Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

DIPLOMOVÁ PRÁCE

Analýza a návrh řešení pro ověření kompatibility softwaru

Plzeň, 2016 Bc. Michal Kinzl

VLOŽIT ORIGINÁL ZADÁNÍ

ČESTNÉ PROHLÁŠENÍ

Prohlašuji, že jsem diplomovou práci na téma Analýza a návrh řešení pro ověření kompatibility softwaru vypracoval samostatně a na základě literatury a pramenů uvedených v seznamu použité literatury.

V Plzni dne 20. června 2016 ……..……

Bc. Michal Kinzl

PODĚKOVÁNÍ

Rád bych poděkoval zejména panu doktoru Holému za poskytnutí potřebných informací a znalostí pro realizaci této diplomové práce a za vedení pravidelných konzultací. Dále bych rád poděkoval svému zaměstnavateli za ochotu týkající se pracovní docházky v průběhu psaní této diplomové práce. V neposlední řadě musím poděkovat svým přátelům, rodině a Ing. Lucii Maškové, kteří mi dodávali potřebnou motivaci a podporu.

ABSTRAKT

Práce se zabývá analýzou aktuální situace na trhu týkající se nástrojů pro ověřování kompatibility a kvality softwaru psaného v programovacím jazyce Java. Analýza dostupných nástrojů je provedena z technického a částečně z obchodního pohledu, kdy byl důraz kladen na možnosti zařazení nástrojů do procesu vývoje, případně etapy testování.

Klíčová slova: životní cyklus vývoje softwaru, testování, kvalita softwaru, statická analýza, Java kompatibilita

ABSTRACT

Diploma thesis analyzes the current market situation regarding tools for verification of the compatibility and software quality written in Java priogramming language. Analysis of available tools is done from technical and partly from bussiness point of view when the focus was on the possibility of tools inclusion into developmnet process or testing phase.

Key words: software development lifecycle, testing, software quality, static analysis, Java compatibility

OBSAH

SEZNAM POUŽITÝCH ZKRATEK

SW - Software

JLS - Java Language Specification

JVM - Java Virtual Machine

QA - Quality Assurance

SA - Static Analysis

IDE - Integrated Development Environment

JAR - Java Archive

JDK - Java Development Kit

API - Application Programming Interface

TCK - Technology Compatibility Kit

FP - False positives

FN - False negatives

POM - Project Object Model

CI - Continuos Integration

1. Úvod

1.1. Úvod do kompatibility softwaru

V dnešní technologické době, kdy software ovládá a řídí valnou většinu elektronických zařízení, by měla kvalita, funkčnost a kompatibilita softwaru představovat nejvyšší prioritu při vývoji jakékoliv softwarové aplikace či systému.

Čas, cena a kvalita jsou veličiny, jejichž správná kombinace rozhoduje o úspěchu softwarových produktů. Ale doba, kdy nejnižší cena vždy bezpodmínečně vyhrávala nabídku, je pryč. Nastal čas, kdy zákazníci volí kvalitu i na úkor vyšší ceny pořízení. Nejen technicky založení lidé, po desítkách let v denním kontaktu s počítači a softwarovými systémy, už ví, že sebemenší softwarová chyba, která nebyla při testování odhalena a ve výsledném produktu se dostane k zákazníkům, může mít v budoucnu obrovské následky. Reálné příklady z minulosti ukazují případy, kdy softwarové chyby způsobily ztráty ve výši stovek milionů dolarů[1] nebo dokonce ztráty mnoha lidských životů[2].

Při vývoji softwarové aplikace dnes už ve většině případů není třeba psát veškerý programový kód na tzv. zelené louce. Daleko častěji si vývojáři pomáhají znovu použitím knihoven, které umožňují využít již napsaný zdrojový kód i v jiných programech, než pro které byly vytvořeny, čímž usnadňují a urychlují softwarový vývoj. Stejně jako samotné systémy se záplatují a opravují, tak i knihovny se aktualizují, případně vznikají nové a upravené verze, a hrozí riziko možné nekompatibility softwaru či některých jeho částí.

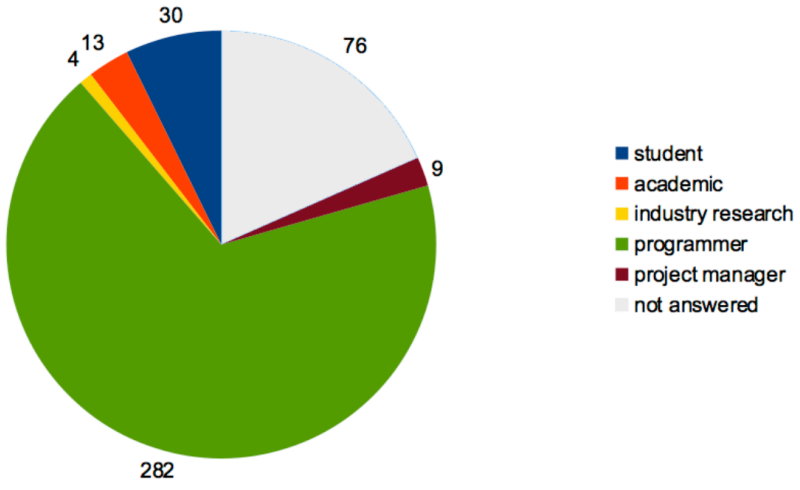
[1] Chyba v dělení s pohyblivou řádovou čárkou u procesorů Intel Pentium [1994].

[2] Problém u přistávacího modulu NASA na Mars [1999].

1.2. MOTIVACE

Dle několika odborných studií [1] je trendem vývoje softwaru v programovacím jazyce Java je používání knihoven třetích stran, které jsou často neustále vyvíjeny a vychází jejich nové verze. Pravidla pro programovací jazyk Java jsou definována v *Java Language Specification*[3] a *Java Virtual Machine Specification*[4], kde jsou ustálena pravidla pro bezpečný vývoj knihoven. Tyto pravidla rozlišují mezi celkem třemi hlavními typy kompatibility - zdrojovou, binární a kompatibilitou chování. Zatímco zdrojová kompatibilita se týká překladu Java zdrojového kódu do souborů tříd (class), binární kompatibilita je konkrétněji definována v *JLS* jako schopnost propojovat (link) bez výskytu chyby. Kompatibilita chování zahrnuje sémantiku kódu, která je provedena za běhu. [2]

V průzkumu, který byl proveden v rámci studie [1], byl vytvořen odborný dotazník, který se ověřoval znalosti vývojářů ohledně typů kompatibility. Celkem odpovědělo 414 vývojářů, z nichž 338 vyplnilo i svůj technický background. Tento výsledek ukázal, že 83,43% z těchto respondentů byli programátoři. Jak sami autoři studie uvádějí, nejednalo se o žádný studentský průzkum, jelikož samotných studentů bylo zastoupeno pouze 30 (8,88%). Rozložení všech respondentů je zobrazeno na grafu 1.

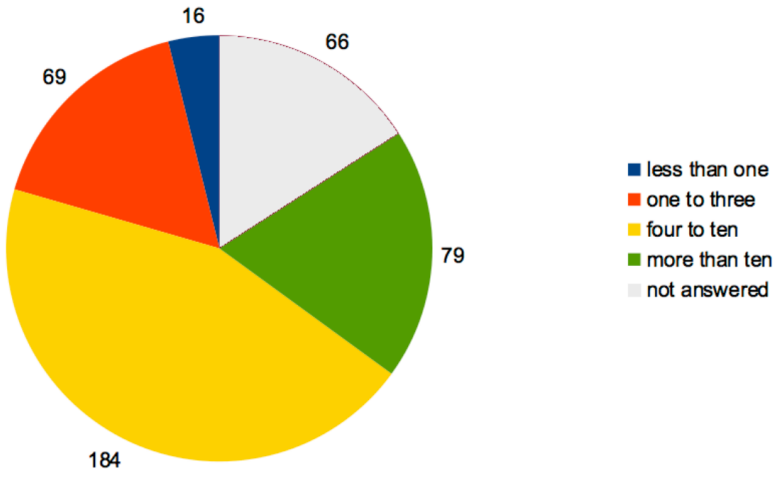


graf 1 - rozložení respondentů dotazníku týkajícího se znalostí kompatibility [1]

[3] Java Language Specification - https://docs.oracle.com/javase/specs/

[4] Java Virtual Machine Specification - https://docs.oracle.com/javase/specs/jvms/se7/html/

Podle vyplněných odpovědí bylo možné také rozlišit respondenty podle jejich zkušeností s programovacím jazykem Java a souvisejícími technologiemi. Graf 2 vizualizuje sumy respondentů podle počtu let zkušeností s Java technologiemi.



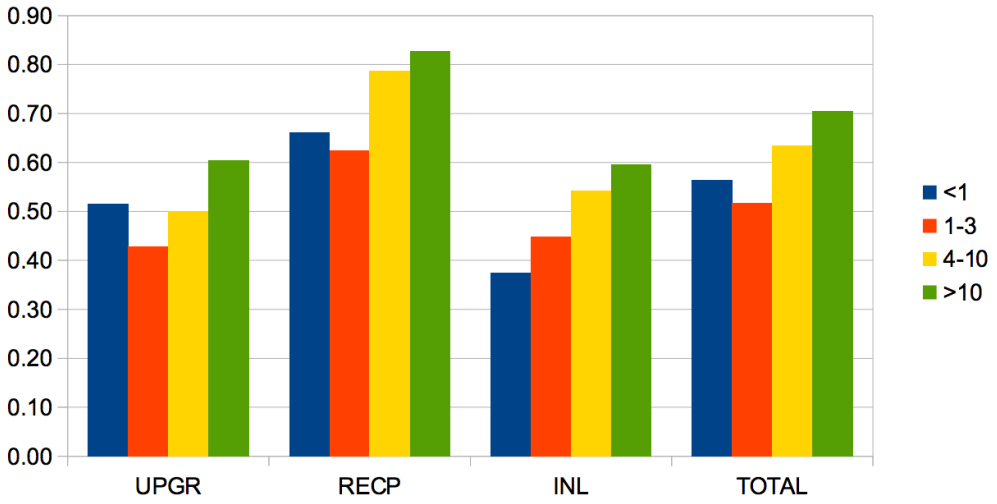
graf 2 - vizualizace respondentů podle zkušeností s Javou v letech [1]

Otázky v dotazníku byly rozdělené do několika kategorií, aby pokryly co nejširší oblast znalostí týkajících se různých typů kompatibility. Autoři pro rozlišení otázek volili zkratkovitá označení, které jednoznačně znázorňovalo kategorii, na jaké znalosti byla otázka zaměřena. Jednotlivé typy označen dále obsahovaly své specifické podkategorie otázek. Použitá označení pro jednotlivé skupiny otázek byla:

UPGR - zaměřeno na otázky týkající se dynamické aktualizace knihoven (dynamic upgrade),
RECP - zaměřeno na otázky tykající se rekompilace a aktualizace knihoven (recompile and upgrade),
INL - zaměřeno na otázky týkající se vkládání konstant (constant inlining).

Výsledky průzkumu ukázaly, že reprezentující respondenti odpověděli pouze na 62% otázek týkající se kompatibility správně. Mezi jednotlivými skupinami otázek byly značné rozdíly ve správnosti odpovědí. Skupina otázek UPGR byla odpovězena správně v 51% případů, RECP v 76% a INL v 52%. Z těchto

výsledků autoři vyhodnotili, že je daleko větší znalost týkající se zdrojové kompatibility než v případě binární a kompatibilitě chování. Na základě rozlišení odpovídajících respondentů bylo možné kategorizovat jednotlivé skupiny podle délky zkušeností s Java technologiemi a graficky vizualizovat jejich úspěšnost odpovědí (graf 3).



graf 3 - vizualizace správnosti odpovědí mezi skupinami respondentů dle délky zkušeností

Z grafu 3 je patrné, že délka zkušeností s Java technologiemi představuje významnou roli ve znalostech týkající se znalostí různých typů kompatibility. Experti a guru ze zkušenostmi delšími než 10 let dosáhli v průzkumu nejlepších výsledků. Naopak nováčci a mírně pokročilí v dané oblasti měli nejnižší procento správných odpovědí skrze všechny skupiny otázek. Je nutné dodat, že reprezentující skupina respondentů byla zastoupena z velké části experty s bohatými programátorskými zkušenosti, proto se předpokládá, že by výsledek průzkumu skrze širší komunitu vývojářů ukázal ještě nižší procento správných odpovědí.

Závěr studie se zaměřuje na několik faktů. Jedním z nich je, že kromě potřeby lepšího vzdělání programátorů, které poskytují univerzity s programátorskými obory a také publikace věnující se dané problematice. Další stanovisko říká, že je zapotřebí lepších nástrojů kontrolující konzistenci softwarových sestavení, což zahrnuje vnitřní návaznosti knihoven a jejich rozhraní (API). Takové nástroje je možné integrovat do nástrojů sestavení vytvářející výsledné

automatizované sestavy. Představitelem takového nástroje je například *Maven,* který pomocí pluginů dokáže v průběhu sestavení provést statickou analýzu byte kódu.

1.3. Cíl práce

Cílem práce je popsat aktuální situaci na trhu týkající se nástrojů pro ověřování kompatibility a kvality softwaru. Nástroje jsou analyzovány z technického pohledu zabývající se funkcemi, možnosti integrace nebo principem průběhu testování. Zároveň je na nástroje nahlíženo částečně i z pohledu obchodního, kde jsou uvedené informace týkající se podpory autorů, velikosti komunity nebo celkových stažení nástroje. Analýza se snaží shrnout možnosti zařazení nástrojů do procesu vývoje, respektive testování.

2. Životní cyklus vývoje softwaru

Softwarové inženýrství je oproti ostatních inženýrským oborům velice mladé odvětví. Až přibližně od poloviny 60.let minulého století se začínají vytvářet první počítačové programy a utváří se pojem softwarové inženýrství. [3] I přesto, že se od té doby vytvořilo mnoho postupů, modelů a metodik jakým způsobem by se měl software vyvíjet, stále je softwarové inženýrství částečně považované za exaktní vědu. Její všechny části nejsou jasně dané a je nutné je přizpůsobovat požadavkům okolního světa. I proto je potřeba při vývoji nahlížet na každý produkt jako na unikát a přizpůsobovat vývojový proces tak, aby se dosáhlo předem stanovených cílů.

Vývoj jakéhokoliv softwaru probíhá v určitých vývojových etapách, jejichž sled a celkový počet závisí na mnoha parametrech. Těmi může být účel použití, cena, termín dodání, typ koncových uživatelů a mnoho dalšího. Jedním z nejpodstatnějších parametrů v ohledu na životní cyklus a model vývoje je rozsáhlost softwaru. Dnes jsou používané systémy tak komplexní, že je v celém procesu vývoje potřeba vícero samostatných etap a důsledná kooperace několika vývojových rolí. Mezi nejznámější role patří patří projektový manažer, architekt, analytik, programátor, a tester, ale existuje jich mnohem více.

Dnes existuje velice široké spektrum modelů a jednotlivých metodik, podle kterých je možné vyvíjet software. Tato kapitola se zabývá pouze některými z nich a popisuje jejich důraz kladen na testování softwaru, což je disciplína, na kterou se tato práce nejvíce zaměřuje.

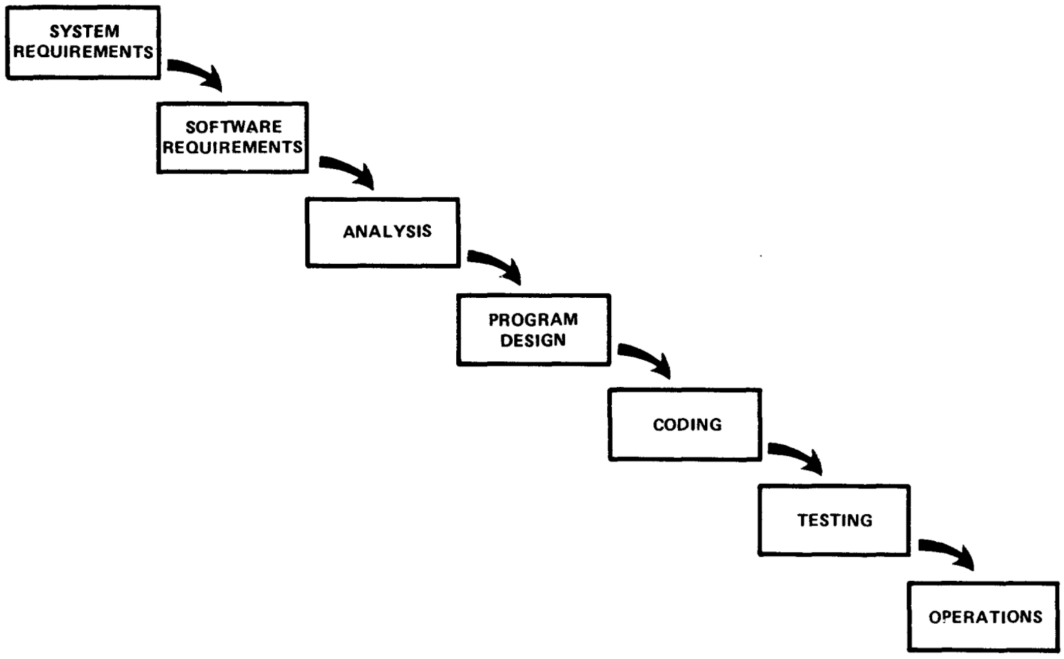
2.1. METODIKY VÝVOJE SOFTWARU

Každá metodika vývoje softwaru popisuje vztahy mezi jednotlivými etapami životního cyklu softwaru a obsahuje procesy, jejichž postupné provedení umožní vyvinout softwarový produkt.

Proto výběr správného metodiky je jedním z počátečních kroků při vývoji libovolného softwaru. Ta musí nejlépe splňovat kritéria pro celkový průběh procesu vývoje softwaru a zahrnovat všechny potřebné etapy. Této problematice se věnuje mnoho odborné literatury. James Chapman se o výběru metodiky zmiňuje ve své knize [4]: „*Složitým problémem při výběru a dodržování metodiky je činit tak s rozumem – poskytnout dostatek procesních disciplín k zajištění kvality vedoucí k obchodnímu úspěchu, ale zároveň se vyvarovat kroků, které představují ztrátu času, snižují produktivitu, demoralizují vývojáře a vytvářejí nepotřebnou administrativu*.“

2.1.1. VODOPÁDOVÝ MODEL

Vodopádový model patří mezi jeden z nejstarších modelů životního cyklu vývoje softwaru a označuje se jako sekvenční vývojový model. Pojmenování vychází z přirovnání posloupnosti jednotlivých etap protékání vody vodopádem. Autorem modelu je označován Winston W. Royce [5], jehož znázornění je vidět na obrázku 1.



obr.1 - Fáze vodopádového modelu pro vývoj softwaru dle W. Winstona Royce

Vodopádový model klade největší důraz na počáteční specifikaci požadavků. Vychází z předpokladu postupné návaznosti vývojových fází, které je třeba splnit, aby se mohlo přejít na další fázi. Je nutné provést každou fázi s maximálním úsilím, protože není možné se v etapách vracet zpět. Proto je nutné jakékoliv nedostatky objevit v průběhu procesu dané fáze a zde je také opravit. Objevení závažné chyby v pokročilé fázi vývoje může znamenat opakování všech fází vývoje, což je velice neefektivní.
Model se využívá u softwarových projektů, kde se věnuje dostatečně dlouhý čas prvotní analýze specifikací. Po jejím vytvoření se předpokládá, že se specifikace požadavků v průběhu vývoje nebude měnit. Tento přístup se označuje jako preskriptivní. Taková metodika vývoje se uplatňuje především u vládních projektů, kde jsou změny v průběhu vývoje nežádoucí. Zajímavostí je, že sám autor ve své publikaci model označuje jako „riskantní a odsouzen k neúspěchu", právě z důvodu nemožnosti se vracet zpět k již provedeným fázím. I přesto je tento model využíván při vývoji softwaru a často velké množství literatury ho označuje jako základní.

Sekvenční fáze vodopádového modelu jsou reprezentovány sedmi po sobě jdoucími fázemi, jejichž účel je popsán níže.

Systémové požadavky
Vstupní fáze modelu slouží pro specifikaci systémových požadavků, které definuje především zákazník. Zahrnuje systémovou konfiguraci, na které software bude provozován.

Softwarové požadavky
Druhá specifikační fáze, slouží pro co nejpřesnější popisu, co má software dělat a jakým způsobem. Opět důraz kladen na přesné pochopení zákazníkových požadavků.

Analýza
V analýze požadavků se přesně a jasně zdokumentují požadavky zákazníka, které je potřeba splnit. Výsledkem je dokument analýzy specifikace.
Návrh programu

Návrh probíhá na základě dokumentu vytvořeného v předchozí fázi. Rozložením zákazníkových požadavků se vytvoří jednotlivé logické moduly a jejich návaznosti. V této fázi je jasně definována softwarová architektura, datové struktury i detailně popsané algoritmy a metody použité v jednotlivých logických modulech.

Implementace

Fáze zabývající se samotným vytvářením zdrojového kódu, kdy dochází k přepsání návrhu do strojově čitelného kódu. Výsledkem je program obsahující funkčnost definovanou ve specifikaci požadavků

Testování

V ideálním případě by se v této fázi měli verifikovat a validovat všechny zákazníkovy požadavky a zajistit, že program neobsahuje žádné chyby. Testování zahrnuje jak hardwarovou i softwarovou část.

Provoz

Závěrečná fáze vodopádového modelu. Výsledný otestovaný program se předává zákazníkovi a nasazuje se do provozu. Do této fáze patří i podpora a údržba systému, která se zpravidla definuje v dokumentu specifikace požadavků.

Výhodou modelu je, že je snadno srozumitelný a využívá se k jednoduché demonstraci návaznosti důležitých etap pro vývoj rozsáhlejšího softwaru. Jednotlivé etapy na sebe kontinuálně navazují, což v praxi znamená, že ukončením jedné etapy přecházíme do etapy následující. Z tohoto pravidla vyplývá i největší nevýhoda vodopádového modelu. Často se při vývoji stává, že je nutné reagovat na změny a tím dochází ke změně specifikace.

