MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Údržba a modernizace webových aplikací

BAKALÁŘSKÁ PRÁCE

Ondřej Božek

Brno, jaro 2008

Prohlášení

Prohlašuji, že tato bakalářská práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Vedoucí práce: Mgr. Jan Pavlovič

Shrnutí

Údržba a modernizace aplikací je velmi důležitou fází životního cyklu webové aplikace a v budoucnu bude její význam pravděpodobně ještě růst. Této problematice však dosud bohužel nebyla věnována pozornost, jakou by zasluhovala. V předkládané práci se pokusím popsat uvedenou problematiku a identifikovat faktory ovlivňující udržovatelnost a běžně používané postupy. Poznatky následně aplikuji na konkrétním případu neudržované webové aplikace.

Klíčová slova

Java, testování, udržovatelnost, údržba software, vývoj software, webové aplikace

Obsah

Kapitola 1

Úvod

Většina oblastí lidské činnosti prochází neustálým vývojem. Je to přirozený proces, důsledek zájmu o danou problematiku. V mnoha oborech je pokrok velmi rychlý. Pokud chce člověk obstát, musí držet krok a neustále se přizpůsobovat. Nástroje, které byly včera moderní, jsou dnes již zastaralé. Pro modernizaci také hraje fakt, že včasné uplatnění nové technologie může pomoci získat cenný náskok před konkurencí.

Tato skutečnost samozřejmě velkou měrou platí i v oboru informačních technologií. Uživatelé postupně zjišťují, že aplikace již přesně nesplňují jejich požadavky. S časem se začínají projevovat skryté chyby, funkcionalita neodpovídá všem současným nárokům. Pokrok v informačních technologiích často zpřístupní nové možnosti, jež se později stanou nutností.

Dobrá správa by měla zabránit klesání užitné hodnoty webové aplikace. Aby v rapidně se rozvíjejícím prostředí internetu obstála, je však častěji potřeba užitnou hodnotu zvyšovat. Je nutné, aby software v tomto rychle měnícím se prostředí co nejlépe plnil své poslání. Webové aplikace proto musí být spolehlivé a co možná nejlépe dostupné. Odchylky od správného fungování či náhlé výpadky je třeba co nejrychleji odstranit. To jsou jedny z hlavních důvodů, proč musí být webové aplikace udržovány a modernizovány, aby plnohodnotně zastávaly svou funkci.

Pokud tedy chceme udržet aplikace v chodu, je nezbytné zajistit jejich průběžnou údržbu. Bohužel ne vždy se tak stane. Není obtížné narazit na zcela neudržované systémy. Pokud situace přinutí provozovatele k započetí prací na systému, je důležité, aby tento systém nebyl příliš zastaralý. V opačném případě je často výhodnější, jej začít budovat úplně znovu. Horní hranicí bývá stáří 5–10 let, v závislosti na konkrétním případě. Pokud systém splňuje tyto požadavky, je vhodné ho nejdříve důkladně prozkoumat a analyzovat. Poté přichází na řadu samotná modernizace.

Cílem této práce je objasnit problematiku udržovatelnosti a modernizace aplikací. Dále se pokusím identifikovat faktory ovlivňující udržovatelnost a popsat postupy běžně používané pro efektivní údržbu. Poznatky následně aplikuji na konkrétním případu neudržované webové aplikace.

V následující kapitole se pokusím osvětli nejdůležitější pojmy potřebné pro orientaci v problematice. V třetí kapitole identifikuji základní faktory ovlivňující udržovatelnosti a modernizovatelnost aplikace a nejčastější problémy s tím spojené. Ve čtvrté kapitole potom popíši postupy udržby a modernizace aplikací, s důrazem na zachování co nejvyšší udržovatelnsti. Pátá kapitola popisuje aplikaci poznatků a postupů uvedených v předcházejících kapitolách na konkrétním případu webové aplikace.

Kapitola 2

Údržba aplikací

2.1 Vývoj software

Z obecného hlediska je vývoj software pojem, užívaný v softwarovém inženýrství, označující proces počátečního vytvoření aplikace a její následné úpravy z rozličných důvodů. V tomto procesu lze tedy rozlišit dvě rozdílné fáze. První označuje samotné vytvoření software, zatímco druhá označuje proces údržby či rozvoje již vytvořené aplikace, dále jen údržba aplikace. Fáze vytvoření software samozřejmě do značné míry určuje předpoklady a potenciál následující fáze údržby aplikace.

Z pohledu této práce je zajímavější fáze údržby aplikace. Profesor Meir Manny Lehman[1] uvedl jako první ze zákonů vývoje software: Software musí být průběžně vyvíjen, nebo se postupně stane nepoužitelným [8]. To znamená, že pokud nebude aplikace průběžně udržována, bude její užitná hodnota klesat, až se nakonec zcela vytratí. Z dlouhodobého hlediska se jedná o nejdůležitější období v životním cyklu typické používané webové aplikace. Rozhoduje o její úspěšnosti a použitelnosti. U většiny systému se též jedná o zdaleka nejnákladnější fázi. Vyžádá si více než 90% prostředků použitých během celého života daného software. Z nákladnosti ani samotné existence této fáze nelze usuzovat na nedostatky ve fázi vytváření aplikace[2]. Správné postupy při vývoji aplikace sice mohou následnou údržbu značně ulehčit. Na druhou stranu úspěšnost dobře vyvinuté aplikace může zapříčinit velkou uživatelskou obec a tudíž velký potenciál k odhalování chyb a vznášení požadavků na funkcionalitu či portabilitu.

2.2 Údržba software

Údržba software je nejčastěji chápána jako oprava chyb v kódu. Nicméně rozličné studie a průzkumy v průběhu let ukázaly, že většina (přes 80%) úsilí vynaloženého na údržbu, je věnována jiným než opravným úkonům [9]. Tento chybný vjem je rozšířen hlavně uživateli, odesílajícími hlášení o chybách v programu, jež jsou ve skutečnosti požadavky na funkční vylepšení systému. Lientz a Swanson identifikovali čtyři základní kategorie údržby aplikace

1. Meir Manny Lehman známý jako „father of software evolution“. Byl profesorem na Imperial College London a později na Middlesex University. Zabýval se fenoménem vývoje software a takzvanými Lehmanovými zákony softwarového vývoje.
2. „...any successful piece of software will inevitably be maintained ...“ [2]

(1980), jež byly později aktualizovány a mezinárodně standardizovány v ISO/IEC 14764 z roku 2006 [ ]:

Korektivní údržba Reaktivní modifikace softwarového produktu provedená po dodání za účelem opravy objevených problémů.

Adaptivní údržba Modifikace softwarového produktu provedená po dodání za účelem udržení použitelnosti software v měnícím se prostředí.

Zdokonalující údržba Modifikace softwarového produktu provedená po dodání za účelem zlepšení výkonu a udržovatelnosti.

Preventivní údržba Modifikace softwarového produktu provedená po dodání za účelem zjištění a opravy skrytých chyb dříve než se tyto chyby projeví.

Všechny z uvedených kategorií jsou uplatněny, pokud nastane podnět pro změnu. I když byly tyto kategorie doplněny či rozšířeny mnoha autory, mezinárodní standart si zachovává původní formu. Klíčové otázky údržby aplikací jsou jak administrativního tak technického rázu. Hlavními administrativními úkoly jsou např.: soulad s prioritami zákazníka, personální otázky, výběr organizace zabezpečující údržbu, odhad a optimalizace nákladů. Technická problematika zahrnuje: omezené porozumění systému, testování, analýza vlivu plánovaných změn, měření udržovatelnosti.

2.3 Ujasnění základních pojmů

Korektivní údržba. Snad každý běžně vyvíjený netriviální software má chyby. Ať se dodavatel snaží sebevíc, lze jen s těží očekávat bezchybné provedení. Už jen proto, že pokoušet se vyvinout perfektní systém bývá finančně neúnosně a neefektivní. Výhodnější je, poskytnout kvalitní údržbu s velkou pravděpodobností nalezení a opravy chyb dříve, než se stihnou významným způsobem projevit. Snaha o dokonalé provedení má smysl pouze u aplikací, zastávajících mimořádně zodpovědné funkce, jako jsou např.: softwarové vybavení raketoplánu, bezpečnostní systém jaderné elektrárny atd. I přes všechnu snahu vývojářů si uživatel nikdy nemůže být úplně jistý bezchybností software. Důkazem je například pád vesmírné sondy Mariner I. (1962), kvůli chybnému opisu ručně psaného kódu do počítače, nebo incident na Sovětském plynovodu, jež se zapsal do dějin jako největší nejaderný výbuch v historii. Důkladné testování, jako základní a nenahraditelný nástroj k odstranění chyb, je navíc u rozsáhlých, komplikovaných systémů velmi nepraktické až nemožné. Tento druh údržby ukusuje ze zdrojů překvapivě malý díl (méně než 20% [9]), přesto jej s čistým svědomím nelze nikdy vynechat.

Adaptivní údržba. Softwarové systémy nefungují v izolaci. Typicky musí interagovat s databázemi, uživatelskými rozhraními, síťovými protokoly, externím software, různými hardwarovými platformami atd. V průmyslovém odvětví informačních technologií se kterýkoliv, nebo všechny tyto prvky mohou měnit během velmi krátké periody. Tento druh údržbářských aktivit bývá klíčový pro udržení, či rozšíření použitelnosti aplikace. Díky rozsáhle adaptivní údržbě je možné systém provozovat v rozmanitém prostředí, jsou redukovány omezující požadavky. Do této kategorie například patří většina úkonů provedených v souvislosti s přípravami na rok 2000.

