Testování a ladění webových aplikací – Handout

Martin Ledvinka
martin.ledvinka@fel.cvut.cz

21. října 2015

1 Testování

Jak již bylo řečeno na cvičení, oblast QA (quality assurance) v software je velice rozsáhlá. Podívejme se tedy alespoň na některé základní koncepty.

1.1 Úrovně testování

QA u softwarového projektu se obvykle zajišťuje na mnoha úrovních. Bráno z pohledu granularity testovaného kódu, dají se tyto úrovně rozdělit na:

Jednotkové (Unit) testy slouží k testování nejmenších *jednotek* software, typicky tedy metod, maximálně tříd. Obvykle členíme jednotkové testy do tříd, kdy jedna třída (možno nazývat *Test suite*) testuje metody jedné aplikační třídy. Metody v Test suite, anotované `@Test`, reprezentují samotné testy (*Test case*). Tyto metody testují jednotlivé metody testované třídy. Pro bližší informace vizte handout o jednotkovém testování z předmětu BI-ATS na https://edux.fit.cvut.cz/courses/BI-ATS/_media/lectures/bi-ats-h03.pdf a pro popis API JUnit též slajdy k cvičení.

Integrační testy slouží k testování integrace tříd či komponent systému. Příkladem mohou být testy business logiky, které pracují s testovací databází a používají tak jak business, tak persistentní vrstvu. V tomto případě nám jde o ověření správné funkcionality větších celků v rámci aplikace. Obvykle se realizují opět s pomocí frameworků typu JUnit.

Funkční testy testují funkcionality v rámci celé aplikace. Může se jednat o průchod celou aplikací nebo o testování jejích vybraných částí. Tak jako tak, obvykle se jedná o test pracující s celým stackem aplikace – tedy uživatelské rozhraní, business vrstva a persistence.

Ruku v ruce s granularitou jde i náročnost testů na čas (vývoj i provádění) a prostředí. Proto je cílem, aby uvedené testovací úrovně tvořily pyramidu, kdy základna složená z unit testů je nejrozsáhlejší, neboť unit testy jsou rychlé na vývoj a spouštění. Funkční (a případně manuální) testy vyžadují prostředí odpovídající produkčnímu a mnohem více času (sekundy a minuty oproti desítkám milisekund v případě unit testů), proto se jich snažíme mít méně.

1.2 Další možná členění testování

Techniky testování se dále mohou členit dle různých kriterií do skupin (ne nutně disjunktních):

Black box a White box *Black box* testy přistupují k testované komponentě jako k černé skřínce, u které známe jen podobu vstupů a očekávaný výstup, ale nevíme nic o tom, jak funguje uvnitř. Příkladem budiž funkční testy aplikace. Oproti tomu při *White box* testování víme, jak testovaná jednotka funguje uvnitř a součástí testu je i ověření tohoto vnitřního chování. Mezi white box testy řadíme např. unit testy.

Akceptační testy ověřují, že testovaná aplikace odpovídá akceptačním kriteriím (požadavkům) zákazníka. Obvykle se jedná o testy na úrovni funkcionálního testování, tedy testy celé aplikace.

Regresní testy se zaměřují na chyby vznikající úpravou existujícího kódu. Pokud si vzpomeneme na příklad ze cvičení, regresní test například může ověřovat, že sčítání a odčítání funguje i po přidání podpory pro násobení a dělení. V podstatě se tedy existující testy, které jsme psali při přidávání původní funkcionality, stávají regresními.

Zátěžové testy ověřují, jak se aplikace chová při zátěži. V takovém případě např. bombardujeme rozhraní aplikace desítkami requestů současně.

Usability testy jsou zaměřeny na to, jak je aplikace pro uživatele použitelná. Jde hlavně o to, aby její uživatelské rozhraní bylo intuitivní, aby bylo případně přístupné pro osoby s postižením např. zraku apod.

Existuje celá řada dalších členění testovacích technik, zde jsme opravdu zmínili jen několik hlavních. Dále existují techniky pro izolaci testů, jako mockování, stubování apod. Opět zde můžeme odkázat na předmět BI-ATS, který se jimi zabývá podrobněji.

1.3 Proč a jak testovat

Důvod, proč testovat, je poměrně zřejmý – zvyšujeme tak naši důvěru v to, že náš kód neobsahuje chyby. Samozřejmě, žádný kód není bez chyb a ani při nejlepší vůli nebudeme schopni otestovat všechny možné kombinace a podmínky vstupů. Ale kvalitní testovací sada dokáže odhalit velkou většinu potenciálních chyb. Opravit chybu odhalenou testem je mnohokrát levnější, než opravovat chybu nahlášenou zákazníkem. Důkladně testovaný kód je také mnohem snazší refaktorovat, protože se můžeme na testovací sadu spolehnout při ověření, že jsme nezměnili chování aplikace.