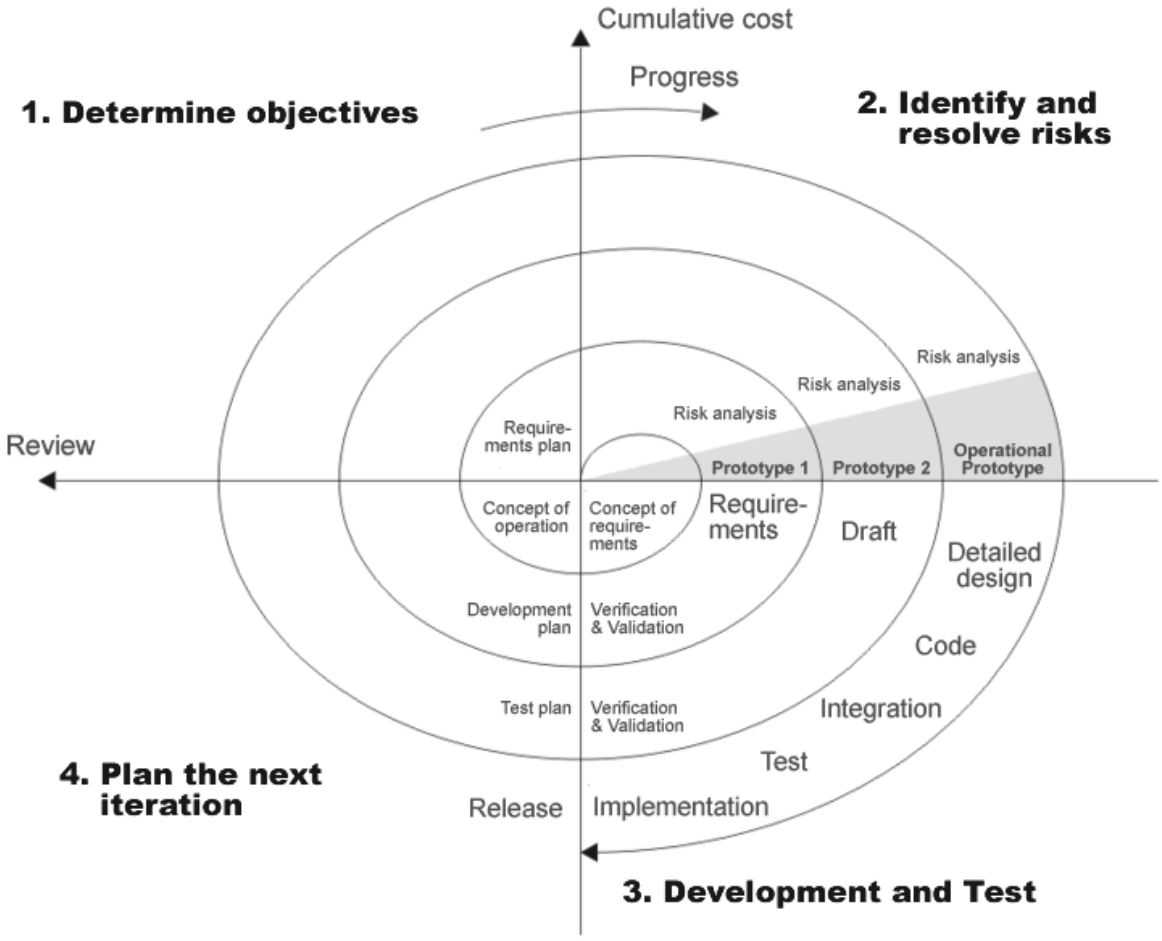
VLIV TESTOVÁNÍ

Fáze testování se u vodopádového modelu nachází téměř na samém konci životního cyklu softwaru. Znamená to, že automatické či ruční testy probíhají až po úplném dokončení implementace, což patří mezi velmi slabé stránky tohoto modelu. Tímto způsobem není možné odhalit chyby vznikající v průběhu vývoje.

Trendem ve vývoji softwaru je pokrýt testy celý proces vývoje a nápravou nalezenými chybami vytvářet kvalitnější software.

2.1.2. SPIRÁLOVÝ MODEL

Dobře známý spirálový model částečně vychází z vodopádového modelu, ale napravuje jeho největší nedostatky pomocí opakujících se fází, dokud nedojde k dokončení produktu. Model byl definován v roce 1986 v článku [6], autorem je Barry Boehm a grafické znázornění cyklických etap je vidět na obrázku 2.



obr. 2 - spirálový model vývoje softwaru podle B. Boehma [6]

Model je rozdělen do čtyř cyklicky opakujících se hlavních kvadrantů, prováděných dokud není softwarového dílo dokončeno dle požadavků zákazníka, respektive podle definované specifikace požadavků. Díky opakované analýze rizik je model vhodnější pro projekty, kde je velká pravděpodobnost úpravy specifikace požadavků v průběhu vývoje projektu.

Spirálový model je tvořen těmito opakujícími se fázemi:

Analýza - určení cílů

Počáteční fáze, obdobné specifikační fázi u vodopádového modelu. Zaměřeno na komunikaci se zákazníkem a pochopení jeho požadavků. Při první iteraci je vyhotovena hrubá specifikace požadavků a v každém dalším cyklu je specifikace detailněji specifikována konzultací se zákazníkem.

Hodnocení - identifikace a řešení rizik

V této fázi se analyzují všechny možná rizika, která mohou při vývoji nastat. Zároveň dochází k analýze možných řešení, která by byla aplikována v případně výskytu daných rizik. Dochází k tzv. prototypování, kdy přichází v úvahu více možných řešení a až následující etapy ukážou, který z prototypů je nejvhodnější.

Vývoj a testování

Fáze, při které dochází k vytváření samotného produktu. V prvních iteracích je vytvořen pouze návrh konceptu funkcí zaměřeným na ověření softwarových a uživatelských požadavků. V následujících iteračních cyklech se vytvoří návrh softwarového produktu a na důraz je kladen na verifikaci a validaci návrhu. Při dalších iteracích dochází k podrobnému návrhu, která zahrnuje implementaci, integraci, testování (unit testy, akceptační testy) a v případě dokončení produktu nasazení do provozu.

Plánování nadcházejících iterací

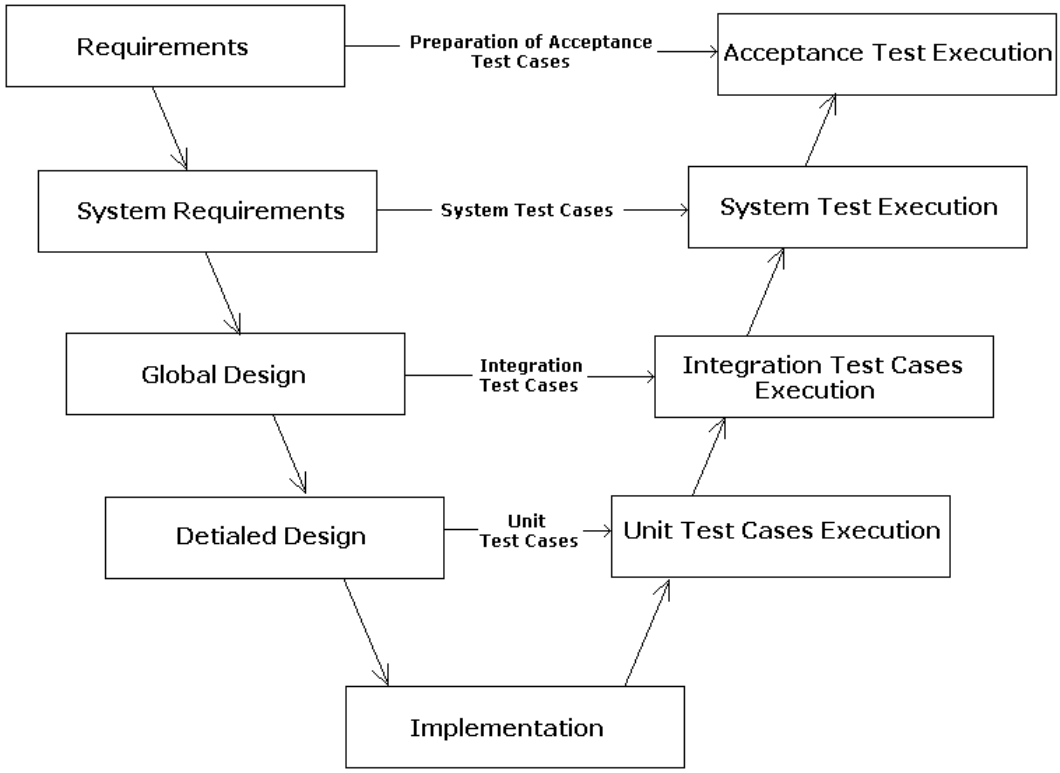
Tato fáze je tím nejvíce odlišujícím elementem od vodopádového modelu. Právě zde dochází k plánování dalšího cyklu a jednotlivých kroků, které se budou v jednotlivých fázích provádět. První iterace je zaměřena na plán požadavků a také na plán životního cyklu vývoje. Při dalších iteracích se hodnotí plán vývoje případně plány integrace a testování. Na základě těchto průběžných plánů se lépe identifikují cíle pro další iterace jednotlivých fází.

VLIV TESTOVÁNÍ

Proces testování je u spirálového modelu propracovanější, než u předešlého vodopádového modelu. Ačkoliv probíhá pouze v jednom kvadrantu vývoje, díky opakujícím se etapám se testování provádí vícekrát v průběhu vývoje softwaru. Důsledkem je možnost reagovat na chyby objevené při vývoji a se nabízí využití více typů testů. Nalézání chyb je efektivnější, ale zdaleka se nedá říci, že by bylo dokonalé. Proces testování je stále zafixován v daném kvadrantu a není možné při vývoji pomocí spirálového modelu testy provádět v jiném čase, než jsou plánovány.

2.1.3. V-MODEL

Vývojový model nazývaný V-model (obrázek 3) se skládá ze dvou hlavních částí. U sestupné levé části se postupuje klasickým vývojem - od sběru požadavků, specifikace systému, návrhu a architektury až po samotnou implementaci. V pravé vzestupné části pak probíhá dílčí testování k ověření odpovídající úrovně modelu.



obr. 3 - vývojové fáze V-modelu [7]

Jednotlivé fáze modelu vytváří typické písmeno „V” a úroveň, na které se nachází naznačuje vztah mezi vývojovou a testovací fází.
Základní myšlenkou tohoto přístupu je postup testování od malých částí aplikace, přes větší funkční celky, přes integraci komplexních fází až po otestování celého systému. Přechod mezi fázemi testování předpokládá úspěšné dokončení předchozí fáze.
Typy testování, které se typicky používají u V-modelu jsou Unit testy, systémové testy, integrační testy nebo akceptační testy. Jednotlivé typy testování jsou podrobně popsány v kapitole 2.3

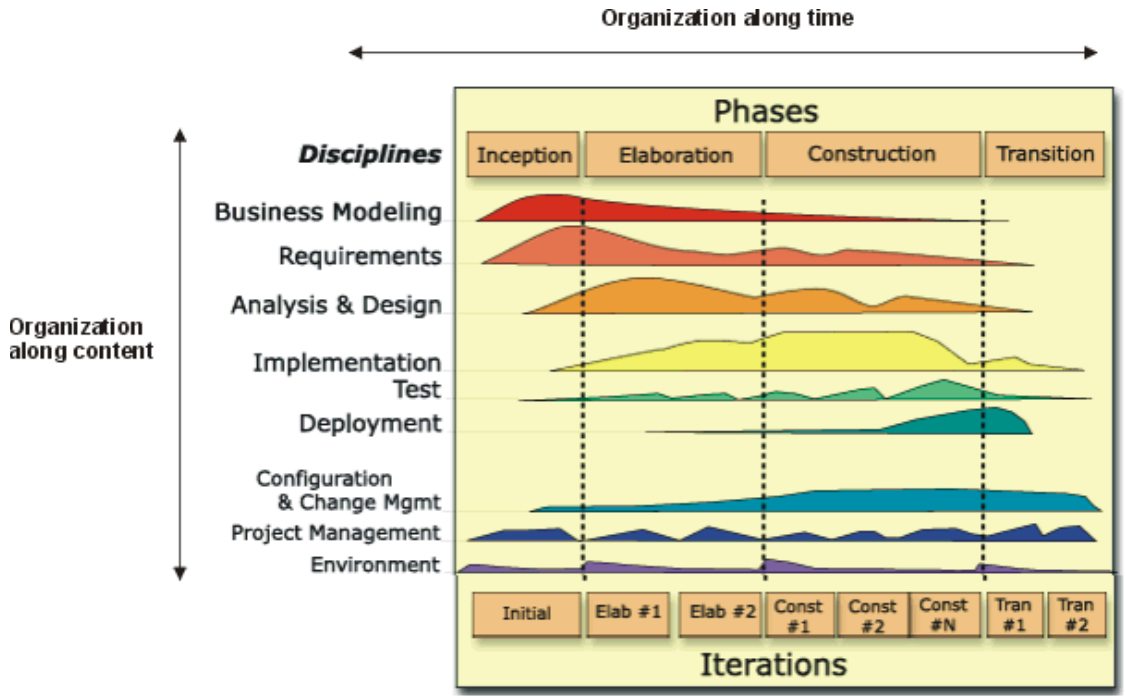
VLIV TESTOVÁNÍ

V-model je vývojovým modelem, který klade důraz na testování ve všech jednotlivých fází vývoje, čímž zajišťuje odhalení co největšího množství chyb. Díky tomu by výsledný softwarový produkt měl být co nejméně chybový a v ideálním případě by všechny závažné chyby měly být eliminovány. Testováním se u V-modelu zabývají všechny vývojové role zainteresované při vývoji.

2.1.4. RUP (RATIONAL UNIFIED PROCESS)

Metodika byla vytvořena společností Rational Software Corporation, dnes samostatná divize IBM. RUP je objektově orientovaný iterativní přístup založený na přístupu řízených případy užití (tzv. use case driven approach).
Základním prvkem modelu je případ užití (tzv. use case), který je definován jako posloupnost akcí prováděná systémem poskytující určitou hodnoty uživateli systému. Pro účely modelování je využíváno jazyka UML (Unified Modeling Language), který byl také vytvořen skupinou Rational Software. [8]

Vývoj softwaru podle RUP je rozdělen do čtyř hlavních fází a každá z fází obsahuje několik vlastních iterací (obrázek 4). Na konci každé fáze je tzv. milník (milestone), který musí být splněn, aby se mohlo přejít do další fáze. Milník je jednoznačně definovaný bod v čase vývoje, kdy je potřeba udělat zásadní rozhodnutí, aby bylo dosaženo klíčových cílů.



obr.4 - Struktura procesů v RUP metodice ve dvou dimenzích [8]

Zahájení (Inception) definuje účel projektu a jeho rozsah..Zaměřuje se také na obchodní model projektu, kde se definují kritéria úspěchu, posuzují rizika a analyzují potřebné zdroje. Dochází k popisu interakce všech subjektů, které mají se systémem komunikovat. Definují se data milníku.

Projektování (Elaboration) je považována jako nejvíce kritická fáze RUPu**.** Je nutné vypracovat výsledný plán projektu zahrnující analýzu rizikových domén a snažit se eliminovat rizikové prvky projektu. Definují se funkční a nefunkční požadavky a také požadavky na výkon.

Konstrukce (Construction) se zabývá samotným vývojem jednotlivých komponent systému a jejich integrace zajišťující výslednou funkčnost. Jedná se o časově nejdelší fázi modelu podle RUP. Výsledný produkt musí být připraven pro předání do rukou koncových uživatelů, ve většině případů se jedná o tzv. beta-release.

Nasazení (Transition) je závěrečná fáze sloužící pro ověření, zda-li je produkt připraven pro nasazení do provozu. Provádějí se úpravy na základně zpětné

vazby od koncových uživatelů. Patří sem i činnosti jako školení a podpora koncových uživatelů, Pokud je produkt dokončen a splňuje všechny požadavky dle specifikace je předán zákazníkovi.

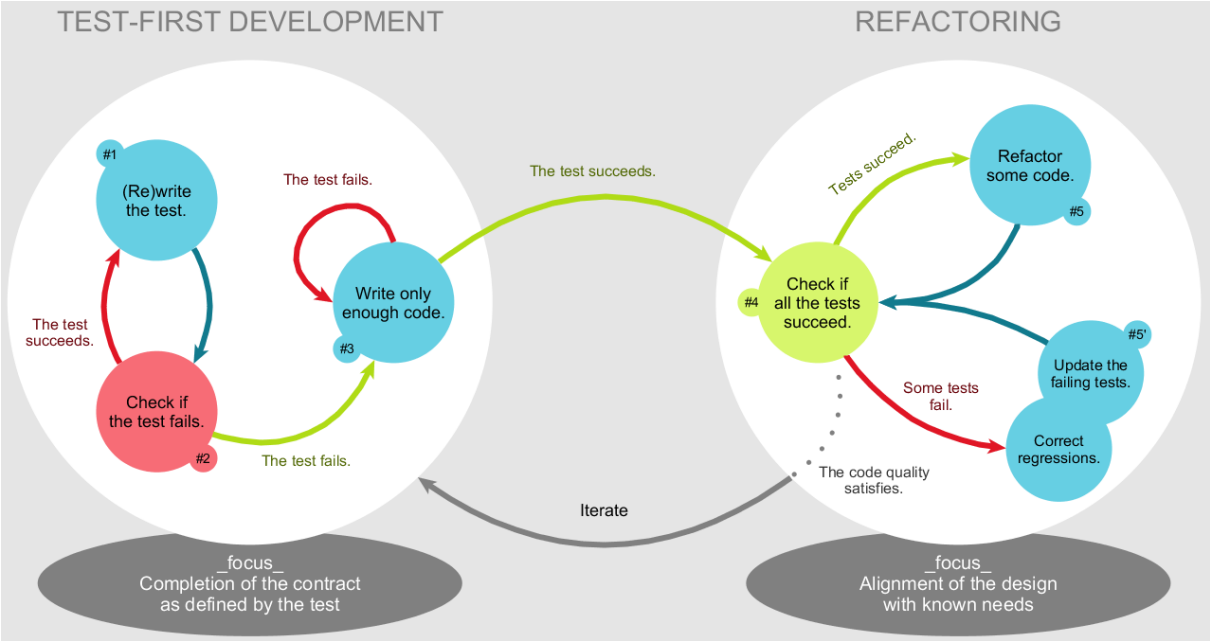
Metodika RUP je dnes často využívána u vývoje rozsáhlých projektů. Výhodou jsou definované milníky, kterých je potřeba docílit při přechodu jednotlivých fází. Oproti ostatním modelům také definuje role a aktivity, které je nutné provádět pro testování v jednotlivých etapách vývoje softwaru.

VLIV TESTOVÁNÍ

RUP je pokročilá metodika, která proces testování zahrnuje ve všech svých etapách vývoje. Počáteční testy specifikace nejsou tolik časově náročné jako u stěžejních etap projektování a konstrukce, kde je věnováno testování nejvíce času. Po nasazení produktu se očekává, že software je již kompletně otestován a poté dochází průběžně k opravám menších nedostatků, na které se přijde až v provozu.

2.1.5. VÝVOJ ŘÍZENÝ TESTY

Název pochází z anglického názvu Test-driven development (TDD) a popisuje metodu vývoje softwaru založenou na prvotní přípravě automatizovaných testů a až následného psaní zdrojového kódu, který musí danými testy úspěšně projít. Metoda vychází z opakování malého vývojového cyklu, kde výsledný softwarový produkt vzniká postupným psaním testů a následně implementací pouze nezbytného zdrojového kódu pro úspěšný výsledek testu. Proces při vývoji SW řízeného testy je znázorněn na obrázku 5.



obr. 5 grafická reprezentace životního cyklu vývoje softwaru pomocí TDD [9]

Postup sekvence vývoje pomocí vývoje řízeného testy popsal ve své knize K. Beck [10]. Následující popis jednotlivých fází je volný překlad z jeho knihy zabývající se tímto vývojovým modelem.

Přidání testu

Každá nová funkce začíná napsáním testu. Je nutné, aby vývojář testu velice jasně rozuměl požadavkům a přesně specifikaci nové funkce. Často se používají modely použití (user stories) pro znázornění všech požadavků, včetně možných výjimek. Nemusí vždy docházet ke psaní nového testu, nezřídka se využívá přepisování již existujících testů.

Spuštění všech testů a zjištění, zda-li nový test selhal

Tato fáze validuje, zda-li byl nový test napsán správně. Pokud by došlo k situaci, že nový test úspěšně prošel, aniž by byl napsán nový kus kódu, značí to chybu v testu a je nutné test předělat. V případě, že testujeme novou funkci, která ještě v kódu není implementována, test musí selhat. Tímto si vývojář či softwarový tester ověřuje správnost testu.

Napsání nezbytného kódu

Pokud po spuštění testu dochází k selhání, nastává fáze opravování kódu. Ten slouží k tomu, aby po znovu otestování test vždy prošel. Kód vytvořený v této fázi není zdaleka finální a nedá se považovat za dokonalý. Toto je chtěný stav, jelikož kód bude upravován v budoucích fázích. Jediným cílem této fáze je úspěšný výsledek testů nad tímto kódem.

Nové spuštění testů

Nyní je potřeba, aby spuštěné testy vždy prošly. V případě, že test selže, je nutné upravit kód, znovu ho otestovat, dokud nebude testování úspěšné. Když je splněna tato podmínka, je možné přejít na další fázi.

Refaktorizace kódu

S průběhem vývoje zdrojový kód softwaru roste a je nutné ho upravovat do výsledné podoby. K tomu slouží tato fáze, kde dochází k refaktorizaci kódu. Odstraňují se duplicity, kód se přesouvá, tak aby logika zdrojového kódu dávala smysl. Výsledkem by měl lépe čitelný a čistí, do kterého se budou snadněji implementovat nové funkce.