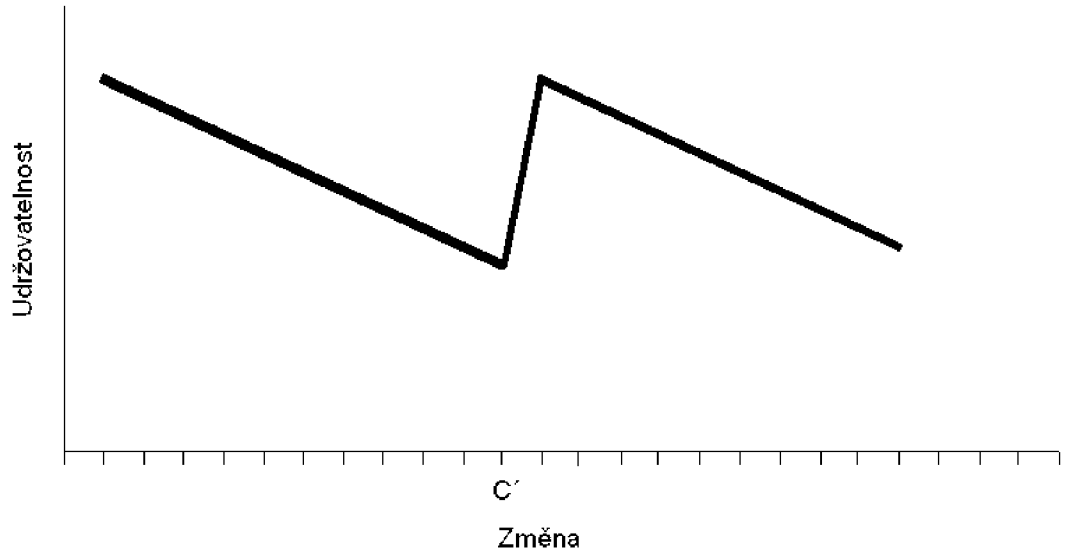
Zdokonalující údržba. Nic není dokonalé a stejně tak uživatelé a distributoři nejsou nikdy úplně spokojeni. Často je podnětem pro vylepšování systému soupeření s konkurencí. I když je systém jednoznačně úspěšný, vždy se najde někdo kdo potřebuje nové, nebo vylepšené součásti. Jindy se objeví popud změnit rozhraní, nebo způsob fungování osvědčené části systému. Intenzivní zdokonalování může vést k podstatné proměně aplikace, jež ji posune na jinou úroveň. Dobrým příkladem tohoto jevu je dramatický pokrok velké části Open Source aplikací. Otevřenost kódu dává mnoha nadšencům cestu k obohacení produktu kapkou své invence.

Preventivní údržba. Důvody pro tuto činnost mohou být čistě interní. Takovéto změny ze své podstaty nebývají pro běžného uživatele přímo rozeznatelné. Avšak nelze je kvůli tomu považovat za méně důležité. Jako v mnoha jiných odvětvích i zde platí že prevence je nejvýhodnějším přístupem. Potenciálně může ušetřit mnoho prostředků a tvoří oporu pro jednodušší spravovatelnost systému v budoucnu. Krom toho má pozitivní vliv na spolehlivost a poskytuje základnu pro budoucí rozvoj aplikace. Zástupcem této kategorie je totiž i akt zkvalitňování dokumentace jež umožňuje rychlejší proniknutí do tajů systému. Samotné seznamování se se systémem, dle studií, vývojářům zabere okolo 50% veškerého pracovního času [4].

Udržovatelnost. Je mnoho definic této vlastnosti software, v jádru velmi podobných. Například: Nenáročnost s jakou může být softwarový systém nebo komponenta upravována za účelem odstranění chyb, zvýšení výkonu nebo jiných vlastností, či přizpůsobení změněnému prostředí [5]. Bohužel pro určení udržovatelnosti neexistuje žádný obecně platný algoritmus a tudíž ji nelze přesně a jednoznačně měřit. Nejčastěji bývá určována expertním posouzením. Naneštěstí s rostoucím počtem zásahů do systému většinou klesá jeho udržovatelnost, pokud nejsou provedeny strukturální změny, viz [Křivka udržovatelnosti]. Jak je uvedeno výše náklady na údržbu jsou vysoké, proto se vyplatí věnovat této vlastnosti zvýšenou pozornost.

Správa konfigurací. Anglicky Configuration Management, odtud zkratka CM. Je to model technické správy zaměřující se na vytvoření a údržbu konzistence provedení produktu a jeho funkčních atributů s požadavky a plány po celý jeho život. CM může být také definován jako správa bezpečnostních prvků a záruk skrze kontrolu změn provedených na hardware, software, firmware, dokumentaci, testech, dokumentaci testů, po celý životní cyklus informačního systému.

Správa konfigurací má tradičně čtyři elementy:



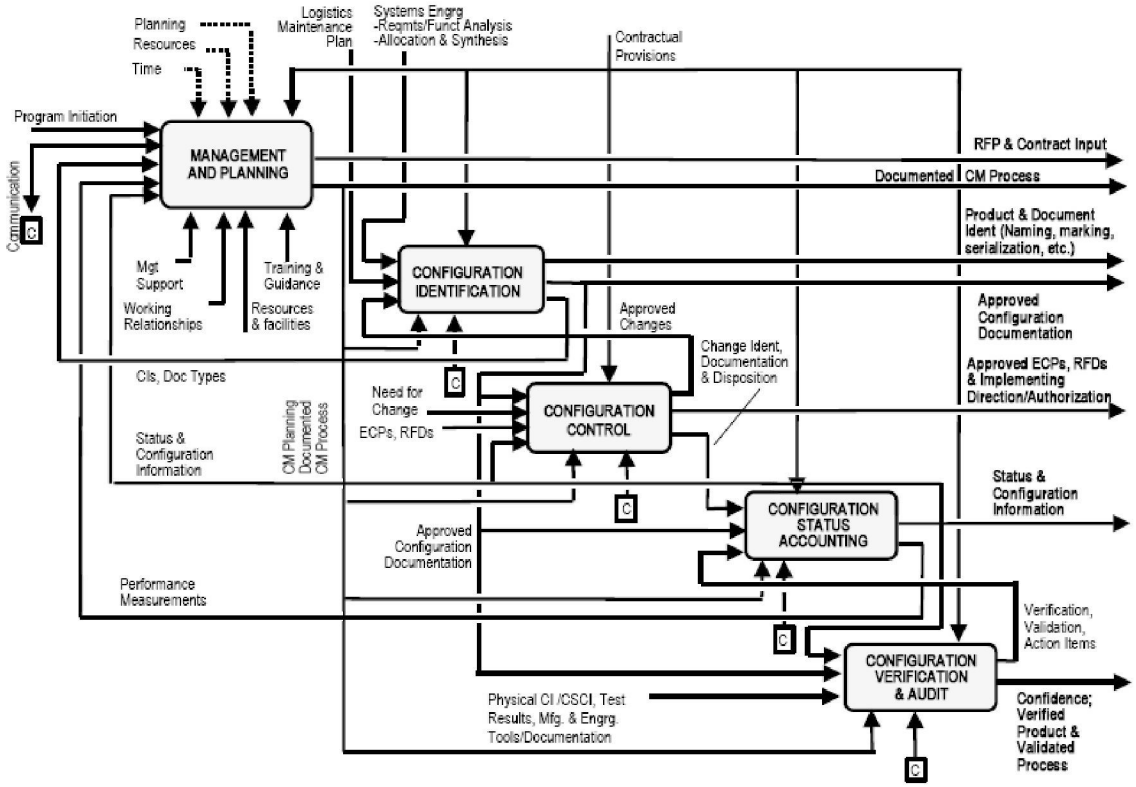
Obrázek 2.1: Křivka udržovatelnosti. Změna C′ (restrukturalizace systému) zvýší (dočasně) udržovatelnost.

Identifikace konfigurace Je to proces identifikace každého atributu konfiguračních položek. Konfigurační položka je produkt (hardware/software), který má význam pro koncového uživatele. Tyto položky jsou zaznamenány v dokumentaci. Toto zaznamenání vynutí ovlivnění procesu formální kontroly změn v konfiguraci, pokud jsou tyto atributy změněny.

Kontrola změn konfigurace Jedná se o skupinu procesů a schvalovacích úrovní nutných ke změně atributů konfiguračních položek.

Evidence stavu konfigurace Schopnost zaznamenat a podat zprávu o atributech konfiguračních položek v jakýkoliv moment.

Verifikace a prověřování konfigurace Rozděluje se na funkcionální a fyzickou část. Je prováděno buď v okamžiku dodání produktu nebo při uplatnění změny atributu položky. Funkcionální část ověřuje, zda jsou funkcionální a výkonostní atributy konfigurační položky splněny, zatímco fyzická část ověřuje, zda je konfikurační položka instalována v souladu s požadavky podrobného dokumentačního plánu.



Obrázek 2.2: Model aktivit správy konfigurací. [11]

Kapitola 3

Analýza

V této kapitole se pokusím popsat základní faktory mající vliv na udržovatelnost aplikací. Identifikuji činitele, jež ulehčují proces zásahu do již existujícího software. Zmíním také některé problémy a negativní vlivy týkající se této problematiky či základní vlastnosti, které by měli aplikace mít, aby mohli být dobře udržovány a modernizovány.

3.1 Základní předpoklady udržovatelnosti aplikací

Rozdíly v udržovatelnosti mohou být mezi softwarovými systémy propastné. Vše začíná u kvality původního návrhu a implementace. V ideálním případě je systém budován s ohledem na to, že bude později udržován a rozvíjen. V takovém případě byly už při návrhu do systému zařazeny různé nástroje a technologie, jež pro samotnou funkci systému nemají podstatný význam. Tyto nástroje však podstatně pomohou při vývoji, údržbě, modernizaci, správě a testování systému. Mezi tyto nástroje a technologie patří pokročilé logování, různé testovací metody, vývojářská menu v systému atd. viz [Postup údržby a modernizace aplikací].

Dalším důležitým faktorem je jak dobře (pokud vůbec) byla aplikace dosud udržována. Pokud je systém průběžně nepřetržitě udržován, běžně dochází k předávání znalostí a informací mezi stávajícími a nově příchozími zaměstnanci. V opačném případě může nastat stav, kdy kvůli fluktuaci zaměstnanců už není k dispozici nikdo s potřenou úrovní znalostí systému. Krajní situace nastane pokud v důsledku předchozího jevu a nahromaděného množství podnětů pro zásah do systému, je po odhadu nákladů zjištěno, že se vyplatí začít pracovat na úplně novém software.

3.2 Dokumentace

Dokumentace tvoří základní podklady pro jakoukoliv práci se systémem. Důležitá je dokumentace designu a struktury, stejně tak dokumentace vývoje softwarového systému. Neúplnost nebo dokonce úplná absence těchto podkladů je problém u jakéhokoliv netriviálního softwarového systému. I lidé, kteří pracovali přímo na vývoji systému po čase ztrácí přehled o návrhu či vlastní implementaci, pokud nemají možnost si tyto znalosti oživit. Špatná dokumentace, která nekoresponduje se skutečným stavem aplikace, obvykle naopak pokusy o zásah do systému značně zkomplikuje. Výsledky analýzy vlivu změn jsou zbytečné. Provedené změny mohou mít neočekávatelné důsledky. Vše se projeví, až při samotném zásahu

do software, což může být fatální. Nespolehlivost těchto podkladů si následně vynutí ověřování všech informací z nich čerpaných. Z toho plyne nemalý nárůst náročnosti jakýchkoliv úprav software.

3.3 Verze sestavení aplikace

S dokumentací také souvisí popis případných verzí sestavení a rozdílů mezi nimi. Je možné, že změna kterou chceme realizovat, je již v některé z verzí sestavení implementována. Případně můžeme zjistit, že některé sestavení je jako výchozí bod příhodnější. Tato dodatečná dokumentace také může mnoho napovědět o vývoji a implementaci aplikace. Každá takováto informace má svoji nezanedbatelnou hodnotu. Byla by škoda mít k dispozici různé verze software, na které bylo vynaloženo úsilí, věnovány prostředky na jejich vývoj, ale přitom je předem vyřadit. Mohlo by se také stát, že budeme usilovně pracovat na něčem co už dávno před námi někdo vyřešil, stáčí se jen podívat na jinou verzi sestavení.