Naším cílem je vždy automatizace testování. Takové testování je levné, rychlé a opakovatelné, tedy mnohem spolehlivější, než manuální. Pozor, manuální testování nelze zcela nahradit, zvláště u aplikací s grafickým uživatelským rozhraním.

1.3.1 Test Driven Development

Co se týče způsobu testování, v posledních letech se prosazuje přístup zvaný Test Driven Development (TDD), při kterém se testy píší ještě dříve, než samotný produkční kód. Ukazuje se totiž, že programátoři mají tendenci přizpůsobovat testy existujícímu kódu a ty tak nutně nevypovídají o jeho kvalitě. V případě TDD je test zároveň jakousi specifikací plánovaného kódu a předejdeme tak situacím, kdy napíšeme kód a pak musíme ohýbat volající kód, aby se přizpůsobil kódu volanému.

TDD razí následující koloběh akcí: test → fail → implement → refactor. Nejprve tedy napíšeme test, který selhává. Pak doplníme nejjednodušší implementaci, která testem projde. Nakonec zrefaktorujeme kód a znovu spustíme testy, dokud nejsme spokojeni s výsledkem. Postup minimální implementace v tomto případě též pomáhá předcházet overengineeringu – implementaci něčeho, co vlastně vůbec nepotřebujeme.

2 Code Coverage

Měření pokrytí kódu testy je alespoň částečně způsobem, jak zjistit, do jaké míry je naše testovací sada kompletní. Jak již bylo řečeno na cvičení, *100% pokrytí testy neznamená bezchybný kód*. Code coverage nám ale dává alespoň základní představu o tom, jak jsou které části aplikace otestovány.

Pro oba hlavní code coverage nástroje v Javě - JaCoCo a Cobertura, existují Maven pluginy, které umožňují reportovat code coverage jako součást buildu. Continuous integration servery jako např. Jenkins pak umožňují sledovat např. trend zvyšování/snižování pokrytí kódu v průběhu času.

3 Logging

Logování je způsob, jak získávat informace o běhu aplikace bez toho, abychom ji krokovali či ji jinak ovlivňovali. Oproti výpisům pomocí `System.out` umožňují logovací frameworky řídit granularitu a frekvenci logování. Díky tomu můžeme např. v rámci vývoje logovat podrobnější informace o aplikaci a poté v produkčním prostředí logovat jen velmi základní data.

Typické úrovně logování jsou (bráno z SLF4J):

- NO
- FATAL
- ERROR
- WARN
- INFO
- DEBUG

- TRACE
- ALL

Logovací frameworky umožňují definovat handlery pro různé logy, obvykle na základě package jmen. Můžete tedy nastavit jeden handler pro třídy vaší aplikace a posílat logy do souboru a další handler pro některý framework, který vaše aplikace používá. Ten nastavíte na logování jen základních informací a přesměrujete je pouze na standardní výstup.

Nezapomínejte, že příliš podrobné logování má nezanedbatelný vliv na výkon aplikace. Logování úrovně DEBUG či TRACE tedy používejte jen během vývoje.

4 TestDemo

Informace k demo projektu z cvičení.

4.1 Code Coverage

Code coverage z testovacího projektu získáte při každém Maven buildu. Ve složce *target*, kam Maven generuje zbuildovanou aplikaci, se nachází též složka *site/jacoco*. V ní je soubor *index.html*, který stačí otevřít v prohlížeči a proklikat si pokrytí testy.

JaCoCo Maven plugin, který zajišťuje změření code coverage, je deklarován v *pom.xml* v sekci plugins.

Code coverage lze též získat přímo v IDE. Do NetBeans je ale třeba doinstalovat plugin, např. TikiOne JaCoCoverage. IntelliJ IDEA podporuje měření code coverage nativně.

4.2 Logging

Logování v našem demo projektu je realizováno pomocí SLF4J a implementace logback. Loggeru se můžete všimnout ve třídě `CalculatorServlet`. V pozdějších projektech bude logování více.

Logovací úrovně a výstup je nastaven v souboru *src/main/resources/logback.xml*. Ten definuje dva *Appendery*, které reprezentují výstupní handlery – jeden pro konzolový výstup, druhý pro výstup do souboru (u souborového výstupu si u Windows budete muset nastavit jinou cestu, uvedená funguje na Linuxu). Dále nastavujeme samotný *logger*, který platí pro všechny Loggery v definovaném package a jeho sub-packages. Tomuto loggeru nastavíme logovací úroveň a appendery, na které má výstup posílat.

Více informací v komentářích v uvedeném souboru a v manuálu logback na http://logback.qos.ch/manual/.