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 olyan változástűrő forráskódot állítsunk elő, melyben a hibák lehetősége minimalizált.

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:

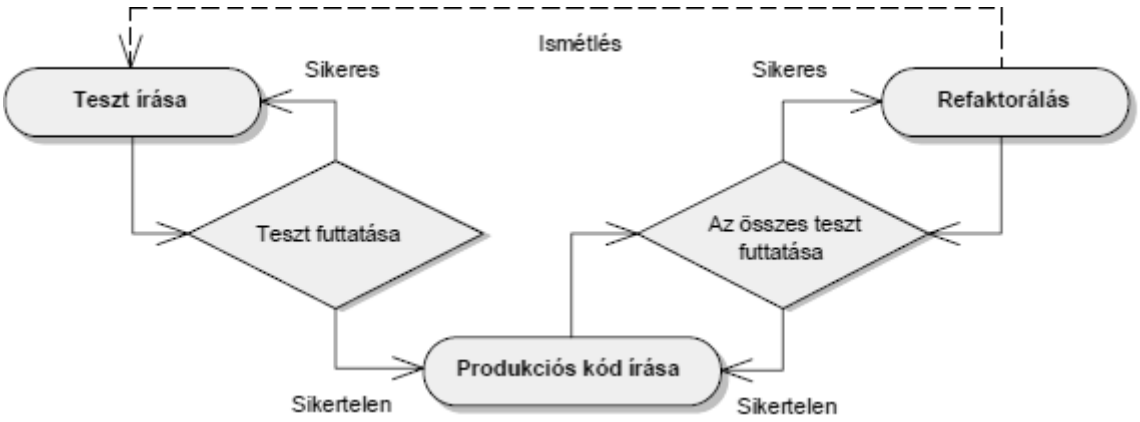
- A tesztek futtatása nem vesz el időt a fejlesztőtől (a manuális teszteléshez képest), és akárhányszor megismételhetők, önellenőrzők (azaz futásuk eredménye egyértelműen sikeres vagy sikertelen, nem szükséges az eredmény utólagos vizsgálata).
- A legtöbb esetben a hibakeresésre fordított idő jelentősen 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 összes szoftverkomponens, funkcionalitásuk 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 (tehát sikertelen és sikeres teszt-lefutásokat jelölnek). Dolgozatomban a `PHPUnit` **[2]** teszt-keretrendszert és a hozzá kapcsolódó eszközöket alkalmazom. Használatukat a fejlesztői dokumentáció keretein belül részletezem.

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

1.2. 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, illetve rengeteg hasznos segédeszköz érhető el hozzá, így ideális környezet 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 szemléltessem működését. Ez az alkalmazás egy egyszerű feladatlista-karbantartó program, használatát és funkcionalitását a 2. fejezetben fejtettem ki.

1.3. A dolgozat felépítése

A következő fejezetben ismertetem a kifejlesztett keretrendszer és a bemutató alkalmazás rendszerkövetelményeit, telepítését, és használatát.

A fejlesztői dokumentációban bemutatom a legfontosabb komponensek működését és egymáshoz való viszonyát. Ismertetem keretrendszer kiterjesztési és továbbfejlesztési lehetőségeit, illetve kitérek a *PHPUnit* segítségével írható tesztek felépítésére, futtatásának módjára is.

2. Felhasználói dokumentáció

2.1. Rendszerkövetelmények

A következő szoftverek a keretrendszer és a bemutató alkalmazás használatához, és a tesztek futtatásához is szükségesek:

- PHP 5.3.0, vagy újabb verzió (ajánlott 5.3.5, vagy újabb)
- Bármilyen kompatibilis webszerver illetve operációs rendszer (ajánlott Apache HTTP Server)
- A tesztek futtatásához: PHPUnit, illetve vfsStream PEAR-csomagok
- A kód-lefedettség jelentésekhez: Xdebug PHP-kiegészítés
- A teljesítménymérési adatok megtekintéséhez: WinCacheGrind vagy KCachegrind (ajánlott ez utóbbi bővebb tudása miatt)

A bemutató alkalmazás működéséhez, használatához szükséges további eszközök:

- MySQL adatbázis szerver (ajánlott 5.1 verzió, vagy újabb)
- A felhasználói felület tesztek futtatásához: Selenium Server (JRE szükséges)
- Bármely korszerű, Selenium-kompatibilis webböngésző (Firefox 4, Safari 5, Internet Explorer 9)

A letöltések és telepítési útmutatók hivatkozásainak listáját a *„6.1. Letöltési linkek és telepítési útmutatók”* melléklet tartalmazza. Ezen felül a dolgozat mellékleteként beadott DVD-lemez tartalmaz egy előtelepített virtuális gépet, melyen megtalálható a fent említett komponensek mindegyike. A gép a `VirtualBox` virtualizációs környezettel használható, melynek telepítői Windows és Linux platformokra szintén helyet kaptak a lemezen.

2.2. A lemezmelléklet könyvtárstruktúrája

A lemezmellékleten három fő könyvtár található:

- `forráskód`: a bemutató alkalmazás és a keretrendszer forráskódja (mivel a PHP interpretált nyelv, lényegében ezek a fájlok futtathatók webszerver-környezetben)

- `telepítők`: a VirtualBox virtualizációs környezet telepítőkészletei Windows operációs rendszerekre és különböző Linux disztribúciókhoz
- `virtuális gép`: ebben a könyvtárban található az a fájl, mely a VirtualBox segítségével elindítható (`kampaai.vbox`), illetve a `login.txt` állományban találjuk a gépre történő bejelentkezéshez szükséges felhasználónevet és jelszót.

A GÉP ELINDÍTÁSA ELŐTT MINDENKÉPPEN MÁSOLJUK A MEREVLEMEZRE AZ EGÉSZ MAPPÁT!

2.3. Telepítés előkészített rendszeren

Az alábbi folyamat opcionális, az alkalmazás kipróbálásához ajánlott az előtelepített virtuális gép használata. A leírt telepítési folyamat szerverkörnyezettől függően lehet, hogy nem teljes, csak a fő lépéseket emeli ki. A leírás nem terjed ki a tesztek futtatásához szükséges infrastruktúra létrehozására.

Amennyiben telepítve rendelkezésre állnak az előző pontban felsorolt szoftverek, a bemutató alkalmazás az alábbi lépésekkel indítható el (feltételezve Linux és Apache HTTP Server használatát, illetve hogy a `DOC_ROOT` beállítása a `/var/www` mappára mutat, a szerver pedig `http://localhost/` címen érhető el, és rendszergazdai jogosultságokkal rendelkezünk az operációs rendszer felett):

1. Másoljuk a `forráskód` könyvtárból a `/var/www` könyvtárba a `framework` és `demoApplication` almappákat. Jogosultságaikat állítsuk be úgy, hogy a webszerver olvasni tudja őket. A `/var/www/demoApplication/logs` könyvtárra írási jogot is adni kell.
2. A webszerver jogosultságait állítsuk be úgy, hogy a keretrendszer könyvtára a szerveren keresztül ne legyen elérhető, az ne szolgáltassa ki tartalmát (a következőket a webszerver `httpd.conf` konfigurációs állományában kell elhelyezni):

```
<Directory "/var/www/framework">
    Order allow,deny
    Deny from all
</Directory>
```

Hasonló megkötéseket kell tennünk a bemutató alkalmazás könyvtárára is, némi kiegészítéssel:

```
<Directory ”/var/www/demoApplication”>
     DirectoryIndex index.php
     Order allow,deny
     Deny from all
</Directory>

<Directory ”/var/www/demoApplication/assets”>
     Allow from all
</Directory>

<FilesMatch ”^(|index\.php)$”>
     Allow from all
</ FilesMatch>
```

Ilyen beállítások mellet csak a stíluslapokat és képeket tartalmazó `assets` könyvtár, illetve az alkalmazás belépési pontja, az `index.php` érhető el a szerveren keresztül. A beállítások módosítása után újra kell indítani a webszervert.

3. Az adatbázisszerveren futtassuk le a `/var/www/demoApplication` könyvtárban található `create_database.sql`, `create_test_database.sql` és `insert_data.sql` fájlokat. Ezek két adatbázist (`demoApplication` és `demoApplicationTest`), és a hozzájuk tartozó két felhasználót készítik el, illetve töltik fel adatokkal.
4. A PHP futtatókörnyezet alapbeállításai megfelelnek, az alkalmazás futásához azonban engedélyezni kell a következő kiegészítőket (ha nem lettek belefordítva a futtatókörnyezetbe): `mysql`, `pdo-mysql`, `mbstring`.

A fenti lépések elvégzése után a bemutató alkalmazás a *„http://localhost/demoApplication/”* címen érhető el web böngészővel.

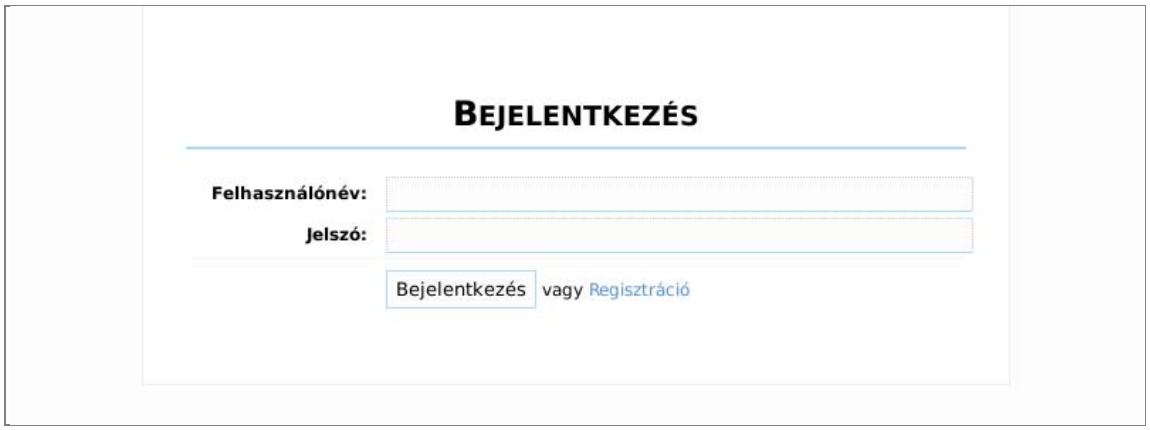
A továbbiakban a virtuális gép használatát feltételezem a felhasználó részéről.

2.4. Bemutató alkalmazás

A keretrendszer bemutatására egy egyszerű feladat- vagy tevékenység-lista karbantartó webes alkalmazást készítettem. A felhasználási esetek úgy lettek kialakítva, hogy kihasználják a keretrendszer minden lényeges, bemutatandó elemét. A fő funkciók a következők:

- Be- és kijelentkezés, regisztráció
- Saját és publikus feladatlista megtekintése (lapozással, 5 soronként)
- Új feladat hozzáadása, saját feladatok szerkesztése, törlése
- Publikus feladatok és felhasználói adataik megtekintése

Az alkalmazást a virtuális gép asztalán található *„Bemutató alkalmazás”* ikon segítségével indíthatjuk el. Ekkor elindul az alapértelmezett webböngésző, és betöltődik a *„http://localhost/”* lokális webszerver által kiszolgált oldal. Az alábbi képernyőfotók helytakarékossági okokból nem tartalmazzák a böngészőablak keretét és eszköztárait.



2. ábra: Az alkalmazás indító képernyője

Amennyiben a felhasználó nem rendelkezik érvényes belépési azonosítóval, a *„Regisztráció”* linkre kattintva készíthet magának egyet:

3. ábra: Új belépési azonosító létrehozása — regisztrációs képernyő

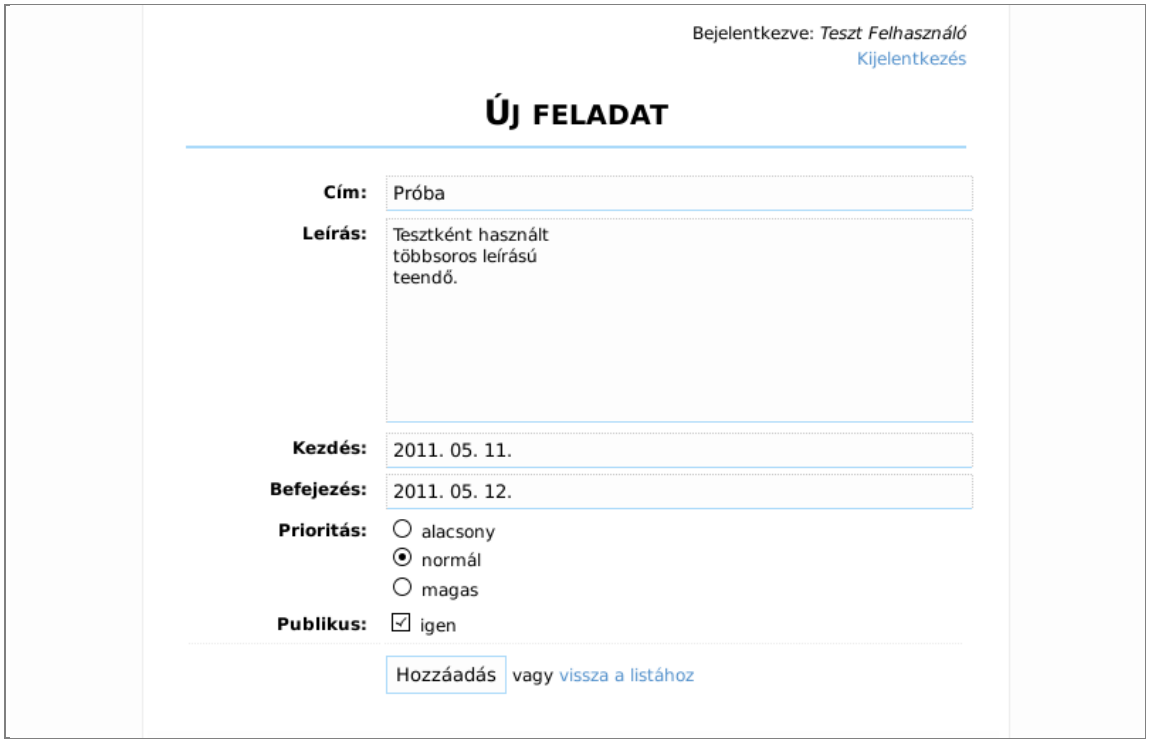
Minden mező kitöltése kötelező. Az alkalmazás ellenőrzi a mezők tartalmát az űrlap elküldésekor, és tájékoztatja a felhasználót az esetleges kitöltési hibákról (ez az alkalmazásban szereplő összes űrlapra igaz). A regisztráció után az alkalmazás visszanavigálja a felhasználót egy link segítségével a bejelentkezés oldalra. Sikeres belépés után a következő képernyő jelenik meg:

4. ábra: Feladatok listája

A bejelentkezés után minden további képernyőn elérhető az alkalmazás jobb felső sarkában a *„Kijelentkezés”* link, mellyel a felhasználó biztonságosan lezárhatja munkamenetét. A feladatok listája két részre bontható: felül a felhasználó saját feladatai jelennek meg (ez esetben még üres, hiszen új felhasználóról van szó), alatta pedig más felhasználók publikusként megjelölt feladatainak listáját láthatjuk. A listák mindig maximum 5 sort tartalmaznak, több elem esetén az alattuk található *„előző oldal”* és *„következő oldal”* linkekkel lapozhatók, egymástól függetlenül. Az adatok kezdési időpont szerinti növekvő sorrendben rendezve jelennek meg.

Feladatot hozzáadni a felső táblázat felett baloldalon elhelyezett *„Új feladat”* linkre kattintva lehet kezdeményezni. Az ekkor megjelenő űrlapot az 5. ábra mutatja. A *„Leírás”* mező opcionális, és többsoros szöveget is tartalmazhat. Az összes többi mezőt kötelező kitölteni. A dátum mezők elfogadott formátuma „*ÉÉÉÉ. HH. NN.*”, ahol *É* az

évszám, `H` a hónap, `N` az adott nap egy számjegye. Az űrlap elküldés után ezt a formátumot is ellenőrzi, illetve hogy a befejezés dátuma nem korábbi a kezdésnél. A `JavaScript` támogatással rendelkező böngészőkben a dátumok bevitelét egy külön komponens segíti, mely az adott dátum mezőbe lépéskor jelenik meg (6. ábra).



5. ábra: Új feladat hozzáadása

6. ábra: Dátumok bevitelét segítő komponens

Ha a feladat *„Publikus”* tulajdonsággal kerül elmentésre, meg fog jelenni más felhasználók számára is, a feladatlista alsó táblázatában (de szerkeszteni vagy törölni nem tudják). Ha valamely mező hibásan lett kitöltve, azt az adott mező alatt megjelenő hibaüzenetek jelzik. A feladat sikeres elmentése után a program két lehetőséget kínál a felhasználó számára: megtekinteti a frissen elkészült bejegyzést, vagy visszatérhet a feladatlistához.