3.4 Verze komponent potřebných pro sestavení a běh aplikace

Jedná se o dvě skupiny software:

- Software potřebný při sestavování aplikace. Jde o různé knihovny, jary, frameworky atd. Např.: Java Genetics Algorithms Package, Junit, Java Server Faces.
- Software potřebný při běhu aplikace. Takzvaný runtime. Patří sem databázové systémy, runtime knihovny, interpretry, virtuální stroje, aplikační servery atd. Např.: databáze Oracle, Java Runtime Enviroment, GlassFish.

Podrobné informace o veškerém tomto software by měli být součástí dokumentace. Důležité jsou verze těchto komponent. Pokud je známe, většinou není problém dohledat podrobné informace. Tyto informace jsou potřeba při následné správě a modernizaci aplikace. Tento software ovlivňuje mnoho vlastností systému. Aktualizace či náhrada některé z těchto komponent může mít pozitivní vliv na výkon aplikace. Může v nich být chyba, která je propagována dále do aplikace. Při rozšiřování systému o novou funkcionalitu, je znalost prostředků nabízených komponentami výhodu, někdy i nutností. Veškeré změny komponent by samozřejmě měly být důkladně otestovány, i když se jedná například jen o aktualizace, viz [Testování aplikace].

3.5 Testování aplikace

Testování software je postup používaný k měření kvality počítačového software. Kvalita není absolutní, je to hodnota platná pro nějakou osobu. V důsledku toho testování nemůže nikdy kompletně ověřit korektnost jakéhokoliv počítačového software. Testování může prokázat přítomnost chyb, nemůže však dokázat jejich absenci. Skýtá ovšem možnost kriticky

porovnat stav a chování produktu vůči specifikaci. Testování je jedním z nástrojů, nezbytných při provádění jakýchkoliv změn v aplikaci.

Do tohoto procesu patří zejména spouštění různých programů za účelem objevení chyb. Testováním není možné spolehlivě odstranit veškeré chyby v aplikaci jednoduše proto, že není možné nasimulovat veškeré vstupní hodnoty a následné zkontrolovat výstupní hodnoty. Chyby v software vznikají následujícím procesem. Programátor udělá chybu (omyl), který vede k defektu ve zdrojovém kódu. Pokud je tato defektní část kódu provedena, systém v určitých situacích produkuje špatné výsledky, jež vedou k selhání. Ne všechny defekty zapřičiní selhání. Například chyby v tzv. mrtvém kódu[1] nikdy nepovedou k selhání. Defekty se mohou proměnit v selhání, pokud se změní prostředí systému. Příklady těchto změn mohou být např.: software provozovaný na novém hardware, změny ve zdrojových datech, interakce s odlišným software a v neposlední řadě se mohou skryté chyby projevit po provedení změn v systému. Z výše uvedeného je zřejmé, jak významnou roli hraje testování při provozování, údržbě a modernizaci aplikací.

Ne příliš často bývá systém vyvinut s optimálními vykonnostnímy parametry. S blížícim se termínem odevzdání aplikace se snižuje důraz kladený na efektivitu kódu. Výkonnost aplikace zvyšuje uživatelský komfort a celkovou užitnou hodnotu. Také při ladění výkonnosti je testování nezbytné. V tomto případě se jedná o zátěžové testování. Tato metodika spočívá ve vytvoření situace kdy určitý počet uživatelů pracuje se systémem a v současném měření výkonnostních parametrů v závislosti na počtu uživatelů. Cílem zátěžového testování není kontrolovat správnost systému, ale zjistit zda systém bude dostatečně rychlý i při větším počtu připojených uživatelů. Zátěžové testování se obvykle provádí s 1,5 násobkem SWL (Safe Working Load), což je bezpečný počet uživatelů, kdy systém obvykle pracuje bez problémů či větších časových prodlev. Tyto zkušební postupy mohou také pomoci ověřit jiné kvalitativní atributy aplikace, jako jsou škálovatelnost, spolehlivost nebo míra využití dostupných prostředků.

Velmi blízko k zátěžovým testům má tzv, stresové testování, pomocí kterého je zjišťována stabilita, robustnost a dostupnost systému. Stresové testování spočívá v testování za očekávanou provozní kapacitou, často až ke krajnímu bodu, za účelem pozorování důsledků.

Vlastnosti aplikací, které mají pozitivní vliv na jejich testovatelnotst:

Spustitelnost Čím lépe aplikace pracuje, tím účinněji může být otestována. Chyby nebrání běhu programu, může být testován a vyvíjen současně.

Přehlednost To co je vidět lze lépe testovat. Aplikace produkují různé výstupy pro různé vstupy. Stavy systému a proměnné jsou viditelné za chodu, faktory ovlivňující výstup jsou zřejmé, nesprávné výstupy jsou lehce identifikovatelné, vnitřní chyby jsou automaticky detekovány a zaznamenávány, zdrojový kód je dostupný.

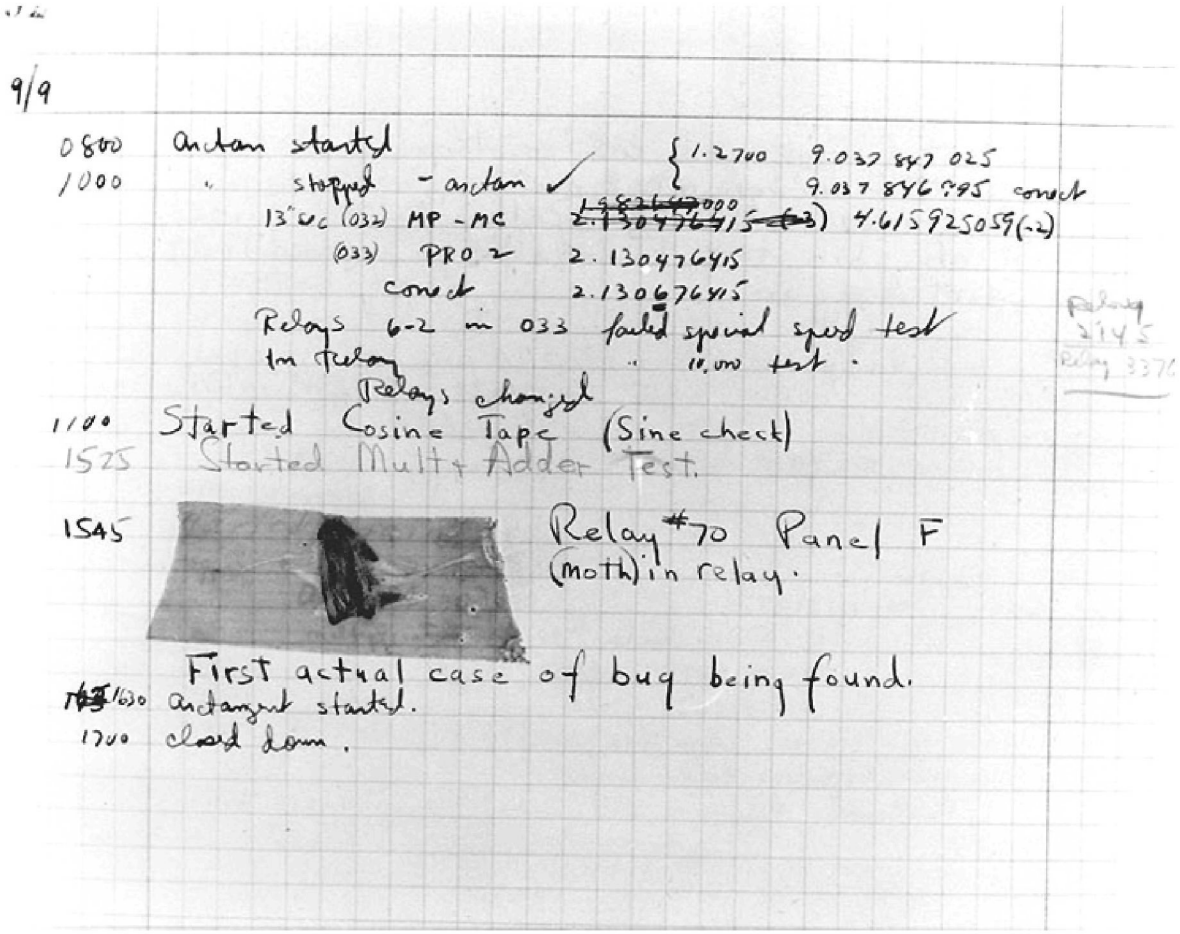
1. Mrtvý kód je část zdrojového kódu programu, která nemůže být nikdy vykonána.

Kontrolovatelnost/Řiditelnost Čím lépe lze software kontrolovat/řídit, tím spíše lze testování automatizovat a optimalizovat. Všechny možné výstupy jsou generovány určitou kombinací vstupů, veškerý kód je spustitelný nějakou kombinací vstupů, vstupní a výstupní formát je konsistentní a strukturovaný, testy mohou být specifikovány, automatizovány a reprodukovány.

Jednoduchost Čím menší je rozsah testované oblasti, tím rychleji a lépe ji lze zkontrolovat. Jednoduchost funkce, struktury, kódu.

Stabilita Čím méně je změn v softwaru, tím lépe. Změny nejsou časté, jsou řízené, neovlivní provedené testy.

Srozumitelnost Čím více informací je k dispozici, tím jsou testy inteligentnější. Srozumitelný návrh, srozumitelné závislosti mezi vnitřními, vnějšími a sdílenými komponentami, dostupnost a srozumitelnost technické dokumentace.



Obrázek 3.1: První počítačový "Bug". Mol chycený mezi spoji relé # 70, Panel F, Mark II. Aiken Relay Calculator, při testování tohoto stroje na Harvardské Univerzitě, 9. září 1947. Operátoři přilepili mola k logu počítače s popisem: "First actual case of bug being found". Také se zmínili, že provedli "debugging" stroje. Tyto termíny se ujaly a používají se do dnes. Wikipedia

Kapitola 4

Postup údržby a modernizace aplikací

Modernizace a údržba aplikací bývá, jak bylo zmíněno výše, o mnoho jednodušší u systémů, které byly dosud nějakým způsobem udržovány. Samozřejmě i mezi plynule udržovanými systémy bývají rozdíly. Záleží na intenzitě a kvalitě údržby i mnoha dalších faktorech. V následujícím se budu zabývat spíše neudržovanými systémy, aby bylo možné postihnout i činnosti, které u udržovaných systémů není nutné většinou provádět.