Opakování pro další iteraci

Pokud software není ve finálním stavu a je potřeba přidat další funkce, opakují se všechny předešlé fáze ve stejném pořadí. Každým cyklem se přidává funkce do systému a velikost kódu roste. Důležité je udržet velikost kroků mezi testováním dostatečně malou, doporučuje se 1-10 úprav mezi každým testováním. V případě, že nový kód nesplňuje požadavky napsaných testů nebo dochází k neočekávanému selhání předchozích testů, vývojář má možnost

vrátit kód do předešlého stavu. Princip postupné integrace, poskytuje zpětně dostupné záchytné body vývoje, kde testy úspěšně proběhly.

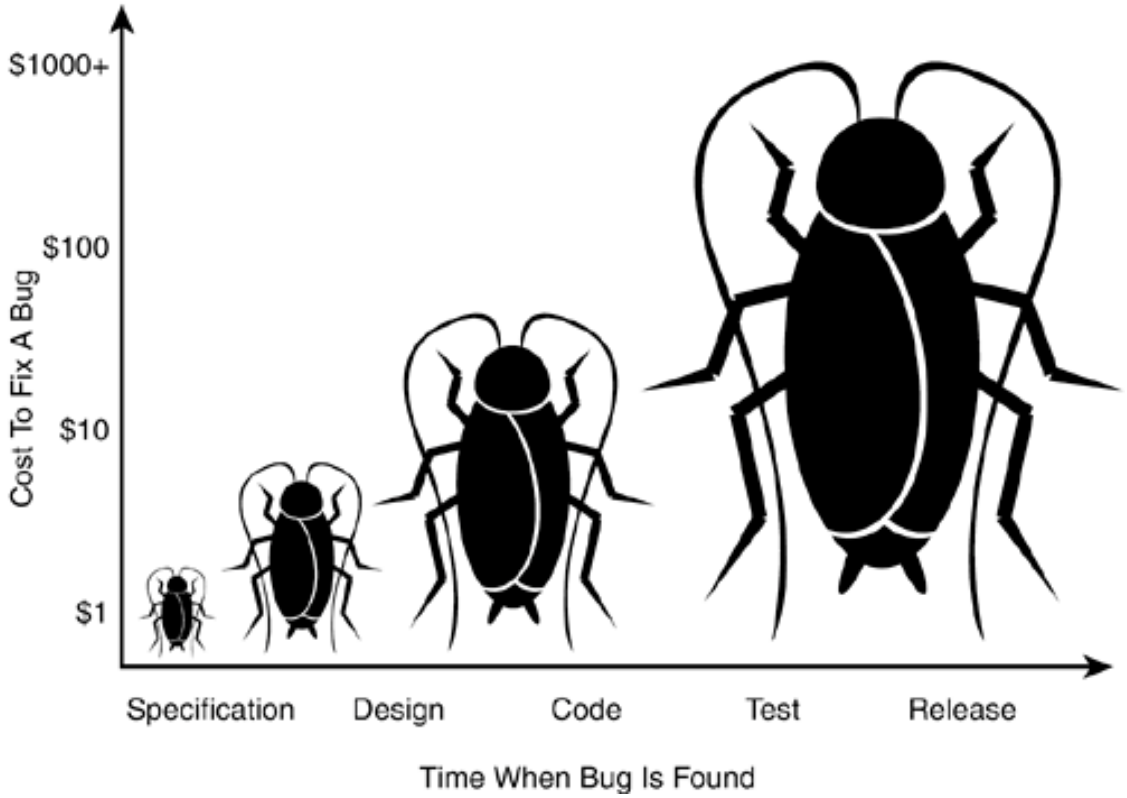
VLIV TESTOVÁNÍ

Testování je proces, na kterém je tento typ vývoje založen a bez kterého by nemohlo být dosaženu výsledku v podobě kvalitního softwarového produktu. Nevýhodou modelu je, že může být použit pouze pro vývoj určitých systémů, u kterých je požadovaného relativně pomalého vývoje s otestováním každé přidané funkce. Při použití vývoje řízeného testy je výhodou jistota úspěšného otestování celého zdrojového kódu pomocí předem napsaných automatizovaných testů.

3. TESTOVÁNÍ SOFTWARU

3.1. ÚVOD DO TESTOVÁNÍ

Dělání chyb je lidská vlastnost a musí s ní být počítáno ve všech oblastech průmyslu, včetně toho softwarového. Pro eliminaci chyb v životním cyklu vývoje softwaru se využívá testování. Existuje mnoho pohledů a mnoho různých druhů testování lišících se například v čase provedení, způsobem provedení, přístupem k testovanému kódu, testující části softwaru apod. Všechny druhy testování mají společný cíl - ověřit kvalitu softwaru pomocí nalézání chyb.
Jelikož je vývoj komplexních systémů proces trvající týdny, měsíce i roky, rozhodujícím faktorem při odhalování chyb je okamžik nalezení chyby. Exponenciální růst nákladů je ilustrován na obrázku 5, ze kterého jasně vyplývá, čím déle je chyba odhalena, tím větší musíme vynaložit náklady na její opravení.



obr. 5 - znázornění nákladů při odhalení chyby v SW vývoji [11]

3.2. TESTOVÁNÍ

Testování je velice obsáhlá disciplína, která zahrnuje mnoho činností v průběhu celého životního cyklu vývoje softwaru. Optimálně by mělo testování probíhat skrze všechny etapy vývoje softwaru, což zahrnuje činnosti několika projektových rolí - softwarových testerů, vývojářů i analytiků.
Mnozí by si mohli myslet, že cílem testování je zbavit software veškerých softwarových chyb. Tak to ale není a ani býti nemůže. Abych toto tvrzení mohl dokázat, je nutné znát definici softwarové chyby. Patton [11] uvedl 5 pravidel pomocí kterých je možné bezpečně identifikovat všechny problémy:

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 nedělá 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.

Důležitým pojmem, který zde Patton používá je specifikace produktu. Tato specifikace vyjadřuje dohodu mezi vývojovým týmem daného softwaru. V ní jsou zahrnuté všechny funkční požadavky zákazníka, které samy o sobě nijak nepopisují výsledný produkt, pouze určují zda-li se má něco vytvořit a jaké funkce zákazník požaduje. Softwarová specifikace všechny tyto informace přebírá a přidává k nim všechny nepsané, ale přesto nezbytné požadavky pro vytvoření softwarového produktu. Výsledkem je, že ze specifikace je zřejmé jak produkt bude vypadat a co bude dělat.

„Cílem softwarového testera je vyhledávat chyby, vyhledat je co nejdříve a zajistit jejich nápravu.” tvrdí Robert Patton ve své knize“.

3.3. Typy testů

Jak již bylo zmíněno, při vývoji softwaru může být testování prováděno mnoha různými druhy testů na odlišných úrovních vývoje. Testy se liší svým významem a zaměřením, co konkrétně testují. Podle této charakteristiky lze konkrétní typy testů doporučit do vhodných etap vývoje softwaru. V této kapitole jsou zmíněné nejčastěji využívané druhy testů pro programovací jazyk Java.

Testy se dají dělit dle mnoha kritérií, autor čerpal ze zdroje věnující se problematice testování od T. Hlavy [12]. Základní rozdělením je dle úhlu pohledu, ze kterého softwarový tester či jiná osoba provádějící testy nahlížejí na testovanou aplikaci. Následující podkapitoly popisují rozdělení dle tohoto kritéria.

White-box (testování bílé skříňky)

Vychází z předpokladu znalosti vnitřní architektury testované aplikace, jedná se tedy o testování z pohledu zdrojového kódu. Tato vlastnost umožňuje otestovat jednotlivé procedury, třídy, funkce, metody apod. Pomocí tohoto testování je možné otestovat všechny průchody kódem nebo otestovat chování aplikace zadáváním neočekávaných vstupních hodnot. Pro provedení white-box testování je důležitá dobrá čitelnost a přehlednost kódu, čímž se myslí správně strukturovaný a okomentovaný kód.

Black-box (testování černé skříňky)

U tohoto testování nemá tester žádnou znalost vnitřní architektury a nezná vnitřní procesy aplikace. Jedinou znalostí jsou vstupní a odpovídající výstupní hodnoty, jejichž správnost je testována. Lze říct, že tento typ představuje testování z pohledu koncového zákazníka. Tester nemusí mít žádnou znalost daného programovacího jazyka, ale ve většině případů je nákladné či téměř nemožné touto metodou otestovat všechny reálné stavy aplikace.

Dalším kritériem, podle kterého lze dělit testy je dle fáze vývoje, ve které se dané druhy testování provádí. Mezi nejvíce používané druhy testů dělíme testy na:

3.3.1. UNIT TESTY

Cílem jednotkových (unit) testů je ověření, zda-li nejmenší kompilovatelné komponenty (například Java třídy) fungují správně.
Typicky jde o testy jednotlivých komponent aplikace na úrovni modulů, objektů a tříd. Tento druh testování často provádějí vývojáři. Těmito testy je ověřováno, zda nová nebo změněná část kódu funguje (tedy nepadá do chyby), a že její funkce odpovídá očekávání. Jedním z nejznámějších nástrojem pro jednotkové testování v Javě je open source framework *jUnit*. Ten je také klíčovou součástí vývoje řízeného testy, kde k tvorbě testu dochází obvykle před tvorbou samotného programu (viz kapitola 1.1.5).

3.3.2. ASSEMBLY TESTY

Úkolem assembly testů je ověřit, že jednotlivé části kódu, testované v rámci unit testů, je možné sestavit (assembly) do funkčního celku. Jde tedy o první test integrace jednotlivých nejmenších komponent. Stejně jako u předchozího jednotkového testování jsou i assembly zpravidla prováděny vývojáři.

3.3.3. INTEGRAČNÍ TESTY

Dnes jedny z nejdůležitějších testů, jelikož převážná většina softwaru využívá modulární strukturu. Cílem integračních testů je ověřit, že spolu všechny komponenty aplikace komunikují a spolupracují, jak bylo specifikováno v požadavcích. Kromě zmíněného lze integraci testovat i mezi komponentou a operačním systémem, ovladačem hardwaru nebo rozhraním dalšího systému. U rozsáhlých vývojových projektů můžou být tyto testy rozděleny na vnitřní a vnější testování. V takovém případě jsou vnitřní integrační testy zaměřeny na testování komponent uvnitř aplikace a vnější integrační testy mohou

ověřovat správnost integrace aplikace s ostatním softwarem, operačním systémem nebo například s konkrétními ovladači. Provedení těchto testů už obvykle není prováděno vývojáři, ale častěji spíše specializovaným softwarovým testerem případně testovacím týmem.

3.3.4. Systémové testy

Systémové testování předchází integračním testům, ale v mnoha případech jsou tyto testy spojeny do jednoho typu. Během systémových testů je ověřována funkčnost aplikace jako celku. Pomocí předem připravených scénářů se simulují možné kroky, které mohou v aplikaci nastat. Testuje se, že software plní úlohu, pro kterou byl vyvinut, že vrací správné výstupy, že byly ošetřeny všechny nestandardní situace a v klade se důraz na to, aby byly pokryty všechny požadavky ze strany zákazníka. Systémové testování probíhá v několika iteracích a při každé z nich je nutné nalezené chyby nahlásit, následně opravit a software znovu otestovat.

3.3.5. Smoke testy

Jedná se o krátké, rychlé a často plně automatizované testy, které lze provádět až po splnění integračních testů. Testy slouží k ověření, že vyvíjený software je připraven pro další fázi testování. Rozsah testů je značně menší oproti ostatním typům testování. Jde o rychlé testy, obvykle obsahující jednoduchý "průchod" skrz aplikaci, které dokáží ověřit všechny zmíněné atributy. Tyto testy jsou prováděny softwarovými testery. Výhodou smoke testů je, že nejsou časově náročné a jejich pravidelným spouštěním (automatizace) v průběhu vývoje se lze ujistit, že přidaný kód nijak neovlivnil žádným způsobem funkčnost aplikace ani ostatní její části, které by ovlivňovaly očekávaný správný výstup.

3.3.6. Akceptační testy

Hlavní testy z pohledu předání softwaru zákazníkovi. Po úspěšném průběhu provedení předchozích testů je možné zahájit akceptační testy, jejich provedení je nezbytné pro předání produktu. Testují, že software splňuje všechny požadavky specifikované zákazníkem v úvodní etapě vývoje. K testování

dochází v prostředí zákazníka, mnohdy s jeho vlastním testovacím týmem, podle předem připravených scénářů. Pokud se v průběhu akceptačních testů vyskytnou nějaké problémy se softwarem, je nutné reportovat chyby vývojovému týmu a zajistit rychlou opravu. Velmi důležitým faktorem je zajistit, aby prodleva mezi nalezením chyby, opravením a dodáním opravené verze aplikace zákazníkovi, byla co nejmenší.

3.3.7. Testy po akceptaci

U velkého počtu softwarových projektů jsou akceptační testy poslední testovací fází před předáním produktu zákazníkovi. Ale nemusí to být pravidlem. Po dokončení vývoje softwaru zákazník může požadovat zajištění podpory pro zakoupený software, což představuje další kategorii testování. Tato podpora představuje, kromě opravování chyb objevených až při provozu, také případné přidání nových funkcí dle požadavků zákazníka, které představují další provádění již dříve zmíněných testů. Tato praktika se využívá u softwarových projektů, kdy nebylo možné reálně a efektivně otestovat ve vývojovém prostředí všechny stavy aplikace. V takovém případě se zákazníkovi odevzdá celý produkt s tím, že část je otestovaná pouze tak, jak bylo možné z vývojového prostředí a počítá se s tím, že bude docházet k opravám nalezených chyb z provozu. Rozšíření funkčnosti aplikace s sebou přináší provedení dalších testů.

U každého konkrétního softwarového projektu je potřeba zvážit ekonomický pohled na vývoj, zda-li se investovaný čas a lidské práce promítne pozitivně na přínosech pro zákazníka. V opačném případě takové úpravy nemají smysl. Proto se také stává, že aplikace obsahuje určitou chybu, o které zákazník ví, ale neovlivňuje nijak jeho aktivity a proto se taková chyba neopravuje.

3.3.8. Progresní testy

Jsou využívány ke kontrole nových funkcí nebo vlastností aplikace. Jsou silně závislé na dobré dokumentaci, která popisuje nově implementované části softwaru. Často využívané u inkrementálních metodik vývoje, například u vývoje řízeného testy.

3.3.9. Regresní testy

Regresní testy naopak ověřují, že novými zásahy do softwaru nebyla narušena funkčnost již dříve vytvořených částí, které změnami neměly být ovlivněny. Jinými slovy se testuje, že oprava chyby nebo přidání nové funkčnosti nezpůsobily novou chybu v již funkčních částech aplikace. Regresní testy se často automatizují, protože u nich jde o opakované provádění stejných operací se známým výsledkem. Jejich cílem je tak zjistit, zda neexistují odchylky mezi výstupem získaným před zásahem do aplikace a výstupem po tomto zásahu. Jelikož u softwarových projektů dochází často k zásahům do aplikace, jsou regresní testy v praxi velice často využívány.

3.3.10. Shrnutí typů testů

Je potřeba zmínit, že všechny výše uvedené druhy testů nemusejí být nutně zastoupeny v životním cyklu vývoje softwaru i přesto, že daná metodika klade na proces testování důraz. Vždy záleží na aktuálním vyvíjeném produktu a především na požadavcích zákazníka, kterému se celý proces plánování přizpůsobuje. Důležité je vybrat vhodnou vývojovou metodiku a zahrnout optimální kombinaci testů, aby vývoj byl efektivní - nákladově i kvalitativně. Vhodný výběr těchto kritérii je otázkou především zkušeností vývojového týmu.

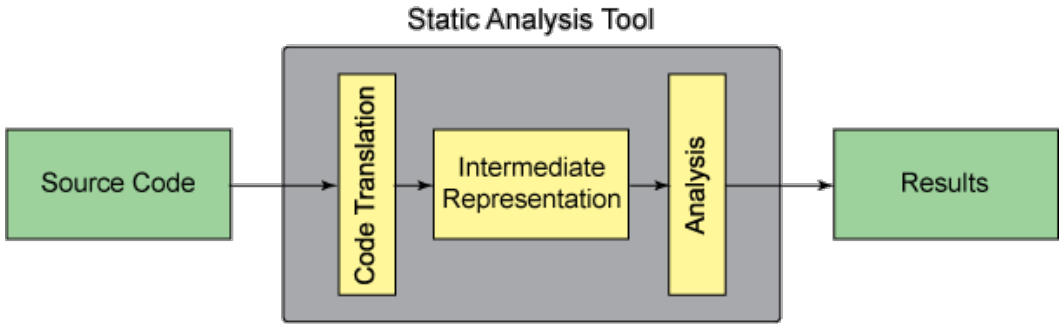
4. STATICKÁ ANALÝZA KÓDU

Kromě klasického testování zmíněného v předchozí kapitole, se velice často provádí tzv. statická analýza kódu (zkráceně SA). SA znamená strukturální analýzu kódu softwaru bez nutnosti jeho běhu. Analýza probíhá zpravidla nad zdrojovým kódem, případně nad objektovým kódem po přeložení, kterému se říká v Javě bytecode. SA detekuje chyby v kódu staticky na základě hledání chybových vzorů. K provádění SA kódu se využívá mnoha dostupných nástrojů, které slouží k odhalování různorodých problémů a nedostatků. Mezi ně například patří:

- bezpečnost,
- zajišťování technického dluhu,
- výkonnost,
- čitelnost kódu,
- odhalování mrtvých částí kódu.

Nástroje pro podporu SA lze integrovat s různými vývojovými prostředími (IDE), či využít nástrojů přímo pro tento účel vytvořených. Mezi takové nástroje patří například *Apache Ant*[5] a Apache *Maven*[6].

Nástroje na SA většinou pracují na principu detekování chybových vzorů. Obecnou strukturu nástroje pro statickou analýzu znázorňuje obrázek 6.



obr. 6 - obecná struktura nástroje pro statickou analýzu [13]

---

[5] *Jedná se o nástroj sloužící k sestavování softwarových aplikací, který umožňuje automatizovat řadu činností zahrnující kompilaci, testování i vytvoření balíku pro distribuci.*

[6] *Podobný nástroj jako Apache Ant, který nabízí komplexnější funkce , které je možné využít skrze celý vývojový životní cyklus produktu.*

Mezi nejčastější nedostatky v kódu bývají dle Václava Pecha [14] označovány:

špatné praktiky

- detekuje kód, který nemusí být v dané situaci chybný, může však časem způsobit problémy nebo je nespolehlivý

správnost

- neboli chyby správnosti, detekují nesprávný kód, který se chová tak, jak jeho tvůrce nezamýšlel

chyby v multijazyčnosti

- odhaluje potenciální problémy s kódováním znaků, nutné brát v úvahu při vývoji vícejazyčného softwaru

zranitelnost kódu

- detekuje použití nezapouzdřených měnitelných objektů, např. umožňuje-li třída vrátit přímo referenci na její měnitelný atribut (bez vytvoření kopie)

vícevláknová korektnost

- detektory problémů týkajících se vláken a synchronizace

výkon

- vyhledává konstrukce zhoršující výkon aplikací použitím nevhodných konstrukcí

porušování stylu kódu

- zaměřuje se na porušování stylu kódu

používání riskantního kódu

- upozorňuje na podezřelý, zbytečný nebo riskantní kód

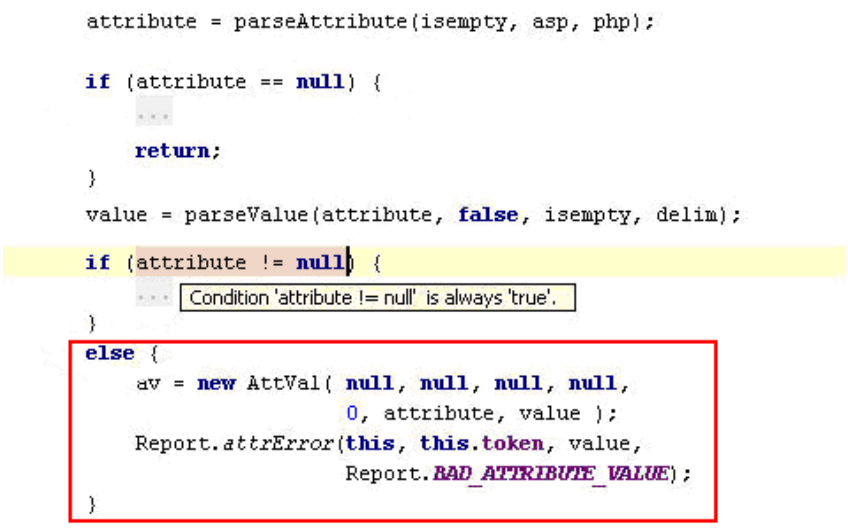
4.1. NÁSTROJE PRO STATICKOU ANALÝZU KÓDU

K nalézání nedostatků v kódu pomocí statické analýzy se využívá celá řada dostupných nástrojů. Některé nástroje jsou závislé na použitém vývojovém prostředí, jiné nabízí možnost integrace s ostatními vývojovými nástroji. V této kapitole jsou uvedené nejpoužívanější nástroje pro SA pro programovací jazyk JAVA, které jsou aktualizované a stále podporované svými vývojáři[7].

[7] Ke dni 4.4.2016.