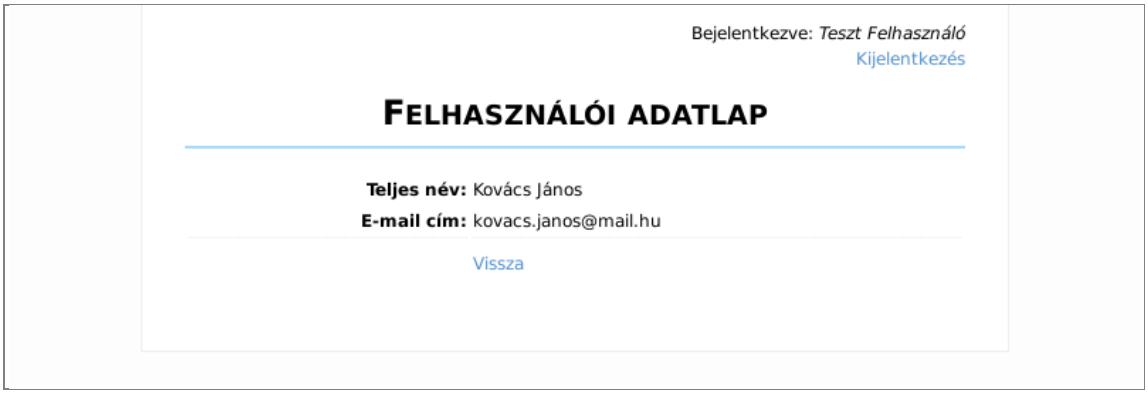
7. ábra: Az újonnan elkészült feladat adatlapja

Miután visszatértünk a feladatlistához látható, hogy az új feladat bekerült a felső táblázatba. Adatait módosítani a sor jobb szélén, *„Műveletek”* oszlopfejléc alatt található *„szerkesztés*”, eltávolítani pedig a *„törlés”* link segítségével tudjuk.

8. ábra: Szerkesztés és törlés linkek a „Saját feladatok” táblázatban

A szerkesztési képernyő egy ugyanolyan űrlapot tartalmaz, mint amilyet az új feladat hozzáadásánál láthattunk, azzal a különbséggel, hogy a mezők előre ki vannak töltve a kiválasztott feladat tulajdonságaival. A törlés linkre kattintva a program megerősítést kér a felhasználótól a feladat tényleges törléséhez.

A feladatlista *„Cím”* oszlopában szereplő szöveges linkek mind az adott feladat adatlapjára visznek (lásd 7. ábra). Az alsó táblázatban (*„Más felhasználók megosztott feladatai”*) egy felhasználó nevére kattintva az adatlapjára kerülünk, mely tartalmazza teljes nevét és regisztrációnál megadott e-mail címét.



9. ábra: Felhasználói adatlap

Az alkalmazás használatának befejezésekor a böngészőablak bezárása előtt érdemes a *„Kijelentkezés”* linkre kattintani, mert hatására munkamenetünk valóban megsemmisül az alkalmazásszerveren is, így illetéktelenek semmiképpen nem vehetik azt igénybe.

2.5. Keretrendszer

A felhasználást egy előre elkészített prototípus segíti, melyből kiindulva bármilyen egyszerű web-alkalmazás létrehozható a komponensek ismeretében (ezek dokumentációját a 3. fejezet tartalmazza). A prototípus a lemezmelléklet `forráskód/prototype` könyvtárában található. Használatához a 2.2. alfejezetben ismertetett telepítési eljárást kell végrehajtani, a `demoApplication` helyett a `prototype` mappát használva. Apache HTTP Server esetén a konfigurációs fájl módosítása nem szükséges, ha *htaccess* fájlok használata engedélyezett. A prototípus alkalmazás szerkezete a fájlrendszerben az alábbi:

```
forráskód/prototype/ ..... A prototípus alkalmazás könyvtára
  assets/ ..... Az alkalmazás kiegészítő elemeinek helye
    css/ ..... Stíluslapok
    images/ ..... Képek
    js/ ..... JavaScript fájlok
    .htaccess ..... Apache HTTP Server konfigurációs fájl
  classes/ ..... Az alkalmazás fő névterének helye
    control/ ..... Vezérlő osztályok
      DefaultController.php ..... Alapértelmezett vezérlő osztály
      ErrorController.php ..... Hibakezelő vezérlő osztály
  config/ ..... Alkalmazáskonfiguráció könyvtára
    application.ini ..... Alkalmazáskonfigurációs fájl
  coverage/ ..... Kódlefedettség-jelentések helye
  tests/ ..... Tesztkódok helye
```

```
forráskód/prototype/ ..... (folytatás)
  view/ ..... Nézet-sablonok főkönyvtára
    default/ ..... Alapértelmezett vezérlő sablonjai
      index.phtml ..... Kezdőlap sablon
    error/ ..... Hibakezelő vezérlő sablonjai
      exception.phtml ..... Kivétel hiba-sablon
      not-found.phtml ..... 404 hiba-sablon
    layout.phtml ..... Fő elrendezési sablon
  .htaccess ..... Apache HTTP Server konfigurációs fájl
  append.php ..... PHPUnit kódlefedettség segéd-szkript
  bootstrap.php ..... keretrendszer és osztálybetöltő inicializálása
  index.php ..... az alkalmazás belépési pontja
  phpunit.xml ..... PHPUnit konfigurációs fájl (tesztekhez)
  phpunit_coverage.php ..... PHPUnit kódlefedettség segéd-szkript
  prepend.php ..... PHPUnit kódlefedettség segéd-szkript
```

Mivel a keretrendszer alapvetően MVC (modell-nézet-vezérlő) **[3]** architektúrájú alkalmazások építését támogatja, új alkalmazás készítése során főleg a `classses/control` és a `view` mappa tartalma gyarapodik, melyek a vezérlő osztályokat és a megjelenítésért felelős sablonokat hivatottak tartalmazni. Ezek elvárt szerkezete, tartalma a fejlesztői dokumentációban ismertetett. A rendszer nem írja le vagy köti meg a modell osztályok kialakítását, a fejlesztő szabadon határozhatja meg ezeket. A bemutató alkalmazás például adabázis-kezelő modell osztályokat definiál és használ (pedig a keretrendszernek nem része az adatbázis-kezelés). A prototípus (és a bemutató alkalmazás) által tartalmazott alkalmazás-indítási szekvencia részeinek (`bootstrap.php` és `index.php`) leírása szintén a 3. fejezetben kapott helyet.

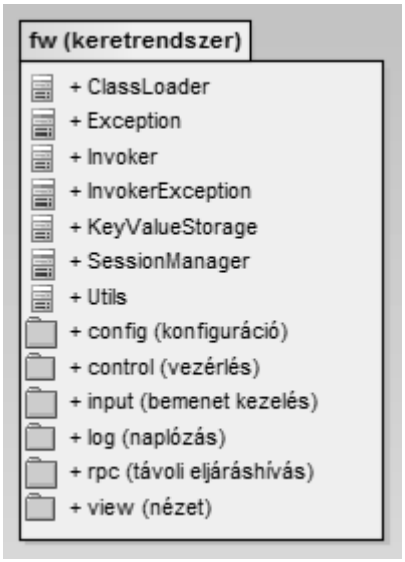
3. Fejlesztői dokumentáció

A bemutató alkalmazás felépítésének részletei csak a keretrendszer bemutatása után nyernek értelmet, ezért a dokumentáció ez utóbbival folytatódik.

3.1. Keretrendszer

3.1.1. KÖNYVTÁRSZERKEZET

A keretrendszer főkönyvtára (a mellékelt virtuális gépen ez a `/var/www/szakdolgozat/framework` könyvtár) öt alkönyvtárt tartalmaz. A `classes` könyvtárban találhatók meg a keretrendszert alkotó osztályok és interfészek. Egyben ez a könyvtár tartalmazza a fő névteret (`\fw`). Minden további névtér ennek egy alkönyvtárában foglal helyet. Struktúráját a 10. ábra szemlélteti. A `coverage` könyvtár a `PHPUnit` által készített kódlefedettségi jelentést tartalmazza (erről még szó esik egy későbbi fejezetben). A `testdox` könyvtárban a futtatott tesztek listáját találjuk meg szintén HTML formátumban. A `tests` könyvtár és alkönyvtárai a fejlesztés során elkészült egység- és integrációs tesztek kódját tartalmazzák. Ennek a könyvtárnak a szerkezete lényegében megegyezik a `classes` könyvtár szerkezetével. Az egyetlen eltérés az `assets` alkönyvtár jelenléte, melyben a tesztekhez szükséges segédosztályok, konfigurációk, adatok kaptak helyet.



10. ábra: A fő névtér osztályai és alnévterei

A főkönyvtárban található `bootstrap.php` a keretrendszer osztálybetöltőjét állítja be (ez a ráépülő alkalmazások számára fontos), a `phpunit.xml` pedig a tesztek futtatásához szükséges konfigurációs fájl.

Az alábbiakban nem ismertetem minden osztály minden egyes metódusát, inkább átfogó képet szeretnék alkotni az egyes csomagokban található osztályok szerepéről és működéséről. A `docs` könyvtárban megtalálható a forráskód megjegyzés-blokkjaiból automatikusan generált alkalmazásprogramozási interfész dokumentáció.

3.1.2. A FŐ NÉVTÉR (`\FW`) OSZTÁLYAI

`\fw\Exception`

A nyelvbe beépített kivétel osztály közvetlen leszármazottja. A keretrendszer minden további kivételosztályának ősosztálya. Célja, hogy a keretrendszer kivételei megkülönböztethetők legyenek a ráépített egyéb alkalmazásosztályokétól. Sem konstanssal, adattaggal vagy művelettel nem egészíti ki ősosztályát.

`\fw\ClassLoader`

Mivel a PHP nyelv nem ad szabályokat, hogy osztályainkat miként rendszerezzünk fájlokba, a fájlok betöltéséért felelős kódot nekünk kell elkészíteni. A osztálybetöltő pont erre a feladatra készült: ha az értelmező nem talál egy osztályt, hozzá fordul. Ekkor a betöltendő osztály neve és névtere alapján (a kettőt együtt teljesen minősített névnek[5] is nevezik) meghatározza a betöltendő fájl útvonalát, majd befűzi a végrehajtási láncba a `require_once` nyelvi szerkezettel. Ezt az osztályt manuálisan kell betölteni. Ez látható a keretrendszer főkönyvtárában található `bootstrap.php` fájlban.

Egy osztálybetöltő példány létrehozásához legalább két paramétert kell megadnunk: az osztályok alapkönyvtárát (`classpath`), és az ehhez tartozó alapértelmezett névteret (`defaultNamespace`). A harmadik, logikai paraméter azt jelzi, hogy regisztrálni kell-e az előtöltőt. Ez azért fontos, mert hiába hozzuk létre a betöltő példányt, ha nem regisztráljuk a futtatókörnyezet számára, nem fogja osztálybetöltésre használni. A regisztrációt nem csak konstruktor paraméterből, hanem később is kérhetjük, a betöltő példány `register()` metódusával, illetve vissza is vonhatjuk az `unregister`() segítségével. Ezzel gyakorlatilag hatástalanítjuk a betöltőt.

A betöltés folyamatát valójában az `autoload(string)` metódus végzi, ezt hívja meg a futtatókörnyezet. Először ellenőrzi, hogy a kapott osztály névtere részét képezi-e a konstruktorban megadottnak. Ha nem, nem foglalkozik a betöltésével (például: a betöltő az „fw” névtérrel jön létre. Ez a példány nem fogja betölteni az `\app\Class` osztályt, de a `\fw\namespace2\Class` osztályt már igen). Az alnévtereket könyvtárnevekké alakítja. Ha a megadott alapkönyvtár `/var/www/classes` volt, és az

[5] fully qualified name

alapértelmezett névtér az „fw”, akkor a `\fw\ns1\ns2\Class` osztályt a `/var/www/classes/ns1/ns2/Class.php` fájlból tölti be.

`\fw\KeyValueStorage`

Kulcs-érték tár. Asszociatív tömbökhöz készült burkoló osztály. Főleg konfigurációs struktúrák és HTTP lekérések paramétereinek tárolásánál használatos. Fő célja, hogy az asszociatív tömbök nehézkes kezelését megkönnyítse, megrövidítve a szükséges kódot. Használatát egyszerű példákkal lehet a legjobban bemutatni:

```
// adat asszociatív tömbökben
$data = array(
  'key1' => 'value1',
  'key2' => array(
    'key21' => 'value21',
    'key22' => array(
      'key221' => 'value221'
    )
  )
);

// burkolás kulcs-érték tárral
$kvs = new \fw\KeyValueStorage($data);

// értékek elérése
$v = $kvs->key1                          // $v == 'value1'
$v = $kvs->get('key3', 'defaultValue')   // $v == 'defaultValue'
$v = $kvs->key2->key22->key221           // $v == 'value221'
$v = $kvs->key2->has('key21')            // $v == true
$v = $kvs->key5->key6->has('key7')       // $v == false
$v = $kvs->key5->key6->key7              // $v: new KeyValueStorage()

// értékek beállítása
$kvs->key3 = 'value3';
$kvs->set('key4', 'value4');
```

Ha a konstruktor paraméterben szereplő tömböt nem adjuk meg, üresen jön létre a tár. Ha megadtuk, és a tömb többszintű, akkor minden szintjét bejárja, és a tömböket mind kulcs-érték tárakra cseréli, így valójában rekurzív struktúrát alkot. Ugyanez a transzformáció történik meg, ha egy kulcsnak a `set` metódussal vagy értékadással tömböt állítunk be. Ha nem létező kulcs értékét kérdezzük le, és nem adtunk meg alapértelmezett értéket (mert például adattagként hivatkoztuk az alternatív szintaxissal), vagy `null`-t adtunk meg, akkor egy üres kulcs-érték tárat kapunk. Ez a mechanizmus ahhoz szükséges, hogy nem létező kulcsok lekérési láncai ne okozzanak futási hibát. A tárat tömbbé konvertálhatjuk a `toArray()` meghívásával.

`\fw\Invoker`

A keretrendszer vezérlése és a távoli eljáráshívást végrehajtó programrészek ennek az osztálynak a segítségével képesek egy karakterláncként, nevével megadott osztályt példányosítani, és meghívni rajta a egy adott metódust a kívánt paraméterekkel.

Konstruktor paraméterként egy osztály teljesen minősített nevét és konstruktor paramétereit várja tömb formájában. A megadott osztályt példányosítja ezek segítségével, és eltárolja a keletkezett példányt. A további metódusok nagyrészt ezen a belső példányon végeznek műveleteket, vizsgálatokat. Ha a megadott osztály nem tölthető be, `\fw\InvokerException` kivétel keletkezik, `MISSING_CLASS` kóddal.

Olykor szükséges megvizsgálni, hogy a belső példány olyan osztályból származik-e, mely implementál egy adott interfészt, vagy leszármazottja egy megadott osztálynak. Az ilyen típusú az ellenőrzéseket a `checkInterface($interfaceName)` illetve a `checkParentClass($parentClassName)` metódussal végezhetjük el. A vizsgálatok eredményét nem visszatérési értékük jelzi, hanem sikertelenség esetén kivételt dobnak `INVALID_IMPLEMENTATION` illetve `INVALID_SUBCLASS` kóddal. Ez az eredményjelzési mód jobban használható együtt az osztály további metódusaival, mert jellemzően egymás után, egy kivételkezelő blokkban kerülnek meghívásra ezek a műveletek.

Az `invoke($methodName, $arguments)` művelet segítségével egy nevével megadott metódust hívhatunk meg a belsőleg eltárolt példányon. A második paraméter a hívás paramétereinek tömbje. Ez belsőleg két további műveletet hajt végre: lekéri a nevezett metódus önelemzési információit a `getMethod($methodName)` hívásával, illetve illeszti ezekhez a megadott paramétereket (`matchArguments(…)`). Az illesztés azt szolgálja, hogy egy asszociatív tömbben, nevükkel kulcsolt paraméterértékek esetén is helyes paramétersorrendet tartson a nevezett metódus híváskor. Ez távoli eljáráshívás implementálásánál hasznos, mert egyes protokollok megengedik a nevekkel ellátott, nem sorrend-tartó paraméterezésű hívásokat. Ha a hivatkozott metódus nem létezik, vagy a paraméterek nem megfelelők, kivétel keletkezik `MISSING_METHOD` vagy `INVALID_PARAMETERS` kóddal.

`\fw\InvokerException`

A `\fw\Invoker` osztály által dobott kivételeket fogja össze, és azok kódját tartalmazza. A `\fw\Exception` osztályból származik. A kódok és értékeik:

```
const MISSING_CLASS           = 1;
const INVALID_IMPLEMENTATION  = 2;
const INVALID_SUBCLASS        = 3;
const MISSING_METHOD          = 4;
const INVALID_PARAMETERS      = 5;
```

`\fw\SessionManager`

Statikus osztály, munkamenetek kezelését segíti. Védelmet tartalmaz a „*session hijacking*” illetve „*session fixation*” támadási technikák ellen **[4]**. Ezt úgy valósítja meg, hogy bejelentkezéskor új munkamenet azonosítót generál, illetve ellenőrizhető, hogy a munkamenet-azonosító ugyanazon klienstől érkezett-e folytatásra, mint amelyikhez létrejött. Ez a funkcionalitás a következő műveletekkel érhető el:

- `start(string)`: Elindítja vagy folytatja az aktuális munkamenetet a `session_start` beépített függvénnyel. A paraméterként megadott nevet használja a munkamenet neveként.
- `login()`: Újragenerálja a munkamenet-azonosítót, majd bejelentkezett munkamenetként jelöli meg az aktuálisat. A kliens IP-címe és a kliensszoftver azonosítója elmentésre kerül a munkamenetben. Felhasználók bejelentkeztetésénél használatos.
- `isValid()`:A munkamenet érvényességének megállapítására szolgál. Visszatérési értéke igaz, ha az aktuális munkamenet eredete megfelel a kliens IP-címének és a kliensszoftver azonosítójának. Miden egyéb esetben hamis.
- `destroy()`: A munkamenet-süti (*session cookie*) törlésére szólítja fel a klienst, és a szerveren is megszűnteti az aktuális munkamenetet. Felhasználók kijelentkeztetésénél használatos.

A munkamenet változók kezelésére nem tartalmaz külön interfészt, azokat továbbra is a beépített `$_SESSION` szuperglobális tömbön keresztül célszerű elérni.