Mezinárodní standard ISO/IEC 14764:2006 [6] rozlišuje 6 základních procesů údržby aplikací:

Implementační proces Skládá se z přípravných a přechodových aktivit jako jsou, koncepce a vytvoření plánu údržby, přípravy pro nakládání s problémy zjištěnými během vytváření software. To vše pokud možno v souladu s CM.

Proces analýzy problémů a modifikací Tento proces je prováděn jakmile se aplikace stane zodpovědností údržbářů. Údržbář musí analyzovat každou připomínku, potvrdit ji (reprodukováním situace) a ověřit její platnost, vyšetřit ji a navrhnout řešení, zdokumentovat připomínku a návrh řešení a konečně získat oprávnění k aplikaci modifikace.

Proces implementace modifikace Náplní tohoto procesu je samotná implementace modifikace na základě připomínky.

Proces schválení modifikace Probíhá na straně autora připomínky. Jeho náplní je ověření zda modifikace vyřešila problém.

Proces migrace Nejedná se o pravidelně vykonávaný proces, neprobíhá na denní bázi. Může a nemusí být prováděn. Je vykonáván např. pokud systém musí být přenesen na jinou platformu bez jakékoliv změny funkcionality.

Vyřazovací proces Tento proces také neprobíhá denně. Jedná se o vyřazení nevhodného kusu software.

4.1 Příprava

Před započetím jakýchkoliv modernizačních či udržbářských prací, je nejdříve potřeba provést analýzu systému. Zmapovat ho a pochopit jeho strukturu. Je třeba shromáždit co nejvíce informací o aplikaci. V tomto je velmi nápomocná dokumentace. Pokud chybí nebo je ve špatném stavu, měli bychom se ji snažit intenzivně doplňovat a upřesňovat už od samého začátku. Dalším krokem je zmapování případných verzí sestavení. V těchto počátečních procesech mohou významně pomoci lidé, kteří se buď přímo podíleli na vývoji, nebo nějakým jiným způsobem získali podrobnější vědomosti o systému.

Dále je vhodné vytvořit plán údržby a modernizace systému. Sepsat všechny nashromážděné požadavky pro zásah do systému a seřadit je logicky tak, aby byly prováděny v co nejefektivnějším pořadí. Při sestavování tohoto plánu je také třeba dbát na prioritu těchto změn. Důležitější změny v aplikaci se samozřejmě provádí dříve tak, aby byla zajištěna co nejlepší funkce systému. Následně je také vhodné vytvořit budoucí plán údržby systému. Je třeba do něj prozíravě zakomponovat všechny předpokládané činnosti potřebné pro optimální chod systému. Dá se očekávat, že se tento plán údržby bude dále vyvíjet s tím jak porostou zkušenosti údržbářů a jak se budou měnit nároky na aplikaci. Vytvoření tohoto plánu má poměrně nízkou naléhavost, většinou se dokončuje až po provedení všech prioritních úkonů. Je však vhodné ho formovat už od samého začátku.

4.2 Testování

Jak bylo zmíněno výše, testování je nepostradatelným nástrojem užívaným pří modernizaci a údržbě webových aplikací. Umožňuje po provedení jakýchkoliv změn v aplikaci ověřit její správnou funkci. Nejdříve je vhodné osvětlit základní pojmy, jejichž znalost je pro správné testování aplikací nutná. Následně bude uvedeno několik základních typů testů hrajících významnou roli v průběhu celé údržby a modernizace webové aplikace.

4.2.1 „Doubles“

„Double“ v překladu dvojník je obecný výraz označující jakýkoliv typ předstíraného objektu užívaného výhradně pro testovací účely.

4.2.1.1 Mock objekty

Mock objekty jsou simulované objekty, které předstírají skutečné objekty, na rozdíl od nich jsou však plně pod kontrolou. Tester většinou vytváří mock objekty, aby mohl s jejich pomocí otestovat nějaký jiný objekt. Mock objekty se podobají například pokusným králíkům. Vědci je mají plně pod kontrolou a mohou na nich důkladně otestovat nejrůznější preparáty, které, pokud projdou testy, budou později nasazeny člověku. Testování těchto experimentálních preparátů přímo na lidech, by asi nebylo tak efektivní.

Mock objekty v unit testech simulují komplexní reálné objekty a jsou tedy užitečné, pokud je obtížné, nebo dokonce nemožné použít reálné objekty v testech. Mock objekty implementují stejné rozhraní jako napodobované objekty, ponechávajíc testovaný objekt v nevědomosti, zda je v kontaktu s reálným či mock objektem. Falešné (viz „fake" objekty) a mock objekty umožňují otestovat a odhalit skryté chyby v takzvaném mrtvém kódu. Tyto skryté chyby by se jinak projevily až při změnách v kódu nebo prostředí systému. Zatímco oživení skryté chyby po zásahu do kódu by nemělo, za předpokladu dobře prováděného testováni, způsobit vážnější problémy, oživení chyby změnou prostředí systému, kterážto bývá nezřídka náhlá a nečekaná, může být velmi nepříjemným překvapením. Podchycení těchto skrytých kostlivců je umožněno schopností mock objektů simulovat i stavy a výstupy, které za normálních okolností nevznikají. Pokud má objekt potřebný pro testování některou z následujících vlastností, stojí za uvážení použít místo něj mock objekt:

- Skutečný objekt dodává nedeterministická data, například: aktuální čas, aktuální teplota, současný stav databáze.
- Některé jeho hodnoty, nebo stavy je obtížné vytvořit nebo reprodukovat, například: chyba v síti, selhání hardware.
- Skutečný objekt je pomalý, například: inicializace celé databáze, přenos velkého množství dat po síťovém spoji s malou propustností.
- Objekt potřebný pro testování neexistuje nebo, nemá odpovídající reakce. Například během testy řízeného vývoje aplikace není neobvyklé testovat objekt, jímž odkazované třídy ještě nejsou implementovány.
- Pro plnohodnotné otestování je potřeba, aby objekt zahrnoval informace či metody výhradně pro účely testování.

Příkladem použití mock objektu je simulace síťového soketu pomocí mock objektu. To umožní testerovi zjistit zda testovaný objekt správně reaguje na širokou škálu stavů, kterých může soket nabývat. Dalším vhodným příkladem je mockování objektu (nahrazení skutečného objektu mock objektem) reprezentujícího připojení k databázi. Toto mock připojení potom může například zapisovat do logu každé uložení dat do databáze. Parsováním logu pak lze zjistit, zda byl proveden správný počet zápisů. Tímto lze odhalit jinak neviditelné výkonnostní ztráty způsobené vývojářem, který strachem ze ztráty dat naprogramoval více volání metody `save()`, tam kde by stačilo volání jen jedno.

4.2.1.2 „Fake" objekty

„Fake" v překladu falešný, nepravý. Rozdíl mezi falešným a mock objektem spočívá v jejich složitosti. Falešný objekt implementuje stejné rozhraní jako napodobovaný objekt, vrací však předem nachystané hodnoty příhodné pro testování. Mock objekt dělá o něco více.

Jeho metody mohou obsahovat vlastní asserty[1]. To znamená, že mock objekt prozkoumá kontext každého volání a výsledky použije pro vytvoření reakce tohoto volání. Mock objekt zahrnuje funkcionalitu související výhradně s testováním.

4.2.1.3 „Dummy" objekty

„Dummy" objekty v překladu klamné objekty, jsou obálky bez jakékoliv funkcionality. Bývají předávány, ale nikdy skutečně použity. Většinou slouží jen pro naplnění seznamu parametrů.

4.2.1.4 Pahýly.

Pahýly z anglického „stubs", jedná se o řetězce kódu poskytující předem připravené odpovědi na volání během testu. Většinou neobsahují nic, co by se netýkalo testu. Mohou také zaznamenávat užitečné inforamace o průběhu testu.

4.2.1.5 Závěrem o doubles

Mock objekty jsou většinou použity k verifikaci chování testované části software. Zatímco ostatní typy dvojníků jsou obvykle schopny verifikovat pouze stavy testované části.

4.2.2 Zkušební případ

Mezi odbornou veřejnosti se pro pojem zkušební případ většinou používá anglický výraz „test case". Je to množina podmínek nebo proměnných, podle kterých testující pozná zda je požadavek nebo případ užití („use case") částečně nebo úplně splněn. Pro rozhodnutí, zda je požadavek bezezbytku splněn, může být potřeba mnoha zkušebních případů. Aby byl program kvalitně otestován, měl by existovat pro každý na něj kladený požadavek alespoň jeden testovací případ. Zkušební případy jsou obvykle sdružovány do testovacích sad („test suites").

Pro testovací případy je charakteristické, že je předem znám vstup stejně jako očekávaný výstup. Vstup i výstup jsou zjištěny (např. spočítány) dříve, než je test vykonán. Ideálně by měl vstup testovat předpoklady[2] a výstup kontrolovat podmínky, jež musí být platné vždy po provedení testovaného kódu. Velmi prospěšné je, pokud do dokumentace testů nebo přímo do komentářů testovacích případů uvedeme popis funkcionality, jež je testována a přípravy potřebné pro provedení testu.

V některých případech je potřeba posoudit výsledky provedeného testu skupinou expertů, aby bylo možné určit, zda je lze považovat za úspěšné. Tato situace může nastat například při zjišťování hodnot výkonu nového produktu. V takovém případě je první test

1. Assertem je myšlen predikát (pravdivý, nepravdivý výraz), umístěný v programu tam kde si prográmator myslí, že tento predikát bude vždy pravdivý.
2. Předpoklady jsou zde myšleny podmínky, jež před provedením testovaného kódu musí vždy platit.

použit jako výchozí hodnota pro příští testování případně následující verze produktu, kde lze hodnoty již porovnávat automaticky.

4.2.3 Unit Testy

Jedná se o poměrně nízkoúrovňové testy. Jsou psány ve stejném jazyce jako testovaný kód. Účelem unit testů je validovat individuální díly (units) zdrojového kódu[3]. V ideálním případě je každý tzv. [zkušební případ] unitu nezávislý na ostatních testovacích případech. Jako pomůcky k testování dílů kódu bývají používány tzv. mock objekty, případně fake objekty právě tak jako automatizované testovací frameworky. Tento typ testů bývá používán vývojáři a údržbáři, aby otestovali, zda kód, který napsali splňuje požadavky a zda se chová tak, jak bylo zamýšleno.

Cílem unit testů je izolovat každou jednotlivou část programu a ověřit, zda je korektní. Unit test vlastně tvoří jednoznačný předpis, jenž musí daná část kódu splnit. Tento nástroj se výborně hodí pro regresní testování[4]. Unit testy nabízí, na oplátku za práci vynaloženou při jejich implementací, několik cenných prvků usnadňujících údržbu aplikace:

Zjednodušení procesu uplatnění změn v aplikaci. Unit testy umožňují programátorovi například provádět refactoring[5] kódu s relativní jistotu, že program stále funguje správně (že se skutečně jednalo o refactoring). Pokud se vývojáři snaží držet dobrého zvyku psát testovací případy pro všechny funkce a metody, neměl by být problém rychle najít a opravit chybu způsobenou změnou v aplikaci. Toto je velmi užitečné, zvláště při provádění regresních testů, jež jsou během údržby aplikací obvykle používány. V prostředí s průběžně probíhajícím unit testováním, jako je trvalá údržba webových aplikací, tyto testy s určitou přesností reflektují zamýšlené chování aplikace. Přesnost reflexe je, mimo jiné, závislá na podrobnosti a šíři pokrytí kódu unit testy.

