EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

MÉDIA- ÉS OKTATÁSINFORMATIKAI TANSZÉK

TESZTVEZÉRELT FEJLESZTÉS BEMUTATÁSA EGY ÁLTALÁNOS CÉLÚ WEBES KERETRENDSZEREN

Abonyi-Tóth Andor

Egyetemi tanársegéd

Karácsony Máté

Programtervező informatikus Bsc,

Nappali tagozat

Budapest, 2011

<Témabejelentő helye>

Tartalom

1. Bevezetés

1.1. Tesztvezérelt fejlesztés

Az elmúlt évtizedekben számos fejlesztési módszertant hoztak létre minőségi szoftvertermékek előállítására. Ezek közül napjainkban egyre népszerűbb a *tesztvezérelt fejlesztés*[1], melynek fő célja, hogy hibamentes, változástűrő forráskódot állítsunk elő.

Alkalmazása során két fő kódbázissal bír egy projekt: a produkciós kód, mely a valódi terméket alkotja, illetve a hozzá készülő tesztkód, melynek célja a produkciós kód automatizált ellenőrzése. A két kódbázis karbantartása pazarlásnak tűnhet, de a megfelelően létrehozott és karbantartott automatizált tesztek jelentős előnyökkel járnak:

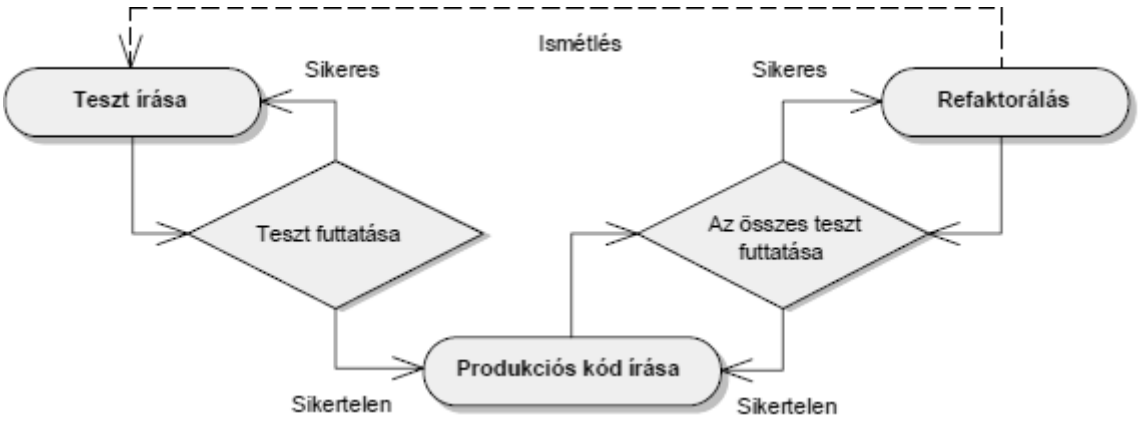
- A tesztek futtatása nem vesz el időt a fejlesztőtől (a manuális teszteléshez képest), és akárhányszor megismételhetők, önellenőrzők (azaz futásuk eredménye egyértelműen sikeres vagy sikertelen, nem szükséges az eredmény utólagos vizsgálata)
- A legtöbb esetben a hibakeresésre fordított idő jelentősen redukálódik, hiszen egy nem teljesülő teszt lokalizálja a hibás kódrészt
- Könnyebb új fejlesztőket bevonni a csapatmunkába, mert a tesztek védőhálóként funkcionálnak, így nem fognak új hibákat bevezetni régebbi komponensekben
- Nem alakul ki a kód egyes részeinek megváltoztatásától való „félelem” a fejlesztőkben (főleg hosszú távú projekteknél fontos)

A munkafolyamat ciklikus, és négy fő tevékenység köré épül: *(a)* tesztek írása, *(b)* a teszteket teljesítő produkciós kód elkészítése, *(c)* a tesztek futtatására, illetve *(d)* e kódok újraszervezésére (refaktorálása), mely során átláthatóbbá tesszük az elkészült forráskódot, illetve szerkezeti javításokat végzünk rajta, mely végső soron emeli a kód minőségét, megkönnyíti megértését, ezáltal a csapatmunkát. Fontos, hogy a teszteket a produkciós kód előtt írjuk, hiszen így fogják befolyásolni annak alakulását,