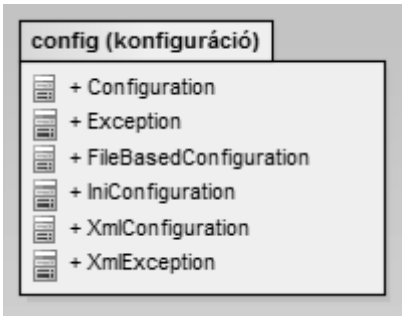
`\fw\Utils`

Egy csak statikus metódusokkal rendelkező segédosztály. Többek közt a naplózás, munkamenet kezelés és a vezérlés egy kis része is használja. Műveletei:

- `getClientIp()`: Visszaadja a kliens IP címét
- `getUserAgent()`:A kérést indító kliens szoftver azonosítója (*user agent string*).
- `setLocation(string)`: A HTTP „Location” fejléc értékét beállítja a megadott paraméter értékre, majd megszakítja a program futását, így HTTP átirányítást végez.

3.1.3. KONFIGURÁCIÓ (`\FW\CONFIG`)

A web-alkalmazások számára különösen fontos, hogy jól konfigurálhatóak legyenek. Ez többnyire abból fakad, hogy jelentős eltérések lehetnek a különböző szerverek kialakításában, illetve kimondottan fejlesztési és tesztelési célokra fenntartott környezetek között is gyorsan kell tudni áthelyezni őket. Ennek legegyszerűbb módja, ha a komponensek konfigurációját fájlokba helyezzük. A keretrendszerben implementált konfiguráció-kezelés háromféle formátumot támogat: PHP tömböket és objektumokat, illetve INI és XML fájlokat képes kezelni. Ezen kívül a futtatási környezetek közti áthelyezést az is segít, hogy a konfigurációk szekciókra osztottak, melyek örökölhetik, illetve felülírhatják egymás beállításait. Így egyszerűen az aktív szekció megadásával választhatunk, hogy melyik környezetbe alkalmas konfigurációt használja az alkalmazás.



11. ábra: A konfigurációs névtér osztályai

A konfigurációk alapját valójában a kulcs-érték tárak (`\fw\KeyValueStorage`) képezik, ebből származik ősosztályuk: a `\fw\control\Configuration`. Ez egy olyan, módosított változata az eredeti kulcs-érték tárnak, ahol a legfelső szintű kulcsok alkotják a konfigurációs szekciókat. Ez a szint közvetlenül elérhetetlen, ugyanakkor mindig van egy kiválasztott szekció, melyet úgy használhatunk, mint az eredeti kulcs-érték tárakat. A kiválasztott szekció neve statikusan tárolódik (lekérdezés: `Configuration::getActiveSection()`), ezért az alkalmazás összes konfigurációján egyszerre cserélhető (`Configuration::setActiveSection($newActiveSection)`). Az alapértelmezett szekció a „`default`” nevet viseli.

A `\fw\control\Configuration` osztály első konstruktor-paramétere egy tömb. Ezt fogja hasonlóképpen burkolni, mint a kulcs-érték tár. Az egyetlen különbséget az

okozza, hogy ha a második paraméter hamis, akkor nem a legfelső szintre, hanem az aktív szekció kulcsa alá töltődnek be az adatok. Ellenkező esetben a paraméterként adott tömb legfelső szintjén lévő kulcsokból keletkeznek a szekciók. Például:

```
use \fw\config\Configuration;

// az alapértelmezett szekció a "default", ebbe töltődnek az adatok
$config = new Configuration(array('key1' => 'value1'), false);

$v = $config->key1;             // $v == 'value1'

// szekció váltás
Configuration::setActiveSection('other');

$v = $config->has('key1');      // $v == false

// az adatok szekciókat is tartalmaznak (itt csak egyet)
$config = new Configuration(
  array(
    'other' => array('key2' => 'value2')
  ),
  true
);

$v = $config->has('key2');      // $v == true
$v = $config->key2;             // $v == 'value2'
```

Lehetőség van konfigurációk összefésülésére is a `merge(Configuration $other)` művelettel. Ez nem változtatja meg a műveletben résztvevő egyik konfigurációt sem, hanem egy újat készít, mely mindkettő adatait tartalmazza. Kulcsütközések esetén a paraméterként megadott konfiguráció értéke kerül alkalmazásra, és nem azé a példányé, melyen meghívták.

A fájl alapú konfigurációk ősosztálya a `\fw\config\FileBasedConfiguration`, absztrakt osztály, mely egységesíti a fájlmegnyitási hibák kezelését, és a fájl értelmezésének folyamatát. Így minden fájl-alapú konfiguráció egyetlen konstruktor-paraméterrel rendelkezik: ez a beolvasni kívánt fájl elérési útvonala. Alosztályai melyek XML és INI formátumokat olvasnak, csupán az absztrakt `_parseFile($filePath)` és `_parseIntoArray($parserResult)` megvalósításban különböznek. Ezek a metódusok felelnek a fájl tartalmának értelmezéséért, illetve az értelmező eredményének asszociatív tömbbé alakításáért, melyből később kulcs-érték tárrá transzformálódik tartalmuk.

INI formátumú fájlokból a `\fw\config\IniConfiguration` osztály segítségével hozhatunk létre konfigurációs objektumot. A szekciókat az INI fájl szekciói alkotják, a kulcsok szintjeit pedig pont karakterrel választjuk el. A szekciók öröklésséhez a kettőspont karaktert használjuk, a szülő szekció neve áll hátul. Példa (INI fájl tartalma):

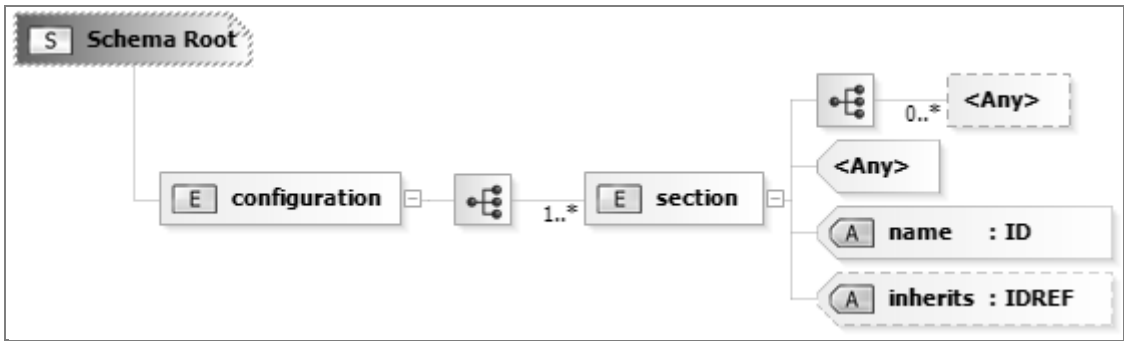
```
[ base ]
key1 = value1
key2.key3.key4 = value4

[ default : base ]
key1 = value100
```

Ezt a fájlt beolvasva a következőket kapjuk (az aktív szekció: „`default`”):

```
$config = new \fw\config\IniConfiguration('config.ini');

$v = $config->key1;                          // $v == value100
$v = $config->key2->key3->key4               // $v == value4
```

A PHP `parse_ini_file` függvénye megengedi, hogy a konfigurációs fájlokban a `define` beépített függvénnyel létrehozott konstansokat használjuk, ezeket kiértékeli a beolvasáskor. Ezen felül egy kulcs értéke tömb is lehet (ez szintén kulcs-érték tárrá transzformálódik majd).

XML konfiguráció beolvasásához a `\fw\config\XmlConfiguration` osztályt használhatjuk. Az értelmezés során elvégez egy ellenőrzést a keretrendszer főkönyvtárán belül található „`classes/config/schema/configuration.xsd`” XML-séma alapján. A séma szerkezete a következő ábrán látható:



12. ábra: Az XML konfigurációk ellenőrző sémája

Ez lényegében csak a szekciók meglétére és formájára tartalmaz megkötést. Fontos, hogy a gyökérelem névterének meg kell adni a „`urn:fw-configuration`” karakterláncot, különben a séma-ellenőrzés sikertelen eredménnyel zárul. Az előző példában szereplő INI konfigurációnak megfelelő XML fájl:

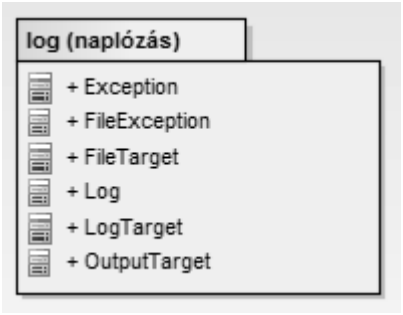
```
<?xml version="1.0" encoding="utf-8"?>
<configuration xmlns="urn:fw-configuration">
     <section name="base">
          <key1>value1</key1>
          <key2>
               <key3 key4="value4"/>
          </key2>
     </section>
     <section name="default" inherits="base">
          <key1>value100</key1>
     </section>
</configuration>
```

Amint a „*key4*” kulcs esetében látható, nem csak egymásba ágyazott elemek, hanem azok attribútumai is felhasználhatók az értékek definiálásához.

A bemutatott konfigurációs osztályok különböző olvasási és értelmezési hibák esetén kivételeket dobnak, melyek őse a `\fw\config\Exception` kivétel osztály. Pontos leírásuk a mellékleten elhelyezett generált dokumentációban található.

3.1.4. NAPLÓZÁS (`\FW\LOG`)

A keretrendszer egy egyszerű és rugalmas naplózási megoldást kínál az alkalmazások részére, melyet a `\fw\log\Log` osztályon keresztül valósít meg. Egy ebből az osztályból létrehozott napló-példányhoz több naplózási célt (*log tartget*) adhatunk meg. Ezek lényegében különböző típusú kimenetek. A névtérben két ilyen konkrét naplózási cél definiált: `OutputTarget` és `FileTarget`. Előbbi közvetlenül a standard kimenetre írja a naplósorokat, míg utóbbi egy megadott fájlba. Közös ősük az absztrakt `LogTartget` osztály, mely a naplósorok formázásáért felelős kódrészletet egységesíti. Egy adott sor tényleges kiírását a `write(…)` metódus implementációjában kell elvégezni (a sortöréseket is ennek a metódusnak kell kiírnia). A naplózás folyamata egy alkalmazás szemszögéből:



13. ábra: A naplózás osztályai

- Létre kell hozni egy példányt `a \fw\log\Log` osztályból. Konstruktor paraméterként egy logikai értéket vár, mellyel be vagy ki lehet kapcsolni a nyomkövetés szintű üzeneteket (lásd lejjebb).
- Naplózási célokat kell hozzáadni a naplóhoz. Ez a napló példány `addTarget` metódusával történik, melynek első paramétere egy naplózási cél példány, második

pedig egy tömb, mely naplózási szint konstansokat tartalmaz. Ezek a napló osztály „`LEVEL_`” kezdetű konstansai. Ha a tömb nem üres, csak a megadott naplózási szintekhez lesz hozzárendelve a cél, egyébként mindhez.

- A napló példányon meg kell hívni valamelyik naplózó metódust, a kívánt szintnek megfelelően. Ez lehet az `error` (hiba), `warning` (figyelmeztetés), `info` (információ) vagy `debug` (nyomkövetés). Mindegyik metódus egy paramétert vár: a naplósor üzenet mezőjét, ami egy tetszőleges karakterlánc. A nyomkövetés szintű naplózási kérelmek még a hozzárendelt naplózási célokat is csak akkor érik el, ha a napló konstruktorában bekapcsoltuk a nyomkövetési módot, vagy később beállítottuk egy `setDebugEnabled(true)` hívással a napló példányon.

A négy naplózási metódus végül az `_invokeWriteOnTargets` privát eljárásban fut össze. Ez megállapítja a hívási láncból az üzenet forrását (melyik osztály kódjából hívtuk meg a naplózást), és minden, az adott szinthez hozzárendelt naplózási célon meghívja a `write` metódust a szint, forrás és üzenet paraméterekkel.

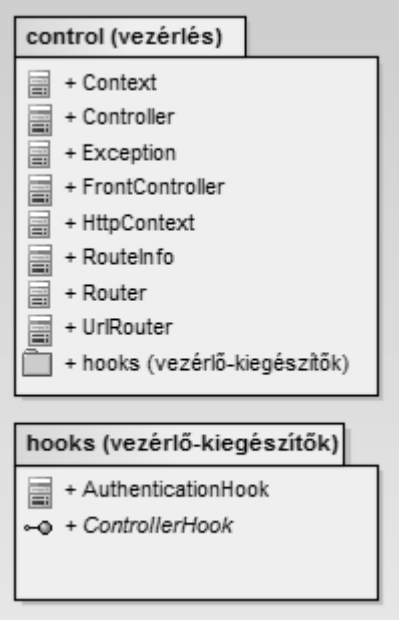
A naplósorok formátumát a naplózási célok konstruktor-paramétereivel állíthatjuk be. Az `OutputTarget` esetében ez az első és egyetlen paraméter. A `FileTarget` először a célfájl teljes elérési útvonalát várja, majd ezután a formátum-sztringet (ez mindkét esetben opcionális paraméter, alapértelmezett: „`%t - [%l] %s: %m (%h)`”). A formátum- sztring az alábbi helyettesítő karaktereket tartalmazhatja:

- `%l`: bejegyzés szintje
- `%s`: bejegyzés forrása
- `%m`: üzenet
- `%t`: dátum és idő
- `%h`: kliens IP címe

A *FileTarget* naplózási cél már a konstruktorában megkísérli megnyitni írásra az első paraméterként megadott fájlt. Hiba esetén `\fw\log\FileException` kivételt dob. (Ezt a felépítést, miszerint konstruktorban nyitjuk meg és destruktorban zárjuk le az erőforrásokat, „*Resource Acquisition Is Initialization*” névvel is illetik. **[10]**)

3.1.5. VEZÉRLÉS (`\FW\CONTROL`)

A vezérlés a keretrendszer legösszetettebb része. Feladata az alkalmazások kódjának áttekinthető, rendszerezett módon történő felosztása kezelhető méretű egységekre. Környezetet nyújt a paraméterek feldolgozáshoz és az eredmény közléséhez.



14. ábra: A vezérlés osztályai

Az alkalmazások kódjának felosztására kézenfekvő módszer, ha a funkcionálisan egybetartozó (például az azonos típusú entitásokat kezelő) részeket egy „vezérlőbe” (*controller*) csoportosítjuk. Egy vezérlő lényegében egy osztály, melynek metódusai az „akciók” (*action*). Ezek egy-egy felhasználói műveltre bekövetkező reakciót jelképeznek. (A bemutató alkalmazás példáján keresztül: a felhasználókkal kapcsolatos funkciók csoportja a `UserController` vezérlő. Ennek metódusai a `loginAction`, `logoutAction` és a `registerAction` akciók, melyek a felhasználók be- és kijelentkezését, illetve regisztrációját kezelik.)

Ez a felosztás természetesen megköveteli, hogy minden futás során kiválasszuk az aktív vezérlőt és akciót, majd végrehajtsuk. Ezt a folyamatot fogja össze a fővezérlő (`\fw\control\FrontController`). A folyamat indítása egy fővezérlő példányon történő `dispatch()` hívással kezdeményezhető. Az aktív vezérlő és akció kiválasztása valójában egy külön komponens: az útválasztó („*router*”) felelőssége. Egy vezérlő-akció párost egy `\fw\control\RouteInfo` struktúra tárol. Ezeken kívül egy harmadik adattaggal is rendelkezik: a paraméterekkel. Ez egy tömb (valójában kulcs-érték tár), mely kiegészítő információkat szolgáltat az adott akció számára. Az útválasztó osztályok fő feladata tehát egy ilyen struktúra előállítása a fővezérlő számára. Ezt az útválasztó osztályok őse, a `\fw\control\Router` osztály `parseRoute($route)` metódusa biztosítja. Mivel absztrakt, minden alosztályban kötelező felülírni, és nincs alapértelmezett implementáció. A dokumentáció további részében a konkrét `\fw\control\UrlRouter` osztályt használom bemutatás céljából. Ez az útválasztó a

modern web-alkalmazásoknál megszokott URL-alapú vezérlés-kiválasztást támogatja. A `parseRoute` metódusának működése a következő:

- Ha nem kapott paraméterként útvonalat, a webszerver által szolgáltatott `PATH_INFO` változó értékét használja fel (ez egy *„/index.php/a/b/c*” alakú URL esetén az „a/b/c” karakterlánc).
- Az útvonalat felbontja a ferde vonalak mentén („/”), az első kapott rész lesz a vezérlő, a második az akció neve. Ha valamelyik része hiányozna az információnak, alapértelmezett értékkel tölti ki. Tehát: ha nincs megadva akció-név, akkor a konfiguráció `router.default_action` beállításában keresi. Ha ott sincs definiálva, az alapértelmezett akció „`index`”. Ugyanez történik abban az esetben, ha a vezérlő megadása is hiányzik. Ekkor `router.default_controller` beállítást keresi, ha nincs jelen, az alapértelmezett vezérlőnevet állítja be: „`default`”.
- Ha vannak további részek, akkor paraméterként tárolja el őket. A paraméterekhez mindig kettesével veszi az egységeket: az első lesz a paraméter neve, a második az értéke. Ha egy paraméternév többször is előfordul, a paraméter értéke egy tömb lesz, mely mindegyik értéket tartalmazza.
- Az így kiolvasott részeket egy `\fw\control\RouteInfo` struktúrában visszaadja.

(Az útválasztó rendelkezik egy `generateRoute(RouteInfo $routeInfo)` metódussal is, mely ennek az inverzét képezi. Ezt linkek generálásához használjuk fel a nézetekben.)