Zjednodušení kontroly integrace jednotlivých částí aplikace. Unit testování pomáhá do jisté míry odstranit nejistotu, zda jednotlivé díly kódu fungují tak, jak mají. Tato technika elegantně zapadá do přihrádky testovacích postupů zdola nahoru. Testování zdola nahoru spočívá ve zkoušení nejdříve izolovaných dílů systému, následně různých, do počtu stále větších, souhrnů těchto dílů. Zda je možné uspokojivě otestovat integraci komponent systému pouze automatizovanými testy, je silně diskutované téma. Během integrace složitých webových aplikací vyplave na povrch mnoho problémů,

3. Díl zdrojového kódu, anglicky unit. Nejmenší testovatelná část aplikace. V procedurálním programování se může jednat o program, funkci, proceduru, atd., zatímco v objektově orientovaném programování je nejmenším dílem metoda nějaké třídy.
4. Regresní testování je jakýkoliv typ testování, jehož cílem je odhalení regresních chyb. Pojem regresní chyba označuje nově vzniklý rozpor software se správným fungováním, na místě dříve odpovídajícímu specifikaci. Regresní chyby se většinou vyskytují jako nechtěné konsekvence změn v programu.
5. Refactoring je jakákoliv změna, která vylepší čitelnost, případně zjednoduší strukturu kódu, aniž by jakkoliv ovlivnila výstup aplikace.

jež v současné době pravděpodobně není možné spolehlivě otestovat bez lidského dohledu. Objevují se názory, že za předpokladu dostatečně širokého spektra automatizovaných testů není lidská spoluúčast nutná. Ve skutečnosti však záleží na charakteristikách vyvíjeného produktu. Rozhodující může být dostupnost lidských zdrojů či rozpočet schválený pro údržbu aplikace.

Vytvoření samokontrolujícího se designu Když je vývoj (počáteční vytvoření) software řízen pomocí testů[6], unit testy se obvykle stanou jakýmsi formálním designem aplikace. Každý test může být chápán jako prvek designu, specifikující jednotku již testuje. Design je tedy sám schopen kontrolovat, zda je dodržován. Je pravda, že takovýto návrh oproti klasickému diagramu něco postrádá, na druhou stranu jsou dnes UML diagramy jednoduše generovatelné ze zdrojových kódů, jež mohou být napsány ve většině moderních jazyků pomocí volně šiřitelných nástrojů. Aplikace jejichž vývoj byl řízen testy, mají pro pozdější údržbu výrazná pozitiva. Jejich součástí je absolutně spolehlivý design. Nedodržení designu je okamžitě a jednoduše odhaleno. Pokud je při vývoji postupováno ve shodě se správným vedením dokumentace unit testů, údržbáři přebírají velice kvalitně popsanou a dobře udržovatelnou aplikaci. Zvolení testy řízeného vývoje webových aplikací je jeden z momentů, kde lze zdánlivě velkou investicí do podrobných unit testů ušetřit mnohem více prostředků ve fázi údržby.

Oddělení rozhraní od implementace Třídy ve webové aplikaci ne zřídka obsahují reference na jiné třídy. Testování takových objektů se může rozrůst o ověřování odkazovaných tříd. Například pokud je třeba prověřit objekt závisející na databázi, tester napíše do testovacího případu kód zabezpečující komunikaci s databází. To je chyba, protože jednotlivé testové případy by neměly být závislé na ničem mimo testovaný díl aplikace. Správným postupem je vytvořit abstraktní rozhraní, jež musí být implementováno třídou databázového připojení. V testovacím případě následně implementuje vytvořené rozhraní vlastním mock objektem, který poskytne testované třídě. Tato izolovaná jednotka potom může být důkladněji otestována nezávisle na omezeních kladených např. aktuálním stavem databáze. Výsledkem je kvalitnější část aplikace, která je také lépe udržovatelná.

4.2.3.1 Závěrem o unit testech

Unit testování je stále jen testování a proto nemůže bezpečně odhalit všechny chyby v aplikaci. Z definice plyne, že tyto testy prověřují pouze jednotlivé díly software. Tudíž nemohou plnohodnotně odhalovat integrační chyby, výkonnostní problémy a jiné celo-systémové vady. Unit testy jsou efektivnější, pokud jsou kombinovány s dalšími typy testů. K plnému

6. Zástupcem testy řízeného programování je například takzvané Extrémní Programování.

využití potenciálu unit testů je potřeba přísná disciplína během celého životního cyklu aplikace. Je nezbytné udržovat přesné záznamy nejen o testech, ale i všech zásazích do kódu provedených na kterémkoliv z dílů software. Použití některého ze systémů pro správu verzí je jedním z velmi významných a dnes již zcela neodmyslitelných předpokladů pro údržbu aplikací. Pokud selže testovací případ, který v předchozí verzi dílu prošel, systém pro správu verzí poskytne seznam všech změn (pokud nějaké existují) provedených na jednotce od posledního úspěšného průchodu testem.

Unit testování musí být udržováno rychlé a snadno proveditelné, všemi prostředky nesnižujícími důkladnost testování (např. mock a falešné objekty). Nenáročné rychlé testování lze provádět často. Díky tomu je usnadněno jeho zakořenění v každodenním pracovním plánu vývojářů. Často prováděné rychlé testování implikuje včasnější odhalení chyby a v neposlední řadě úsporu prostředků, ať z důvodu nižší náročnosti opravy brzy detekovaných chyb nebo zmenšení režie spojené s prováděním testů. Aby práce na systému byly skutečně efektivní, je také velmi důležité co nejdříve přezkoumat a korigovat všechny jednotky software, na kterých jsou testováním odhaleny nedostatky. V opačném případě by se aplikace, úměrně se vzrůstajícím počtem selhávajících testovacích případů, desynchronizovala vzhledem k unit testům. Celé unit testování by pak ztrácelo na efektivitě a nakonec i na významu.

Unit testy bývají v drtivé většině automatizované. I když existuje i manuální přístup k této technice, má oproti automatizovanému provádění řadu nevýhod a rizik nabourávajících filozofii unit testů. Manuální postup může být použit snad jen ve výjimečných případech nevyzpytatelných testů vyžadujících dohled inteligentní bytosti. Při užití automatizovaného přístupu je jednotka respektive díl systému jako subjekt unit testu prováděn uvnitř testovacího frameworku, mimo aplikační systém či kontext volání. Tímto je plně realizována izolace, čili nezávislost testované jednotky. Unit testovací frameworky byly vytvořeny pro velkou škálu programovacích jazyků. V zásadě je možné provádět tuto techniku testování i bez podpory konkrétního frameworku. V tom případě je třeba napsat klientský kód, schopný během testů nezávisle provozovat jednotky systému s podporou mechanizmů potřebných pro testování (asserty, výjimky, signalizace selhání, atd.). Tento přístup může být výhodný, pokud existují nezanedbatelné překážky pro začlenění běžného unit testování. Na druhou stranu lze s těží dosáhnout kvalit a funkcionality pokročilého testovacího frameworku.

V současné době je k dispozici nepřeberné množství frameworků pro všemožné jazyky od Javy, přes JavaScript, XML, XSLT, SQL, až po MATLAB. Jedním z prvních testovacích frameworků byl SUnit implementovaný kolem roku 1994 v jazyce Smalltalk. SUnit je zakládajícím člen v současnosti nejrozšířenější rodiny testovacích frameworků takzvané xUnit. V roce 1998 byl SUnit přepsán do Javy – vznikl JUnit. Zkušenosti získané používáním JUnitu hrály důležitou roli během vzniku testy řízeného paradigmatu vývoje aplikací. Při diskuzi o testy řízeném vývoji aplikací je často předpokládána aspoň částečná znalost JUnitu. Jedná se tedy o jeden z nejvýznamějších a nejpoužívanějších unit testovacích frameworků. Implementce JUnitu jsou dostupné v mnoha jazycích pod (většinou) logicky pozměněným názvem: PyUnit (Python), CPPUnit (C++), NUnit (C#), atd.

Používání unit testů je tradičně stimulem pro vytváření soudržných částí kódu, jež jsou poměrně málo provázány se zbytkem aplikace. Soudržná část kódu se projevuje silně souvislým kódem a blízkým zaměřením smyslů a funkcí kódu. Čím soudržnější kód je, tím intenzivněji se u něj projevují vlastnosti jako jsou robustnost, spolehlivost, znovu použitelnost a srozumitelnost.

4.2.4 Integrační testy

„Celek je více než souhrn jeho částí."[ ]

Jde o testy, ve kterých jsou jednotlivé části nebo komponenty aplikace testovány dohromady jako skupina. Integrační testy jsou prováděny v pořadí po unit testech a před takzvanými systémovými testy. Integrační testy berou jako vstup díly aplikace, jež byly otestovány unit testy, seskupí je a testuje ve velkých agregátech dle integračního testovacího plánu. Výstupem by měl být integrovaný systém připravený pro fázi systémových testů.

Existuje několik technik integračního testování:

Velký třesk. Tento přístup spočívá ve zkompletování velké části nebo všech modulů aplikace a jejich následném otestování. Metoda Velkého třesku je velmi efektivní hlavně co se týče časové náročnosti. Na druhé straně musí být dobře provedena, jednotlivé testovací případy musí dobře pokrýt integraci systému. V opačném případě může dojít ke zkomplikování celého procesu integračních testů a nedostatečnému otestování. Jednou z metod velkého třesku je takzvané testování dle modelu použití systému. Tato metoda spočívá v testování celého systému zatížením co nejvíce se blížícím skutečnému využití systému po jeho zavedení. Za otestování aplikace tímto způsobem může být považováno i její sestavení např. pomocí Mavenu[7].

Shora dolů. Začíná se hlavním řídícím modulem a postupuje se směrem dolů, buď do hloubky nebo do šířky. Podřízené moduly jsou na počátku nahrazeny takzvanými pahýly. Tyto pahýly jsou postupně, buď podle přístupu do hloubky nebo do šířky, nahrazovány skutečnými moduly. Po každém nahrazení je systém testován. Problematické může být, pokud jsou očekávány významné výstupy z podřízených modulů, které lze poté těžko nahradit pahýly. Nevýhodou tohoto přístupu je nutnost vytvářet někdy i složité pahýly.