[1] Test Driven Development (TDD)

megírásukhoz pedig a rendelkezésre álló követelményspecifikációkat és használati eseteket[2] kell figyelembe venni. A tesztek újrafuttatásával bármikor meggyőződhetünk róla, hogy az utolsó fázis végeztével is működőképes maradt az adott szoftverkomponens, funkcionalitása nem változott. A fejlesztési folyamat lépései az alábbiak **[1]**:

1. Írjunk egy tesztet
2. Tegyük lefordíthatóvá
3. Ellenőrizzük, hogy futtatása sikertelen (*piros csík*[3])
4. Írjuk meg a produkciós kódot, mely átmegy a teszten
5. Futtassuk újra a tesztet, ha sikertelen, térjünk vissza az előző lépésre, ha sikeres (*zöld csík*[4]), haladjunk tovább
6. Refaktoráljuk az elkészült teszt- és produkciós kódot
7. Ellenőrizzük, hogy még mindig sikeresen lefut minden teszt (ha nem, a 4. lépéstől folytassuk)
8. Ismételjük a lépéseket előről, amíg elkészül a kívánt funkcionalitás



1. ábra: A tesztvezérelt fejlesztés folyamata

Az angol terminológia szerinti „piros csík” illetve „zöld csík” kifejezések a tesztek futtatásához használt keretrendszerek jellemző grafikus eredménykijelzéseire utalnak. Dolgozatomban a *PHPUnit* **[2]** teszt-keretrendszert használom, mely szöveges kimenettel rendelkezik. Egy teszt futási eredményét az alábbi karakterek egyikével jelzi:

[2] use case
[3] red bar
[4] green bar

- **.** (pont karakter): sikeres lefutás
- **F** (fault): sikertelen lefutás
- **E** (error): sikertelen lefutás fordítási (PHP nyelv esetében értelmezési) hiba miatt
- **S** (skipped): átugrott teszt (nem futtatható le valamely függőség hiányában)
- **I** (incomplete): befejezetlennek jelölt teszt

Egy tesztcsomag futtatásának eredménye pedig a fenti karakterek sorozata, és ha minden teszt sikeres, csak pont karakterekből (.) áll. Ezt a zöld, minden egyéb esetet a piros csík megjelenésének tekintünk.

1.2. A tesztek típusai

A teszteket alapvetően három típusba sorolhatjuk ebben a témakörben:

- A legsűrűbben és legtöbbször végrehajtott, egy-egy osztály publikus metódusainak tesztelését végző kódokat *egységteszteknek*[5] nevezzük. Egy osztályhoz általában egy teszt-osztály tartozik, egy metódushoz pedig egy vagy több teszt-metódus. Dolgozatom főként ezzel a típussal foglalkozik.
- Az *integrációs tesztek*[6] több osztály vagy alrendszer összehangolt működését ellenőrzik.
- A *funkcionális tesztek*, vagy *elfogadási tesztek*[7] azt hivatottak biztosítani, hogy a program funkcionalitása az elvárásoknak megfelelő. (Ezek gyakran felhasználói felület tesztek, melyek automatizálása meglehetősen körülményes, de ezt is be fogom mutatni.)

1.3. Keretrendszer és bemutató alkalmazás

A dolgozathoz készült szoftver egy PHP nyelven írt webes keretrendszer. A választásom azért esett erre a nyelvre, mert rendkívül népszerű, mivel könnyen tanulható, ugyanakkor tapasztalataim szerint nem használják elég hatékonyan a kisebb szoftverfejlesztő cégeknél — népszerű a módszer böngészőben „tesztelés” frissítés gombbal, és ez elég rossz minőségi mutatókat eredményezhet — így ideális környezet

[5] unit test
[6] integration test
[7] acceptance test

nyújt a tesztvezérelt fejlesztés bemutatására. Az elkészült keretrendszer az alapvető vezérlési funkcióján túl alábbi feladatokhoz nyújt segítséget:

- konfiguráció
- naplózás
- input-validáció
- authentikáció és munkamenet-kezelés
- távoli eljáráshívás támogatása különböző protokollokon

A keretrendszerhez készítettem egy bemutató alkalmazást is, hogy könnyebben megérthetővé tegyem működését. Ez az alkalmazás egy egyszerű feladatlista-karbantartó program, használatát és funkcionalitását a 2. fejezetben fejtettem ki.