4.1.1. IntelliJ IDEA

Jedná se o kompletní IDE, které nabízí mnoho nadstandardních funkcí oproti ostatním klasickým vývojovým prostředím. Jednou z funkcí je také analýza kódu. Analýza probíhá tzv. on-the-fly[8] a detekuje kompilační a runtime chyby a navrhuje opravy a vylepšení před komplikací kódu. Nástroj umožňuje také analyzovat návaznosti a vztahy v kódu, čímž pomáhá porozumět struktuře zdrojového kódu včetně vztahů mezi komponentami softwaru. Rozšířenými funkcemi je vyhledávání nedosažitelného kódu[9] (obrázek 7), nepoužitých deklarací nebo vytváření inspekčních profilů, pomocí kterých lze chování a citlivost prostředí nastavit dle preferencí.



obr. 7 - detekce nedosažitelného kódu pomocí analýzy nástroje IntelliJ IDEA [15]

4.1.2. FindBugs

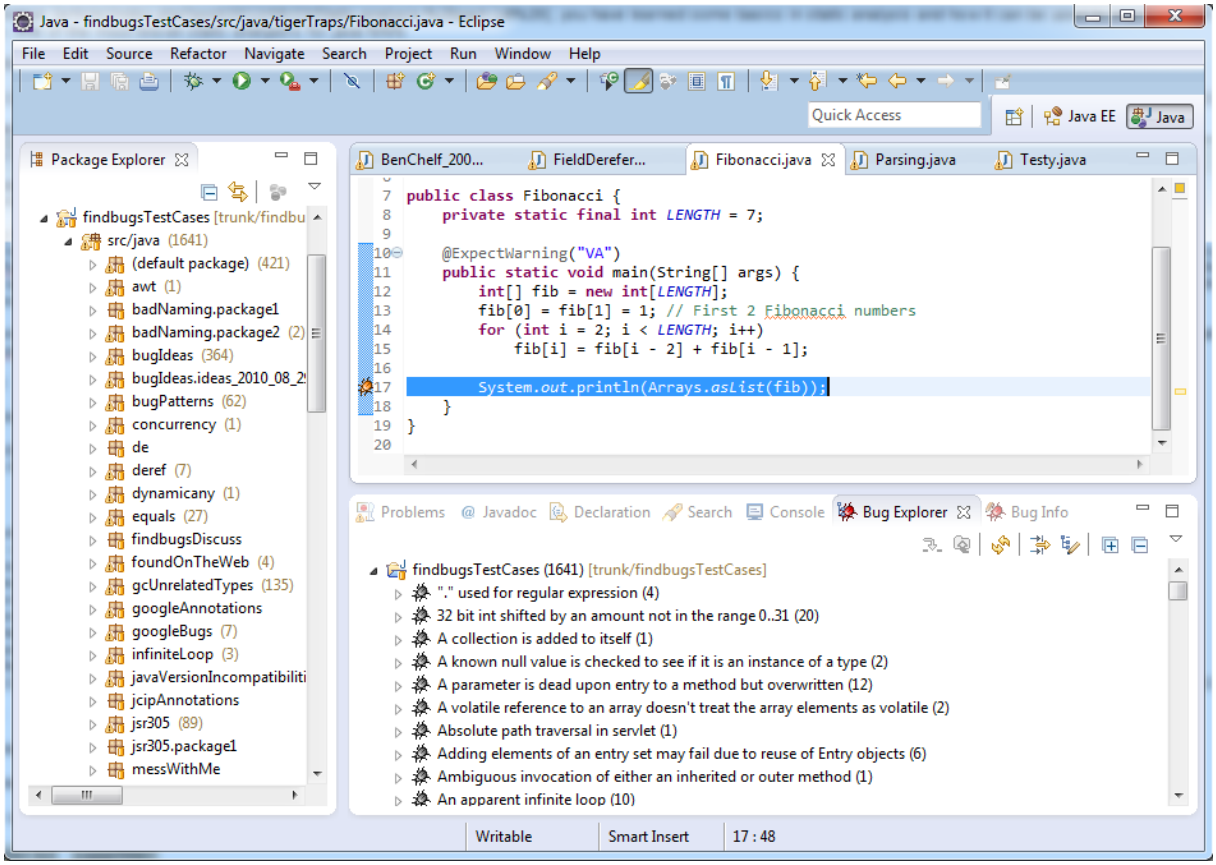
Jedná se o volně šiřitelný nástroj, který je určen pro statickou analýzu pouze programovacího jazyka Java. Je zdarma a distribuce probíhá pod GNU licencí[10]. Výhodou nástroje je, že může být používán buď samostatně či být integrován do IDE (*Eclipse*, *BlueJ*, *NetBeans*). Findbugs analyzuje bytecode,

[8] Výraz on-the-fly vyjadřuje v tomto kontextu činnost prováděnou přímo po napsání kódu.

[9] Představuje zdrojový kód, který není nikdy vykonán, protože k němu neexistuje žádná cesta ze zbytku kódu.

[10] Podle informací uvedených na oficiální stránce http://findbugs.sourceforge.net/.

což znamená, že samotný zdrojový kód není pro provedení analýzy zapotřebí. Findbugs je založen na tzv. chybových vzorech, jejichž známou sekvenci vyhledává v přeložených class souborech. Nástroj je velice hojně využíván komunitou Java vývojářů a díky tomu nabízí propracovanou správu reportů a historie chyb. Aktuální verze je 3.0.1[11], která obsahuje přes 850 chybových vzorů [16]. Ukázka prohlížeče chyb (bug explorer) nástroje FindBugs v IDE Eclipse je zobrazena na obrázku 8. Nalezené chyby jsou vypsány v seznamu, kde jsou zobrazeny doporučení jak lze chybu opravit. Číslo v závorce určuje řádek zdrojového souboru, na kterém se daná chyba nachází.



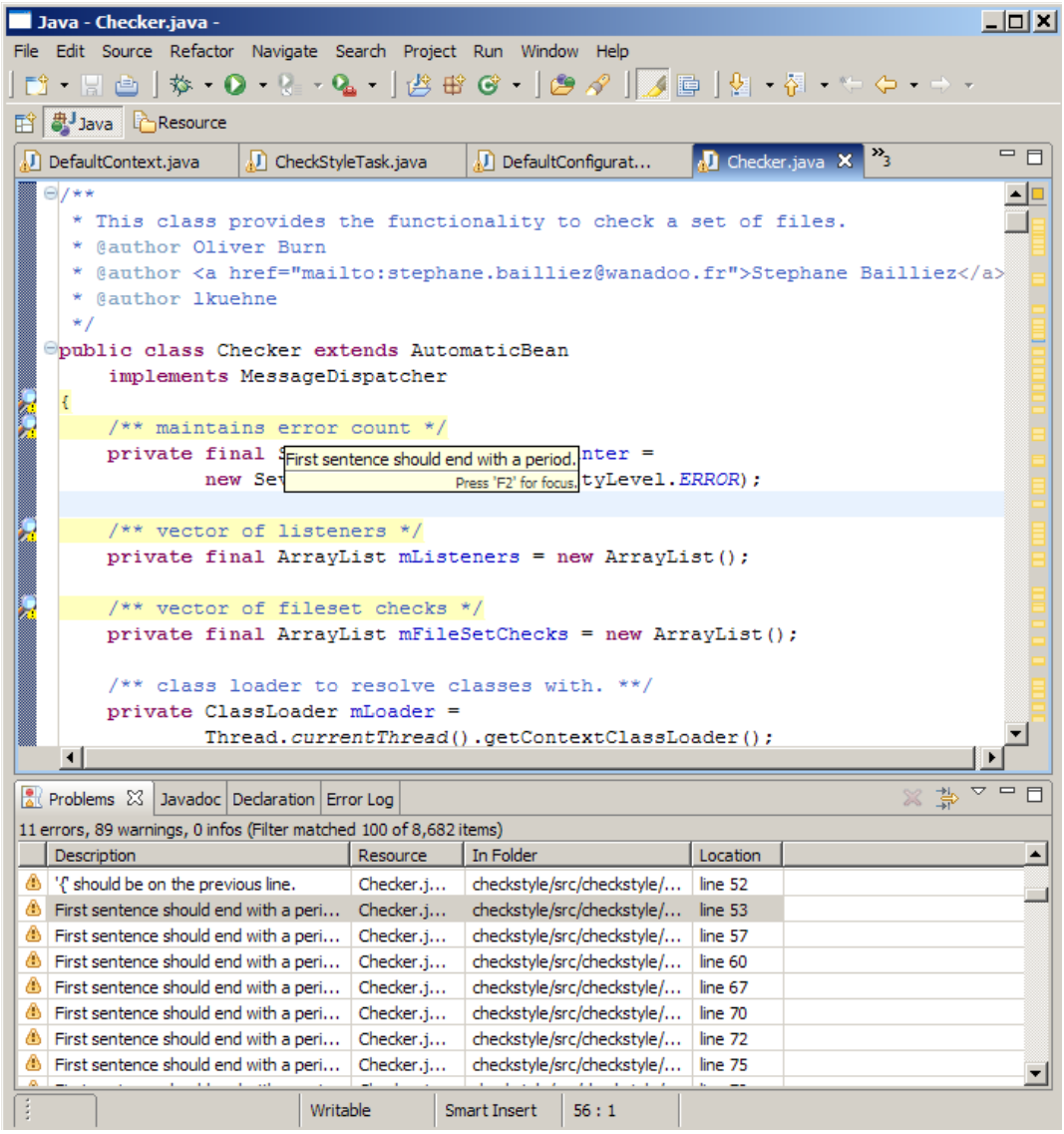
obr. 8 - plugin nástroje FindBugs integrovaný v IDE Eclipse [17]

4.1.3. CHECKSTYLE

Další z volně dostupných nástrojů pro kontrolu Java kódu. Slouží pro automatizaci testů. Nástroj nabízí širokou možnost přizpůsobení a podporuje

[11] Ke dni 8.4.2016.

většinu kódových konvencí[12], pro které lze nástroj využít. Výhodou *Checkstylu* je možnost integrace se všemi známými vývojovými prostředími a developerskými nástroji, např. *IntelliJ IDEA*, *Eclipse*, *NetBeans*, *SonarQube*, *Maven*, apod. Aktuální verze je 6.17[13], která obsahuje přes 300 chybových vzorů, nazývaných *checks* [18]. Plugin nástroje *Checkstyle* integrovaný do IDE *Eclipse* je zobrazen na obrázku 9. V rychlém náhledu je vidět popis chyby, zdrojový soubor, kde se chyba vyskytuje, cesta složky zdrojového soubory a číslo řádku, kde lze chybu lokalizovat.



obr. 9 - grafické prostředí IDE Eclipse s využitím pluginu nástroje SA Checkstyle [19]

[12] Kódovou konvencí se má na mysli Java Language Coding Standard, např. Sun Code Convention, Google Java Style, apod.

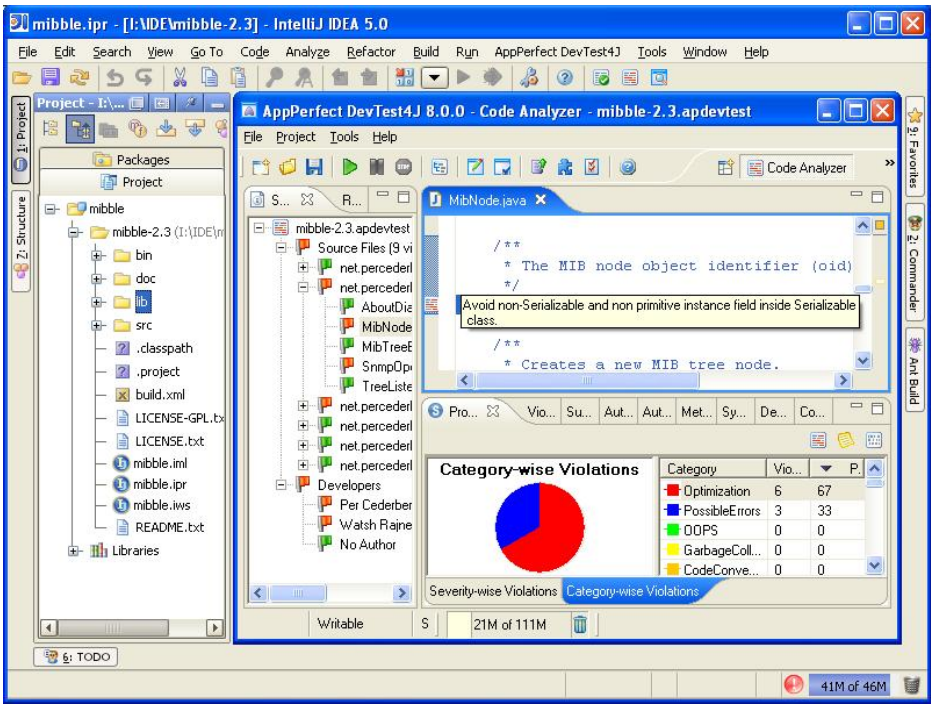
[13] Ke dni 8.4.2016.

4.1.4. PMD

Nástroj *PMD* je analyzér zdrojového kódu pro vícero programovacích a skriptovacích jazyků. Zaměřuje se na nejčastější nedostatky jako jsou nevyužité proměnné, prázdné try/catch bloky, nalézání mrtvého a nedosažitelného kódu, duplicity v kódu apod. Nástroj se vyžívá jako plugin do některého z IDE (*Eclipse*, *IntelliJ*, *NetBeans*) nebo pokročilejšího vývojového nástroje (*Maven*, *Ant*). Aktuální verze 5.4.1 obsahuje řádově stovky pravidel pro jazyk Java. [20]

4.1.5. APPPERFECT CODE ANALYZER

Analyzér vytvořen pro využití s prostředím *Eclipse*. Zaměřuje se na dva jasně stanovené cíle - kontrolu Java zdrojového kódu a vynucení používání dobrých praktik při psaní kódu, které mohou při jakékoliv revizi kódu ušetřit mnoho času pro lepší čitelnost kódu. *AppPerfect Code Analyzer* je možné integrovat do mnoha nástrojů (např. *Maven, Ant, Jenkins*), což umožňuje použití nástroje v komplexních vývojových prostředích. Ukázka integrace ve vývojovém prostředí *IntelliJ Idea* je zobrazena na obrázku 10. Kromě klasických lokalizačních ukazatelů nástroj nabízí také grafickou vizualizaci kategorií vyskytujících se chyb [21]



obr. 10 - nástroj App Code Analyzer v IDE IntelliJ IDEA [22]

Nástroj obsahuje přes 750 vnitřních pravidel pro analýzu Java/JSP kódu a také obsahuje funkci automatické rychlé opravy (*quick fixes*), kterých je v poslední verzi dostupných přibližně 180. Samozřejmostí je vytvoření a přidání vlastních pravidel či rychlých oprav. Nástroj je dostupný ke stažení na Eclipse Marketplace [23].

4.1.6. SonarQube

Jedná se o open source nástroj označovaný jako komplexní projektový analyzér. (*high-level project analyzer*) [24]. Toto označení se používá díky zajišťování široké možnosti analýzy nad projektem, čímž poskytuje daleko širší možnosti oproti klasické SA. *SonarQube* je platforma pro řízení kvality umožňující průběžnou analýzu a zabývá se měřením technické kvality celého softwarového projektu. Nástroj podporuje více než 20 programovacích jazyků (Java, C#, C/C++, PL/SQL, Cobol, PHP, atd.)[14] skrze volně dostupné pluginy, které obsahují parsery[15] a analyzátory pro daný jazyk. Kromě toho umožňuje integraci s nástroji třetích stran, například *FindBugs*, *Checkstyle*, *Maven*, *Ant* nebo *Gradle*.

Pro analýzu projektu v programovacím jazyce Java slouží Java plugin[16] dostupný na oficiálních stránkách. Nabízí více než 300 pravidel analyzujících kódové konvence, detekci chyb a bezpečnostních problémů. Analýza probíhá nad kompilovaným Java kódem.

SonarQube umožňuje analýzu nad mnoha projekty zároveň a dokáže nad nimi vyhodnocovat souhrnné výsledky pomocí mnoha grafických panelů (*dashboard*). Dashboard zobrazující souhrnné informace nad vícero projekty je zobrazen na obrázku 11 (*global dashobard*).

---

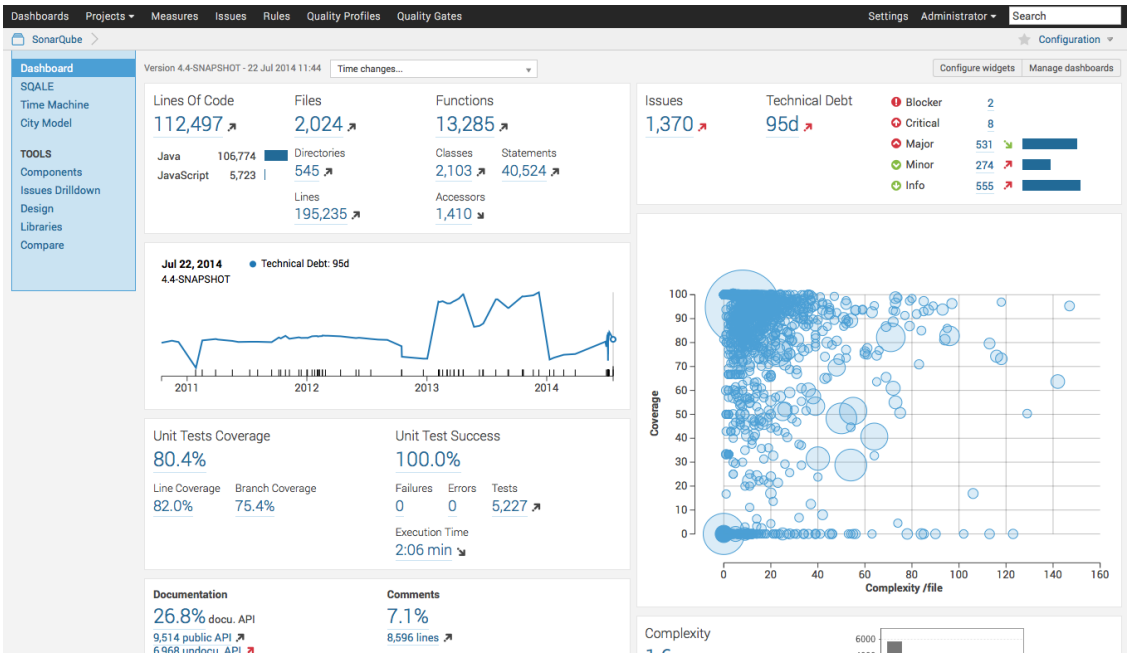
[14] Dle oficiálních stránek - http://docs.sonarqube.org/display/PLUG/Plugin+Library

[15] Program pro vykonávání syntaktické analýzy.

[16] Java Plugin - http://docs.sonarqube.org/display/PLUG/Java+Plugin

obr.11 - globální dashboard nástroje SonarQube [25]

Globální dashboard zobrazuje souhrnné informace všech projektů, jako například celkový počet řádek kódu, celkový počet chyb a varování nebo pokrytí testy. Dále rozdělení kritických, majoritních a minoritních nedostatků v projektech či historický vývoj řádek kódu nad jednotlivými projekty. Pokud chceme zjistit konkrétní informace týkající se pouze daného projektu, lze využít projektový dashboard (*project dashboard*), který zobrazuje obrázek 12.



obr. 12 - projektový dashboard nástroje SonarQube [25]

Projektový dashboard zobrazuje detailnější informace týkající se pouze daného projektu. Patří sem například počet tříd, konstant či přistupujících objektů. Procento kódu pokrytého unit testy, jejich celkový počet a úspěšnost. Dashboard také znázorňuje kolik procent kódu je okomentováno.
Aktuální verze[17] je SonarQube 5.5 a Java Plugin 3.13.1, obojí volně dostupné z oficiálních stránek nástroje SonarQube. [26]

4.1.7. LAPSE+

Nástroj *LAPSE+* je založený na statické analýze kódu, která slouží k nalézání bezpečnostních chyb v aplikacích napsaných v Java Enterprise Edition. Je volně přístupný a distribuován pod licencí GNU.
Zaměřuje se na zranitelnost webových aplikací na základě získání nedůvěryhodných dat, jejichž zdrojem může být například HTTP požadavek nebo cookie soubor. Nástroj detekuje několik druhů zranitelností mezi které například například patří:

- podvrhování URL (*URL tampering),*
- manipulace s hlavičkou (header manipulation),
- „otrava" souborů cookie (cookie poisoning),
- vsunutí kódu (*SQL injection)*.

K detekci těmto potencionálním zranitelnostem nástroj využívá tři základní kroky. Nejdříve identifikuje části kódu, které mohou být zdrojem nedůvěryhodné injekce dat, Ve druhém kroku nástroj v kódu nalézá takové části kódu, které mohou útok nedůvěryhodnými daty šířit a ovlivnit tak chování aplikace. V posledním kroku se zjišťuje, jestli z nalezených části kódu ve druhém kroku je možné dosáhnout na části kódu z bodu jedna. Pokud ano, testovaná aplikace obsahuje chybu v zabezpečení a je zranitelná. [27]

4.1.8. FORTIFY STATIC CODE ANALYZER

Univerzální komerční nástroj od společnosti HP, který se zaměřuje na bezpečnost aplikací pomocí statické analýzy kódu. Cílem nástroje je

[17] Ke dni 23.5.2016.

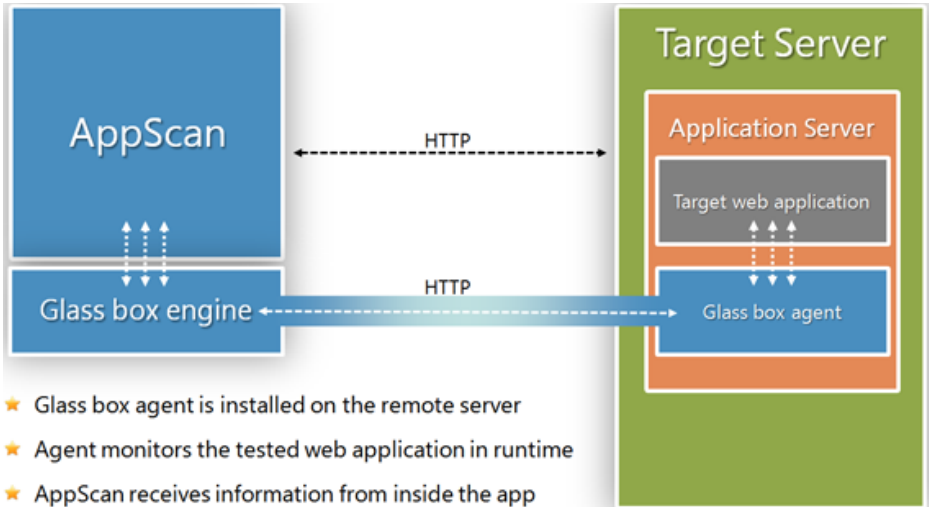
provádět testování kódu co nejčastěji a co nejdříve v průběhu vývoje softwaru. Nástroj nalézá příčiny zranitelnosti v kódu, nabízí opravení bezpečnostních chyb pomocí rychlých náprav a při vývoji navrhuje osvědčené postupy pro psaní méně zranitelného kódu. Podporuje více než 22 programovacích jazyků (Java, C/C++, Python, PHP, HTML, Ruby, apod.) a detekuje chyby ve více než 600 kategoriích zranitelnosti. Podporuje všechny hlavní vývojové platformy, stejně tak IDE (*Eclipse*, *IntelliJ IDEA*, *Microsoft Visual Studio*, apod.). [28]

4.1.9. IBM Security AppScan Source

Jedná se o komerční nástroj společnosti IBM, který se zaměřuje především na zabezpečení webových aplikací a zabezpečení mobilních aplikací pro Android a iOS. Nástroj podporuje několik programovacích jazyků - Java, JavaScript, HTML5, Cordova a Objective-C.