Az aktív vezérlő-akció páros kiválasztása után osztály- és metódusnévvé kell transzformálni őket. Ez úgy történik, hogy a vezérlőnév a „`Controller`”, míg az akció neve az „`Action`” utótagot kapja (a két névmegállapító függvény a `getControllerClassName` és a `getActionMethodName`). Az osztály névtere a fővezérlő konstruktorparaméterében bejuttatott konfigurációs objektumból ered. A névteret a `controller.namespace` beállításban keresi. A konfiguráció tartalmazhat még egy fontos beállítást arra vonatkozóan, hogy az ebbe a névtérbe tartozó osztályok milyen útvonalon elérhetők. Ez a beállítás `controller.classpath`. A két érték ismeretében beállítható egy osztálybetöltő. Ezt a fővezérlő meg is teszi a konstruktora végén.

Az osztály- és metódusnév ismeretében végrehajtható a kívánt akció. Ezt a korábban bemutatott, a fő névtér osztályai közé tartotó `\fw\Invoker` osztály segítségével végzi el. Az akcióhoz tartozó metódus meghívása előtt még ellenőrzi, hogy a létrehozott vezérlő-példány valóban alosztálya-e a `\fw\control\Controller` osztálynak. Ez egyrészt védelmet nyújt, mert nem lehet nem vezérlőnek készített osztályt ilyen módon meghívni, másrészt lehetőséget nyújt a vezérlő-környezet injektálására a konstruktoron keresztül.

A vezérlő-környezet (`\fw\control\Context`) minden, amit egy akció fel tud használni feladatai végrehajtásához. Egy akció-metódus törzsén belül a `$this->_context` kifejezéssel érhetjük el. Ezen keresztül jutunk hozzá az aktuális útvonal-információhoz, a fővezérlőhöz, a nézethez és a konfigurációhoz. A keretrendszer alapértelmezett vezérlő-környezete ennek egy leszármazottja, a `\fw\control\HttpContext`. Ez a HTTP GET illetve POST paraméterek kulcs-érték táron keresztüli elérésével egészíti ki az eredeti kontextust, valamint alapértelmezett nézete a sablonokkal dolgozó `\fw\view\TemplateView` (ennek részletei a következő alfejezetben találhatók).

Ha vezérlő osztály vagy metódus nem található, a fővezérlő elkéri az útválasztótól a hiba-vezérlőt és a megfelelő akciót a `getErrorController()` és a `getNotFoundAction()` meghívásával. Ezek a `router.error_controller` és a `router.not_found_action` beállítások értékét adják vissza. Ha ezek nem találhatók, „`error`” és „`not-found`” lesz az eredmény.

Egyéb hiba esetén (például ha kivétel keletkezik az akció-metódus futása során) hasonló eljárást folytat: szintén a hiba-vezérlőt veszi elő az előbb ismertetett módon, de most a kivételek kezeléséért felelős akciót kéri le a `getExceptionAction()` metódussal. Ez a `router.exception_action` beállításban keres, alapértelmezett értéke *„exception”. A kivételt kezelő akció-metódust úgy kell megírni, hogy az semmilyen körülmények között ne dobhasson további kivételeket!*

Összefoglalva az alapértelmezéseket (üres konfiguráció esetén):

- Ha nincs elegendő útvonal információ a vezérlő-akció páros meghatározásához, akkor a `DefaultController indexAction` metódusa jut érvényre. Ha csak az akció hiányzik, akkor a megadott vezérlő `indexAction` metódusa.
- Ha a hivatkozott osztály vagy metódus nem létezik, az `ErrorController notFoundAction` metódusa hívódik meg.
- Végül, ha kivétel keletkezett az akció-metódus végrehajtása közben, akkor az `ErrorController exceptionAction` metódusa fut le.

A hibakezelők hívása valójában belső átirányítással történik. Ezt a fővezérlő `forward` metódusa végzi. Ezt a metódust akciók belsejéből is használhatjuk (így például megoldható, hogy bizonyos paraméterek hiánya esetén egy másik oldalon kössön ki a felhasználó). Külső átirányítás is végezhető a `forwardExternal` segítségével. Ez valójában a „`Location`” HTTP fejléc segítségével átnavigálja a klienst (persze csak a `\fw\control\UrlRouter` esetében). Mindkét metódus egy új útvonal-információt tartalmazó `\fw\control\RouteInfo` típusú struktúrát vár. Az utolsó átirányítási lehetőség a `forwardToDefault`, mely az alapértelmezett vezérlő alapértelmezett akciójára visz (az alkalmazás „kezdőlapja”). Ez egy logikai értékű paraméterrel rendelkezik, igaz érték esetén külső, egyébként belső átirányítást használ. Ha átirányítást használunk egy akció-metódus törzséből, akkor annak futása azonnal megszakad egy speciális kivétel hatására (`\fw\control\Exception`, `STOPPED_BY_FORWARD` kóddal). Ezt a kivételt „elnyeli” a fővezérlő, és az új útvonalnak megfelelő akció-metódusra helyezi a vezérlést.

Az akció-metódus lefuttatása után a fővezérlő utasítja a nézetet, hogy hajtsa végre az automatikus kiértékelését, amennyiben azt nem kapcsolták ki (további részletek a következő alfejezetben).

A fővezérlő további képessége, hogy közvetlenül az aktív akció-metódus elé és mögé ágyazhatunk be olyan vezérlő-kiegészítőket („*controller hook*”), melyek mindig lefutnak. Ezek segítségével az akciók számára transzparens módon oldható meg jogosultságkezelés vagy gyorsítótárazás. Egy vezérlő-kiegészítő létrehozásához a `\fw\control\hooks\ControllerHook` interfészt kell implementálni. Ennek mindössze

egy művelete van: `execute(Context $context)`. A megvalósítás törzsében használható átirányítás.

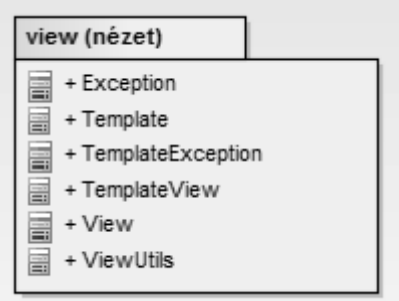
A keretrendszer szolgáltat egy vezérlő-kiegészítő vázát is, felhasználók authentikálásához. Ez a `\fw\control\hooks\AuthenticationHook` osztály. Alkalmazásainkban ezt úgy kell implementálnunk, hogy felülírjuk a `validateSession(Context $context)` absztrakt metódust olyan módon, hogy érvényes, bejelentkezett munkamenet esetén igaz, egyébként hamis értéket adjon vissza. Két konfigurációs értéket is olvas: az `auth.login_controller` és az `auth.login_action` határozzák meg azt a vezérlő-akció párost, mely a bejelentkező nézetet szolgáltatja a felhasználónak, érvénytelen munkamenet esetén ugyanis erre az oldalra végez belső átirányítást. Szintén a konfigurációból határozhatók meg azon vezérlő-akció párosok, melyek elérése publikus, azaz nem hajtja végre rajtuk az ellenőrzést (ezek például „`auth.public.<VEZÉRLŐ-NÉV>.<AKCIÓ-NÉV> = true`” formában hozhatók létre INI konfiguráció esetén).

A vezérlő-kiegészítőket szintén konfiguráció útján aktiválhatjuk. Az alábbi INI formátumú példa-konfiguráció három elő- (*pre-hook*) és két utó-futtatott kiegészítőt (*post-hook*) állít be:

```
…
controller.pre_hooks[] = \app\control\hooks\PreHook1
controller.pre_hooks[] = \app\control\hooks\PreHook2
controller.pre_hooks[] = \app\control\hooks\PreHook3
…
controller.post_hooks[] = \app\control\hooks\PostHook1
controller.post_hooks[] = \app\control\hooks\PostHook2
…
```

3.1.6. NÉZET (`\FW\VIEW`)

A keretrendszerrel megvalósított alkalmazások felhasználói felületének összeállításáért felelős osztályok a `\fw\view` csomagban találhatóak. A nézet osztályok ősét alkotó `View` absztrakt osztály egy rendkívül egyszerű interfészt definiál, hiszen a nézetek nagyon különbözőek lehetnek akár egy alkalmazáson belül is, így pont csak annyit követel meg, amely a vezérléssel történő integrálást lehetővé tette. Közös tulajdonságuk, hogy ki lehet



15. ábra: A nézet osztályai

értékelni őket, azaz kérhetjük kimenetük előállítását. Ezt a `render($returnsOutput)` hívásával kérhetjük egy nézet-példánytól. A logikai értékkel kitölthető paraméter azt határozza meg, hogy kiértékelésének eredményét azonnal a standard kimenetre írva (hamis érték), vagy visszatérési értékként (igaz érték) szeretnénk megkapni. További közös tulajdonság, hogy a vezérlés automatikusan meghívja-e ezt a metódust (`setAutoRender(bool)` hívásával állítható). Ez kényelmi szempontokat szolgál, hogy ne kelljen minden vezérlő-akció végén manuálisan kérni a kimenet létrehozását. Kikapcsolására azért van lehetőség, mert néhány ritka esetben, például távoli eljáráshívásnál nem használunk nézeteket, mivel a kimenet nem a felhasználói felületre, hanem egy másik programhoz kerül.

A keretrendszer alapértelmezett nézete (`\fw\view\TemplateView`) egy klasszikus sablon-alapú nézet egyszerűsített változata. Konstruktor paraméterként egy konfigurációs objektumot vár, melyből kiolvassa a `view.template_directory` beállítás értékét. A továbbiakban ebből a könyvtárból próbálja meg beölteni a sablon-fájlokat. Alapvetően két sablonnal dolgozik: egy elrendezési (*layout*), és egy akció (*action*) sablonnal. Mindkét sablon megállapításához figyelembe veszi a vezérlő-környezet (`Context`) aktuális útvonal-információját (`RouteInfo`).

Az akció sablont úgy választja ki, hogy a vezérlő nevéből könyvtárnevet képez, melyet hozzáilleszt a konfigurációban megadott alap sablon-könyvtárhoz. A sablon nevét maga az akció neve adja. Például, ha az alapértelmezett sablon könyvtár a „`/var/www/view`”, az aktuális vezérlő „`controller`”, és akcio az „`action`”, akkor a „`/var/www/view/controller/action`” sablont tölti be. Ezt egy `\fw\view\Template` példány létrehozásával éri el, mely a sablon nevét kiegészíti a „*.phtml*” kiterjesztéssel. A sablon példányt később a nézet `getActionTemplate()` műveletével lehet lekérdezni, illetve a `setActionTemplate(Template $template)` metódussal beállítani. Ha kézzel akarunk beállítani egy sablont, annak létrehozásához érdemes a `createTemplate($templateName)` függvényt meghívni, mert így az alap sablon-könyvtár átadódik a sablon konstruktorának a nézet egy belső adattagjából, és nem kell azt manuálisan megadnunk.

Az elrendezés sablon kiválasztásához három dolgot vesz figyelembe. Először megpróbálja az adott vezérlőhöz és akcióhoz tartozó egyedi elrendezést beolvasni a konfigurációból. Ezt a `view.layout.<VEZÉRLŐ-NÉV>.<AKCIÓ-NÉV>` beállításban keresi. Ha nem létezik ilyen beállítás, a `view.default_layout` értékével próbálkozik. Amennyiben ez sem létezik, a „`layout`” nevű sablont tölti be. Ennek eredménye „`/var/www/view`” alapkönyvtár mellett a „`/var/www/view/layout.phtml`” betöltése.

A sablonok PHP kódot is tartalmazhatnak. A felhasználható változókat úgy kapják, hogy a sablon példányon tetszőleges adattagot beállítva azok a kiértékelés során a sablon kódjában lokális változók lesznek. Példa (sablon kódja, `example.phtml`):

```
<?php echo $variable1; ?>
```

Ezt betöltve, majd változót behelyettesítve:

```
// sablon léterhozása
$template = new \fw\view\Template('example');

// változó beállítása
$template->variable1 = 'value1';

// sablon kiértékelése, miután lefutott: $v == 'value1'
$v = $template->evaluate(true);
```

Mint látszik, a sablonok kiértékelése a nézetekkel szemben nem a `render(…)`, hanem az azonos paraméterezésű `evaluate(…)` metódussal történik.

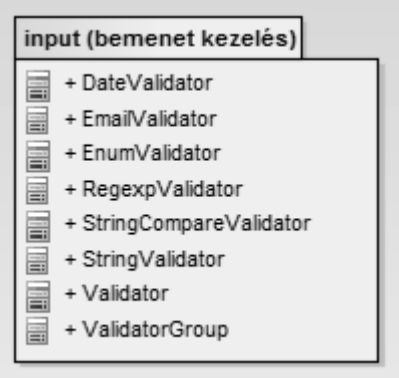
A sablon-nézet kiértékelése során az történik, hogy kiértékeli a beállított akció-sablont, majd ennek eredményét az elrendezés-sablon `layoutContents` adattagjába tölti. Ezután kiértékeli az elrendezés-sablont is, és ez lesz az eredménye a nézet kiértékelésének. Tehát egy értelmes elrendezés sablon legalább az alábbit tartalmazza:

```
<?php echo $layoutContents; ?>
```

Ha ez nem található meg az elrendezés-sablonban, akkor az akció-sablonok tartalma nem kerül bele a végső kimenetbe.

3.1.7. BEMENET ELLENŐRZÉS (`\FW\INPUT`)

Ebben a csomagban a klienstől érkezett adatok ellenőrzésére és szűrésére szolgáló segédosztályokat találhatunk. Biztonsági szempontból kulcsfontosságú a web-alkalmazások bemenetére érkező adatok ellenőrzése, mert ez az egyik fő támadási pontjuk. Az adatokat nem csak ellenőrizni, hanem szűrni is kell, mert speciálisan előkészített adatok támadókódként juthatnak vissza a kliensbe (*XSS* támadások **[8]**).



16. ábra: A bemenet kezelés névtere, osztályai

A különböző típusú bemeneti adatokhoz a keretrendszer különböző validációs osztályokat kínál. Ezek közös őse a `\fw\input\Validator` absztrakt osztály, mely három közös tulajdonságukat fogja össze: azt, hogy az adott input-mező jelenléte kötelező-e (`required(bool)`, ez alapértelmezésben hamis), illetve hogy szűrni kell-e a tartalmát, mielőtt tovább dolgozik vele az alkalmazás (`sanitize(bool)`, alapértelmezett értéke igaz), továbbá felületet biztosít a legutolsó ellenőrzési hibák lekérdezéséhez (`getLastErrors()`). Egy absztrakt műveletet tartalmaz, ez a `validate(mixed)`, mely egyetlen paramétere az ellenőrizni kívánt változó, referencia szerinti átadással. A referencia szerinti átadás azért szükséges, hogy a változó tartalmának módosítása (tipikusan a szűrés) érvényre jusson a művelet hatókörén kívül is. A metódust úgy kell felülírni, hogy logikai értékkel térjen vissza, mely az ellenőrzés sikerességét jelzi, illetve karban kell tartania a `$_lastErrors` védett adattagot (ki kell üríteni, és be kell jegyezni a sikertelenség okát). Az osztály konstruktora szintén védett, és a leszármazottak sem teszik publikussá, egy ellenőrző példány létrehozásához a statikus `build()` metódust kell használni az adott ellenőrző típuson. Ez azért került így kialakításra, mert konstruktorhívás után a PHP nyelvben nem lehet továbbláncolni a műveleteket:

```
// érvénytelen
$validator = new StringValidator()->reguired()->minLength(2);

// statikus build() metódussal
$validator = StringValidator::build()->reguired()->minLength(2);
```

A PHP nyelv statikus öröklődése és a késői statikus kötés (*„late static binding”*) lehetőség kihasználásával a `build()` metódust nem kell minden validációs osztályon

implementálni. A tulajdonságok metódusláncon keresztüli beállítása a nagyobb áttekinthetőség miatt került bevezetésre, ha konstruktor paramétereket használnánk erre a célra, akkor a paraméterek nevei a hívás helyén nincsenek jelen, és nem látszik azonnal, melyik milyen értéket kap. Az alábbi felsorolásban a keretrendszer ellenőrző osztályai és tulajdonságaik leírása található:

- `StringValidator`: karakterláncok ellenőrzése és szűrése szolgál, így csak karakterláncokat fogad el. érvényes értékként. A `minLength(int)` és `maxLength(int)` metódusok a vizsgált karakterlánc elvárt minimális és maximális hosszának beállítására használhatók. Minkét beállítás egész szám paramétert fogad. A maximum hossz nulla értékre állítása kikapcsolja ellenőrzését, ez az alapértelmezett, de érdemes beállítani a felhasználás helyén. A karakterláncból minden olyan elemet eltávolít, mely támadókód visszajuttatására lehet alkalmas (például HTML elemek, illetve átalakítja az alacsony ASCII-kódú karaktereket).
- `StringCompareValidator`: a vizsgált karakterláncot egy előre megadotthoz hasonlítja. A vizsgálat eredménye sikeres, ha megegyeznek. Az összehasonlítás alapjául szolgáló karakterláncot a `compareTo(string)` metódussal állíthatjuk be (alapértelmezett az üres karakterlánc). Ugyanazt a szűrést végzi el, mint a `StringValidator`. Kettős, megerősítéses adatbevitelnél (például jelszavak) használatos.
- `RegexpValidator`: a karakterlánc-ellenőrző specializált változata, mely reguláris kifejezést illeszt a bemenetre. Ha a kifejezés illeszkedik, az ellenőrzés sikeres. A reguláris kifejezést a `pattern(string)` metódussal állíthatjuk be, alapértelmezésben üres karakterlánc. Minden egyéb tulajdonságát örökli a `StringValidator` osztálytól.
- `EmailValidator`: ez is a karakterlánc-ellenőrző specializált változata, kimondottan e-mail címek formai ellenőrzésére és szűrésére. Az örökölteken túl saját beállításokat nem implementál.
- `EnumValidator`: azt ellenőrzi, hogy a vizsgált karakterláncot tartalmazza-e az előre megadott értékhalmaz. Az elfogadott értékek halmaza az `acceptValue(string)` művelettel bővíthető. Főleg rádiógomb, vagy választólista típusú űrlapelemektől érkezett bemenetre alkalmazható.