Zdola nahoru. Nejnižší moduly jsou spojeny do skupin (clusterů), které provádí specifické funkce. Tyto skupiny jsou otestovány. Vzniklé skupiny jsou dále spojovány po struktuře aplikace směrem výše a následně testovány. Tato forma testování je užitečná pouze když jsou připraveny všechny nebo velká část modulů stejné úrovně. Postup zdola nahoru umožňuje jednoduše odhadnout postup testování například v procentech.

Často používaným frameworkem pro testování webových aplikací je Cactus. Je to nástroj postavený na základech JUnitu. Jeho velkou výhodou je, že testy běží přímo v aplikačním serveru, takže lze otestovat fungování aplikace v prostředí jejího budoucího nasazení. Dále poskytuje spoustu tříd usnadňujících testování specifik javových webových aplikací. Mezi tyto třídy patří například: ServletTestCase, FilterTestCase, JspTestCase, atd. Existuje mnoho

7. Maven je nástroj pro správu softwarových projektů v širokém slova smyslu.

dalších frameworků vhodných pro integrační testování webových aplikací. Pro otestování integrace databáze je vhodný DbUnit. Speciálně pro webové aplikace lze použít HttpUnit.

4.2.5 Zátěžové testy

Cílem zátěžového testování není, na rozdíl od unit testů, kontrolovat správnost systému, ale zjistit, zda systém poskytne dostatečný výkon i při větším počtu současně obsluhovaných uživatelů. Jinak řečeno, jde o testování z jedné perspektivy, jehož účelem je zjistit, jak rychle nějaký aspekt systému pracuje pod určitou zátěží. Pomocí zátěžových testů lze ověřovat i jiné vlastnosti systému, jako jsou škálovatelnost, spolehlivost a náročnost na zdroje.

Zátěžové testy mohou být použity k různým účelům:

- Mohou prokazovat zda systém splňuje výkonnostní kritéria.
- Mohou sloužit pro porovnávání parametrů různých aplikací nebo verzí jedné aplikace. Toho se využívá např. během optimalizace aplikace.
- Zátěžové testy lze použít k zjišťování parametrů jednotlivých částí systému a nalézt tak úzká hrdla.
- Pomocí zátěžových testů lze identifikovat operace, které nejvíce zatěžují nebo nějakým způsobem zpomalují systém a soustředit pozornost na odpovídající místa systému.

Nezřídka se stává, že nejsou nijak konkrétně specifikovány požadavky na výkony systému při různých počtech současně obsluhovaných uživatelů. Není proto možné systém v tomto smyslu přesně kontrolovat. Častěji jsou zátěžové testy používány k ladění celkové výkonnosti systému. Základní ideou je nalezení tzv. nejslabšího článku. Nejslabší článek je taková část systému, jejíž zvýšení výkonnosti povede k nárůstu výkonu celého systému. Nalezení tohoto místa někdy může být obtížné. Některé testovací nástroje proto nabízí monitorování prostředků na serveru, kde běží aplikace. Sledovány jsou např. doba trvání transakcí, rychlost přístupu k databázím, síťová režie, vytížení CPU, a mnoho dalších. Nejslabší články jsou potom hledány analýzou těchto údajů v souvislosti s naměřenými výkonnostními parametry.

Extrémním příkladem špatného postupu hledání nejslabšího článku v systému je příběh společnosti, která vydala velké prostředky na optimalizaci aplikace aniž by byla provedena důkladná analýza. Výsledkem bylo přepsání „čekací smyčky“ aplikace, která byla identifikována jako místo, ve kterém aplikace stráví nejvíce času. Je zřejmé, že i kdyby se jim podařilo vytvořit nejefektivnější čekací smyčku na světě, nezlepšili by celkovou výkonnost ani o píď.

Během zátěžového testování je rozhodující, aby se testy co nejvíce podobaly skutečnému zatížení systému. V praxi je to často velmi obtížné. Úplná shoda testů se skutečným provozem je přímo nemožná. Přesný odhad je možný pouze ve velmi jednoduchých aplikacích.

Důvodem je přítomnost určité náhodnosti v povaze reálného zatížení systému. Tento problém pomáhají řešit nástroje, které průběžně monitorují provoz systému. Získané údaje následně použijí při automatizované adaptaci testů nebo je jen jednoduše zaznamenají a jejich využití přenechají vývojářům.

Pro vytvoření správných zátěžových testů jsou velmi užitečné tyto informace:

- Co je předmětem zátěžového testu? Které subsystémy, rozhraní, komponenty atd. jsou oborem tohoto testu?
- Kolik je očekáváno uživatelů aplikace? Kolik ve špičce a kolik mimo špičku?
- Jakou konfiguraci bude mít server (hardware, software), na kterém aplikace poběží? Sem patří i síťové připojení serveru.
- Jaká je struktura zatížení aplikace podle jejích komponent? (např.: přihlášení 10%, vyhledávání 30%, výběr produktů 20%, vyúčtování 40%)

Činnosti, které je většinou potřeba vykonat pro provedení výkonnostních testů aplikace:

Prozkoumání prostředí testů. Prozkoumání fyzického a vývojového prostředí testů, stejně jako dostupných zdrojů a nástrojů. Fyzické prostředí zahrnuje hardware a síťové prostředí. Dobré povědomí o tomto prostředí je základním předpokladem pro vývoj efektivních testů a rozpoznání největších úskalí, s nimiž se bude potřeba během testů vypořádat. Je potřeba toto povědomí udržovat konzistentní se skutečným stavem.

Stanovení výkonnostních cílů. Je potřeba stanovit omezení a cíle jako jsou rychlost odezvy, propustnost a náročnost na zdroje. Lze říci, že rychlost odezvy se týká uživatelů, propustnost je obchodní záležitostí a využití zdrojů je systémovou doménou. Existují také jiná kritéria, která se netýkají těchto tří skupin, např. ideální konfigurace vedoucí k žádoucím výkonnostním charakteristikám systému.

Plánování a návrh testů. Zmapování klíčových scénářů, určení proměnlivosti uživatelů a způsobu jak tyto činitele nasimulovat. Definování testovacích dat a specifikování parametrů aplikace, jež je potřeba změřit. Využití těchto informací ke zhotovení modelů zátěže systému, jež mají být implementovány, otestovány a analyzovány.

Nastavení prostředí testů. Příprava prostředí testů, nástrojů a zdrojů potřebných k provedení testů v závislosti na možnostech a komponentách těmto testům dostupných. Zjištění, zda je prostředí připraveno pro monitorování zdrojů tak, jak je potřeba.

Implementace testů. Vyvinutí testů ve shodě s jejich návrhem.

Provedení testů. Spuštění a monitorování testů. Kontrola testů, testovacích dat a shromážděných výsledků testů. Spuštění zkontrolovaného testu za současného sledování testu a jeho prostředí.

Analýza výsledků, vytvoření hlášení a nové testování. Shromáždění a následné předání získaných výsledků. Analýza dat jak individuálně, tak se všemi zainteresovanými stranami. Určení priorit zbývajících testů a jejich provedení tak, jak je potřeba. Pokud jsou všechny parametry v pořádku, žádný z limitů není překročen a jsou získány všechny potřebné informace, byly dokončeny testy daného scénáře na dané konfiguraci.

Oblíbeným nástrojem pro zátěžové testování aplikací je JMeter od Apache Software Foundation. JMeter je čistě javová aplikace použitelná na většině operačních systémů. Původně byl zaměřen pouze na webové aplikace, ale posupně bylo spektrum jeho použití rozšířeno. Nabízí široké možnosti, včetně přehledných grafů naměřeného výkonu při různé zátěži. Mezi další frameworky vhodné pro zátěžové testování patří např.: LoadSim postavený na základech JMeteru, nebo CLIF.

4.2.6 Funkční Testy/GUI

Tyto testy se řadí do skupiny takzvaných systémových testů. Systémové testy jsou většinou prováděny na již kompletním integrovaném systému. Systémové testy lze zařadit do kategorie takzvaných „black box“ testů. Takovéto testy by neměly vyžadovat žádnou znalost vnitřní struktury nebo kódu aplikace. Systémové testy by měly být prováděny na systému jako celku, opírajíc se o požadavky na funkcionalitu systému.

Obrázek 4.1: Black box

Jedná se o testování produktu používajícího grafické uživatelské rozhraní, za účelem ověření, zda splňuje danou specifikaci. Tester se musí vypořádat s dodatečnou funkcionalitou poskytnutou GUI. GUI činí testování o mnoho náročnějším z mnoha důvodů, např.: událostmi řízená povaha GUI, nevyžádané události, mnoho různých cest do/ze stejného stavu, nekonečná doména vstupů. Komplikovaná povaha GUI tvoří silný předpoklad pro

presenci chyb, způsobených vývojářem, jenž může jen těžko nějak kontrolovat obrovskou vstupní doménu.

Prvním činitelem komplikovanosti GUI je množství proveditelných operací. Dalším faktorem hrajícím významnou roli je sekvenční problém. Tedy nějaká funkcionalita aplikace může být zpřístupněna sekvencí nějakých operací. Tyto dva činitele mohou být dohromady velkou komplikací, pokud chce tester vytvářet testy manuálně.

Regresní testy jsou také velkým problémem. GUI se totiž velmi často mění, i když v pozadí se nacházející aplikace se nezměnila vůbec. Na rozdíl od ostatních typů testů i drobné změny GUI mohou zapříčinit nefunkčnost testů. Tato nefunkčnost neznamená chybu programátora. Je potřeba testy upravit, aby byly v souladu s novou verzí rozhraní.

Funkční testy jsou důležité, protože ověřují, zda je systém v pořádku připraven pro nasazení. Tyto testy významným způsobem doplňují unit testy. Unit testy mohou dobře ověřit funkčnost všech částí kódu, ne však celý systém jako takový. Funkční testy mohou pomoci podchytit mnoho chyb, které by s pouhým, třeba i velmi důkladným, unit testováním nebylo možné odhalit. Unit testy říkají, že kód *dělá věci správně*; funkční testy říkají, že kód *dělá správné věci*. Dá se říci, že funkční testy použitelným způsobem definují funkční systém. Udržovaný balík funkčních testů:

- Užitečným způsobem popisuje požadavky na funkcionalitu systém.
- Dává vývojářům jistotu, že aplikace tyto požadavky splňuje.

Pokud se má programátor vypořádat s otestováním GUI, zásadní roli hrají softwarové nástroje, které přitom využije. Jedním z vhodných nástrojů pro otestování GUI webových aplikací je Selenium. Testy tohoto nástroje běží přímo v prohlížeči, stejně tak jako při běžném užívání testované aplikace jejími uživateli. Selenium podporuje všechny běžně používané prohlížeče, je proto možné jej použít i pro testování kompatibility GUI s webovými prohlížeči. Pro Selenium je k dispozici i integrované vývojové prostředí ve formě pluginu do prohlížeče Firefox. Tento plugin mimo jiné umožňuje vytvářet testovací skripty zaznamenáváním interakcí testera s prohlížečem. Díky této funkci mohou rychle vzniknout komplikované testy. Obzvláště užitečné je zaznamenávání interakce s prohlížečem pro začátečníky ve fázi seznamování se se Seleniem. Skripty testů mohou být zaznamenány v mnoha jazycích jako jsou: HTML, Ruby, Java.