Pomocí nástroje lze identifikovat bezpečností chyby nedostatky v kódu v rané fázi vývoje. Umožňuje nastavovat a spravovat bezpečností politiky dle preferencí vývojářů. Výrobce uvádí, že dokáže zanalyzovat více než milion řádek kódu za hodinu včetně komplexních podnikových řešení.

AppScan využívá tzv. Glass box testování, což je technologie vyvinutá a patentovaná společností IBM. Tato metoda využívá přítomnosti Glass box agenta na serveru, kde běží testovaná webová aplikace (obrázek 8).



obr. 8 - diagram komunikace nástroje AppScan pomocí Glass box testování [29]

Tento agent analyzuje chování aplikace přímo z běhového prostředí, kde monitoruje chování rizikových metod představující potencionální zranitelnost aplikace. Agent veškeré informace, získané z vnitřku aplikace, posílá nástroji *AppScan*, kde je přítomný tzv. *Glass box engine*, který informace zpracovává a předává do samotného nástroje.

4.1.10. Nedostatky nástrojů statické analýzy

Používání nástrojů SA zajišťuje při vývoji efektivnější nalézání nedostatků a chyb ve zdrojovém kódu a slouží k dodržování nadefinovaných programovacích pravidel. Výčet v předchozí kapitole 3.1 ukazuje, že nástrojů existuje celá řada, ale ne všechny nástroje poskytují již v základním nastavení potřebná pravidla k vytvoření dostatečně kvalitního kódu. Vývojových standardů a pravidel se používá mnoho a záleží na zvycích softwarové společnosti či na požadavcích zákazníka, které standardy je žádoucí u softwarového vývoje aplikovat.

Pokud spustíme libovolný nástroj SA nad rozsáhlejším softwarovým projektem, pravděpodobně nástroj odhalí ve zdrojovém kódu stovky až tisíce více či méně podezřelých nálezů. Podstatné ovšem je, že zdaleka ne všechny tyto nálezy jsou pro daný projekt důležité a není potřeba se zabývat jejich opravou. Obvykle postačuje některé z těchto nálezů pouze omezit či ignorovat na základě závažnosti významu jejich nedostatku. Takovým nálezům, které SA označí jako nedostatek, ale ve skutečnosti žádnou hrozbu v závislosti na projektu nepředstavují, se v praxi říká *false positives*. Opačným případem, kdy SA nerozpozná slabinu v kódu, která ve skutečnosti hrozbu může představovat, jsou takzvané. *false negatives*. Právě z tohoto důvodu většina nástrojů SA obsahuje možnost nastavení vlastních pravidel, která mají být statickou analýzou

4.2. Nástroje pro sestavení

Kromě klasických nástrojů pro SA (kapitola 3.1), které slouží převážně pro opravu lepší bezpečnosti kódu, existují i další nástroje usnadňující programátorům opakující se činnosti při vývoji SW projektu. Mezi takové velké pomocníky patří nástroje pro sestavení (*build*) softwarových aplikací. Jejich hlavním úkolem je především automatizovat opakující se činnosti (správa, řízení, testování, sestavení), které je nutné provádět při každém vytvoření nového buildu aplikace. Výhodou těchto nástrojů jsou široké možnost i přizpůsobení a integrace s ostatními nástroji. Proto je možné takový nástroj nastavit přesně dle našich preferencí a pomocí pluginů přidat funkce potřebné pro náš projekt. V této kapitole budou zmíněny známé a používané nástroje pro sestavení využívané pro projekty psané v jazyce Java.

4.2.1. Maven

Vznik sestavovacího nástroje *Maven* se datuje k roku 2002, kdy vznikl jako podprojekt pro usnadnění práce při vývoji projektu Apache Turbine. Od svého vzniku prošel nástroj spoustou vylepšení a v roce 2010 byla vydána verze *Maven* 3, která je dnes stále podporována.

Existuje hned několik důvodů, proč je *Maven* velice oblíbený v komunitě nejen Java vývojářů. Mezi jeho přednosti patří nezávislost na IDE a typu projektu a tak vývojáři mohou upřednostnit své oblíbené preference. Jednou z hlavních funkcí je správa závislostí, kdy nástroj analyzuje závislosti projektu na externích knihovnách (JAR soubory), případně externích nástrojích. *Maven* umožňuje standardizovat životní cyklus vývoje softwaru a zároveň automatizovat mnoho různých činností pomocí jednoho nástroje. Maven klade důraz na TDD (kapitola 1.1.5), kdy je výhoda mít projekt pokrytý jednotlivými testy pro různé části kódu. V neposlední řadě nástroj usnadní práci díky automatickému generování dokumentace a sestavení aplikace [30]. Modulární architektura nástroje umožňuje integraci se spoustou dostupných pluginů dle potřebných funkčních požadavků. Samotný nástroj neobsahuje žádné grafické prostředí a ovládá se pouze pomocí příkazové řádky.

Princip

Základním principem nástroje *Maven* je vytvoření modelu, kterému se říká Project Object Model (POM). Tento model popisuje projekt softwarový projekt pomocí zdrojového kódu včetně všech závislostí na externích knihovnách. Také popisuje celý proces buildování včetně všech částí vývoje projektu jako například shromažďování informací o zdrojovém kódu nebo spouštění jednotlivých testů. Celý POM je popsán pomocí XML struktury v souboru *pom.xml* (obrázek 9), který se nachází zpravidla v kořenovém adresáři. Každý projekt má svůj *pom.xml* soubor, který může dědit vlastnosti od nadřazeného projektu se svým vlastním *pom.xml* souborem.

```
<project>
 <modelVersion>4.0.0</modelVersion>

 <!-- identifikátory projektu -->

 <groupId>com.mycompany.app</groupId>
 <artifactId>my-app</artifactId>
 <version>1.0</version>

 <!-- závislosti na externích knihovnách -->

 <dependencies>
 <dependency>

 <!-- upřesnění konkrétní knihovny -->

 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <version>3.8.1</version>

 <!-- definice oblasti použití knihovny -->

 <scope>test</scope>

 </dependency>
 </dependencies>
</project>
```

obrázek 9 - základní struktura pom.xml souboru popisují Project Object Model [31]

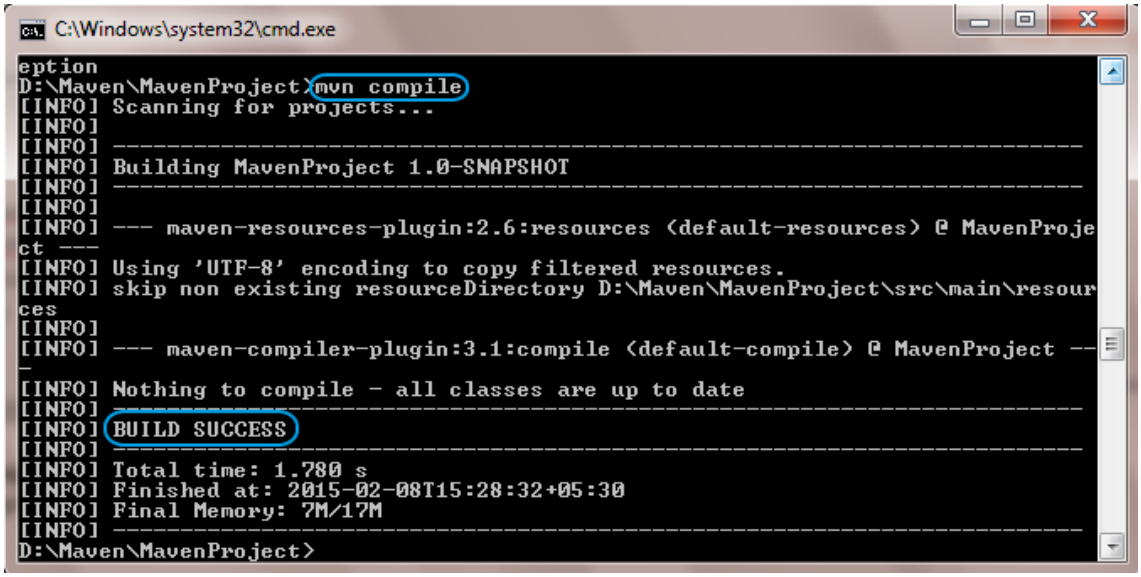
V souboru *pom.xml* je možné nadefinovat závislosti na externích knihovnách a nástroj tyto knihovny následně automaticky vyhledá a nainstaluje. Vyhledávání knihoven probíhá v definovaných úložištích (*repository*), kdy standardní volbou je *Maven central repository*[18], které je veřejné přístupné. *Maven* umožňuje

[18] Také se nazývá jako global repository - http://repo1.maven.org/maven2/

nastavit si přístup na soukromé či firemní repository s obsahem potřebných externích knihoven.

Další funkcí je možnost přizpůsobit si vlastní fáze, které bude naše sestavení aplikace obsahovat. To znamená, že je například možné nastavit si v jaké konkrétní fázi se mají spouštět pluginy, kdy se mají spouštět testy nebo v jaké fázi má *Maven* skončit. Samozřejmostí nástroje je výsledný build nahrát přímo na předem daný server.

Na obrázku 10 je zobrazen rozhraní v příkazové řádce, pomocí kterého se *Maven* standardně ovládá.



obrázek 10 - buildování projektu pomocí příkazové řádky [32]

Aktuální verze nástroje *Maven* je 3.3.3 a je dostupná z oficiálních stránek nástroje.[19]

4.2.2. ANT

Jedná se o starší a také velice známý sestavovací nástroj pro jazyk Java, jehož vznik je podle zdroje [33] datován k roku 2000. Program je volně dostupný (pod *The Apache Software Licence*) na oficiálních stránkách[20] nástroje. Nástroj je kompletně napsán v programovacím jazyku Java a také je používán především pro projekty v tomto jazyce. Vychází ze známého sestavovacího nástroje

[19] Ke dni 26.5.2016. Oficiální stránky nástroje *Maven* - http://maven.apache.org/

[20] Oficiální stránky nástroje Ant - http://ant.apache.org/

Make[21] *pro programovací jazyk C,* ale oproti němu přináší několik zásadních vylepšení. Hlavními výhodami nástroje *Ant* oproti nástroji *Make* je nezávislost na platformě[22], použití přehlednější syntaxe ve formátu XML (stejně jako *Maven*), vyšší rychlost sestavování Java aplikací nebo existence specifických konstrukcí pro build Java aplikací (podpora JAR souborů, generování dokumentace javadoc, podpora výstupu *JUnit*) [30].
Stejně jako v případě *Mavenu*, ani *Ant* neobsahuje žádné vlastní grafické prostředí a standardním prostředním je příkazová řádka. Díky možnosti integrace do různých vývojových prostředí umožňuje zvolit si vlastní preferovaný způsob ovládání nástroje.

Princip

Každému projektu je vytvořen vlastní buildovací soubor, v případě nástroje *Ant* označovaného jako *build.xml.* V takovém souboru je projekt reprezentován pomocí kořenového elementu *project,* který je definován několika atributy viz tabulka 1.

atribut	popis
name	jméno projektu
default	název defaultního cíle, který se bude provádět, pokud na příkazovém řádku při spouštění neuvedete jiný
basedir	kořenový adresář projektu
description	pro zapsání podrobnější informací o účelu projektu

tabulka 1 - atributy elementu project [33]

Některé z nich jsou povinné (atribut *default)*, jiné z nich jsou naopak nepovinné (atributy *name* a *basedir).* Atribut *description* představuje volitelnou možnost použití. Každý takový projekt má definovaný určitý počet cílů (*targets*) představující úlohy, které mají být spuštěny (například překlad aplikace). Při spuštění nástroje je možné definovat, jakých cílů chceme dosáhnout. Atributy elementu target popisuje tabulka 2.

---

[21] Starší sestavovací nástroj sloužící k buildu aplikací v programovacím jazyce C.

[22] V informatice označení pro pracovní prostředí označující hardware i software.

atribut	popis
name	jméno (název) cíle
depends	seznam cílů, na kterých je tento cíl závislý
if	jméno parametru (property), který musí být nastaven, aby byl cíl prováděn
unless	jméno parametru (property), který nesmí být nastaven, aby se cíl prováděl
description	krátký popis činnosti cíle

tabulka 2 - atributy elementu target [33]

Každý z definovaných cílů je prováděn maximálně jednou a v případě vzniku chyby nástroj *Ant* přeruší činnost a nepokračuje ve vykonání dalších cílů.
Každý z cílů se dále skládá z jednotlivých úloh (*task*), které jsou prováděné v sekvenčním pořadí. Provádění úloh závisí i na tom, zda je daná úloha potřeba provést. Například při neexistenci novějšího class souboru není třeba provádět nový překlad zdrojového souboru.
V celém projektu je navíc možné používat množinu parametrů (*properties*). Všechny parametry jsou definovány jménem a svou hodnotou a mohou být použity v jednotlivých atributech úloh.
Jednoduchý příklad jak může vypadat struktura souboru *build.xml* je zobrazena na obrázku 11.

```
<?xmlversion="1.0"?>
<project name="test" default="compile" basedir=".">

   <property name="src" value="."/>
   <property name="build" value="build"/>

   <target name="init">
     <mkdirdir="${build}"/>
   </target>

   <target name="compile" depends="init">
     <!--Compilethejavacode-->
     <javac srcdir="${src}" destdir="${build}"/>
   </target>

</project>
```

obrázek 11 - jednoduchý příklad build.xml nástroje Ant [34]

Ant umožňuje integraci se spoustou nástrojů pro samostatnou SA, tak se kompletními vývojovými prostředími. Nástroj je zdarma a je možné ho stáhnout z oficiálních stránek. Aktuální verzí je *Ant* 1.9.7[23].

Ačkoliv se může zdát, že nástroje *Ant* a *Maven* jsou si velice podobné, skutečnost je taková, že se jedná vcelku o odlišné nástroje, které však slouží k podobnému primárnímu účelu, tj. sestavování aplikací. Následující tabulka 3 obecně shrnuje rozdíly mezi těmito nástroji, informace jsou převzaté od Libora Jelínka ze zdroje [30].

	Ant	**Maven**
vznik	pokračovatel např. Make z jazyka C	založil novou generaci nástrojů
jméno skriptu	build.xml	pom.xml
syntaxe script	XML	XML
obsah skriptu	jak něco dělat	co dělat (není potřeba znát jak)
podpora v IDE	perfektní	velmi dobrá
vývojář projektu musí znát	konkrétní build.xml	pouze Maven obecně

tabulka 3 - obecné porovnání nástrojů Ant a Maven

[23]Ke dni 29.5.2016.

4.3. NÁSTROJE PRO TESTOVÁNÍ

Při vývoji softwaru lze, kromě nástrojů určených přímo pro statickou analýzu (kapitola 3.1) a nástrojů pro sestavení (kapitola 3.2), využít samostatné aplikace od různých vývojářů, které se zaměřují přímo na konkrétní specializovanou činnost. Takové nástroje lze používat jak samostatně, tak u některých využít možnost integrace s dalšími nástroji případně prostředími. V této kapitole budou zastoupeny nástroje sloužící k zajištění lepší kompatibility a kvality softwaru pro programovací jazyk Java. Vybrané nástroje se zabývají především rozdíly mezi různými verzemi Java tříd případně analýzou závislostí na externích knihovnách.

4.3.1. BNDTOOLS

Bndtools je volně dostupný nástroj (open source software dle oficiálních stránek[24]), který je založen na základě bnd[25] a *Eclipsu*. *Bndtools* se nabízí jako plugin na Eclipse Marketspace. Slouží především k vývoji OSGi (*Open Services Gateway initiative*) balíků (*bundle)* sloužící pro vývoj modulů či aplikací. Motivací pro vznik nástroje bylo to, že vývoj OSGi je v základním prostředí Eclipse velice obtížný.

OSGi framework představuje specifikaci dynamického modulárního systému pro jazyk Java[26]. Takový systém nabízí kompletní infrastrukturu pro spolupráci modulů skrze služby. Výhodou modulárního systému je možnost manipulace s moduly za běhu. Jedná se o velice vyspělou a velice používanou modulární technologii v programovacím jazyku Java. Využití nachází například ve větších aplikačních serverech. [35]

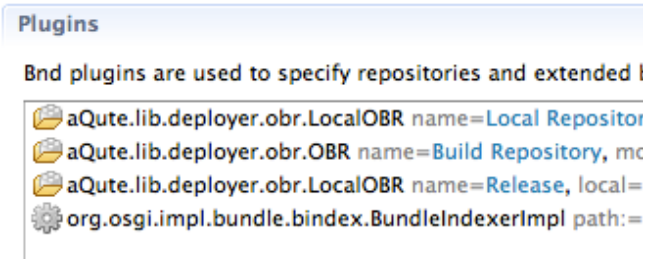
Nástroj pracuje přímo s class soubory pro vytvoření manifest souboru, což zahrnuje závislosti na Java balíčcích (*package*) pro kompilaci. Závislosti mezi

[24] Oficiální stránky nástroje Bndtools - http://bndtools.org/.

[25] Bnd je knihovna využívající se pro drtivou většinu OSGi vývoje a byla uznána jako standard OSGI aliancí.

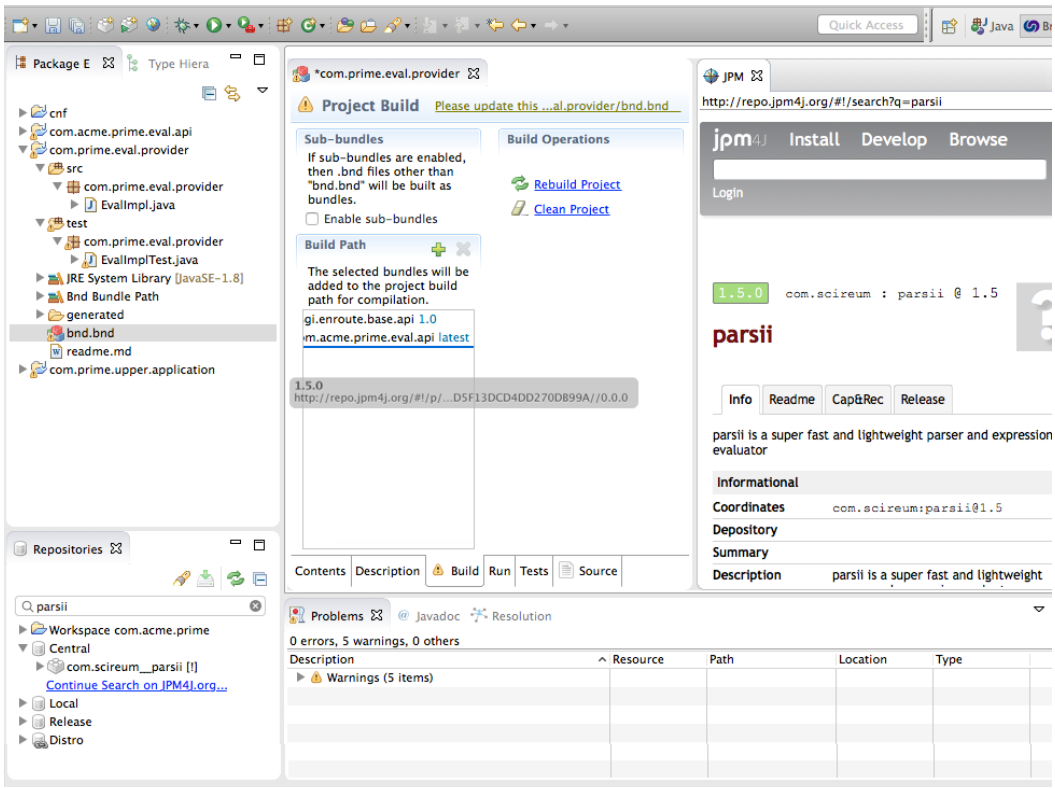
[26] Definice převzatá z https://cs.wikipedia.org/wiki/OSGi_Service_Platform

jednotlivými balíky vycházejí z používaných úložišť (*repositories*), které nástroj používá při sestavování (*build-time*) a při běhu (*runtime*). Úložiště představuje datový sklad balíků, které mohou být umístěné téměř kdekoliv - v pracovním adresáři (*workspace*), uložené na lokálním systému nebo mohou být dostupné ze vzdáleného serveru. Příklad různých typů úložišť je vidět na obrázku 11.



obrázek 11 - příklad nadefinovaných úložišť [36]

V případě chybějící balíku (obrázek 12) pro kompilaci projektu nástroj ukáže chybějící knihovnu (typicky JAR soubor) včetně různých verzí a umožní její přidání skrze získání pomocí úložiště.



obrázek 12 - přidání chybějící knihovny pomocí nástroje bndtools [37]

Bndtools vyhledává balíky třemi různými způsoby [36]:

- vyhledáním balíků uvedených v Build Path podle indexu BSN (*Bundle Symbolic Name*) a verze,
- vyhledáním balíků potřebných k vyřešení seznamu požadavků (*Requirements list*),
- vyhledáním balíků uvedených v seznamu běžících balíků (*Run Bundle list*) podle indexu BSN a verze.

Úložiště jsou implementované jako pluginy a teoreticky je tak možné použít libovolný typ úložiště, v případě, že by pro něj byl naprogramován plugin do nástroje *bndtools*. Ten umožňuje integraci s nástrojem *Maven* (kapitola 3.1.1), což umožňuje využít také *Maven* repository. Aktuální verzí je *bndtools* 3.2[27].