- `DateValidator`: a reguláris kifejezés alapján ellenőrző osztály speciális változata, mely csak megadott formátumú dátumokat fogad el. Az alapértelmezett elfogadott dátumformátum „*`ÉÉÉÉ. HH. NN.`*”, ahol `É` az év, `H` a hónap, `N` pedig a nap egy számjegye. A formátum megváltozatásához az `inputFormat(string)` és az örökölt `pattern(string)` összehangolt hívása szükséges. Előbbi a karakterlánc időbélyeggé alakításához szükséges, utóbbi pedig a formai ellenőrzéshez. Az ellenőrző akkor is hibát jelez, ha a dátum formátuma érvényes, de tartalma értelmetlen (például: „*2010. 14. 86.*”). A dátumot képes az alkalmazás számára a bemenetitől eltérő formátumra konvertálni, ezt a formátumot `outputFormat(string)` segítségével adhatjuk meg. A dátumot vizsgálhatjuk más, előre megadott dátumhoz képest is, hogy előtte, vagy utána van-e, a dátumok egyenlőséget megengedve vagy sem. Erre az alábbi műveleteket használhatjuk: `before(string)`, `after(string)`, `notAfter(string)`, `notBefore(string)`. A paraméterként várt dátum szintén az `inputString` által meghatározott formátumban kell, hogy legyen.

Létezik egy speciális bemenet-ellenőrző is, mely bemeneti változók egy halmazát képes egyszerre ellenőrizni: ez a `ValidatorGroup`. Létrehozása után az `addValidator Validator $validator, $key, $defaultValue = null)` szignatúrájú művelettel adhatunk hozzá ellenőrzőket. Használatát az alábbi példa szemlélteti:

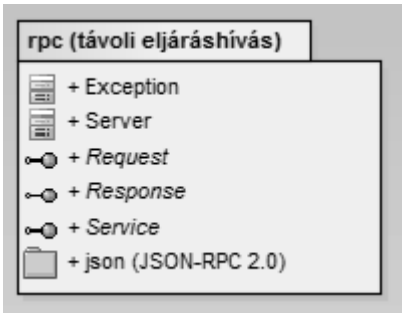
```
// paraméterek előkészítése
$parameters = $_POST;

// csoportos ellenőrző előkészítése
$validatorGroup = ValidatorGroup::build()
  ->addValidator(
    StringValidator::build()->reguired(true)->maxLength(50),
    'name'
  )
  ->addValidator(EmailValidator::build()->reguired(), 'email')
  ->addValidator(StringValidator::build(), 'description', '')
  ->addValidator(
    DateValidator::build()->required()->notBefore('1900. 01. 01.'),
    'birthday'
  );

// paraméterhalmaz csoportos ellenőrzése
// az eredmény igaz, vagy hamis érték
$result = $validatorGroup ->validate($parameters);
// ellenőrzési hibák lekérdezése
$errors = $validatorGroup ->getLastErrors();
```

Összefoglalva: a HTTP POST kérésen keresztül érkezett változókat ellenőrizzük és szűrjük. A `name` paraméter karakterlánc, kötelező megadni, és maximum 50 karakter

hosszú lehet. Az `email` paraméter csak érvényes e-mail címet tartalmazhat, és szintén kötelező megadni. A `description` paraméter egy nem kötelező karakterlánc, alapértelmezett értéke üres sztring (az `addValidator` harmadik paramétere). Az utolsó, `birthday` paraméterben egy kötelező dátumot vár, mely nem lehet „1900. 01. 01." dátum előtti. Az ellenőrzés után a `$parameters` tömbben már a szűrt, biztonságos értékek találhatók (azért nem használhatjuk a `$_POST` változót közvetlenül, mert az egy szuperglobális tömb a PHP nyelvben, és referenciaként nincs értelme átadni). Amennyiben az ellenőrzés sikertelen, a hibák egy kulcs-érték tárban (`\fw\KeyValueStorage`) állnak rendelkezésre. Ez a bemenet minden paraméternevéhez tartalmaz egy kulcsot, melyekben ismét egy-egy kulcs-érték tárat találunk. Ezeken a struktúrákon már a hiba típusa a kulcs, értéke pedig kiegészítő információ. A hiba típusa egy validációs osztály egy hiba-konstansa (például `StringValidator::ERROR_MAX_LENGTH`), értéke pedig további információ a hibáról (az előző példát követve ez egy egész szám, az átlépett maximális karakterszám).

3.1.8. TÁVOLI ELJÁRÁSHÍVÁS (`\FW\RPC`)

Távoli eljáráshívás segítségével elkészült alkalmazásainkat integrálhatjuk más, külső forrásból elérhető szolgáltatásokkal. Ennek támogatására célszerű akár több protokoll kezelését is implementálni. Mivel működésük alapvetően hasonló, érdemes egy absztrakt szerver osztályban megfogalmazni. Egy ilyen funkcionalitást nyújtó szerver tömören a következőkre képes:



17. ábra: A távoli eljáráshívás névtere és osztályai

- A kapott bemenet (kérés) dekódolása, ellenőrzése
- A kérésben meghatározott művelet végrehajtása a kapott paraméterekkel
- Az eredmény (válasz) összeállítása és elküldése

A `\fw\rpc\Server` absztrakt osztály az ilyen elven működő kiszolgálók építését teszi lehetővé. Az új osztály leszármaztatásával és a kapcsolódó interfészek (`\fw\rpc\Request és \fw\rpc\Response`) implementálásával bármilyen, hasonló elven működő protokoll támogatása implementálható (ilyen például az `XML-RPC`, `AMF`,

SOAP vagy a keretrendszerben meg is valósított *JSON-RPC*). A szerver konstruktor paraméterként egy konfigurációs objektumot vár (*\fw\config\Configuration*), melyet két célra is használ: egyrészt eltárolja (a *$_config* védett adattagba), hogy a meghívott szolgáltatás használni tudja az eredmény előállításához, illetve két értéket megkísérel lekérni saját osztálybetöltőjének létrehozásához. Ez a két beállítás az *rpc.services.classpath* és az *rpc.services.namespace*. Saját osztálybetöltőt azért hoz létre, mert a szerver által meghívott eljárás mindig egy osztály egy metódusa. A névtér megadása pedig önmagában is fontos, ha a kérésben szereplő metódus-hivatkozás nem tartalmazza (illetve biztosítható vele, hogy csak a megfelelő névtérben lévő osztályok metódusai legyenek hívhatók). A kiszolgálási hívásláncot ezután a *handle()* metódus meghívásával indíthatjuk, melynek visszatérési értéke már a küldésre kész válasz csomag. Egy tipikus felhasználás:

```
// kérés átvétele
$request = file_get_contents('php://input');

// feldolgozás, eljáráshívás
$server = new \fw\rpc\json\Server($configuration);
$response = $server->handle($request)

// válasz küldése
echo $response;
```

A példában a *\fw\rpc\json* névtérben implementált *JSON-RPC 2.0* **[9]** protokollt implementáló szervert példányosítottam, hiszen a *\fw\rpc\Server* osztály absztrakt. A fenti példakódot egy web-szerveren alkalmazva bemenet a klienstől érkezett HTTP kérés törzse, a kimenet pedig a HTTP válasz törzsében lesz.

A kérés interfész (*\fw\rpc\Request*) két műveletet határoz meg. A *decode($rawData)* arra szolgál, hogy a kérést az átvitel formátumáról átalakítsa. Visszatérési értéke nincs, a kérés példány adattagjait kell beállítania (mint ahogyan a *\fw\rpc\json\Request* osztályban látható). Értelmezési hiba esetén a metódusnak *\fw\rpc\Exception* kivételt kell dobnia a megfelelő kóddal. Ennek legegyszerűbb módja az alábbi:

```
use \fw\rpc\Exception;
…
throw Exception::create(Exception::PARSE_ERROR);
…
```

A másik művelet, melyet implementálni kell az *isValid()*, mely egy logikai érték visszaadásával jelzi, hogy a kérés mezőinek tartalma érvényes-e. Ez azért szükséges,

mert az adatfolyam dekódolásának sikere még nem jelenti azt, hogy a tartalmazott üzenet egy érvényes kérés az adott protokollban (például *JSON* sorosítással nem csak *JSON-RPC* kérés üzenetek kódolhatók).

A szerver a kérés feldolgozása során tehát először dekódolja azt, majd ellenőrzi érvényességét. Értelmezési hiba esetén az absztrakt *_handleParseError*() metódust hívja meg, érvénytelen kérést tapasztalva pedig a *_handleInvalidRequest()* műveletet. Ezeket úgy kell implementálni a leszármaztatott, konkrét protokollt kezelő szerverekben, hogy a válasz objektumon beállítsák a megfelelő hibát. A metódusok törzsében a válasz-objektumra a *$this->getResponse()* kifejezéssel hivatkozhatunk.

A miután a kérés-objektum rendelkezésre áll, dekódolt és érvényes, végrehajthatjuk az általa megfogalmazott eljáráshívást. Ezt az *_invokeService()* absztrakt metódus implementációjában kell megtennünk. A metódus törzsén belül a *$this->getRequest()* kifejezéssel érhetjük el a kérés-objektumot. Miután megállapítottuk a hivatkozott osztály és metódus nevét, végre kell hajtani a metódushívást, majd visszatérni annak eredményével. Érdemes a korábban bemutatott *\fw\Invoker* osztály segítségét igénybe venni a híváshoz. Az eljáráshívás végrehajtása előtt célszerű letesztelni, hogy a létrehozott példány implementálja a *\fw\rpc\Service* interfészt, és a *setConfiguration* meghívásával átadni neki a konfigurációt, így felhasználhatja azt az eredmény előállításához. Az *_invokeService()* kizárólag *\fw\rpc\Exception* típusú kivételt dobhat a következő kódok valamelyikével:

- *METHOD_NOT_FOUND*: a kérésben hivatkozott metódus nem található
- *INVALID_PARAMS*: a metódus nem hívható meg a megadott paraméterekkel
- *INTERNAL_ERROR*: bármilyen, egyéb hiba esetén

A kivétel létrehozásához ismét érdemes a *\fw\rpc\Exception::create($code)* statikus metódust használni (mint ahogy a kérés dekódolásánál). Ha a hívás közben a fenti kivételek egyike kiváltódott, meghívja a hozzá tartozó hibakezelő metódust, melyek sorrendben: a *_handleMethodNotFound()*, a *_handleInvalidParams()*, vagy a *_handleInternalError()*. Javasolt implementációjuk hasonló, mint az értelmezési hiba vagy érvénytelen kérés esetén. Amennyiben a hívás kivétel nélkül lefutott, a

szerver meghívja az `_applyInvokeResult($result)` absztrakt műveletet, melynek átadja az eljáráshívás eredményét. Ezt úgy kell felülírni, hogy a válasz-objektumon beállítsa a kapott eredményt (válasz-objektumra ismét a `$this->getResponse()` kifejezéssel hivatkozhatunk).

A válasz interfész (`\fw\rpc\Response`) mindössze egy metódust követel meg, ez az `encode()`. Arra szolgál, hogy a válasz objektum mezőinek adatát az adott protokollban érvényes válasszá transzformálja, majd visszaadja ezt az elkódolt reprezentációt (implementációjára példa a `\fw\rpc\json\Response` osztályban látható). A válasz osztályok felelőssége, hogy tárolni és kódolni tudják a nemcsak az eljáráshívás eredményét, de akár egy fellépett hibát is.

3.1.9. A TESZTEK FUTTATÁSA

Dolgozatomban a `PHPUnit` teszt-keretrendszert használom, mely szöveges kimenettel rendelkezik. A keretrendszer tesztjeit úgy tudjuk lefuttatni, hogy a főkönyvtárában állva (ahol a `bootstrap.php` is található, a mellékelt virtuális gépen ez a `/var/www/szakdolgozat/framework` könyvtár), kiadjuk a következő parancsot:

```
user@host:/var/www/szakdolgozat/framework$ phpunit --stderr
```

A parancs végén található `--stderr` kapcsoló a standard hibakimenetre irányítja a tesztek eredményjelzését. Erre azért van szükség, mert néhány teszt érzékeny arra, ha a futása közben írás történik standard kimeneten (ilyenek speciálisan a munkamenet-kezeléssel foglalkozó kódrészletek). A parancs végrehajtásakor egy PHP értelmező indul, mely betölti a `PHPUnit` kódját. Ez induláskor automatikusan beolvassa a `phpunit.xml` konfigurációs fájlt az aktuális könyvtárból. A konfigurációban megfogalmazottak szerint belép a `tests` könyvtárba, és minden „*Test.php*” végződésű fájlt teszt kódnak tekintve megpróbál lefuttatni.

Az egyes tesztek futási eredményét az alábbi karakterek egyikével jelzi:

- **.** (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. Ha minden teszt lefutott, egy összegző sorban kiírja a futási időt és a felhasznált memóriát. A kimenet az alábbihoz hasonló:

```
user@host:/var/www/szakdolgozat/framework$ phpunit --stderr
PHPUnit 3.5.12 by Sebastian Bergmann.
.................................................  63 / 390 ( 16%)
................................................. 126 / 390 ( 32%)
................................................. 189 / 390 ( 48%)
................................................. 252 / 390 ( 64%)
................................................. 315 / 390 ( 80%)
................................................. 378 / 390 ( 96%)
............
Time: 12 seconds, Memory: 14.75Mb
OK (390 tests, 545 assertions)
Generating code coverage report, this may take a moment.
```

3.1.10. 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*[6] 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*[7] több osztály vagy alrendszer összehangolt működését ellenőrzik.
- A *funkcionális tesztek*, vagy *elfogadási tesztek*[8] azt hivatottak biztosítani, hogy a program funkcionalitása az elvárásoknak megfelelő. (Ezek gyakran felhasználói felület tesztek, melyek automatizálása olykor meglehetősen körülményes. A bemutató alkalmazás fejlesztői dokumentációjában erre is található példa.)

3.1.11. A TESZTEK ANATÓMIÁJA

A `PHPUnit` keretrendszerben a tesztek a teszt-osztályok metódusai. A teszt-osztályok szülője mindig a `PHPUnit_Framework_TestCase` osztály, vagy ennek egy leszármazottja. Az elterjedt elnevezési konvenció szerint egy „`Osztaly`” nevű osztály

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

egységtesztjeit a „`OsztalyTest`” nevű osztály tartalmazza. A kifejlesztett keretrendszer szintén ezt követi.

Egy teszt-osztály futtatása során a következő történik (leegyszerűsítve): a futtató rendszer kigyűjti azon metódusokat, melyek neve „`test`” kezdetű. Minden ilyen metódus lefuttatása előtt megkíséreli meghívni a `setUp()`, utána pedig a `tearDown()` műveleteket. Ezeket nem kötelező implementálni. Arra szolgálnak, hogy a teszt metódusok megoszthassák egymással azt a kódot, amely a tesztekhez szükséges környezetet (angolul: *„fixture”*) létrehozza. Tekintsük például a `KeyValueStorage` osztály tesztjeit:

```
<?php
namespace fw\tests;

use \fw\KeyValueStorage;
use \PHPUnit_Framework_TestCase;

class KeyValueStorageTest extends PHPUnit_Framework_TestCase
{
  private $_storage;

  public function setUp()
  {
    $this->_storage = new KeyValueStorage();
  }

  public function tearDown()
  {
    unset($this->_storage);
  }

  public function testDoesNotHaveNonexistentKey()
  {
    …
  }

  public function testGetReturnsStorageForNonexistentKey()
  {
    …
  }

  …
}
```

Látható, hogy esetünkben a *„fixture*” egy darab kulcs-érték tárból áll, hiszen ez az osztály ennek a metódusait teszteli. Mivel a tesztek függetlenek egymástól, tetszőleges sorrendben végrehajthatók, és ha az egyik lefutása sikertelen, attól a többit még futtathatjuk tovább. Szemétgyűjtővel rendelkező nyelvek esetén (mint amilyen a PHP is) szokás a `tearDown()` műveletet elhagyni, amennyiben nem használunk külső

erőforrásokat (például fájlok megnyitása), és nem hozunk létre sok nehézsúlyú objektumot, melyek lassítják a tesztek futását **[2]**. Bizonyos esetekben nem hozunk létre minden teszthez friss környezetet (*fresh fixture*), hanem megosztjuk a tesztek között (*shared fixture*).

A tesztek általában négy fázisra oszthatók **[5]**:

1. A tesztelni kívánt objektum vagy objektumok előkészítése
2. A tesztelni kívánt művelet végrehajtása
3. Az eredmények kiértékelése
4. A teszt során lefoglalt erőforrások felszabadítása

Az 1-es és 4-es pontok sokszor kiszervezhetők a feljebb említett `setUp()` és `tearDown()` műveletekbe. A 3-as pont kivitelezéséhez, azaz a teszteredmény meghatározásához a teszt-keretrendszer értékvizsgáló függvényeit vesszük igénybe. Ezek az „assert” kezdetű névvel ellátott metódusok. Az alábbi táblázat tartalmazza a legegyszerűbbeket és egyben legfontosabbakat:

Név (paraméterek):	**Mit vizsgál:**
assertTrue(kifejezés)	kifejezés paraméter értéke logikai igaz
assertFalse(kifejezés)	kifejezés paraméter értéke logikai hamis
assertEquals(elvártÉrték, aktuálisÉrték)	elvártÉrték és aktuálisÉrték megegyezik
assertNotEquals(elvártÉrték, aktuálisÉrték)	az előző ellentéte

Ha az általuk vizsgált feltétel nem teljesül, a tesztet sikertelennek ítélik meg. Ekkor az értékvizsgáló művelet típusától függő hibaüzenetet kapunk a futtatórendszertől. A teszt metódusokban általában érdemes egyetlen értékvizsgálatot alkalmazni, mert a hiba helye hamarabb lokalizálható ezzel a módszerrel **[5]**. Ha mégis több értékvizsgálat használatára kényszerülünk, és ezek azonos típusúak (például két-három `assertTrue` hívás egymás után), különbséget tehetünk köztük, ha egyedi üzenettel látjuk el őket. Ez könnyen kivitelezhető, mert minden értékvizsgáló függvény utolsó, opcionális paramétere egy ilyen üzenet.