4.2.7 Regresní testy

Regresní testy jsou jakékoliv testy usilující o odhalení regresních chyb. Regresní chyby se objeví kdykoliv nějaká funkcionalita, dříve fungující tak, jak bylo zamýšleno, přestane fungovat nebo již nefunguje stejný způsobem jako bylo původně plánováno. Typicky se tyto chyby objevují jako nechtěné konsekvence zásahů do software. Zkušenost ukázala, že tento typ chyb je vcelku běžný. Někdy je problém způsoben špatnou kontrolou revizí. Jindy je regresní chyba zaviněna špatnou opravou dříve objevené chyby, která napravila problém jen z části, časem se tato špatně opravená chyba projeví na jiném místě. Často se však regresní chyby objeví v přepracované části software, v nové části jsou napáchány stejné chyby

jako se vyskytly v původní implementaci. Proto je dobrým zvykem po jakémkoliv zásahu do aplikace provést celou sadu regresních testů. Běžně se regresní testy spouští po každém úspěšném sestavení software, případně každou noc nebo každý týden. Pravidelné regresní testy jsou pevnou částí extrémního programování

TYPY REGRESÍ:

Lokální Zásah do aplikace vyvolá chybu.

Demaskující Zásah do aplikace odhalí již existující chybu.

Vzdálené Změna jedné části poruší správnou funkčnost jiné části aplikace.

Kapitola 5

Případová studie – Waste Site Energy Calculator

V této kapitole se pokusím aplikovat výše uvedené postupy na konkrétním případu. Pokusím se provést údržbu a modernizaci webové aplikace Waste Site Energy Calculator, dále jen Calculator. Jedním z cílů také bude zajistit co možná nejvyšší udržovatelnost této aplikace.

Calculator je webový nástroj pro odhad energetických požadavků technologií pro dekontaminaci zamořených území. Calculator byl připraven za spolupráce U.S. Enviromental Protection Agency (US EPA) a Northwest Pollution Prevention Resource Center (PPRC). Programátorskou podporu poskytla Fakulta Informatiky Masarykovy University.

Současná verze Calculatoru je 1.0, poskytuje deset tehnologií pro ošetření půdy a pět skupin kontaminantů. Je navržen tak, aby umožnil uživateli srovnání jednotlivých technologií s defoultními parametry. Pokud je předmětem zájmu uživatele území s konkrétními parametry, může v aplikaci tyto údaje specifikovat. Základní rysy Calculatoru jsou:

- Výpočet spotřeby energie vybraných technologií a srovnání s ostatními relevantními tehnologiemi (odstraňujícími stejný kontaminant ze stejného média).
- Výpočet vzdušných emisí vyprodukovaných dekontaminačním procesem.
- Odhad potenciálních úspor energie skrze optimalizaci a průběžným zdokonalováním nápravného procesu.
- Výpočet GWP[1].
- Výpočet potenciálních redukcí GWP.
- Prezentace energetické efektivity nápravy území užitím enviromentálních ekvivalentů.

Hodnoty lze zadávat jak v imperiálních tak metrických jednotkách. Pro významné výpočty jsou použity genetické algoritmy. Výstupy aplikace lze tisknout nebo uložit v TXT, PDF nebo HTML formátu. Budoucí verze kalkulátoru umožní uživatelům registraci s následné spravováni a bezpečné sdílení dat na webu. [ ]

1. GWP (Global Warming Potential) je měřítko určující jak moc dané množství skleníkového plynu přispívá ke globálnímu oteplvání.

5.1 Dokumentace

Dokumentace Calculatoru je na přijatelné úrovni. Drtivá většina tříd je okomentována, i když docela stručně. Z komentářů lze pomocí Javadocu vygenerovat API dokumentaci ve formátu HTML. Vygenerovaná dokumentace poskytne velmi užitečný a přehledný popis API. K dispozici jsou také stručné popisy základních algoritmů a definice základních pojmů. Ideální by bylo kdyby byly komentáře, ze kterých se generuje API dokumentace, o něco podrobnější, na škodu by také nebyl nějaký dokument popisující širší souvislosti Calculatoru. V průběhu prací na Calculatoru jsem tedy komentáře doplňoval, tak aby z nich bylo možné postřehnout fungování jednotlivých částí Calculatoru i souvislosti mezi nimi.

5.2 Verze sestavení

Pro vývoj Calculatoru byl použit moderní systém pro správu verzí Subversion. Tato skutečnost má pozitivní vliv na mnoho aspektů údržby a vývoje této webové aplikace. Díky Subversion jsou přesně, přehledně a dohledatelně evidovány všechny zásahy do systému. Proto by neměl být problém rychle odstraňovat regresní chyby. Dalším významným přínosem Subversion je správa jednotlivých verzí sestavení či speciálních verzí aplikace.

Při prozkoumávání repositáře jsem narazil na několik verzí sestavení Calculatoru. Protože nebyla k dispozici žádná dokumentace popisující rozdíly mezi verzemi, všechny jsem prozkoumal a pokusil se zjistit, v čem se od sebe liší. Jednu po druhé jsem zkompiloval a nasadil na aplikačním serveru, abych postřehl rozdíly v chování. V hledání rozdílů se ukázal jako užitečný nástroj pro porovnání souborů a adresářů, konkrétně KDiff3.

Seznam rozdílů mezi verzemi:

1.0

- Původní verze uzavřená v určitý moment.

1.0.1

- Opraveno zaokrouhlování v třídě `graphs.ValueBar` na řádku 277.

1.1

- Odstraněn `batik.jar`.
- velká část tříd nyní implementuje interface Serializable
- V `Icineration.java` a `ThermalDesorption.java` změněn výpočet nového BSSI v metodě `public double computeImpr_separation(String BSSIDec)`.
- Přidány technologie LandTreatment, PhytohydraulicControl, SoilWashing.
- V `Site.java` změněn atribut `public static int GRAPH_HEIGHT` z 300 na 380.

- Nějaké změny v CSS.
- V `contaminants.xml` změněna hodnota atributu `low` elementu `CONCENTRATION` z 1500 na 500.

2.0

- Přidány nové jary `batik.jar`, `commons-beanutils.jar`, `commons-collections.jar`, `commons-digester.jar`, `commons-fileupload.jar`, `commons-logging.jar`, `commons-validator.jar`, `jakarta-oro.jar`, `jaxen-1.1-beta-6.jar`, `struts.jar`. Soubor `dom4J-1.4.jar` aktualizován na verzi 1.6.1.
- Přepracována struktura balíků a zdrojových kódů.
- Přidány balíky `managers`, `datafields`, `holders` a třídy v nich.
- Použit MVC Framework Struts.
- Aplikace internacionalizována (properties soubor).
- Nová vyjímka `CalculatorException`.
- Nový JavaScript soubor `basic.js`.

2.1

- Nově je možno pracovat jak s metrickým tak imperiálním jednotkovým systémem a převádět hodnoty mezi nimi. Dosud byl k dispozici jen imperiální systém.

2.2

- Nově jsou použity genetické algoritmy. S nimi související jary, `jgap.jar`, `jgap-examples.jar`, `jgap-tests.jar`. Nový balík s třídami týkajícími se genetických algoritmů `ga`. V mnoha třídách přidán kód týkající se genetických algoritmů.
- Opraven bug v souboru `Graph.java`.
- V mnoha třídách nyní podrobnější logování.
- V třídě `LandTreatment` je proměnná `pretreatment` nyní typu String (byla boolean).
- V třídách technologií LandTreatment, Landfilling a PhytohydraulicControl přibyla možnost find.

5.3 Správa projektu – Maven 2

Dalším krokem během procesu údržby Calculatoru bylo použití aplikace pro správu softwarových projektů. Touto aplikací byl Maven 2, dále jen Maven. Použití Mavenu má mnoho výhod mezi něž patří například jednotná struktura projektů, jednoduchý uniformní proces sestavení, správa knihoven, automatické generování dokumentace nebo webových stránek projektu atd.

Pro převod projektu pod Maven byl potřeba nejdříve vytvořit nový mavenovský projekt pro webové aplikace, tedy projekt typu „maven-archetype-webapp". Je důležité zvolit správný typ projektu, protože se od toho odvíjí adresářová struktura, způsob sestavení a typ výsledného balíku. Následně musely být soubory projektu přesunuty na správná místa v adresářové struktuře mavenovského projektu. Kvůli této změně adresářové struktury projektu, bylo potřeba změnit některé cesty nastavené v konfiguračních souborech Calculatoru (zejména cesty k properties souborům).

Dalším krokem bylo nastavení konfiguračního souboru mavenovského projektu takzvaného POM (Project Object Model). Jedná se o soubor `pom.xml`, jak vidno ve formátu XML. Nejdříve bylo v tomto souboru nutné nastavit typ zdrojových kódů na JDK 5.0. Toho byl docíleno přidáním následujícího do POM:

```
<build>
   <plugins>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-compiler-plugin</artifactId>
         <version>2.0.2</version>
         <configuration>
            <source>1.5</source>
            <target>1.5</target>
         </configuration>
      </plugin>
   </plugins>
</build>
```

Poté bylo potřeba najít ke všem knihovnám používaným Calculatorem příslušný soubor `maven-metadata.xml`. V tomto souboru jsou informace o všech verzích knihovny. Zapsáním těchto informací o knihovně v příslušné verzi do POM je knihovna přidána do projektu. Maven si tuto knihovnu, když bude potřeba, automaticky stáhne z lokálního případně vzdáleného repositáře. Pokud k dané knihovně nelze z nějakého důvodu získat `maven-metadata.xml`, není problém tuto knihovnu manuálně zavést do lokálního repositáře. Aby si každý kdo na projektu pracuje nemusel zavádět knihovnu do lokálního repositáře, je potřeba nastavit vlastní vzdálený repositář, ze kterého si budou všichni ostatní stahovat knihovny používané v projektu.

Příklad dat, která je potřeba přidat do POM aby se knihovna (JUnit 4.4) stala součástí projektu:

```
<dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.4</version>
   <scope>test</scope>
</dependency>
```

5.4 Testování Calculatoru

5.4.1 Unit testy

Bohužel během vývoje Calculátoru nebyly implementovány žádné unit testy. Jak bylo zmíněno v podkapitole [Unit Testy], byl tímto Calculator ochuzen o mnoho výhod, nepříznivě se to odrazí také na jeho udržovatelnosti. Je dosti obtížné a neefektivní dodatečně dopisovat unit testy. Proto jsem se omezil na psaní těchto testů pouze pro nové části aplikace, případně pro stávající části podezřelé z výskytu chyby.