4.3.2. JAPICMP

Japicmp (ze zkratky Java API Compare) je nástroj sloužící k jedinému účelu, čímž je porovnání dvou verzí JAR archivu. Ovládá se pomocí příkazové řádky a spouští se pomocí příkazu:

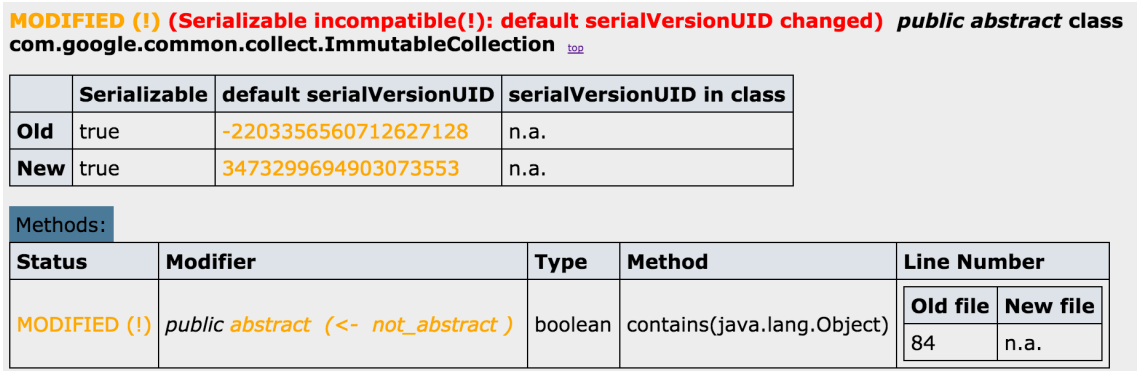
```
java -jar japicmp-0.8.0-jar-with-dependencies.jar -n new-version.jar -o old-version.jar
```

Dle oficiálních stránek nástroje [38] je možné *Japicmp* použít také jako knihovnu do libovolného Java projektu a je dostupný v centrálním úložišti nástroje *Maven*. *Maven* plugin umožňuje výstupy nástroje *Japicmp* integrovat do reportů pro *Findbugs* (kapitola 3.1.2) nebo *Checkstyle* (kapitola 3.1.3).
Hlavní předností tohoto nástroje je jednoduchost a rychlost. Při porovnání dvou JAR archivu obsahujících přibližně 1700 tříd souborů v každé z nich trvá méně než vteřinu. Proto je možné integrovat tuto funkci do každého sestavení bez známky ovlivnění výkonu či delšího času provedení.

[27] Ke dni 29.5.2016.

Funkce

- porovnání dvou JAR archivů bez nutnosti přidání všech závislostí do classpath,
- report z nástroje je možné vypsat v příkazové řádce, případně volitelně do XML nebo HTML souboru (obrázek 13),
- rozlišení mezi zdrojovou a binární kompatibilitou,
- porovnávány jsou všechny změny mezi třídami/metodami/poli.



MODIFIED (!) (Serializable incompatible(!): default serialVersionUID changed) ***public abstract*** **class com.google.common.collect.ImmutableCollection** top

	Serializable	default serialVersionUID	serialVersionUID in class
Old	true	-2203356560712627128	n.a.
New	true	3473299694903073553	n.a.

Methods:

Status	Modifier	Type	Method	Line Number	
				Old file	New file
MODIFIED (!)	*public abstract (<- not_abstract)*	boolean	contains(java.lang.Object)	84	n.a.

obrázek 13 - ukázka HTML reportu nástroje japicmp [38]

Nástroj je volně dostupný na GitHubu[28] a aktuální verzí je Japicmp 0.7.2[29] přičemž je nutné zmínit, že nové verze programu vycházejí poměrně často a podpora autorů je na vysoké úrovni..

4.3.3. JAPI COMPILANCE CHECKER

Java API Compilance Checker (zkratka Japicc) je nástroj pro kontrolu zpětné kompatibility knihoven JAVA, které analyzuje na binární úrovni. Nástroj cíli na vývojáře SW knihoven, případně programátorům specializující se na zpětnou kompatibilitu. Je volně dostupný na GitHubu[30] a je stále aktualizován. Aktuální verze je *Japicc* 1.7[31].

---

[28] Github nástroje *Japicmp* - https://github.com/siom79/japicmp.

[29] Ke dni 29.5.2016.

[30] GitHub nástroje *Japicc* - https://github.com/lvc/japi-compliance-checker.

[31] Ke dni 29.5.2016.

Nástroj funguje na principu kontroly deklarace nových a starých tříd jednotlivých verzí knihoven a provádí analýzu všech změn, které mohly způsobit nekompatibilitu - odstranění metod, odstraněním polí z třídy, přidáním abstraktní metody, apod. Výstup aplikace je vidět na obrázku 14.

	Severity	Count
Added Methods	-	93
Removed Methods	High	4
Problems with Data Types	High	0
	Medium	0
	Low	0
Problems with Methods	High	3
	Medium	0
	Low	2

obrázek 14 - report z nástroje Japicc [39]

4.3.4. CLIRR

Clirr je menší, volně dostupný nástroj, pomocí kterého je možné kontrolovat Java knihovny se staršími verzemi na binární a zdrojové úrovni. Principem nástroje je vstup dvou odlišných JAR souborů a ten vrátí seznam změn ve veřejném rozhraní (API).
Nástroj lze buď integrovat do určitého buildovacího nástroje (*Ant*, *Maven*) nebo používat samostatně pomocí rozhraní příkazové řádky, kde se nástroj spouští pomocí příkazu:

JAVA -JAR CLIRR-CORE-@VERSION@-UBER.JAR

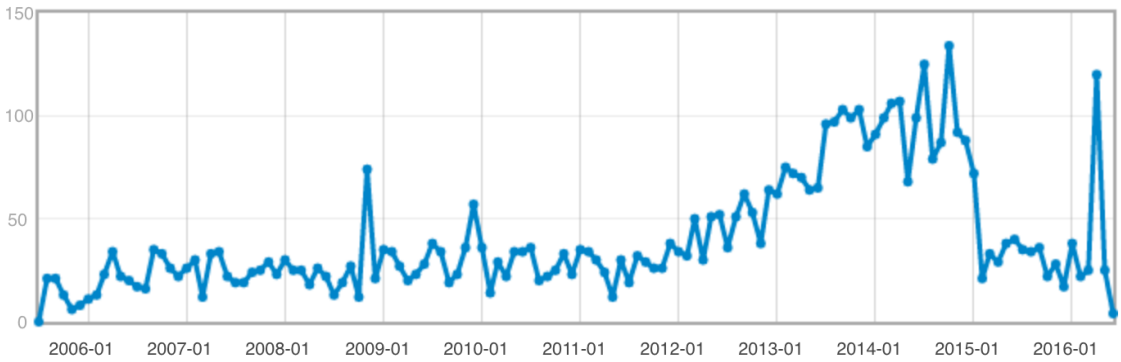
V případě použití v nějakém ze zmíněných buildovacích nástrojů, je *Clirr* možné nastavit tak, aby přerušil proces sestavení v případě detekce nějaké nekompatibility. Při využití v CI nástroj dokáže automaticky zabránit nechtěné příčinně binární nebo zdrojové nekompatibilitě aplikace. [40]

Funkce

- reportování o všech změnách v API mezi různými verzí Java archivů,

- vyhodnocuje každou změnu na základě binární a zdrojové kompatibility,
- podporuje reporty v plain textu nebo XML
- umožňuje zachycovat chyby a na jejich základě upozornit uživatele chybovou hláškou/varováním, přerušit build proces, apod.
- existence pluginu do *Ant*, *Maven*

Jako nevýhodu nástroje lze považovat, že již delší dobu neproběhla žádná aktualizace či jiná modernizace nástroje. Poslední nová verze vyšla v roce 2007 a to verze *Clirr* 0.7. I přesto je však nástroj stále využíván komunitou vývojářů, což je vidět na obrázku 14, který zobrazuje graf se součty stažení za jednotlivé měsíce v období leden 2006 až duben 2016. Celkem si stáhlo nástroj téměř 5400 uživatelů. Nástroj je dostupný zdarma ze stránek Sourceforge[32].



obrázek 14 - počty stažení nástroje Clirr od roku 2006 do roku 2016 [40]

4.3.5. JDIFF

Další z nástrojů zabývajících se analýzou rozdílu mezi různými verzemi Java archivů. Název *JDiff* vychází z anglického Java Difference a nástroj se označuje jako javadoc doclet[33].

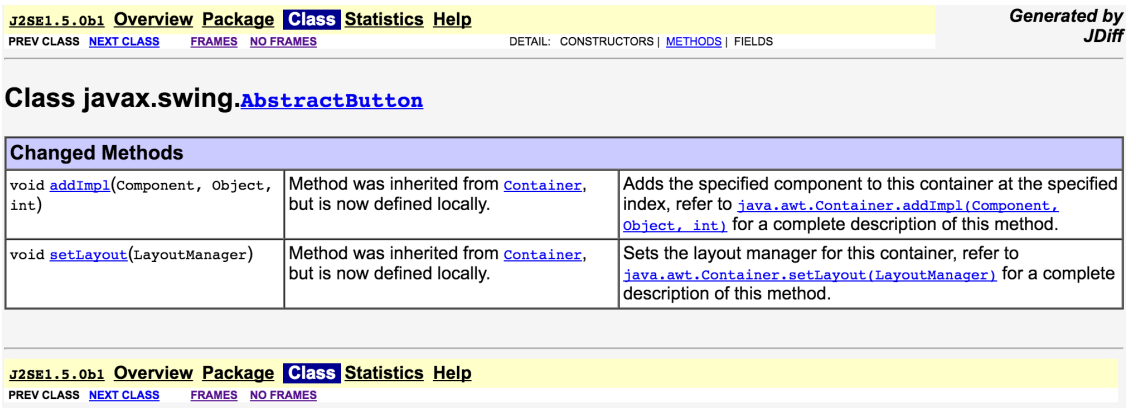
Výstupem nástroje je HTML report, který informuje uživatele o všech změnách, které se týkají balíčků, tříd, konstruktorů, metod a polí ovlivňující API a tím pádem mají vliv na kompatibilitu aplikace. Nástroj lze využít především při snaze získat detailní popis změn, které se udály mezi dvěma verzemi stejné

[32] Sourceforge Clirr - https://sourceforge.net/projects/clirr/files/

[33] Doclet je program pracující s nástrojem javadoc pro generování dokumentace ze zdrojového Java kódu.

aplikace. Porovnávání probíhá pouze na úrovni API, tzn. že se nevyhodnocují změny, které se dějí po provedení daného zdrojového kódu. [41]

Ukázka HTML reportu s popisem změn v ukázkové třídě je zobrazena na obrázku 15.



J2SE1.5.0b1 Overview Package Class Statistics Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES DETAIL: CONSTRUCTORS | METHODS | FIELDS

Generated by JDiff

Class javax.swing.AbstractButton

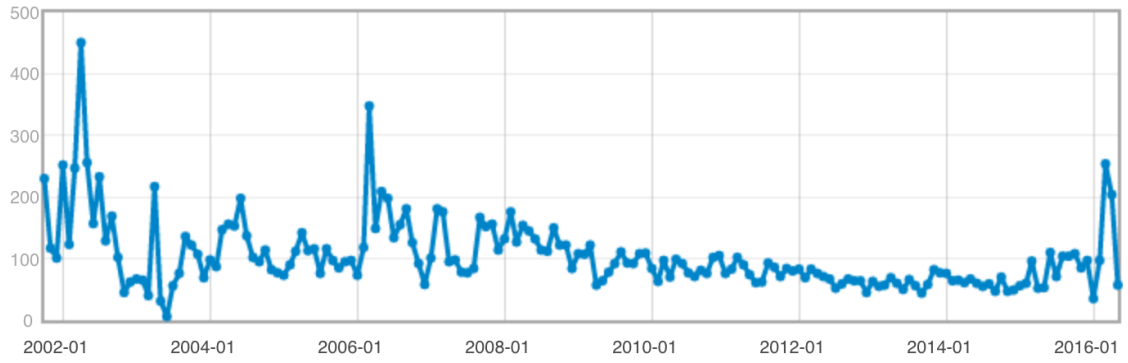
Changed Methods		
void addImpl(Component, Object, int)	Method was inherited from Container, but is now defined locally.	Adds the specified component to this container at the specified index, refer to java.awt.Container.addImpl(Component, Object, int) for a complete description of this method.
void setLayout(LayoutManager)	Method was inherited from Container, but is now defined locally.	Sets the layout manager for this container, refer to java.awt.Container.setLayout(LayoutManager) for a complete description of this method.

J2SE1.5.0b1 Overview Package Class Statistics Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES

obrázek 15 - report o změně ve třídě javax.swing.AbstractButton [41]

Nástroj je dostupný zdarma ze stránek Sourceforge[34]. Nástroj je díky své jednoduchosti a detailnímu popisu změn oblíben v komunitě vývojářů, což dokazují počty stažení zobrazené na obrázku 16. Nástroj byl od roku 2002 stažen více než 13 000 uživateli.



obrázek 16 - celkové počty stažení nástroje JDiff od roku 2002 [41]

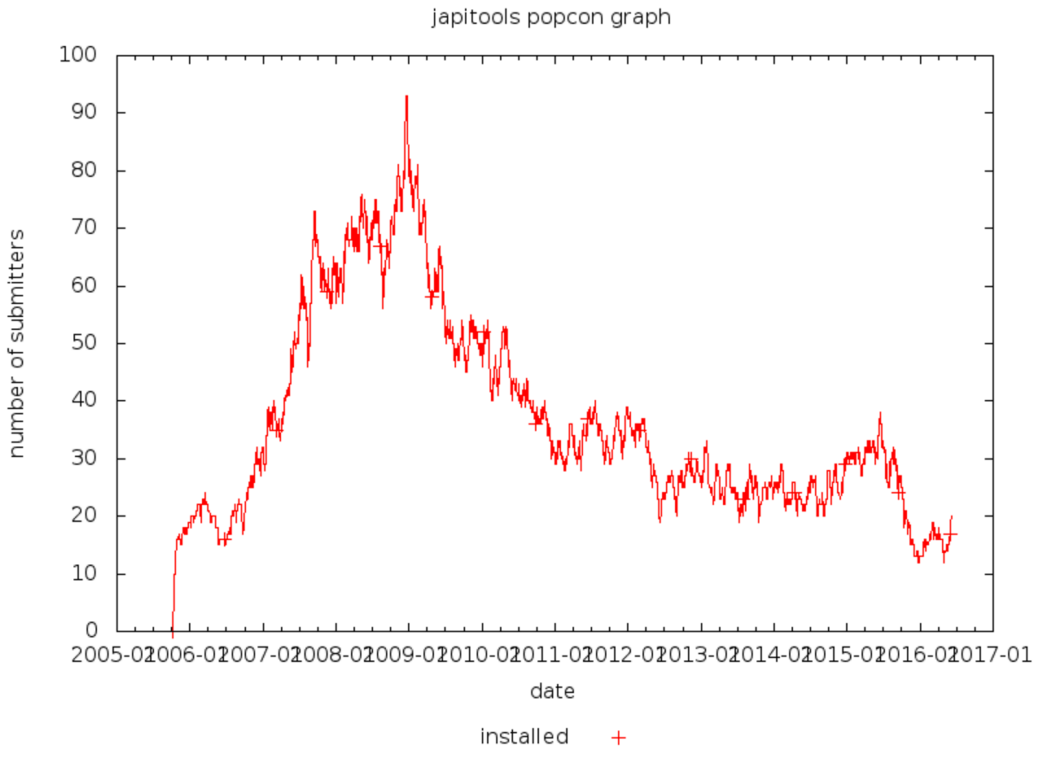
[34] Sourceforge JDiff - https://sourceforge.net/projects/javadiff/.

4.3.6. JAPITOOLS

Japitools (ze zkratky Java API Compatibility Testing Tools) je nástroj skládající se ze dvou jednoduchých programů sloužící pro testování Java API kompatibility. Ty byly původně vyvinuty pro testování libovolné implementace Java kódu se Sun JDK[35]. Nástroj ale umožňuje také testování zpětné kompatibility mezi libovolnými verzemi API.

Prvním program, *Japize,* je napsán v Javě a jeho výstupem je kompletní výpis API všech Java tříd ve strojově čitelném formátu. Druhý program se jmenuje *Japicombat.* Ten porovná libovolné dva výpisy API a porovná je pro ověření binární kompatibility podle specifikace uvedené v *Java Language Specification*[36]*. [42]*

Podle Debian Package trackeru [43] je možné znázornit počty stažení nástroje na platformě Linux/Debian. Graf s počty stažení od roku 2006 do roku 20016 je zobrazen na obrázku 17.



obrázek 17 - počty stažení balíčku japitools vizualizovány pomocí Debian package trackeru [43]

[35] Sun JDK - soubor základních nástrojů pro vývoj a běh aplikací pro platformu Java vyvinut původně společností Sun Microsystems (od roku 2010 patří pod Oracle).

[36] Kompletní dokumentaci jednotlivých verzí Java Language Specification lze nalézt na stránkách - http://docs.oracle.com/javase/specs/.

Aktuální verze *Japitools* 0.9.7 byla uvolněna v Listopadu 2006 a ačkoliv je nástroj svou komunitou stále využíván, nové verze či aktualizace již nebudou svými autory vydávány. Nástroj je možné získat z oficiálních stránek [42] nebo stáhnout jako balíček do OS typu Linux/Debian z *Debian Package Trackeru*.

4.3.7. SIGTEST

Nástroj *SigTest (*ze zkratky Signature Test*)* je open source projekt, který se skládá opět z kolekce vícero programů. Původním důvodem vzniku bylo vyvinout aplikace, které by pomáhaly při vytváření Java testovacích scénářů pro ověření kompatibility (označení TCK[37]).

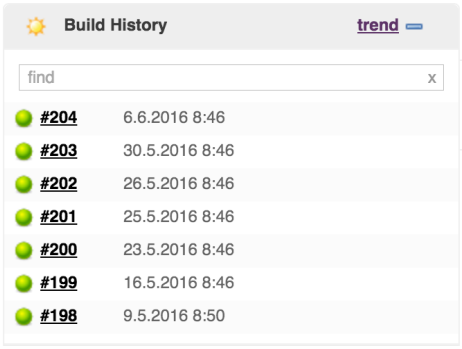
První program se nazývá *Signature Test Tool* a slouží ke snadnému porovnání signatur dvou různých implementací nebo porovnání rozdílných verzí stejného API. U porovnání implementací dochází k ověření, zda-li implementace obsahuje všechny potřebné členy API nutné pro zajištění kompatibility, reportuje nově přidané členy a kontroluje specifické chování těchto nových členů API. Při porovnání dvou různých verzí stejného API se ověřuje možnost, zda-li je starou verzi možné nahradit její novější verzi bez ovlivnění jakýkoliv dalších částí aplikace.
Druhý program je pojmenován jako *API Coverage Tool* a slouží pro analýzu odhadu pokrytí vybraného API testovací sadou. Pro zdroj analýzy je využíván podpisový soubor reprezentující specifikaci daného API. Vyhodnocení analýzy je založeno na počtu členů veřejných tříd (*public class*), kteří odpovídají API specifikaci. [44]

V roce 2014 přešel projekt *SigTest* pod OpenJDK[38] a je dostupná ke stažení ze stránek CloudBees [45]. Nástroj má velice silnou podporu svých autorů a nové sestavení vycházejí v krátkých intervalech, což dokazuje obrázek 18.

---

[37] Technology Compatibility Kit je soubor testovacích scénářů pro technologii Java.

[38] OpenJDK je open source implementace Java platformy dostupná zdarma.



obrázek 18 - historie posledních sestavení nástroje SigTest [45]

Z obrázku je vidět, že aktuální verzí[39] je sestavení s číselným označením 204. Nová sestavení budou k dispozici v následujících dnech a zaměřují se především na odstranění nedostatků předešlých verzí (reportováno na stránkách projektu jako open issues).

Podle oficiální dokumentace projektu [46] je pro spuštění nástroje *SigTest* nutné mít nainstalován *Ant* ve verzi 1.7 a vyšší. Zároveň je možné využít existující *sigtest-plugin* dostupný pro nástroj *Maven*, který umožňuje využít funkce pomocí sestavovacích skriptů.

4.3.8. REVAPI

Dalším z řady nástrojů pro analýzu API je program napsaný v Javě, který je znám pod názvem *Revapi (*ze zkratky Java Api Checker*)*. Kromě samotné analýzy API, pod kterou nespadají pouze Java třídy, ale také různé konfigurační soubory, deskriptory nebo schémata, se kterými dokáže nástroj pracovat, je druhou hlavní funkcí sledování všech změn v API, které se v ní odehrály během vývoje. Nástroj umožňuje vývojáři zvýraznit všechny vybrané úmyslné změny, které jsou v některých případech nevyhnutelné. [47]

Revapi je navržen jako rozšiřitelný nástroj, který je teoreticky schopen pracovat s libovolným jazykem (nemusí být nutně programovací, stačí když bude mít rozložitelnou strukturu). V současné době však existuje podpora pouze pro API jazyku Java. Hlavními vlastnostmi nástroje jsou:

[39] Ke dni 8.6.2016.