Az összes támogatott értékvizsgáló függvény szignatúrája megtalálható a `PHPUnit` felhasználói dokumentációjában. Bonyolultabb esetekre és a tesztek kódjának tisztán

tartására saját értékvizsgáló műveleteket is létrehozhatunk, mint ahogy a `ClassLoaderTest` osztály `assertClassLoaded` metódusában is látható.

A `PHPUnit` működése szabályozható különleges megjegyzés blokk elhelyezésével is. A következő kódrészletben a megjegyzésben szereplő „@” karakterekkel kezdődő részeket annotációknak nevezzük:

```
/**
 * @annotáció1 paraméter
 * @annotáció2 paraméter
 */
public function test…()
```

Van egy speciális eset. amikor a teszt sikeressége nem dönthető el értékvizsgálattal: ha az elvárt működés az, hogy a vizsgált művelet kivételt dob. Az erre kifejlesztett „`@expectedException <KIVÉTEL-OSZTÁLY-NEVE>`” formájú annotációval azonban ez is kivitelzhető. A keretrenszer `InvokerTest` osztálya többnyire ilyen teszteket tartalmaz. Néhány különleges osztály esetében kötött sorrendű teszteket kell végeznünk. Ez a „`@depends <TESZT-METÓDUS-NEVE>`” annotációval érhető el. A keretrendszer ilyen teszteket használ a `SessionManager` tesztelésére.

Ha egy tesztet több kezdeti értékre is le szeretnénk futtatni, kézenfekvő adatszolgáltató („*dataProvider*”) metódusok létrehozása. Tekintsük az alábbi példát (`StringValidatorTest` osztályból, kissé átalakítva):

```
/**
 * @dataProvider stringsDataProvider
 */
public function testAcceptsStringsOnly($isValid, $value)
{
   // „fixture” elkészítése
   $validator = StringValidator::build();
   // tesztelni kívánt művelet elvégzése
   $result = $validator->validate($value);
   // értékvizsgálat
   $this->assertEquals($isValid, $result);
}

public function stringsDataProvider()
{
   return array(
      array(true, 'test'),
      array(true, '42'),
      array(true, ''),
      array(false, 0),
      array(false, 42.0),
      array(false, null)
   );
}
```

A teszteket futtató környezet felismeri a teszt-metódus felett elhelyezett `@dataProvider` annotációt, és az ott megadott függvény visszatérési értékét úgy használja fel, hogy minden egyes elemét egy új futásnak felelteti meg. A példában látható adaszolgáltató függvény egy 6 elemű tömböt ad vissza, így hat alkalommal lesz lefuttatva a `testAcceptsStringsOnly` metódus. Paraméterei mindig az adott indexen lévő tömb elemei, tehát először (true, 'test'), majd a (true, '42')… paraméterekkel kerül meghívásra (egy 6×2 méretű mátrixnak is felfogható, ahol a sorok száma a futások száma, az oszlopoké pedig a paramétereké, és minden sorban egy futtatás paraméterei helyezkednek el).

Egyes metódusok csak úgy tesztelhetők le teljesen, ha olyan futási ágra kényszerítjük a kódot, melyre egyébként csak speciális esetben jut be. Ilyenek például a hibakezelést végző részek. Ezek könnyebb teszteléséhez ideiglenesen megváltoztathatjuk az osztályok működését anélkül, hogy külön a tesztekhez specializált alosztályokat kellene létrehoznunk. A működés megváltoztatása két célt szolgálhat: *(a)* függvény visszatérési értékének felülbírálása vagy kivétel dobása, illetve *(b)* megfigyelési pont létesítése a függvény hívásának helyén.

A két estre együttes példát a `ValidatorGroupTest` osztályban találunk:

```
private function _getValidatorErrorStub(array $lastErrors = array())
{
   $validatorMock = $this->getMock(
      '\\fw\\input\\Validator',
      array('validate', 'getLastErrors'),
      …
   );
   $validatorMock->expects($this->once())
                 ->method('validate')
                 ->will($this->returnValue(false));
   $validatorMock->expects($this->once())
                 ->method('getLastErrors')
                 ->will($this->returnValue($lastErrors));
   return $validatorMock;
}
```

A kódrészleten látható függvény olyan bemenet-ellenőrző példányt készít a `\fw\input\Validator` osztályból (`getMock` hívással), melyen a `validate` művelet hamisat, a `getLastErrors` pedig a függvény paramétereként kapott tömböt adja vissza bármilyen paraméterrel történő hívásra. A hívásoknak nem csak a visszatérési értéke van meghamisítva a teszt ideje alatt, de hívásaik száma is megfigyelés alatt áll: a

`$validatorMock->expects($this->once())` hatására a `PHPUnit` értékvizsgálatot helyez el a függvény elejére, és ha nem (ebben az esetben) pontosan egyszer kerül meghívásra, a teszt elbukik. A `once()` hívás helyén még az alábbi megkötések tehetők a függvényre:

- `any()`: akárhányszor lefuthat
- `never()`: nem hívhatják meg egyszer sem
- `atLeastOnce()`: legalább egyszer meg kell, hogy hívják
- `exactly(count)`: pontosan `count` alkalommal meg kell, hogy hívják
- `at(index)`: a pontosan `index`-edik alkalommal történő hívás esetére határozhatunk meg visszatérési értéket vagy paramétervizsgálatokat.

Érdemes felhívni rá a figyelmet, hogy a bemutatott kódrészletben a `getMock()` hívás látszólag egy absztrakt osztályt példányosít *(*`\fw\input\Validator`*)*. Ez természetesen nem lehetséges. A háttérben az történik, hogy a `PHPUnit` futás közben legenerálja egy belőle származó osztály kódját, melyen a műveletek a kívánt módon vannak felülírva. Ezt a kódot átadja az értelmezőnek az `eval()` beépített függvény segítségével, majd ennek az osztálynak egy példányával tér vissza. Az ilyen módon keletkezett, megváltoztatott viselkedésű példányokat az angol terminológia „`stub`" vagy „`mock object`" névvel illeti. Valójában absztrakt osztályokhoz külön változata létezik a függvénynek (`getMockForAbstractClass`), mely csak a megadott osztályban található absztrakt műveletek helyettesítésére van felkészítve. Ez a bemutatott példa helyén nem volt elegendő, mert az absztrakt osztályban konkrét megvalósítással bíró `getLastErrors` felülírása is szükséges volt a tesztek elvégzéséhez.

Fontos kiemelni, hogy az ilyen helyettesítő objektumok alkalmazásához laza csatolásnak kell lennie az osztályok között, hogy a tesztek elvégzéséhez cserélhetők legyenek a komponensek (tehát a függőségeiket nem példányosítják, hanem megkapják az objektumok). Jó példát szolgáltat erre a `\fw\control\FrontController` osztály `getRouter` és `setRouter` művelete. Ezek segítségével a fő vezérlőben cserélhető az útválasztó példány, így a teszteknél helyettesítő objektumokkal kiváltható az alapértelmezett. Az ilyen típusú laza csatolást

a leggyakrabban függőség-injektálásnak (*dependency injection*) nevezik **[6]**. Három fő típusa különböztethető meg, a keretrendszer mindegyiket alkalmazza:

1. *Konstruktor-injektálás:* a konfigurációs objektum továbbadása így történik a legtöbb esetben. Az adott függőséget egy konstruktor paraméterben juttatjuk be az objektumba. (Gyakori, teszteléshez nem megfelelő gyakorlat, hogy a konfigurációs osztályt statikus adattagokkal érik el. Ez azért helytelen, mert a statikus hivatkozások nem helyettesíthetők a tesztek alatt.)
2. *Setter-injektálás:* a függőséget egy beállító-függvényen keresztül juttatjuk be az objektumba. Ilyen például a `\fw\control\FrontController setRouter` és `setContext`, valamint a `\fw\rpc\Server setRequest` és `setResponse` művelete.
3. *Interfész-injektálás: lényegében* a setter-injektálás egy speciális esete. A függőségeket beállító függvények szignatúráját egy interfészbe szervezzük. Ilyen célt szolgál a `\fw\rpc\Service` interfész, mely a távoli eljáráshívás szolgáltatások számára biztosít hozzáférést a konfigurációhoz.

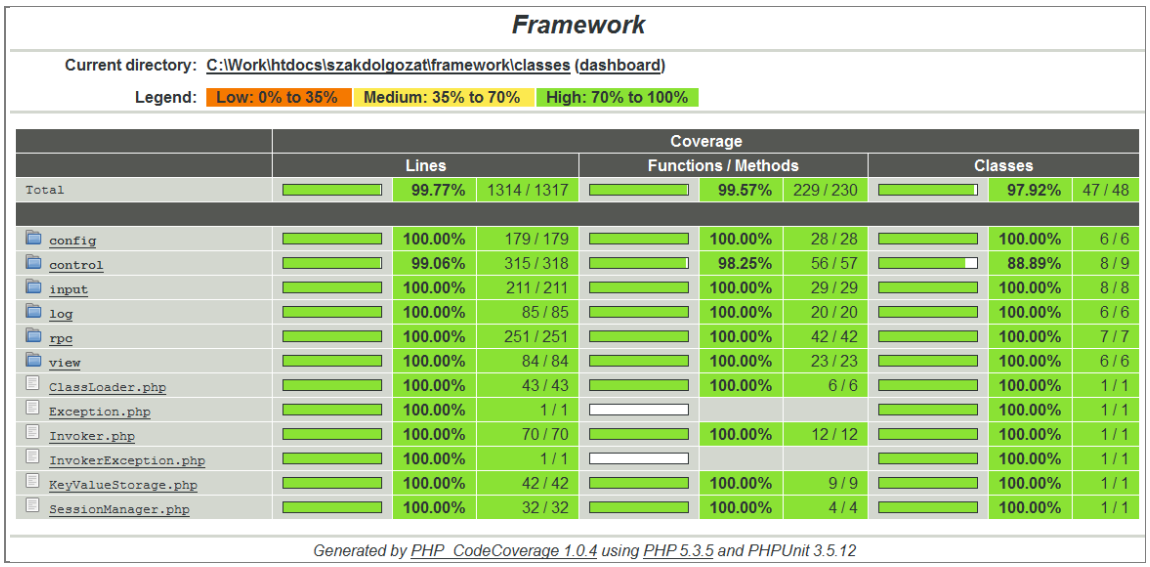
Az objektumok függőségeinek tesztek alatti cseréjére nem csak az kényszeríthet minket, hogy egyébként elérhetetlen kódrészeket is lefuttassunk. Van néhány web-alkalmazásokkal kapcsolatos probléma, mely kimondottan nem tesztelhető egy parancssorból futtatott teszt-környezetben. Ilyen például a HTTP fejlécek beállítása. Ezekben az esetekben az adott műveleteket kiszervezik külön osztályokba, melyeket más úton letesztelnek. A tesztek idejére pedig helyettesítik őket olyan implementációkkal, melyek valójában nem hajtják végre az adott műveletet.

Az olyan speciális teszteket, amelyek adatbázisok és felhasználói felületek teszteléséhez szükségesek, a bemutató alkalmazás fejlesztői dokumentációjának végén ismetetem, hiszen a keretrendszer nem tartalmaz olyan elemet, melyhez szükség lenne ezekre a teszt-típusokra.

3.1.12. KÓDLEFEDETTSÉG

A kódlefedettségi jelentés (*code coverage report*) kiváló kiegészítője az automatizált tesztelésnek. Segítségével ellenőrizhető, hogy minden tesztelni kívánt végrehajtási ág valóban lefut, és a megfelelő tesztek a megfelelő ágakat járják be. A `PHPUnit` képes

ilyen jelentést generálni. Ezt a keretrendszer főkönyvtárán belül található `coverage` könyvtár tartalmazza HTML formátumban (az `index.html` fájlt kell megnyitni).



Framework

Current directory: C:\Work\htdocs\szakdolgozat\framework\classes (dashboard)

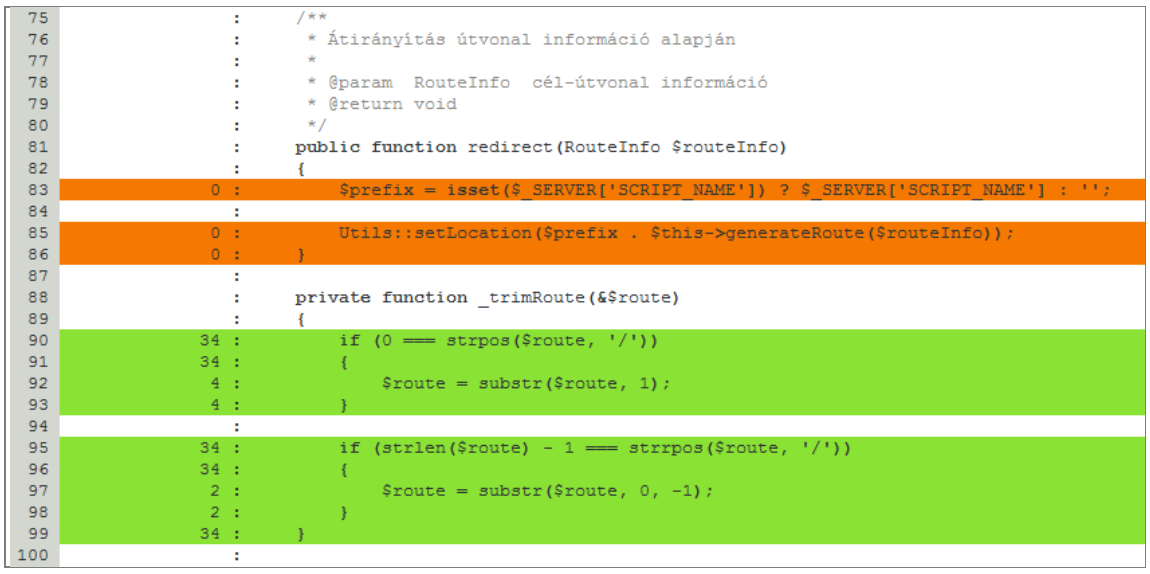
Legend: Low: 0% to 35% | Medium: 35% to 70% | High: 70% to 100%

	Coverage								
	Lines			Functions / Methods			Classes		
Total		99.77%	1314 / 1317		99.57%	229 / 230		97.92%	47 / 48
config		100.00%	179 / 179		100.00%	28 / 28		100.00%	6 / 6
control		99.06%	315 / 318		98.25%	56 / 57		88.89%	8 / 9
input		100.00%	211 / 211		100.00%	29 / 29		100.00%	8 / 8
log		100.00%	85 / 85		100.00%	20 / 20		100.00%	6 / 6
rpc		100.00%	251 / 251		100.00%	42 / 42		100.00%	7 / 7
view		100.00%	84 / 84		100.00%	23 / 23		100.00%	6 / 6
ClassLoader.php		100.00%	43 / 43		100.00%	6 / 6		100.00%	1 / 1
Exception.php		100.00%	1 / 1					100.00%	1 / 1
Invoker.php		100.00%	70 / 70		100.00%	12 / 12		100.00%	1 / 1
InvokerException.php		100.00%	1 / 1					100.00%	1 / 1
KeyValueStorage.php		100.00%	42 / 42		100.00%	9 / 9		100.00%	1 / 1
SessionManager.php		100.00%	32 / 32		100.00%	4 / 4		100.00%	1 / 1

Generated by PHP_CodeCoverage 1.0.4 using PHP 5.3.5 and PHPUnit 3.5.12

18. ábra: A keretrendszerhez készült kódlefedettségi jelentés kezdőlapja

A jelentés tartalmazza, hogy mely osztályok mely metódusai, azoknak mely sorai érintettek a tesztek által. A piros háttérszínű sorok nem futottak a tesztek során (tehát nem lefedettek a tesztek által).



```
 75                :     /**
 76                :      * Átirányítás útvonal információ alapján
 77                :      *
 78                :      * @param  RouteInfo  cél-útvonal információ
 79                :      * @return void
 80                :      */
 81                :     public function redirect(RouteInfo $routeInfo)
 82                :     {
 83              0 :         $prefix = isset($_SERVER['SCRIPT_NAME']) ? $_SERVER['SCRIPT_NAME'] : '';
 84                :
 85              0 :         Utils::setLocation($prefix . $this->generateRoute($routeInfo));
 86              0 :     }
 87                :
 88                :     private function _trimRoute(&$route)
 89                :     {
 90             34 :         if (0 === strpos($route, '/'))
 91             34 :         {
 92              4 :             $route = substr($route, 1);
 93              4 :         }
 94                :
 95             34 :         if (strlen($route) - 1 === strrpos($route, '/'))
 96             34 :         {
 97              2 :             $route = substr($route, 0, -1);
 98              2 :         }
 99             34 :     }
100                :
```