5.4.2 Funkční testy

Součástí Calculatoru bylo několik balíků funkčních testů využívajících framework JUnit. Předmětem těchto testů byla hlavně správná funkce technologií a některé základní vlastnosti aplikace. K mému údivu však ani jeden z testů neprošel. Testy byly napsány někdy na počátku vývoje Calculatoru a nebyly vůbec udržovány. Jak se během vývoje Calculatoru měnilo jeho rozhraní testy s ním postupně přestaly korespondovat. Musel jsem tedy nejdříve aktualizovat testy, aby mohly fungovat s aktuální verzí Calculatoru. Protože byly testy napsány v dnes již zastaralé verzi JUnitu, konkrétně 3.x, při jejich přepisování jsem použil novější syntax JUnitu zavedenou od verze 4.0. Aktualizoval jsem tedy knihovnu JUnit na verzi 4.4 a použil anotace a některé jiné novinky JUnitu přispívající ke srozumitelnějším, rychlejším a pohodlnějším testům. I když byly testy přizpůsobeny, stále jich hodně neprošlo. Opravil jsem tedy funkcionalitu Calculatoru, tak aby byla v souladu s funkčními testy.

5.4.3 Zátěžové testy

Pro zátěžové testy jsem použil nástroj JMeter. Nejsou ale k dispozici žádné konkrétní parametry, které by aplikace co do výkonnosti měla splňovat. Proto jsem aplikaci otestoval spíše pro kontrolu zda je schopná pracovat pod nějakou rozumnou zátěží. Pro testování bylo zvoleno 100 vláken (ekvivalent uživatele), kteří provedou zvolenou posloupnost operací 3krát za sebou. Při této zátěži aplikace podávala přijatelný výkon.

5.4.4 Ostatní testy

Vytvořil jsem několik testů GUI v Seleniu IDE pomocí funkce záznamu interakcí s prohlížečem. Dále jsem napsal několik testů za pomocí frameworku Cactus. Tyto testy společně se sestavením Calculatoru v Mavenu dohromady tvoří skupinu funkčních, integračních a GUI

testů. Nelze totiž říci, že testy Selenia jsou zaměřeny pouze na GUI, chtě nechtě tyto testy do určité míry kontrolují i funkčnost a správnou integraci aplikace. Stejně tak testy Cactusu nekontrolují jen bezproblémovost integrace aplikace. Tyto testy proběhly v pořádku jak na Tomcatu 5, pro který byla aplikace vyvíjena, tak na Glassfishi V2.

5.5 Instrukce pro instalaci Calculatoru

Pro sestavení Calculatoru je potřeba mít k dispozici:

- JDK 5.0 nebo novější.
- Maven 2.0.3 nebo novější.
- Nějaký aplikační server, nejlépe vyzkoušený Glassfish V2 či Tomcat 5 nebo novější.
- Přístup k internetu, aby mohl Maven stáhnout potřebné knihovny.

Vstoupíme do adresáře se zdrojovými soubory Calculatoru. Je dobré nejdříve nastavit v `src/main/webapp/WEB-INF/log4j.properties` cestu k souboru, do kterého budou zapisovány logy Calculatoru. Na tomto místě lze také nastavit úroveň logování. Nyní v nejvyšší úrovni adresáře se zdrojovými soubory Calculatoru příkazem

```
mvn package
```

spustíme proces sestavení Calculatoru. K tomu je potřeba mít nainstalovaný Maven a cestu k němu nastavenou v `PATH`. Součástí procesu sestavení Calculatoru je i provedení testů. Pokud chceme testy vynechat, např. z časových důvodů, je potřeba přidat přepínač pro vynechání testů. Celý příkaz pro sestavení by potom byl:

```
mvn package -Dmaven.test.skip=true
```

Po úspěšném sestavení vznikne adresář `target` a v něm soubor `Calculator.war`. Nyní již stačí jen vytvořený `Calculator.war` nasadit na námi zvoleném aplikačním serveru.

Za zmínku stojí ještě další dva příkazy Mavenu, které lze volat v adresáři se zdrojovými kódy Calculatoru.

```
mvn site
```

vygeneruje webové stránky, obsahující zajímavé informace o projektu. Tyto stránky budou vytvořeny v `target/site`.

```
mvn javadoc:javadoc
```

vygeneruje Javadoc Calculatoru. Javadoc bude umístěn v `target/site/apidocs`.

Kapitola 6

Závěr

Pro udržení vysoké použitelnosti, kvality a s tím spojeného úspěchu software je třeba zajistit stále lepší údržbu a intenzivnější proces modernizace tohoto software. Rozšiřující se konkurence a zvyšující se nároky na aplikace vytvářejí tlak, kterému je třeba neustále čelit v zájmu udržení užitné hodnoty software. Pokud mluvíme o webových aplikacích je výše uvedené ještě zřetelnější. V prostředí internetu již bez kvalitní údržby a její podpory v podstatě nelze z dlouhodobého hlediska obstát.

V této práci jsem postihl několik hlavních faktorů ovlivňujících udržovatelnost aplikace. Uvedl jsme nástroje, které mohou významným způsobem ulehčit tuto fázi životního cyklu software a postupy, jež by měli být dodržovány pro zajištění pro vývojáře velmi užitečných vlastností aplikace. Také jsem se pokusil tyto poznatky aplikovat na konkrétním případu neudržované webové aplikace. Bylo dosaženo hlavních cílů, tedy zvýšit její udržovatelnost, přehlednost a rozšiřitelnost. Zde se ukázalo jako velmi důležité testování, které má pravděpodobně největší přínos. Testování zjednodušuje veškeré zásahy do aplikací a do jisté míry ověřuje jejich správnost.

S testováním souvisí moderní metody vývoje aplikací. Stařičký vodopádový vývoj software překonává iterativní způsob vývoje. Jde hlavně o agilní metody vývoje aplikací, jako je testy řízené Extrémní Programování. Tyto metody zajistí průběžným revidováním aplikace během jejího vývoje mnohem větší soulad s nároky a skutečnými požadavky uživatele. Je vyvíjeno jen to, co je skutečně potřeba, tím klesá zbytečná komplikovanost systému. Jednodušší systém – jednodušší údržba. Princip Extrémního programování zahrnuje časté změny a zásahy do vyvíjeného software a také je značně ulehčuje. Relativní nenáročnost zásahů do aplikace je poté radostně vítána ve fázi údržby či modernizace aplikací. Použitím uvedených metod vývoje software lze pravděpodobně nejvýrazněji ovlivnit budoucí udržovatelnost a úspěšnost webové aplikace.

Literatura

[1] Aristoteles: *Metafyzika*, <http://en.wikipedia.org/wiki/Metaphysics_%28Aristotle%29> (květen 2008) . 4.2.4

[2] Brooks, F.: *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1975, 0-201-83595-9. 2

[3] Pavlovič, J. a Mahutová, K.: *Energy Management at Waste Clean up Sites, Avoiding Secondary Air Impacts to Human Health. In /Environmental Health in Central and Eastern Europe/ U.S.A.*, Springer, 270 s., 2006, 1-4020-4844-0. 5

[4] Fjeldstad, R. a Hamlen, W.: *Application program maintenance report to our respondents, Tutorial on Software Maintenance*, IEEE Computer Society Press, 13–27, 1983. 2.3

[5] IEEE Standard Glossary of Software Engineering Terminology, IEEE, 1990, <http://standards.ieee.org/reading/ieee/std_public/description/se/610.12-1990_desc.html> (duben 2008) . 2.3

[6] BS ISO/IEC 14764:2006, BSI, 58, 31. říjen 2006, 0-580-46596-9, <http://www.bsi-global.com/en/Shop/Publication-Detail/?pid=000000000030115150> (duben 2008) . 2.2, 4

[7] Land, R.: *Measurements of Software Maintainability*, Mälardalen University, Department of Computer Engineering, 2002, <http://www.idt.mdh.se/~rld/Publications.html#Measurements_of_Software_Maintainabilit> (duben 2008) .

[8] Lehman, M. a Ramil, J. a Wernick, P. a Perry, D. a Tursky, W.: *Metrics and laws of software evolution – the nineties view*, 20–32, 5.8.1997, 0-8186-8093-8. 2.1

[9] Pigosky, T.: *Practical Software Maintenance, Best Practices for Managing Your Software Investment*, Wiley Computer Publishing, New York, 1997, 978-0-471-17001-3. 2.2, 2.3

[10] Wikipedia, the free encyclopedia: The First "Computer Bug", 23. březen 2007, <http://en.wikipedia.org/wiki/Image:H96566k.jpg> (duben 2008) .

[11] Wikipedia, the free encyclopedia: Configuration management, 12. únor 2008, <http://en.wikipedia.org/wiki/Configuration_management> (duben 2008) . 2.2

Rejstřík

Příloha A

Odkazy na zmíněný software

Odkazy na software zmíněný v této práci:

Cactus <http://jakarta.apache.org/cactus> (květen 2008)
Framework vhodný pro integrační testování javových webových apliací.

CLIF <http://clif.objectweb.org> (květen 2008)
Framework vhodný pro zátěžové testování jakéhokoliv systému dostupného z JVM.

DbUnit <http://dbunit.sourceforge.net> (květen 2008)
Framework pro testování databází a jejich integrace v systému.

HttpUnit <http://HttpUnit.sourceforge.net> (květen 2008)
Nástroj pro testování webových technologií prostřednictvím protokolu HTTP.

JMeter <http://jakarta.apache.org/jmeter> (květen 2008)
Aplikace napsaná v jazyce Java vhodná pro zátěžové testováni webových aplikací.

JUnit <http://www.junit.org> (květen 2008)
Rozšířený framework pro unit testování javových aplikací.

KDiff3 <http://kdiff3.sourceforge.net> (květen 2008)
Nástroj pro porovnávání souborů a adresářů.

LoadSim <http://loadsim.sourceforge.net> (květen 2008)
Simulátor zátěže webových aplikací.

Maven <http://maven.apache.org> (květen 2008)
Nástroj pro správu softwarových projektů

Selenium <http://selenium.openqa.org> (květen 2008)
Nástroj pro testováni uživatelského rozhraní a kompatibility webových aplikací

SUnit <http://sunit.sourceforge.net> (květen 2008)
Původní framework pro unit testování napsaný v jazyce SmallTalk.

Příloha B

Obsah přiloženého CD

Součástí práce je i přiložené CD, kde jsou k dispozici:

- zdrojový kód této práce ve formátu XML
- zdrojový kód této práce ve LaTeX
- tato práce ve formátu PDF, vygenerovaná pomocí aplikace XSLT 2
- zdrojové kódy webové aplikace Waste Site Energy Calculator
- dokumentace webové aplikace Waste Site Energy Calculator
- výsledky testů webové aplikace Waste Site Energy Calculator