- velký počet kontrol API umožňující kategorizovaných podle vlivu na zdroj, zahrnuje binární a sémantickou kompatibilitu,
- sledování zvyklostí v kontrolovaných knihovnách včetně jejich závislostí
- podpora Java 8 konstrukcí
- pokročilé filtrovací funkce pro kontrolu a reklasifikaci zjištěných problémů, včetně možnosti filtrovat dle přítomnosti anotace
- *Maven* plugin zahrnující automatickou kontrolu závislostí

Pro spuštění nástroje je nutná Java 8 update 40 nebo novější verze. Nástroj může být používán jako samostatná distribuce, vložením do Java projektu (viz obrázek 19) či jako *Maven* plugin. [48]

```
Revapi revapi = Revapi.builder().withAllExtensionsFromThreadContextClassLoader().build();

AnalysisContext analysisContext = AnalysisContext.builder()
    .withOldAPI(API.of(...))
    .withNewAPI(API.of(...))
    .withConfigurationFromJSON("json").build();

revapi.analyze(analysisContext);
```

obrázek 19 - ukázka vložení funkce nástroje do Java kódu [48]

Revapi jako své přímé konkurenty bere nástroje *Japicmp* (kapitola 3.3.2), *Clirr* (kapitola 3.3.4) a *SigTest* (kapitola 3.3.7). I přesto se autoři domnívají, že jejich nástroj nabízí oproti konkurentům funkce, pomocí kterých dosáhnete výsledků, jichž nejste schopni dosáhnout pomocí zmíněných předešlých nástrojů. Bližší porovnání jednotlivých nástrojů je shrnuto v kapitole 4.3.11.

Aktuální verzí[40] je *Revapi 0.5.2,* která byla vydána v březnu 2016. Veškeré balíky, aplikace a pluginy nástroje jsou dostupné ke stažení na oficiálních stránkách projektu.

[40] Ke dni 8.6.2016.

4.3.9. JAR COMPARE

Jedná se o malý nástroj, který umožňuje porovnávat Java třídy uvnitř Java archivu a zobrazit specifické rozdíly mezi třídami. Nástroj provádí hlubokou analýzu porovnání, využívá dekompilaci class souborů a zobrazuje rozdíly přímo na daných řádkách zdrojového kódu. *Jar Compare* se hodí na použití v případech:

- kdy je nutné ověřit, zda-li došlo ke změnám mezi verzemi SW nebo sestaveními bez nutnosti mít uložen zdrojový kód pro každou verzi,
- kdy chceme sledovat změny v externích knihovnách, které SW používá.

Nástroj se spouští pomocí příkazu

JAVA -CLASSPATH JARC [-D"FULL_PATH_TO_DIFF"] LEFT.JAR RIGHT.JAR

nebo

JAVA -JAR JARC.ZIP [-D"FULL_PATH_TO_DIFF"] LEFT.JAR RIGHT.JAR.

Výstupem aplikace je xml soubor, který znázorňuje jednotlivé rozdíly řádek zdrojového kódu. Ukázka výstupu nástroje je zobrazena na obrázku 20.

```
▼<xml>
  ▼<item name="Class3.class">
    ▼<disassembly>
      ▼<!--
          there is a difference in disassembly; the left disassembly is in left tag, the right - in right tag diff result in diff
       -->
      ▼<diff>
         <line n="1">4,5d3</line>
         <line n="2">< java.lang.String field2;</line>
         <line n="3"><</line>
       </diff>
      ▼<left>
         <line n="1">public class Class3 extends java.lang.Object{</line>
         <line n="2">java.lang.String field1;</line>
         <line n="3"></line>
         <line n="4">java.lang.String field2;</line>
         <line n="5"></line>
         <line n="6">public Class3();</line>
         <line n="7">Code:</line>
         <line n="8">0:.aload_0</line>
        ▼<line n="9">
           1:.invokespecial.#1; //Method java/lang/Object."<init>":()V
```

obrázek 20 - výstupní soubor xml obsahující rozdíly kontrolovaných archivů [49]

Aplikace je dostupná jako samostatný archiv a podle oficiálních stránek [49] neumožňuje žádné možnosti integrace s nástroji jako *Maven* nebo *Ant*.
Není známa žádná podpora programu a poslední verzí, která byla uvolněna je 1.1 z roku 2006. Dostupná ke stažení je rovněž ze stránek projektu.

4.3.10. JAVA COMPATIBILITY CHECKER

Jedná se o nástroj, který byl vyvinut skupinou spadající pod výzkumné centrum NTIS[41] na Západočeské Univerzitě v Plzni.

Jádrem nástroje je *Java Compatibilty Checker* (zkráceně *JaCC*). Jedná se o sadu klientských nástrojů, které pomocí metody reverzního inženýrství překládá existující binárních soubory do paměti modulů a provádí jejich kontrolu. Jednoduchý diagram, jak nástroj funguje je zobrazen na obrázku 21. Po vytvoření a kontrole modelů jsou nalezené problémy nahlášeny uživateli. [50]

obrázek 21 -workflow nástroje JaCC [50]

Nástroj je vhodné použít v několika fázích vývoje softwaru a to především v etapách vývoje, návrhu, testování nebo při verifikaci kompatibility.

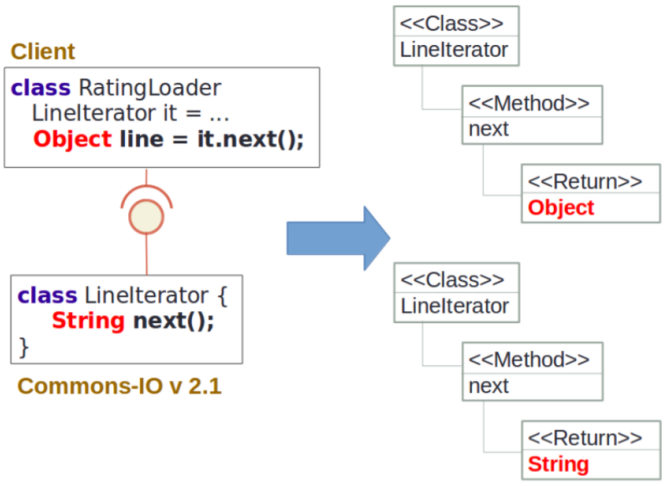
Mezi použití může patřit zajišťování kvality (Quality Assurance), kdy lze pomocí nástroje odhalit nekonzistenci či způsobenou nekompatibilitu v použitém API, ať už se jedná o kontrolu knihoven vlastních nebo třetích stran. Dalším z vhodných použití se jeví nasazení nástroje při integračních projektech, kde se využívá hodně externích knihoven, jenž obsahují neznámý kód a jeho vnitřní vazby nejsou známy. Autoři v dokumentaci nástroje dále uvádějí jako další možné použití nástroje při potřebě softwarové certifikace. Aby certifikační autorita mohla udělit nějakému softwaru certifikát, je nutné ověřit kompletní kód aplikace a odhalit všechny možné nedostatky, k čemuž může pomoci reverzní analýza.

Princip

Nástroj funguje na základě analýzy byte kódu a jeho verifikace. Do analyzéru vstupuje proud byte kódu ze tříd získaných rozbalením JAR souboru. Tento kód

[41] NTIS - New Technologies for Information Society

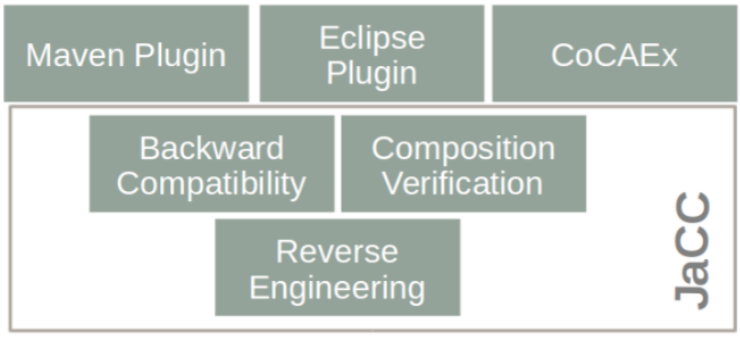
je parsován pomocí ASM[42] frameworku a jednotlivé prvky API jsou po rekonstrukci ve formě připomínající Java třídy z původního zdrojového kódu. Důležité je, aby tyto prvky byly dostupné při jediném průchodu aplikací. Ukázka reverzní analýzy je znázorněna na obrázku 22.



obrázek 22 - princip reverzní analýzy nástroje JaCC [50]

Možnosti integrace

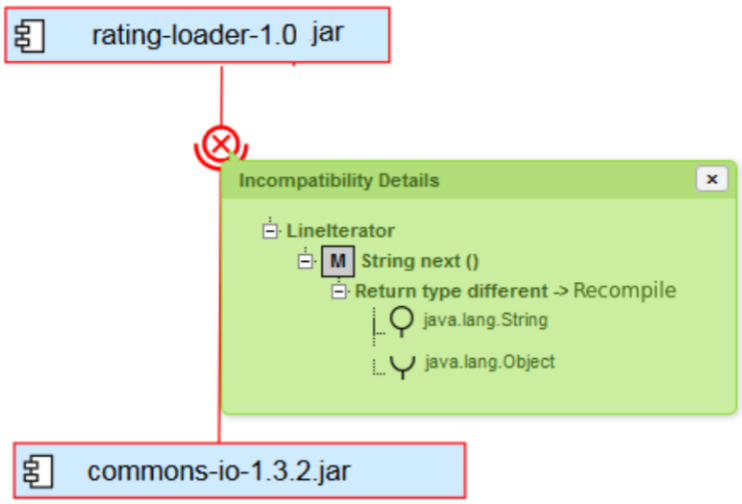
Díky použité architektuře nástroje (obrázek 23) je možné nástroj možné integrovat do ostatních nástrojů, případně vývojových prostředí. V Současné době existuje *Maven* plugin, kdy v průběhu sestavení probíhá verifikace pomocí *JaCC.* Také existuje integrace do populárního IDE Eclipse, kdy jsou poté verifikační reporty z nástroje zobrazeny přímo v Eclipse konzoly a kromě toho plugin poskytuje dva nové uživatelské pohledy zobrazeny v okně prostředí (JaCC Results View a JaCC Markers View).



[42] ASM je framework sloužící pro analýzu a manipulaci Java byte kódu. Oficiální stránky - http://asm.ow2.org/.

obrázek 23 - architektura nástroje JaCC [50]

Kromě zmíněných integračních možností nástroj spolupracuje také s aplikací *CoCAEx* (Complex Component Applications Explorer), která byla vyvinuta rovněž skupinou ReliSA[43] na Katedře informatiky a výpočetní techniky. Primárním účelem aplikace je průchod a průzkum velkých softwarových diagramů s přehlednou grafickou vizualizací pro uživatele. V kombinaci s nástrojem *JaCC* umožní *CoCAEx* zobrazit výsledky v přehledné grafické formě včetně viditelných návazností. Ukázku grafického výstupu nalezené nekompatibility ukazuje obrázek 24.



obrázek 24 - grafická vizualizace nástroje CoCAEx nalezené nekompatibility

Nástroj JaCC zatím není volně k dispozici, stále probíhá jeho vývoj a tak nejsou známé žádné podrobnosti o způsobech distribuce, aktualizacích a podpoře od autorů.

4.3.11. Porovnání nástrojů pro testování

V této kapitole je uvedeno porovnání všech popsaných nástrojů zabývající se ověřením kompatibility a testováním. Vzájemné porovnání je bráno z konzultačního hlediska a zaměřuje se na aktuální pohled na nástroj, jeho podporu, komunitu, oblíbenost a další kritéria. Tabulka zahrnující porovnání

[43] ReliSA je název výzkumné skupiny na ZČU v Plzni, jejíž celý název je Reliable Software Architectures research group.

(tabulka 4) obsahuje volně dostupné informace, jako například poslední aktualizaci nástroje, možnosti integrace, celkový počet stažení, počet nevyřešených problémů apod. Jsou vyplněné pouze ty informace, které bylo možné dohledat z volně dostupných zdrojů, zejména z oficiálních stránek daných nástrojů (uvedeno vždy v odpovídající kapitole nástroje), případně z dokumentací daných nástrojů [GitHub, Stackoverflow, Google, SoundBees].

	Bndtools	Japicmp	Japicc	Clirr	JDiff	Japitools	Sigtest	Revapi	Jar compare	JaCC
poslední aktualizace	Leden 2016 (v 3.2)	Květen 2016 (v 0.7.2)	Duben 2016 (v 1.7)	Květen 2007 (v 0.7)	Duben 2013 (v 1.1.1)	Listopad 2006 (v 0.9.7)	Květen 2016 (v 3.1)	Březen 2016 (v 0.5.2)	2006 (v1.1)	2016
možnosti integrace s dalšími nástroji	Maven, Gradle, Eclipse	Maven, Findbugs & Checkstyle reporty	Ne	Maven, Ant	Maven plugin	Ne	Maven, Ant	Maven, Ant	Maven, Ant	Maven, Eclipse, CoCAEX
počet celkových stažení	5831 za poslední rok z Eclipse MP	307 (pouze GitHub)	X	5348	13624	X	X	X	X	X
trend stahování nástroje	konstantní, 500 stažení měsíčně (EMP)	vzrůstající	X	klesající, 20 stažení měsíčně	kolísavá, 50-100 stažení měsíčně	X	X	X	X	X
počet otevřených issues	146	12 (github)	7	7 (již se neaktualizuje)	8	1 (neaktuální)	X	9	X	X
samostatně použitelné	Ne (nutný Eclipse)	Ano	Ano	Ano	Ano	Ano	Ne (nutný Ant)	Ano	Ano	Ano
Počet autorů	Celkem 22, kolem 5 aktivních	Celkem 10, hlavní 1 aktivní	Celkem 3, pouze 1 aktivní	1 (Lars Kuhne)	1 (Matthew B. Doar)	celkem 7, hlavní 1	X	1 (Lukáš Krejčí)	X	3 (Ježek, Holý, Daněk)
počet odpovědí (stackoverflow)	338	7	15	31	45	9	5	26	13	X
počet Google výsledků za rok 2016	132	21	170	104	90	4	21	90		X
počet commitů	2856	493	55	X	X	X	X	796	X	X
počet uvolněných verzí	45	23	12	3	10	2	3	7	1	ještě neuvolněno

tabulka 4 - porovnání nástrojů pro testování z konzultačního pohledu

Z tabulky je patrné, že vybrané atributy porovnávaných nástrojů se mezi sebou velmi liší. První atribut se zaměřuje na aktuálnost nástroje a uvádí poslední vydanou dostupnou verzi[44]. Mezi nástroje, které jsou již delší dobu bez podpory a nebyla vydána aktuální verze, patří zejména *Clirr*, *Japitools*, *Jar compare* a také *JDiff*. Zajímavostí je, že tyto nástroje jsou stále hojně využívány komunitou Java vývojáři či testery. Druhý ukazatel zmiňuje možnosti integrace daného

[44] Uváděno ke dni 12.6.2016.

nástroje zejména s nástroji umožňující sestavit výslednou verzi aplikace. Mezi nejčastěji podporované nástroje pro build patří jednoznačně *Maven*, který je pro sestavení Java aplikací velmi oblíben díky vysoké míře přizpůsobení.

Další ukazatel porovnával celkový počet stažení, jehož číslo je uvedené jen u nástrojů, kde tato informace byla dohledatelná z oficiálních zdrojů. S tímto ukazatelem souvisí i trend stahování, který porovnával, zda-li je o nástroj větší či menší zájem z dlouhodobějšího měřítka stahování uživateli. U nástrojů, jejichž komunity se podílejí na testování nástroje a poskytují autorům zpětnou vazbu, je uvedena i aktuální informace týkající se počtu otevřených nedostatků (issues).

Protože ne všechny nástroje je možné využívat jako samostatné aplikace, ale už od počátku byly vyvíjeny jako doplněk k jinému nástroji či IDE, je u každého nástroje uvedena možnost samostatného použití. Zajímavým ukazatelem je také počet autorů, kteří se v průběhu vývoje podíleli na samotném programování[45].

Další dva ukazatele porovnávají četnost výskytů daného nástroje na serveru Stackoverflow[46], případně ve vyhledávači Google, kde se započítávaly pouze odkazy s datem od 1.1.2016 do 12.6.2016[47].

Poslední atributy zmiňují počet commitů, pokud je dostupný zdrojový kód daného nástroje a celkový počet uvolněných a dostupných verzí[48].

---

[45] Autorem se bere osoba, přispívající přidáváním případně úpravou zdrojového kódu.

[46] Stackoverflow je komunita více než 4,7 milionu vývojářů. Oficiální web . http://stackoverflow.com.

[47] Toto omezení data bylo aplikováno z důvodu příliš velkého počtu odkazů u některých nástrojů. Zároveň novější informace je více relevantní než odkaz starý několik let. Samozřejmě tento ukazatel znevýhodňuje nástroje, jež dosáhly vrcholu používání v delší minulosti.

[48] Uváděny jsou všechny dohledatelné verze daných nástrojů z dostupných zdrojů.

5. Návrh řešení pro ověřování kvality SW

5.1. Úvod do návrhu ověřování kvality softwaru

Při vývoji Java projektu prochází produkt několika vývojovými fázemi (kapitola). V jednotlivých etapách lze zajišťovat kvalitu softwaru zejména provedením odpovídajících testů, které ověřují správnost, funkčnost, bezpečnost, integritu nebo čistotu zdrojového kódu dané části systému.

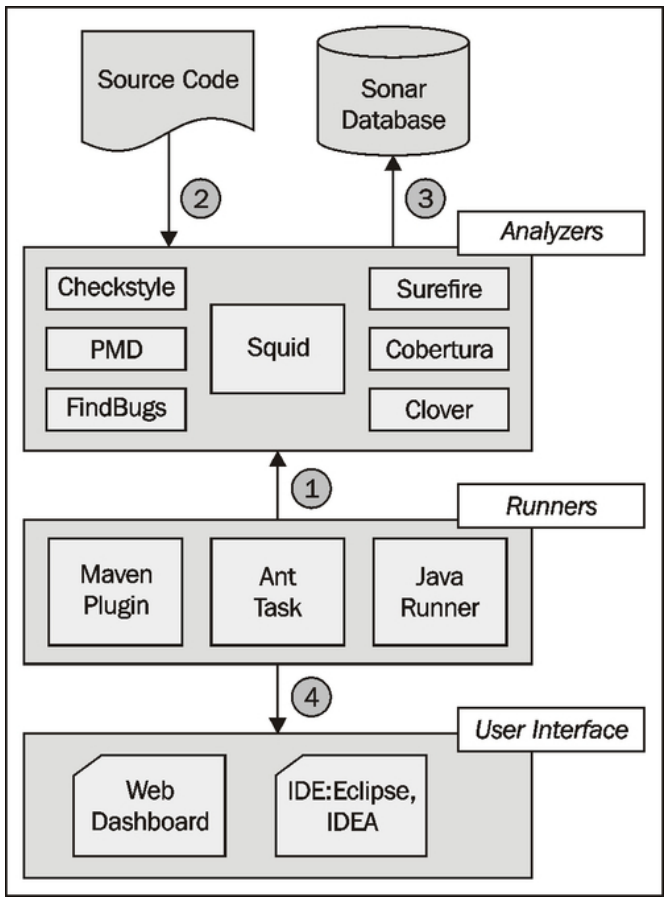
V této kapitole budou uvedeny návrhy řešení pro ověřování kvality softwaru zahrnující ověření statické analýzy i kompatibility pro projekty psané v programovacím jazyku Java. Smyslem této kapitoly je ukázat možnost integrace vybraných popsaných nástrojů pro ověření statické analýzy (kapitola), testování (kapitola 4.2) a sestavení (kapitola 4.3).

Následující příklady diagramů integrace a kombinace použití vývojového prostředí, daných typů testů a vybraných nástrojů jsou vztahovány k obecnému Java projektu s důrazem vyvinout kvalitní software. V reálném případě budou daná kritéria ovlivněna podmínkami zadavatele, technologickými možnosti vývojové společnosti nebo zákazníka. Z tohoto důvodu nelze tvrdit, že některý z návrhů řešení je vhodný pro jakýkoliv Java projekt a vždy bude v první řadě záviset na podmínkách, které si zadavatel a zprostředkovatel softwarového systému dohodnou. Architektura návrhů je vytvořena z celkového pohledu na projekt a nezabývá se programátorskými detaily a specifikacemi.

5.2. Návrh řešení pro ověřování kvality SW pomocí platformy SonarQube

Velice oblíbeným nástrojem k zajišťování kvality softwaru je platforma *SonarQube*, která je podrobněji popsána v kapitole 4.1.6. Díky svým širokým možnostem integrace lze nástroj využít v kombinaci s mnoha dalšími nástroji zajišťujícími statickou analýzu kódu nebo či ověřování kompatibility softwaru.

Následující obrázek 25, převzatý z literatury [51], znázorňuje nejvyšší úroveň komponent v návrhu řešení pro ověření kvality softwaru a jejich interakce mezi nimi.



obrázek 25 - diagram komponent pro ověření kvality SW pomocí platformy SonarQube [51]

Ve znázorněném návrhu je vyvolána analýza kódu některým z nástrojů, který zajišťuje průběh jednotlivých procesů SA. Mezi ty patří například *Maven* nebo *Ant*. Poté Sonar přijme požadavek a spustí analýzu projektu, která může být prováděna nad zdrojovým kódem i nad byte kódem. Analýza je prováděna pomocí analyzátorů, jejichž počet a konkrétní pravidla závisí na aktivovaném profilu Sonaru. Mezi často využívané analyzátory, které je možné použít v kombinaci s nástrojem *SonarQube,* patří zejména *Checkstyle* (kapitola 4.1.2), *PMD* (kapitola 4.1.4) nebo *FindBugs* (kapitola 4.1.3). Sonar mimo to obsahuje i vlastní řešení pro analýzu kódu, která se nazývá *Squid*. Analýza kódu lze rozložit mezi více nástrojů a pomocí pravidel lze jejich nastavení přizpůsobit tak, aby odpovídalo požadavkům projektu. V praxi to znamená, že můžeme využít nástroj *Checkstyle* pro provedení statické analýzy kódu a zároveň využít nástroj *Findbugs* pro odhalení výkonnostních (performance) chyb. Kombinace pravidel a nástrojů závisí vždy na daném projektu a požadavků na výsledný softwarový produkt. Výsledky z celého průběhu analýzy od všech pomocných nástrojů jsou

uloženy do databáze, která uchovává veškeré odhalené chyby a varování slouží pro sledování historie a odpovídajících změn.V poslední fázi dochází k aktualizaci uživatelského prostředí a jeho komponent pomocí nových dat získaných z celého průběhu analýzy. Na výstup se lze podívat pomocí webového rozhraní, kde *SonarQube* nabízí přehledné grafické dashboardy vizualizující stav projektu. Zde může uživatel různých rolí (programátor, projekt manažer, SW tester) nalézt informace týkající se všech odhalených chyb, nedostatků, chybných konvencí nebo bezpečnostních rizik. Sonar umožňuje nastavit reportování také do samotného vývojového prostředí, kde lze zkontrolovat a opravit reportovaný kód. Mezi podporované IDE patří například *Eclipse* nebo *intelliJ IDEA*.