19. ábra: A tesztek által lefedett és lefedetlen kódsorok a jelentésben

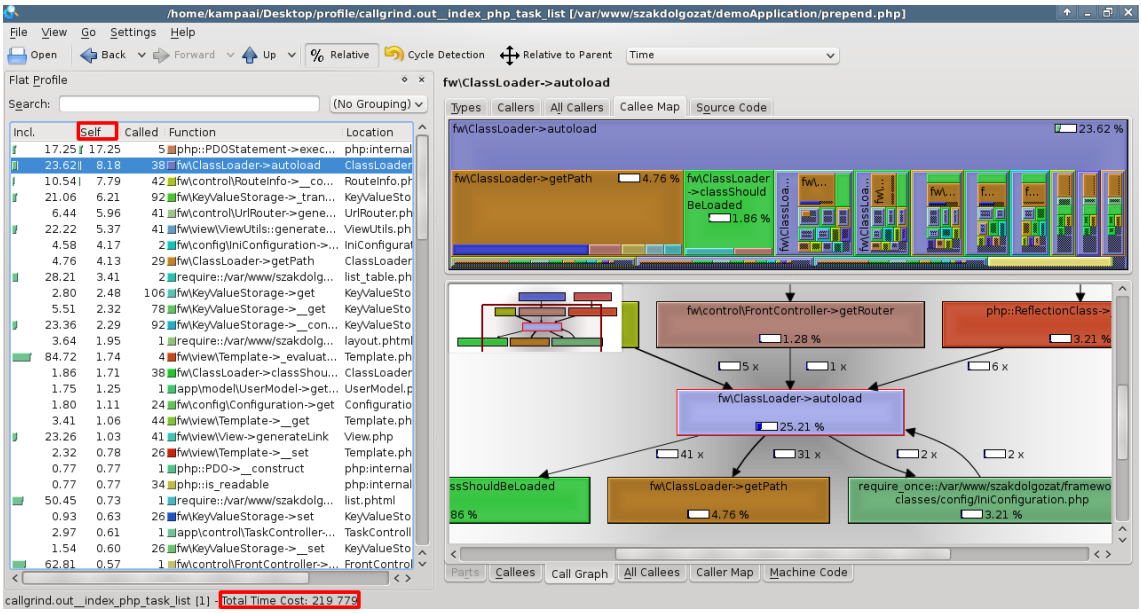
3.1.13. Kiterjesztési lehetőségek

A keretrendszer működése számos ponton átalakítható, kiterjeszthető a rá épülő alkalmazások által. Ezek közül a legfontosabbak:

- Új naplózási célok implementálása a `\fw\log\LogTarget` örökítésével. A rendszer jelenleg csak a standard kimenetre vagy egy megadott fájlba tud naplózni. Egy elképzelhető kiterjesztés például, hogy megvalósítjuk a közvetlenül az operációs rendszer vagy a webszerver rendszernaplójába történő naplózást. Így az alkalmazások esetleges riasztásai hamarabb elérhetik a rendszergazdákat.
- Abban az esetben, ha nem egy klasszikus web-alkalmazást készítünk a keretrendszerrel, hanem például egy összetett SMS-átjárót *(SMS-gateway)*, egészen más útválasztásra lehet szükségünk, mint amit a `\fw\control\UrlRouter` garantál (a felhozott példában elképzelhető egy, az SMS üzenetének prefixe alapján megvalósított útválasztás). Ilyen esetben a vezérlési-környezet objektumok sem megfelelők az alapértelmezett `\fw\control\HttpContext` által szolgáltatott formában, ez szintén cserélhető részét képezi a keretrendszernek.
- Nem minden alkalmazás számára megfelelő az alapértelmezett nézetként beépített sablonkezelő rendszer. A `\fw\view\View` absztrakt osztályból származó új nézetosztályokkal ez bármikor kicserélhető. (lásd: bemutató alkalmazás fejlesztői dokumentációja).
- Lehetőségünk van saját bemenet-ellenőrző osztályok létrehozására a `\fw\input\Validator` leszármaztatásával, ha a keretrendszerhez kapott készlet nem elegendő űrlapjaink átvizsgálásához.
- Amennyiben nem elégendő az *AJAX*-kommunikációhoz használható `JSON-RPC 2.0` távoli eljáráshívás implementáció, kiegészíthetjük további protokollok támogatásával (például `AMF`, `SOAP`, `XML-RPC`, `WDDX`). Ehhez a `\fw\rpc` csomag `Request` és `Response` interfészeit kell megvalósítanunk, valamint származtatnunk a `Server` absztrakt osztályból.

3.1.14. TELJESÍTMÉNY, OPTIMALIZÁCIÓ

A kódlefedettségi jelentések adatait gyűjtő `Xdebug` kiegészítés a kód profilozásához szükséges adatokkal is szolgál. A mellékelt virtuális gépen ez úgy van beállítva, hogy a `/var/www/szakdolgozat/demoApplication/profile` mappába írja az adatokat. Ez a könyvtár az asztalon található *„profile"* szimbolikus linkkel is elérhető. Az itt található „`callgrind.out.*`" fájlok a `KCachegrind` adatelemző szoftverrel nyithatók meg, mely szintén telepítve van a gépen.



20. ábra: Teljesítmény elemzése KCachegrind segítségével

A képen két fontos rész van bekeretezve: a baloldalon alul található „*Total Time Cost*” mellett álló szám (ez esetben 219779) a futáshoz szükséges időt tartalmazza mikro-szekundumokban, ezt kell minimalizálni a szoftver nagyobb futási sebességéhez. Természetesen ez a szám bizonyos hibával értendő, hiszen maga a mérés befolyásolja ezt az eredményt. A másik fontos adat a függvények saját futási ideje („*Self*”, bal oldalt fent a táblázat oszlopfejléce). Ez alapján csökkenő sorrendbe rendezve a táblázatot a legtetején található függvények optimalizálását kell elvégeznünk, mert ezek futottak a legtovább.

A keretrendszer esetében jellemzően az osztálybetöltés (sokszori futása miatt), illetve a kulcs-érték tárak (`\fw\KeyValueStorage`) transzformációja időigényes, így ezeket érdemes optimalizálni. Mindkét problémára az jelent megoldást, ha gyorsító-tárakat építünk hozzájuk. Ezek a tárak lényegében generált kódfájlok, melyek speciális struktúrákat feltöltő programrészeket tartalmaznak. Az előtöltő esetében csupán egy asszociatív tömbre van szükségünk, melyben minden osztály teljesen minősített nevéhez hozzá van rendelve az azt tartalmazó kódfájl abszolút útvonala a fájlrendszerben. A konfigurációnál szintén hasonló módszert lehet alkalmazni: INI vagy XML fájlok beolvasása helyett olyan PHP kódot generálunk, mely közvetlenül egymásba ágyazott kulcs-érték tárakat hoz létre, és ezt a kódfájlt töltjük be.

3.2. Bemutató alkalmazás

3.2.1. KÖNYVTÁRSZERKEZET

A bemutató alkalmazás főkönyvtárának szerkezete (a mellékelt virtuális gépen ez a `/var/www/szakdolgozat/demoApplication` könyvtár, és a lista nem teljes):

```
demoApplication/ ..... A bemutató alkalmazás könyvtára
  assets/ ..... Az alkalmazás kiegészítő elemei
    css/ ..... Stíluslapok
    images/ ..... Képek
    js/ ..... JavaScript fájlok
  classes/ ..... Az alkalmazás fő névterének helye
    control/ ..... Vezérlő osztályok
      hooks/ ..... Vezérlő-kiegészítők
    model/ ..... Modell osztályok
    services/ ..... Szolgáltatás osztályok
    view/ ..... Nézet osztályok
  config/ ..... Alkalmazáskonfiguráció könyvtára
    application.ini ..... Alkalmazáskonfigurációs fájl
  coverage/ ..... Kódlefedettség-jelentések helye
  logs/ ..... Alkalmazásnaplók
  testdox/ ..... Futtatott tesztek listája
  tests/ ..... Tesztkódok helye
  view/ ..... Nézet-sablonok főkönyvtára
    error/ ..... Hibakezelő vezérlő sablonjai
    task/ ..... Feladat vezérlő sablonjai
    user/ ..... Felhasználó vezérlő sablonjai
    layout.phtml ..... Fő elrendezési sablon
  append.php ..... PHPUnit kódlefedettség segéd-szkript
  bootstrap.php ..... Keretrendszer és osztálybetöltő inicializálása
  create_database.sql ..... Adatbázis szkript
  create test_database.sql ..... Adatbázis szkript
  index.php ..... Az alkalmazás belépési pontja
  insert data.sql ..... Adatbázis szkript
  phpunit.xml ..... PHPUnit konfigurációs fájl (tesztekhez)
  phpunit coverage.php ..... PHPUnit kódlefedettség segéd-szkript
  prepend.php ..... PHPUnit kódlefedettség segéd-szkript
```

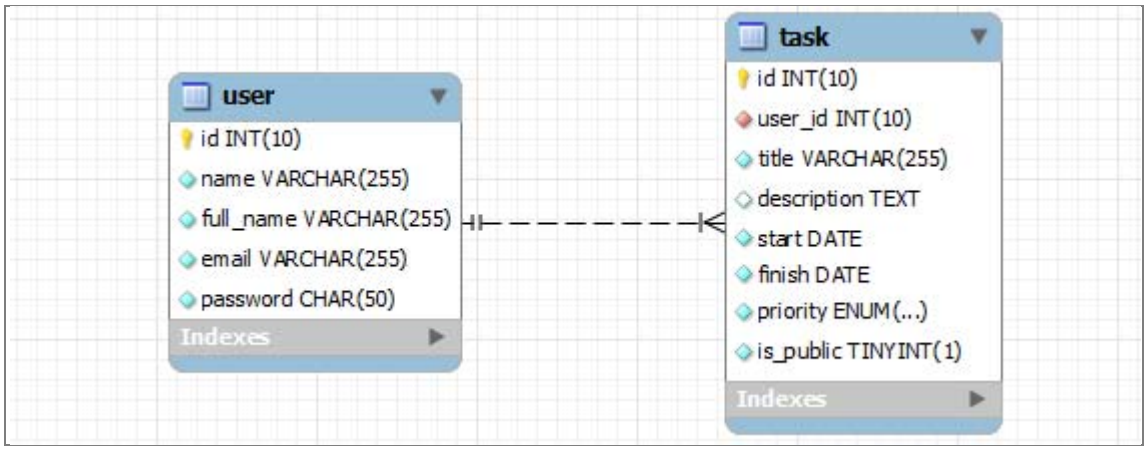
3.2.2. BELÉPÉSI PONT, INDÍTÁSI SZEKVENCIA

Az alkalmazás belépési pontja az `index.php` fájl eleje. Először az alkalmazás többnyire útvonalakat tartalmazó globális konstansait és osztálybetöltőjét állítjuk be a `bootstrap.php` behívásával. A látható, hogy az alkalmazás osztálybetöltőjének beállítása előtt befűzi a keretrendszer `bootstrap.php` szkriptjét is. Erre azért van szükség, hogy a keretrendszer osztályai (és így a `\fw\ClassLoader`) elérhető legyen.

Ezután az `index.php`-ban létrehozunk egy konfigurációs objektumot az `application.ini` fájlból. A fővezérlő indítása (`dispatch`) előtt beállítjuk rajta a környezetet (`Context`) egy saját példányra, melyben a nézetet előzőleg kicseréltük.

3.2.3. ADATMODELL

Az alkalmazás kétféle entitással dolgozik: felhasználó („*user*”) és feladat („*task*”). Ezeket `MySQL` adatbázisban tároljuk. A táblák szerkezete és kapcsolata:



21. ábra: A bemutató alkalmazás adattáblái

A felhasználó tábla mezőnevei és típusai, fentről lefelé:

- Azonosító (egész szám, elsődleges kulcs)
- Név (karakterlánc, maximális hossz 255, egyedi kulcs)
- Teljes név (karakterlánc, maximális hossz 255)
- E-mail cím (karakterlánc, maximális hossz 255, egyedi kulcs)
- Jelszó (karakterlánc, hossz 50)

A feladat tábla mezőnevei és típusai, fentről lefelé:

- Azonosító (egész szám, elsődleges kulcs)
- Felhasználó azonosító (egész szám, idegen kulcs)
- Cím (karakterlánc, maximális hossz 255)
- Leírás (szöveg)
- Kezdés (dátum)
- Befejezés (dátum)
- Prioritás (alacsony / normál / magas)

- Publikus? (logikai)

Az adattáblák kezelését az `\app\model\UserModel` és `\app\model\TaskModel` osztályok végzik. A két modell osztály közös őse az `\app\model\DatabaseModel`, mely az adatbázis-csatlakozási funkcionalitást egyesítette. A táblák egy-egy rekordját a `\app\model\User` és `\app\model\Task` osztályok reprezentálják. A felhasználók modelljének műveletei:

- Egy felhasználó lekérdezése azonosító (`getUserById`), név (`getUserByName`), vagy e-mail cím alapján (`getUserByEmail`)
- Felhasználó hozzáadása (`addUser`)
- Jelszó hash generálás és ellenőrzés (`generateSalt`, `hashPassword`, `validatePassword`)

A jelszó hash-ek képzése úgy történik, hogy a megadott jelszó mellé generálunk egy 10 számjegyű hexadecimális számot. Ezt beírjuk a jelszó mögé, majd SHA1 algoritmussal elkódoljuk. A kész 40 karakteres kódot a közepénél megszakítva befűzzük a 10 karakteres generált részt. A kész kódban nem látszanak a vágás határvonalai, mert mindkét karakterlánc csak hexadecimális számjegyeket tartalmazott. A jelszót úgy ellenőrizzük, hogy az eredeti kódjából kivágjuk a 20. és 30. karakter közti 10 karakter hosszú részt, és ezt használva egy véletlenszerű karaktersorozat helyett, a beírt jelszót is elkódoljuk. Ha a két kód megegyezik, a jelszavak is megegyeznek.

A feladatok modelljének műveletei:

- Egy feladat lekérdezése azonosító alapján (`getTaskById`)
- Egy felhasználó feladatainak megszámlálása (`countTasksByUserId`), és lekérdezése (`getTasksByUserId`)
- Publikus feladatok lekérdezése, megszámlálása egy felhasználó kihagyásával (`getOtherUsersPublicTasks`, `countOtherUsersPublicTasks`)
- Feladat hozzáadása (`addTask`)
- Feladat módosítása (`updateTask`)
- Feladat törlése azonosító alapján (`deleteTaskById`)

Az alkalmazás konfigurációjában két adatbázis-kapcsolat definiált. Ezek közül az egyik a produkciós módban történő üzemeléshez, a másik az automatizált tesztekhez szükséges.

3.2.4. VEZÉRLÉS

Az alkalmazás négy vezérlő osztályt implementál az `\app\control` névtérben. Az `ErrorController` a hibák (kivételek, hibás címek) kezeléséért felel. Az `RpcController` egy `JSON-RPC 2.0` protokollt támogató távoli eljáráshívás átjárót hoz létre. Rajta keresztül érhető el az `\app\services\TaskService`, amely feladat entitások lekérdezésére használható. A `UserController` felhasználókkal, a `TaskController` pedig a feladatokkal kapcsolatos tevékenységeket foglalja magában. A `TaskController` egyben az alapértelmezett vezérlő. A felhasználói dokumentáció nem említi, de az `ErrorController` és a `UserController` naplózást végez az alkalmazás `logs` könyvtárába: a hibákat, illetve sikeres be- és kijelentkezéseket, regisztrációkat (csak nyomkövetés módban) rögzítik.

Két vezérlés-kiegészítő is implementálásra került az `\app\control\hooks` névtérben: az egyik az `AuthenticationHook`, a keretrendszer részét képező, ugyanilyen nevű absztrakt osztályának egy konkrét megvalósítása (a „`user`" munkamenet változót vizsgálja, illetve azt, hogy az adatbázisban is szerepel-e a hozzá tartozó felhasználó). A másik kiegészítő a `LoggedInLayoutHook`, ami elhelyezi az elrendezés sablon fejlécében a felhasználó nevét és a kijelentkezés linket. Mindkét kiegészítő elő-futtatott (*pre-hook*) módban van beállítva.

3.2.5. NÉZET

Mivel a keretrendszerben nincs támogatás a bemenet-ellenőrzők hibakódjainak üzenetekké alakítására, ezért az alkalmazásban kellett módot találni rá. A legegyszerűbben úgy lehetett elérhetővé tenni egy ilyen függvényt, ha közvetlenül a sablonokban hivatkozhatunk rájuk. Ezért lett létrehozva a `CustomTemplateView` osztály, mely a `\fw\view\TemplateView` leszármazottja, és a `getValidatorMessage`, illetve a `getValidationResultRow` metódusokkal lett bővítve a könnyebb használat érdekében. A nézet cseréjét az `index.php` fájlban láthatjuk: a fővezérlő beindítása előtt kicseréljük a vezérlési környezetet egy olyan példányra, amelyen ki van cserélve a

nézet is. A sablonok főkönyvtára az alkalmazás `view` mappája. Az alapértelmezett elrendezés sablon az ebben található `layout.phtml`.