1.4. A dolgozat felépítése

A következő fejezetben ismertetem a kifejlesztett keretrendszer és a bemutató alkalmazás rendszerkövetelményeit, telepítését, és használatát. Kitérek a *PHPUnit* segítségével írt tesztek felépítésére, illetve futtatásának módjára is.

A fejlesztői dokumentációban bemutatom a legfontosabb komponensek működését és egymáshoz való viszonyát, továbbá ismertetem keretrendszer kiterjesztési lehetőségeit.

2. Felhasználói dokumentáció

2.1. Rendszerkövetelmények

A keretrendszer és a bemutató alkalmazás azonos rendszerkövetelményekkel rendelkeznek, melyek az alábbiak:

- PHP 5.3.0, vagy újabb verzió (ajánlott 5.3.5)
- Bármilyen kompatibilis web szerver illetve operációs rendszer
- A tesztek futtatásához: PHPUnit[8]
- A kód-lefedettség jelentésekhez: Xdebug[9] PHP-kiegészítés

[8] http://www.phpunit.de/
[9] http://www.xdebug.org/

3. Fejlesztői dokumentáció

4. Összegzés

5. Irodalomjegyzék

[1] Becoming Agile: Test a little, code a little - a practical introduction to TDD, h*ttp://danbunea.blogspot.com/2005/12/test-little-code-little-practical.htm* (2011. április)

[2] The PHP Unit Testing framework *https://github.com/sebastianbergmann/phpunit/* (2011. április)

6. Mellékletek

6.1. Ábrák listája

6.2. Tesztek listája

Osztály: *fw\ClassLoader*

- ✓ Register
- ✓ Unregister
- ✓ Auto register
- ✓ Load existing class in global namespace
- ✓ Load existing class in global namespace as slash
- ✓ Load existing class in default namespace
- ✓ Class outside default namespace will not be loaded
- ✓ Exception thrown on nonexistent class

Osztály: *fw\KeyValueStorage*

- ✓ Does not have nonexistent key
- ✓ Get returns null for nonexistent key
- ✓ Get with default value
- ✓ Get with magic method
- ✓ Set with magic method
- ✓ Set transforms array
- ✓ Set returns storage
- ✓ Has works on array wrapper
- ✓ Get works on array wrapper
- ✓ Set works on array wrapper
- ✓ Can be converted to array

Osztály: *fw\config\Configuration*

- ✓ Active section can be changed and trimmed
- ✓ Active section cant be changed to invalid name
- ✓ Different section data are independent
- ✓ Merge

Osztály: *fw\config\FileBasedConfiguration*

- ✓ Construction with missing file throws exception

Osztály: *fw\config\IniConfiguration*

- ✓ Constructor throws exception on incorrect ini file
- ✓ Constructor throws exception on invalid section name
- ✓ Constructor throws exception on missing parent section
- ✓ Can parse valid ini file
- ✓ Section inheritance

Osztály: *fw\config\XmlConfiguration*

- ✓ Constructor throws exception on incorrect xml file
- ✓ Constructor throws exception on invalid xml file
- ✓ Constructor throws exception on missing parent section
- ✓ Can parse valid xml file
- ✓ Section inheritance

Osztály: *fw\log\Log*

- ✓ New log has no log target
- ✓ Log has target after adding one
- ✓ Adding same log target twice results in added once
- ✓ Log have zero targets after all removed
- ✓ Writing error invokes write on targets
- ✓ Writing warning invokes write on targets
- ✓ Writing info invokes write on targets
- ✓ Debug disabled by default
- ✓ Writing debug does not invoke write on targets when disabled
- ✓ Writing debug invokes write on targets when enabled
- ✓ Adding target with not matching level will not log
- ✓ Adding target with invalid level throws exception

Osztály: *fw\log\LogTarget*

- ✓ Formatting works with default format string
- ✓ Formatting works with custom format strings

Osztály: *fw\log\FileTarget*

- ✓ Constructor throws exception when permission denied
- ✓ Writes formatted line with line feed to file

Osztály: fw\log\OutputTarget

- ✓ Writes formatted line with line feed to output

Osztály: *fw\control\RouteInfo*

- ✓ Controller name can be set
- ✓ Action name can be set
- ✓ Constructor works with array parameters
- ✓ Constructor works with key value storage parameters
- ✓ Constructor creates key value storage on incorrect parameters