Pro ověření kompatibility softwaru pomocí konkrétního nástroje třetí strany je možné využít nástroje, které byly podrobněji popsány v kapitole 4.3. Ke zmíněnému návrhu by šlo využít rozsáhlých funkcí *Bndtools,* který je pravidelně aktualizován a dle porovnání (kapitola 4.3.11) patří mezi nejoblíbenější testovací nástroj v IDE *Eclipse*.

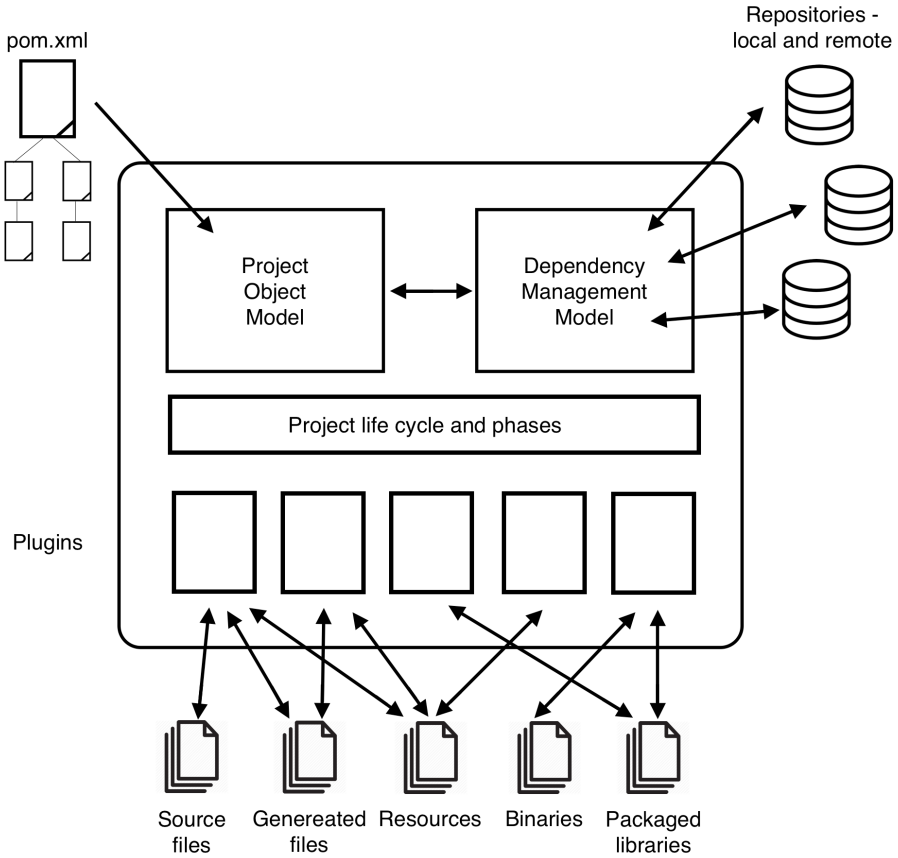
Ve spojitě integrovaném (countinuosly integrated) prostředí se proces spouští automatizovaně pomocí sestavovacího serveru. Ten zkontroluje zdrojový kód v daném úložišti (může být lokální i vzdálené) a provede kompilaci a spuštění všech jednotkových a integračních testů, po jejichž provedení se vytvoří potřebná sestavení. Po těchto krocích nastává průběh analýzy projektu pomocí *SonarQube* popsaný výše.

U obsáhlých projektů je zmíněná procedura všech postupně navazujících kroků časově náročným procesem a proto je třeba spuštění přesně plánovat. Často se tak v praxi využívá takzvaných nočních sestavení (night builds), kdy je analýza spuštěna v nočních hodinách, tedy v čase, kdy jsou vývojáři neaktivní. V příštím přístupu do systému *SonarQube* již budou mít dostupné veškeré aktualizované informace ohledně vývoje projektu.

5.2.1. NÁVRH ŘEŠENÍ PRO OVĚŘENÍ KVALITY SW S VYUŽITÍM PLUGINU REVAPI V NÁSTROJI MAVEN

Sestavovací nástroj *Maven* (kapitola 4.2.1) slouží pro vytvoření výsledných sestavení aplikace (build), jejichž vytvoření je podmíněno splněním nadefinovaných postupných fází, které musí být úspěšně provedeny. Mezi výchozí fáze patří - validace, kompilace, testování, vytváření balíků, instalace a nasazení. V případě specifických potřeb projektu je možné další fáze přidat, mezi často využívané patří *clean (čištění)* a *site* vytvářející dokumentaci projektu [52].

Maven je založen na architektuře umožňující použít doplňkové moduly (plugins), což umožňuje použít funkce nástrojů třetích stran v jednotlivých fázích sestavování. Funkční model nástroje *Maven* je zobrazen na obrázku 26.



obrázek 26 - funkční model nástroje Maven s vrstvou pluginů [52]

Konfigurace týkající se souboru *pom.xml* a připojení k úložištím byla blíže popsána v kapitole 4.2.1. Nyní se zaměříme na samotné testovací přídavné

moduly (plugins), které lze při sestavovacím procesu použít. Těch existuje velké množství a výhodou je, že nové pluginy stále vycházejí od vývojářů třetích stran.

Několik základních přídavných modulů je obsaženo hned v základní instalaci - například *surefire,* který spustí provedení jednotkových testů pomocí frameworku *JUnit, nebo failsafe,* který vyvolá spuštění integračních testů je-li plugin tak nakonfigurován. Pokud chceme pro testování využít funkce některého z nástrojů třetích stran, který je možný použít v kombinaci s platformou *Maven,* lze ho přidat do sestavovacího nástroje úpravou souboru *pom.xml*.

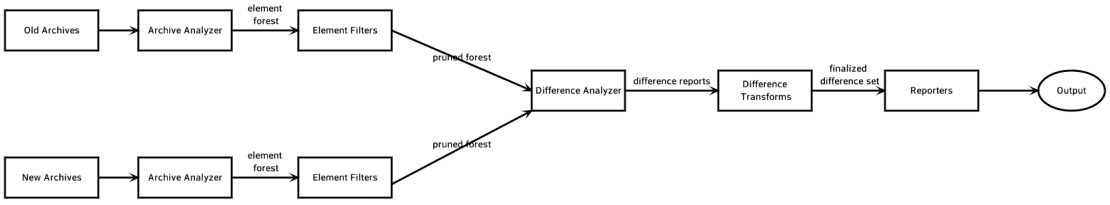
Pokud chceme při sestavování aplikace spustit testování analyzující změny, které se odehrály v Java API od poslední verze, lze k tomu využít přídavný modul nástroje *Revapi* (kapitola 4.3.8). Pro přidání funkcí nástroje *Revapi* do sestavovacího nástroje *Maven* je třeba provést následující změny v souboru *pom.xml* [47]:

```
<BUILD>
  ...
  <PLUGINS>
    ...
    <PLUGIN>
      <GROUPID>ORG.REVAPI</GROUPID>
      <ARTIFACTID>REVAPI-MAVEN-PLUGIN</ARTIFACTID>
      <VERSION>{VERSION}</VERSION>
      <DEPENDENCIES>
        <DEPENDENCY>
          <GROUPID>ORG.REVAPI</GROUPID>
          <ARTIFACTID>REVAPI-JAVA</ARTIFACTID>
          <VERSION>{VERSION}</VERSION>
        </DEPENDENCY>
        ...
      </DEPENDENCIES>
      <EXECUTIONS>
        <EXECUTION>
          <ID>CHECK</ID>
          <GOALS><GOAL>CHECK</GOAL><GOALS>
        </EXECUTION>
      </EXECUTIONS>
    </PLUGIN>
    ...
  </PLUGINS>
  ...
</BUILD>
```

Po provedení předchozí úpravy a po zahrnutí org.revapi:revapi-basic-features:XXX do *Maven* artefaktu (XXX označuje verzi) lze nyní využít specifické cíle (goals) přidané nástrojem Revapi. Těmito cíli jsou:

revapi:check - používá se jako součást sestavení pro zajištění konfigurovatelné úrovně kompatibility,
revapi:report - slouží pro generování jednoduchého reportu pro generované stránky *Maven* projektu,
revapi:validate-configuration - účelem je ověření konfigurace *Revapi,* zejména pro ladění problému s konfigurací,
revapi:update-versions - aktualizuje popisek verze v *pom.xml* na základě semver[49] pravidel,
revapi:update-release-properties - aktualizuje soubor *release.properties* s uvolňovanými verzemi, tak jak je stanoveno podle semver pravidel.

Diagram, který znázorňuje jakým způsobem nástroj *Revapi* porovnává změny v Java API mezi dvěma archivy, je zobrazen na obrázku 27.



obrázek 27 - diagram průběhu porovnání Java API nástrojem Revapi [47]

Pokud nedošlo při sestavování k problémům, které by zastavily proces sestavení, výsledný *Maven* report nyní obsahuje i výstup týkající se ověření kompatibility pomocí pluginu nástroje *Revapi*.

Podobným způsobem by se daly do procesu sestavování přidat další rozšiřující moduly nástrojů třetích stran, které podporují integraci s platformou *Maven*. Celkový výčet z analyzovaných nástrojů je uveden v tabulce 4 v kapitole 4.3.11.

[49] Semver je zkratka z anglického výrazu Semantic Versioning, pravidla jsou k nalezení na oficiálních stránkách http://semver.org/.

5.2.2. Zhodnocení návrhů řešení

Předešlé uvedené návrhy řešení slouží k otestování Java projektu v průběhu vytváření sestavení (build) aplikací.

První návrh je zaměřený na kvalitu kódu provedením statické analýzy pomocí platformy *SonarQube* a přídavných nástrojů třetích stran jako *PMD*, *Checkstyle* nebo *FindBugs.* Výhodou tohoto návrhu je automatizace všech procesů a možnost naplánování průběhu sestavení včetně zmíněných testů na dobu, kdy jsou vývojáři neaktivní. Také snadná integrace s podporovanými IDE umožní jednoduchou konfiguraci dle požadavků softwarového produktu. Jako nevýhodu návrhu můžeme považovat omezení se na používání pouze nástrojů, které je možné integrovat s nástrojem *SonarQube,* například již zmíněná podporovaná IDE.

Druhý návrh se více soustředí na otestování kompatibility softwaru, konkrétně porovnání API dvou různých verzí Java archivu. Toho je docíleno pomocí pluginu nástroje *Revapi,* který je možné použít s oblíbeným sestavovacím nástrojem *Maven*. Výhodou takového řešení jsou široké možnosti integrace umožňující využít mnoha doplňkových modulů nástrojů třetích stran. To vývojářům či testerům umožní použít využít takovou sadu nástrojů, jejichž funkcemi je možné dosáhnout požadavků na celkovou kvalitu a kompatibilitu softwarového produktu. Nevýhodou může být složitější konfigurace celého procesu sestavení, zejména při použití většího množství nástrojů, zahrnující veškeré testy, které mají být před výsledným sestavením aplikace provedeny.

6. Závěr

Tato práce ve své první části vysvětlila přínosy testování softwarového produktu v různých etapách životního cyklu vývoje softwaru. Byly představeny příklady používaných vývojových metodik se zaměřením na proces testování softwaru. Zároveň byly popsány nejčastěji používané druhy testů sloužící pro otestování kompletního softwarového produktu v průběhu vývoje se zaměřením na etapy životního cyklu a vývojové role v projektovém týmu.

Ve druhé části byly zhodnoceny existující nástroje třetích stran, které slouží pro statickou analýzu Java kódu. Ta představuje nezbytnou činnost pro zajištění kvality softwaru. Dále byly detailně popsány a porovnány vybrané nástroje dostupné na trhu, pomocí kterých lze testovat kompatibilitu softwaru psaného v programovacím jazyce Java.

V poslední části byla navržena a zhodnocena řešení pro ověření kvality softwaru s využitím analyzovaných nástrojů. Vhodným řešením pro běžné Java softwarové projekty může být řešení založené na platformě *SonarQube* s použitím přídavných pluginů pro ověření kompatibility softwaru za pomoci a integrovaných nástrojů lze efektivně provést statickou analýzu kódu.

Kvalita softwaru představuje velice důležitou roli v současném vývoji Java aplikací, kdy se stále více využívají již existující knihovny třetích stran, jejichž vnitřní vazby nemusí být vždy dobře známé. Z tohoto důvodu lze pomocí dobře zvolených nástrojů a navržených testů předejít budoucí nekompatibilitě systému či některých jeho komponent.

ZDROJE

[1] DIETRICH, JEZEK, BRADA. What Java Developers Know About Compatibility, And Why This Matters [online]. 2014, říjen [cit 2016-06-12]. Dostupné z: http://arxiv.org/pdf/1408.2607v1.pdf

[2] DARCY D. Joseph. Kinds of Compatibility: Source, Binary, and Behavioral [online]. 2008, duben [cit. 2016-06-12]. Dostupné z: https://blogs.oracle.com/darcy/entry/kinds_of_compatibility

[3] WIRTH, Niklaus. A Brief History of Software Engineering [online]. 2008, 10 [cit. 2016-02-29]. Dostupné z: http://people.inf.ethz.ch/wirth/Miscellaneous/IEEE-Annals.pdf

[4] CHAPMAN, James. Software Development Methodology [online]. Washington, DC. Dostupné z: http://www.hyperthot.com/pm_sdm.htm

[5] ROYCE, Winston. Managing the Development of Large Software Systems [online]. Proceedings of IEEE WESCON, 1970. Dostupné z: http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf

[6] BOEHM, Barry W. A Spiral Model of Software Development and Enhancement. TRW Defense Syst. Group, Redondo Beach, CA, roč. 21, č. 5, August 1988. ISBN: 0018-9162

[7] SURSHETWAR, Laleen. Manual testing tutorials [online]. 2014. Dostupné z: http://laleensurshetwar.blogspot.cz/2014/05/v-model.html

[8] Rational Unified Process : Best Practices for Software Development Teams, [online]. Rational Software. Dostupné z: http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf

[9] PIGEON, Xavier. A graphical representation of the Test-Driven Development lifecycle, Wikipedia, 2014. Dostupné z: https://en.wikipedia.org/wiki/Test-driven_development#/media/File:TDD_Global_Lifecycle.png

[10] BECK, Kent. Test-driven development: by example. Boston: Addison-Wesley, c2003. ISBN 0321146530.

[11] PATTON, Ron. Testování softwaru. Vyd. 1. Praha: Computer Press, 2002. Programování. ISBN 80-7226-636-5.

[12] HLAVA, Tomáš. Druhy, typy a kategorie testů [online]. 2011. Dostupné z: http://testovanisoftwaru.cz/category/druhy-typy-a-kategorie-testu/

[13] J*ONES, M. Tim*, IBM, *Static and dynamic testing in the software development life cycle, září 2013,* IBM [online]. Dostupné z: http://www.ibm.com/developerworks/library/se-static/

[14] *PECH, Václav, Statická analýza kódu - za kód bez chyb,* JetBrains. Cit. březen, 2016 [online]. Dostupné z: http://java.cz/dwn/1003/8159_czjug-vaclav-pech-static-code-analysis.pdf

[15] INTELLIJ IDEA HELP, *Code Inspection. Cit. duben 2016 [online]. Dostupné z:* https://www.jetbrains.com/help/idea/15.0/code-inspection.html

[16] OFFICIAL WEBSITE OF FINDBUGS, *Findbugs - Find bugs in Java Programs*, cit. 8.4.2016 [online]. Dostupné z: http://findbugs.sourceforge.net/index.html

[17] Zenika BLOG, FindBugs, a static analysis tool, vyd. srpen 2012. Obrázek dostupný z: http://blog.zenika.com/wp-content/uploads/2015/07/test_findbugs_on_project.PNG

[18] OFFICIAL WEBSITE OF CHECKSTYLE, *Checkstyle*, cit. 8.4.2016 [online]. Dostupné z: http://checkstyle.sourceforge.net/index.html

[19] Official Eclipse website, Eclipse Checkstyle Plugin - Setting up a project. Obrázek dostupný z: http://eclipse-cs.sourceforge.net/#!/project-setup

[20] Official website PMD, PMD tool [online]. Cit. 22.5. 2016. Dostupné z: http://pmd.github.io/

[21] AppPerfect, AppPerfect Java Code Test [online]. Cit. 22.5.2016. Dostupné z: http://www.appperfect.com/products/java-code-test.html

[22] Official AppPerfect website, IntelliJ IDEA IDE Integration - Screenshots. Obrázek dostupný z: http://www.appperfect.com/images/eclipse-netbeans/idea_java_code_test.png

[23] Eclipse Marketplace, AppPerfect Code Analyzer [online]. Cit. 22.5.2016. Dostupné z: https://marketplace.eclipse.org/content/appperfect-code-analyzer

[24] Zeroturnaround, Static code analysis tools for source code, byte code and the big picture, vyd. duben 2014. Cit. 22.5.2016 [online]. Dostupné z: https://zeroturnaround.com/rebellabs/developers-guide-static-code-analysis-findbugs-checkstyle-pmd-coverity-sonarqube/2/

[25] SonarQube Documentation, Reporting, vyd. duben 2015 [online]. Dostupné z: http://docs.sonarqube.org/display/SONARQUBE50/Reporting

[26] Official SonarQube website, SonarQube [online]. Dostupné z: http://www.sonarqube.org/

[27] OWASP Lapse Project, Lapse [online]. Cit. 26.5.2016. Dostupné z: https://www.owasp.org/index.php/OWASP_LAPSE_Project

[28] HPE Security Fortify Static Code Analyzer Data Sheet, *Build better code and secure your software. Cit. dubbed 2016. [online] Dostupné z:* http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/

[29] Through the Looking-Glass, *IBM Application Security Insider, vyd. Listopad 2011. Cit. 10.4.2016 [online]. Dostupné z:* http://blog.watchfire.com/wfblog/2011/11/through-the-looking-glass.html

[30] JELÍNEK, Libor. Maven build a project management Java aplikací — příručka školení, vyd. červenec 2015. Cit. 26.5.2016. [online]. Dostupné z: http://docplayer.cz/1200947-Maven-build-a-project-management-java-aplikaci-prirucka-skoleni-libor-jelinek-virtage-software-ljelinek-virtage-cz.html

[31] Apache Maven, Project Object Model. Dostupné z: https://cs.wikipedia.org/wiki/Apache_Maven

[32] Software testing tutorials and automation, How To Create and Run Selenium Test In Maven Project From Command Prompt [online]. Dostupné z: http://www.software-testing-tutorials-automation.com/2015/04/how-to-create-and-run-selenium-test-in.html

[33] Java skripta VŠE, Nástroj Ant [online]. Cit. 29.5.2016. Dostupné z: http://java.vse.cz/pdf/Skripta383-ant.pdf

[34] Official manual website for Ant, Writing a simple buildfile [online]. Cit. 29.5.2016. Dostupné z: https://ant.apache.org/manual/using.html

[35] OSGi Alliance, The Dynamic Module System for Java [online]. Cit. 29.5.2016. Dostupné z: https://www.osgi.org/developer/architecture/

[36] Bndtools, Bndtools Repositories [online]. Cit 29.5.2016. Dostupné z: http://bndtools.org/repositories.html

[37] OSGi Alliance, Dependencies [online]. Dostupné z: http://enroute.osgi.org/img/tutorial_base/dependencies-repo-5.png

[38] GitHub, Japicmp. [online]. Cit. 29.5.2016. Dostupné z: https://github.com/siom79/japicmp

[39] StackOverFlow, How to identify a missing method (Binary Compatibility) in a JAR statically [online]. Dostupné z: http://i.stack.imgur.com/S73Ds.jpg

[40] Sourceforge Project Page, Clirr [online]. Cit. 29.5.2016. Dostupné z: http://clirr.sourceforge.net/

[41] Sourdeforge, JDiff - An HTML Report of API Differences [online]. Cit. 29.5 2016. Dostupné z http://jdiff.sourceforge.net/

[42] Official website of Japitools, Java API Compatibility Testing Tools [online]. Cit. 7.6.2016. Dostupné z: http://www.sab39.org/Software/Japitools/42/

[43] Debian package tracker, Japitools [online]. Dostupné z: https://tracker.debian.org/pkg/japitools

[44] OpenJDK Wiki, SigTest [online]. Cit. 7.6.2016. Dostupné z: https://wiki.openjdk.java.net/display/CodeTools/SigTest

[45] CloudBees, Project SigTest [online]. Dostupné z: https://adopt-openjdk.ci.cloudbees.com/job/sigtest/

[46] Docs Oracle, SigTest User's guide, Únor 2011 [online]. Cit. 8.6.2016. Dostupné z: http://docs.oracle.com/javame/test-tools/sigtest/2_2/sigtest2_2_usersguide.pdf

[47] Official Revapi website, Revapi - Full-featured API checker for Java and beyond [online]. Cit. 8.6.2016. Dostupné z: http://revapi.org/

[48] GitHub, Revapi [online]. Cit. 8.6.2016. Dostupné z: https://github.com/revapi/revapi

[49] Official Jar Compare website, Jar Compare [online]. Cit. 8.6.2016. Dostupné z: http://extradata.com/products/jarc/

[50] Ježek, Holý, Daněk, Preventing Composition Problems in Modular Java Applications, 2013. Cit.10.6.2016. Západočeská Univerzita v Plzni.

[51] ARAPIDIS, Charalampos S. Sonar code quality testing essentials: achieve higher levels of software quality with sonar. Birmingham: Packt Publishing, 2012

[52] IT Academy, SoftServe. Java, Automation Build. Maven for QC, leden 2015 [online]. CIt.18.6.2016. Dostupné z: http://www.slideshare.net/ssuser220b38/maven-54639726

PŘÍLOHY

K této diplomové práci je přiloženo CD s originálem diplomové práce.