3.2.6. A TESZTEK FUTTATÁSA

A tesztek futtatása előtt az alkalmazást fejlesztői üzemmódba kell kapcsolni. Ez a lépés azért szükséges, mert az adatbázis tesztek egy másik, (de azonos szerkezettel rendelkező) adatbázishoz csatlakoznak, mint amivel az alkalmazás produkciós módban fut. Ez azt jelenti, hogy az aktív konfigurációs szekciót („*production*”) fejlesztőire kell váltani („*development*”). Ezt az alkalmazás főkönyvtárában található `bootstrap.php` fájl szerkesztésével tehetjük meg, a 11. sor módosításával. Az eredeti kódsor:

```
define('ACTIVE_CONFIGURATION', 'production');
```

A módosítás elvégzése után:

```
define('ACTIVE_CONFIGURATION', 'development');
```

A tesztek futtatását ismét egy terminálból kiadott paranccsal indíthatjuk, a bemutató alkalmazás főkönyvtárában állva:

```
user@host:/var/www/szakdolgozat/demoApplication$ phpunit
```

A keretrendszernél alkalmazott `--stderr` kapcsolóra ez esetben nincs szükség. A parancs kiadása után megkezdődik a felhasználói felületet és az adatmodell tesztjeinek futtatása. Mivel a felhasználói felület teszteléséhez ténylegesen webböngésző-példányokat indít a rendszer, és intenzíven használják az adatbázist is, ezek a tesztek lassabban futnak a keretrendszernél megszokottnál.

3.3.7. SPECIÁLIS TESZTEK: MODELLEK ÉS ADATBÁZISOK

Ahogyan az előző fejezetben láthattuk, a bemutató alkalmazás két adatbázis-egyedet kezel. Mindegyikhez tartozik egy-egy modell osztály, mely összefogja lekérdező és módosító műveleteiket. A következőkben e két osztály (`\app\model\TaskModel` és `\app\model\UserModel`) egységtesztelését mutatom be.

Mivel mindkét osztály kapcsolódik adatbázishoz, ezt a közös logikát kiemeltem egy közös ősbe, mely az `\app\model\DatabaseModel` néven elérhető. Az adatbázis-csatlakozás egy hétköznapi egységteszttel próbára tehető, ezért ezt itt nem részletezem. A másik két osztály tesztelése azért nehézkes, mert minden teszt előtt megfelelő állapotban (azaz megfelelő adatokkal feltöltve) kell lennie az adatbázisnak. A

`PHPUnit` ennek segítésére tartalmaz egy speciális teszt-ősosztályt: ez a `PHPUnit_Extensions_Database_TestCase`. Minden teszt osztálynak két műveletet kötelező felülírni rajta: `getConnection()` és `getDataset()`. Az első azért felel, hogy az adatbázis-kapcsolat rendelkezésre álljon számára, a második segítségével pedig meghatározhatjuk azt az adathalmazt, melyet a tesztek előtt az adatbázisba tölt. A `getConnection()` javasolt implementációja**[2]**:

```
public function getConnection()
{
   return $this->createDefaultDBConnection($pdo, $databaseName);
}
```

A paramétereknél `$pdo` egy érvényes PHP-PDO adatbázis kapcsolat objektum, `$databaseName` a kezelni kívánt adatbázis neve. A modelleket tesztelő osztályok a metódus elején létrehoznak egy modell példányt, és a `$pdo` változó helyett saját megnyitott kapcsolatukat, `$databaseName` helyett pedig a konfigurációból kapott adatbázisnevet adják át. A `getDataset()` metódust úgy kell implementálni, hogy visszatérési értéke `PHPUnit_Extensions_Database_DataSet_IDataSet` típusú legyen. A legegyszerűbb az, ha XML fájlból töltjük be a kiindulási adathalmazt. A modell osztályok tesztelése szintén így történik. Ekkor a metódus törzse egyetlen sorból áll:

```
return $this->createFlatXmlDataSet($xmlFilePath);
```

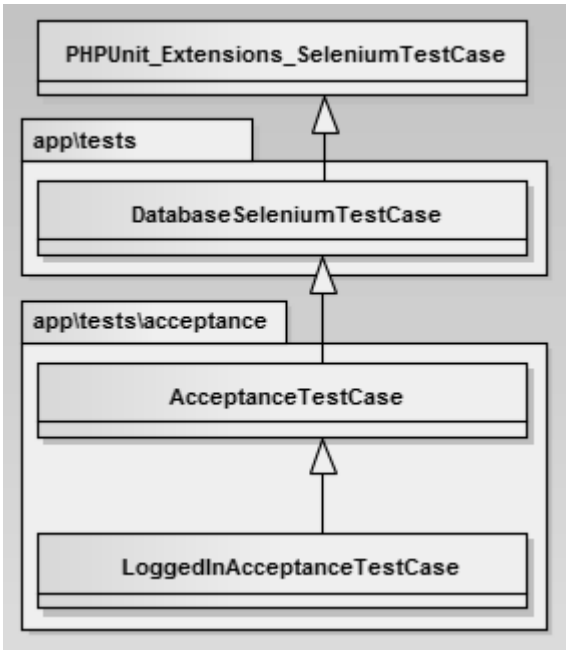
Az `$xmlFilePath` paraméter egy érvényes XML fájl elérési útvonala. A `PHPUnit` minden teszt lefuttatása előtt alapértelmezésben „*clean insert*” műveletet végez ezzel az adathalmazzal, azaz a táblák tartalmát eldobja, és a fájlban szereplő rekordokkal tölti fel őket. Így a tesztmetódusok már szabadon beszúrhatnak, módosíthatnak, törölhetnek és lekérdezhetnek rekordokat, mivel a kiindulási állapot ismert, könnyű meggyőződni az eredmény helyességéről. Az XML fájl pontos felépítéséről szintén a `PHPUnit` kézikönyvében találhatók információk **[2]**.

3.3.8. SPECIÁLIS TESZTEK: FELHASZNÁLÓI FELÜLET

A felhasználói felület tesztek az elfogadási tesztek (funkcionális tesztek, vagy ügyfél tesztek) kategóriájába tartoznak. Ezek a tesztek a bemutató alkalmatzás fökönyvtárán belül a `tests/acceptance` könyvtárban és alkönyvtáraiban kaptak helyet. Végrehajtásukhoz a `Selenium` nevű speciális szoftverre van szükség (elérhetősége a „*6.1. Letöltési linkek és telepítési útmutatók*” fejezetben megtalálható). A szoftver alapvetően két fő részre osztható: `Selenium Server` és `Selenium Core` (avagy

`Selenium Test Runner`). A szerver egy Java nyelven írt program, amely parancsokat fogad HTTP protokollon. Képes elindítani különböző böngészőket, és aktiválni, illetve parancsokkal ellátni a bennük futó teszt-végrehajtó környezetet. Ez a környezet a `Selenium Core`, melyet JavaScript és HTML kód alkot. A tesztelni kívánt alkalmazást egy keretben („*frame*”) nyitja meg, és a szervertől kapott parancsok alapján akciókat és értékvizsgálatokat végez. Az értékvizsgálatok eredményét visszaküldi a `Selenium Server`-nek.

A `PHPUnit` tartalmaz egy speciális teszt-osztályt, mely képes kommunikálni ezzel a szerverrel: ez a `PHPUnit_Extensions_SeleniumTestCase`. A bemutató alkalmazás elfogadási tesztjei mind erre épülnek, néhány további segédosztály beiktatásával az öröklési láncba. Mivel a tesztek egymástól függetlenül is futtathatók, az adatbázist mindig alapállapotba kell helyezni egy teszt megkezdése előtt. Ezt nyújtja a `DatabaseSeleniumTestCase`, mely az előző fejezetben bemutatott adatbázis-teszteknél használt segédfüggvényekkel egészíti ki az eredeti, `Selenium`-ot vezérlő osztályt. Az `AcceptanceTestCase` ennek a konkrét teszt-adatbázishoz csatlakozó leszármazottja. Az öröklési lánc végén található `LoggedInAcceptanceTestCase` azért szükséges, mert a bejelentkező- és regisztrációs-oldal kivételével minden további funkcionalitás érvényes munkamenetet követel meg. Minden teszt indítása új böngészőpéldányban történik, ezért a munkamenet nem adódik át két teszt között. Ezt úgy hidalja át, hogy minden teszt megkezdése előtt automatikusan beír egy ismert, a teszt adatbázisban szereplő felhasználónevet és jelszót, majd bejelentkezik.



22. ábra: Az elfogadási tesztek ősosztályainak hierarchiája

Egy teszt metódus tipikusan a következőket teszi:

1. Megnyitja a tesztelni kívánt oldalt

2. Felhasználói akciókat végez
3. Értékvizsgálatokkal megállapítja a teszt kimenetelét

A tesztekben legtöbbször használt akciókat az alábbi táblázat tartalmazza:

Név (paraméterek):	**Művelet:**
open(cím)	új címre navigálja a böngészőt
clickAndWait(lokátor)	A meghatározott elemre kattint, és vár az új oldal betöltésére
type(lokátor, érték)	A meghatározott űrlap elembe beírja a megadott értéket

A lokátor paraméter egy HTML elem speciális hivatkozását tartalmazó karakterlánc. Ez többnyire egy CSS-szelektor, XPath- vagy DOM-kifejezés. További akciók és a lokátorok pontos leírása a `Selenium` parancsok dokumentációjában található **[7]**. Ugyanitt találjuk az összes értékvizsgálatot paramétereikkel együtt. A legfontosabbak:

Név (paraméterek):	**Mit vizsgál:**
assertLocation(cím)	a böngésző aktuális címe egyezik-e a megadottal
assertTextPresent(minta)	van-e a megadott mintára illeszkedő szöveg az oldalon
assertElementPresent(lokátor)	létezik-e a lokátorral megadott elem
assertTable(lokátor, minta)	az adott táblázat-cella szövege illeszkedik-e a mintára

A minta valójában egyszerű szöveg, vagy reguláris kifejezés is lehet, ha „regexp:” előtaggal látjuk el. Az egységtesztekhez képest ezek a teszt-metódusok általában egynél jóval több értékvizsgálatot tartalmaznak.

Az elkészített tesztek valójában a felhasználói dokumentációban ismertetett eseteket fedik, illetve minden vezérlő minden vezérlő-akciójához készült egy teszt, és a vezérlőnek megfelelő nevű könyvtárba került (a kivételek kezeléséért felelős hiba-oldalt nem számítva). A generált kódlefedettségi jelentés (a `coverage` könyvtárban) sajnos nem a valós állapotot mutatja, mert a külső elirányításra végződő futási ágakat működésénél fogva nem ismeri fel.

3.2.9. Teljesítmény, optimalizáció

A keretrendszer fejlesztői dokumentációjának végén bemutatott teljesítménymérési módszerrel megállapítható, hogy a legtöbb időt természetesen az adatbázis elérése veszi el. Ezt úgy lehet csökkenteni, ha az adatokat nem mindig az adatbázisszervertől kérjük le. Megtehető ugyanis, hogy a lekérdezett rekordokat egy memória-gyorsítótár szervernek adjuk át, így a következő alkalommal már onnan is elérhetők. Mivel ezek

végig a memóriában tartják a szükséges adatokat, lényegesen gyorsabbak, mint lemezről elérni őket egy adatbázis-szerveren át. Arra azonban figyelni kell, hogy az adatokat módosítás esetén érvénytelenítsük a gyorsítótárban. PHP platformhoz ajánlott memória-gyorsítótár szerver például a `Memcached` (*http://memcached.org/*), vagy a `redis` (*http://redis.io/*).

Miden PHP-alapú web-alkalmazás általánosságban is tovább gyorsítható bájtkód-gyorsítótárazás révén. Ez az értelmező munkáját segíti azzal, hogy a lexikális és szintaktikai elemzéseket nem kell minden futtatásnál végrehajtania, mert a forráskód helyett annak bájtkódra fordított változatát tároljuk a szerveren. A gyorsításon túl így ezek védik is az alkalmazást, hiszen nem kell kiadnunk a forráskódot. Ilyen gyorsítótár a `Zend Guard` (*http://www.zend.com/en/products/guard/*) illetve az `Alternative PHP Cache` (*http://hu.php.net/apc*).

4. Összegzés

4.1. Tapasztalatok

A dolgozat elkészítése során szerzett tapasztalataim alapján egy ilyen kisméretű projekt esetén, további karbantartás hiányában az időtakarékosságot figyelembe véve nem célravezető a módszer alkalmazása, ugyanakkor a különféle tesztek bemutatásához pont eléggé változatos. Hosszabb élettartam, és többfős fejlesztés esetén azonban érezhető előnyökkel jár egy ilyen típusú munkamódszer bevezetése. Az automatizált tesztek által nyújtott „védelmet” már a keretrendszer kifejlesztése közben megtapasztalhattam, mert nagyobb méretű refaktorálások után is mindig működőképes maradt a kód, ha a tesztek sikeresen lefutottak. Hosszú távon ezzel rengeteg hibakeresésre fordított időt lehet megspórolni, és optimalizálásra vagy további fejlesztésekre felhasználni.

4.2. Továbbfejlesztési lehetőségek

4.2.1. KERETRENDSZER

Amint a fejlesztői dokumentáció optimalizációval foglalkozó alfejezetében már kitértem rá, érdemes implementálni az osztálybetöltő és a konfiguráció gyorsítótár-kezelését. A keretrendszerre épülő alkalmazások üzemeltetését segíthetné, ha a naplózás integrálásra kerülne a rendszer többi komponensével, és azok működését automatikusan tudná követni. Napjainkban szintén fontos, hogy egy szoftver több nyelven is képes legyen kommunikálni felhasználóival (különösen igaz ez a weben). Erre a célra érdemes egy lokalizációval foglalkozó csomagot megvalósítani, mely képes nézeteket, hibaüzeneteket, naplókat, útvonalakat több nyelven is kezelni.

4.2.2. BEMUTATÓ ALKALMAZÁS

A bemutató alkalmazás elsősorban a keretrendszer könnyebb megértését szolgálja, így a keretrendszer továbbfejlesztése esetén természetesen az új funkcionalitásokat ebben alkalmazni kell. További lehetőség a továbbfejlesztésre, ha a szintén az optimalizációs szekcióban említett memória-gyorsítótárazással egészítjük ki az adatbázis-eléréseket.

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] PHPUnit Manual *http://www.phpunit.de/manual/3.5/en/phpunit-book.pdf* (2011. április)

[3] MVC — XEROX PARC 1978-79 *http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html* (2011. május)

[4] Session hijacking attack - OWASP *https://www.owasp.org/index.php/Session_hijacking_attack* (2011. május)

[5] Gerard Meszaros: XUnit test patterns: refactoring test code, Addison-Wesley, 2007, [833], ISBN-10: 0-13-149505-4, ISBN-13: 978-0-13-149505-0

[6] Inversion of Control Containers and the Dependency Injection pattern *http://martinfowler.com/articles/injection.html* (2011. május)

[7] Selenium reference *http://release.seleniumhq.org/selenium-core/0.8.0/reference.html* (2011. május)

[8] Cross-site Scripting (XSS) - OWASP *https://www.owasp.org/index.php/XSS* (2011. május)

[9] JSON-RPC 2.0 Specification *http://groups.google.com/group/json-rpc/web/json-rpc-2-0* (2011. május)

[10] Bjarne Stroustrup: The Design and Evolution of C++, Addison-Wesley, 1994, [480], ISBN-10: 0-201-54330-3, ISBN-13: 978-0201543308

6. Mellékletek

6.1. Letöltési linkek és telepítési útmutatók

Az alábbi hivatkozások 2011. május hónap szerinti állapotot tükrözik.

- **Apache HTTP Server:** *http://httpd.apache.org/*
- **PHP:** *http://www.php.net/*
- **PHP Xdebug kiterjesztés:** *http://www.xdebug.org/*
- **PHPUnit PEAR-csomag:** *https://github.com/sebastianbergmann/phpunit/#readme*
- **vfsStream PEAR-csomag:** *http://code.google.com/p/bovigo/wiki/vfsStream*
- **WinCacheGrind:** *http://sourceforge.net/projects/wincachegrind/*
- **KCachegrind:** *http://kcachegrind.sourceforge.net/html/Download.html*
- **Mysql Community Edition:** *http://www.mysql.com/downloads/mysql/*
- **Oracle Java Runtime Environment:** *http://www.oracle.com/technetwork/java/javase/downloads/jre-6u25-download-346243.html*
- **Selenium Server:** *http://seleniumhq.org/download/*
- **Mozilla Firefox 4:** *http://www.mozilla.com/hu/firefox/*

6.2. Ábrák listája

6.3. Elfogadási tesztek listája

Az alábbi lista generált, a `PHPUnit „testdox”` kimenetének egy részlete.

app\tests\acceptance\user\RegisterPage

✓ Form elements and register link present

✓ Validation errors present on empty form submit

✓ Error present on user name unique check fail

✓ Error present on email unique check fail

✓ Login link is present on successful registration

app\tests\acceptance\user\ViewPage

✓ Data elements and back link present

✓ Forwards to task list if user does not exists

✓ Forwards to task list if user id does not provided

app\tests\acceptance\user\LogoutPage

✓ Logout deletes cookie and forwards to login

app\tests\acceptance\user\LoginPage

✓ Form elements and register link present

✓ Validation errors present on empty form submit

✓ Error present on invalid credentials

✓ Forwards to task list on valid credentials

app\tests\acceptance\error\NotFoundPage

✓ Shows not found page on invalid route

app\tests\acceptance\task\DeletePage

✓ Confirmation text and links are present

✓ Back link is present and task removed on successful delete

✓ Show error page if task does not exists

✓ Show error page if task id does not provided

✓ Show error page if task is not own

app\tests\acceptance\task\EditPage

✓ Form elements filled correctly and back link present

✓ Validation errors present on empty form submit

✓ Links are present and view works on successful operation

✓ Show error page if task does not exists

✓ Show error page if task id does not provided

✓ Show error page if task is not own

app\tests\acceptance\task\ListPage

✓ Table headers are correct and new link present

app\tests\acceptance\task\ViewPage

✓ Data elements and back link present

✓ User full name is present at public task view

✓ Show error page if task does not exists

✓ Show error page if task id does not provided

✓ Show error page if task is not public or own

app\tests\acceptance\task\NewPage

✓ Form elements and back link present

✓ Validation errors present on empty form submit

✓ Links are present and view works on successful operation