennedy
Jean-Francois Thibeault
Larry Sampson
Andrew Corbett
International Sales
Manny Ramirez - TL
Zafar Quadri – India
* - Part time
** - Student

FIM UHK
UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Rokitanského 62, 500 03 Hradec Králové, tel: 493 331 111, fax: 493 332 235

Zadání k závěrečné práci

Jméno a příjmení studenta: **Ludmila Kellerová**

Obor studia: Informační management (5)
Jméno a příjmení vedoucího práce: **Filip Malý**

Název práce:
Testování webových aplikací

Název práce v AJ:
Web application testing

Podtitul práce:

Podtitul práce v AJ:

Cíl práce: Popsat teorii testování se zaměřením na webové aplikace,provést praktickou analýzu test. postupů ve firmě Polarion Software,navrhnout a implementovat pro firmu vlastní testy v rámci vývoje produktu Polarion ALM

Osnova práce:
1. Úvod
2. Cíl a zvolená metodika práce
3. Analýza problému
4. Návrh testů
5. Implementace
6. Shrnutí výsledků
7. Závěr
8. Seznam použité literatury
9. Přílohy

Projednáno dne: 13. 10. 2014

Podpis studenta

Podpis vedoucího práce