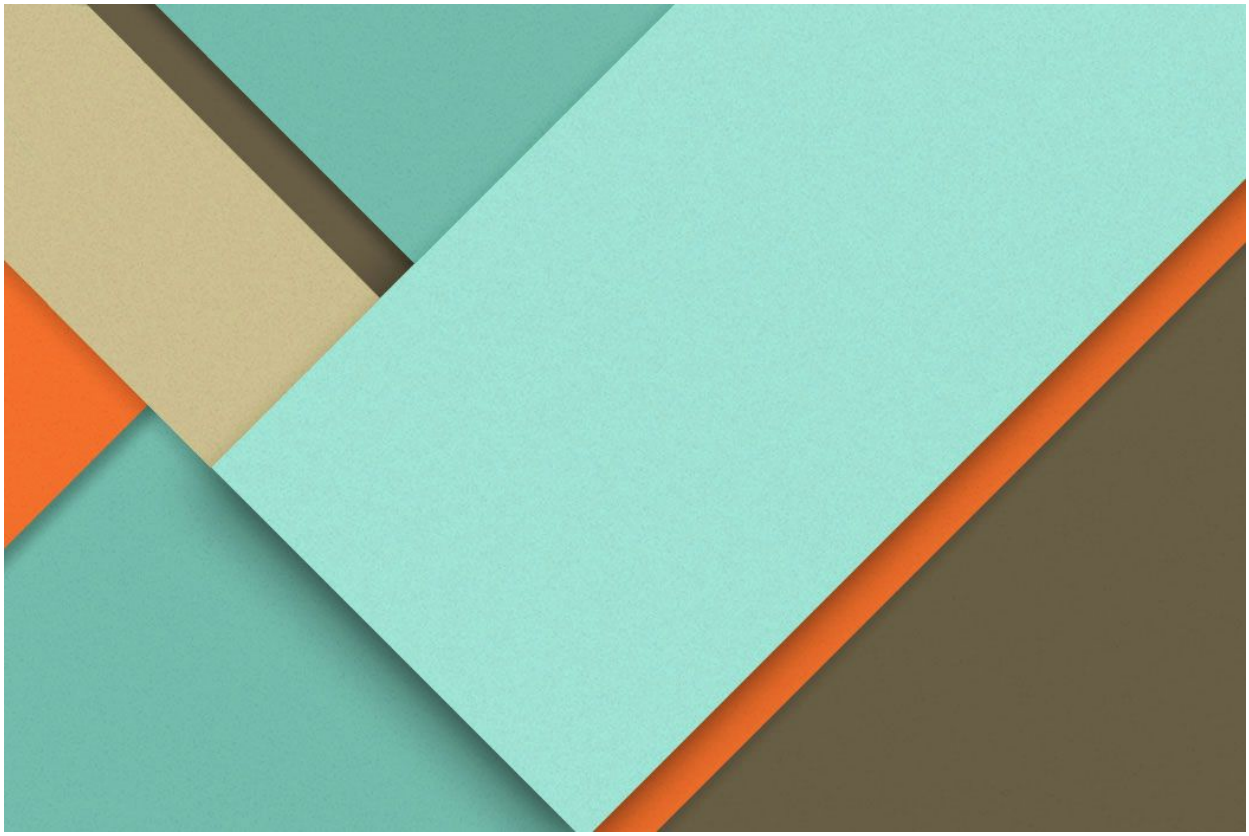


# Test Reporter: Követelmény Specifikáció

10.11.2015

Sűrű Balázs, Szekeres Imre

## Áttekintés

A mai fejlesztési módszerek (főképp az agilisek) nagy hangsúlyt fektetnek a megírt kód minőségére, azon belül is annak megfelelő tesztelésére. Azonban a tesztek sikeres lefutásán túlmenően egyéb metrikák, paraméterek, mérőszámok is megjelentek, hogy információt szolgáltassanaka arról, hogy a futtatott tesztek miképpen fedik le az adott projectet, melyik részeit jobban, kevésbé, és ez alapján kívánnak nyújtani valamiféle minőségi és megbízhatósági becslést. Ennek érdekében az egyes tesztelő keretrendszerek rendszerint valamiféle kimeneti fájllal is szolgálnak, amik tartalmazzák ezen adatok valamely halmazait.

Az általunk készítendő Test Reporter vastagkliens alkalmazás feladata az lesz, hogy a fent említett kimeneti fájlokból általa elkészített riportokat megjelenítse, historikusan tárolja, valamint szöveges dokumentumot képezzen ezen riportokból, továbbáelőhívhatóvá tegye korábbi állapotaikat is.

## Funkciók (Use-Case -ek)

1. Grafikus felület nyújtása
2. Projectek felvétele teszt keretrendszer megadásával
3. Projectek listázása
4. Támogatott teszt keretrendszerek (és azok elvárt teszt kimenet formátumának) listázása
5. Teszt kimenet beolvasása és abból (virtuális) riport készítése
6. Riport megjelenítése
7. Riportok (project szerint) historikus tárolása
8. Riportok szerializálása
9. Riportok listázása adott projecthez
10. Riport visszatöltése (dátum alapján)
11. Riport-hoz komment felvétele és módosítása
12. Historikus adatok alapján az egyes projectek riportjaiból aggregált adatok, metrikák különböző nézetekben megjelenítése
13. Új teszt kimenet felvételével a nézetek frissítése

14. Az egyes aggregált nézetek egymástól független kezelhetősége (pl adott dátumig bezárólag vett aggregáció)
15. Bemeneti file-ok feltakarítása (konfigurálható)
16. (Shell) parancsok parametrizált végrehajtása
17. Telepítésre és futtatásra shell szkriptek (windows, linux) szolgálnak

## Extra funkciók

1. Még a nagyméretű ( > 100 MB) teszt kimenetek feldolgozása se okozzon 1 percnél több várakozást.
2. A rendszer könnyen kiterjeszthető legyen esetlegesen más teszt keretrendszerek támogatására
3. Az egyes elemek csatolása (laza, szoros) tegye lehetővé a könnyű tesztelhetőséget

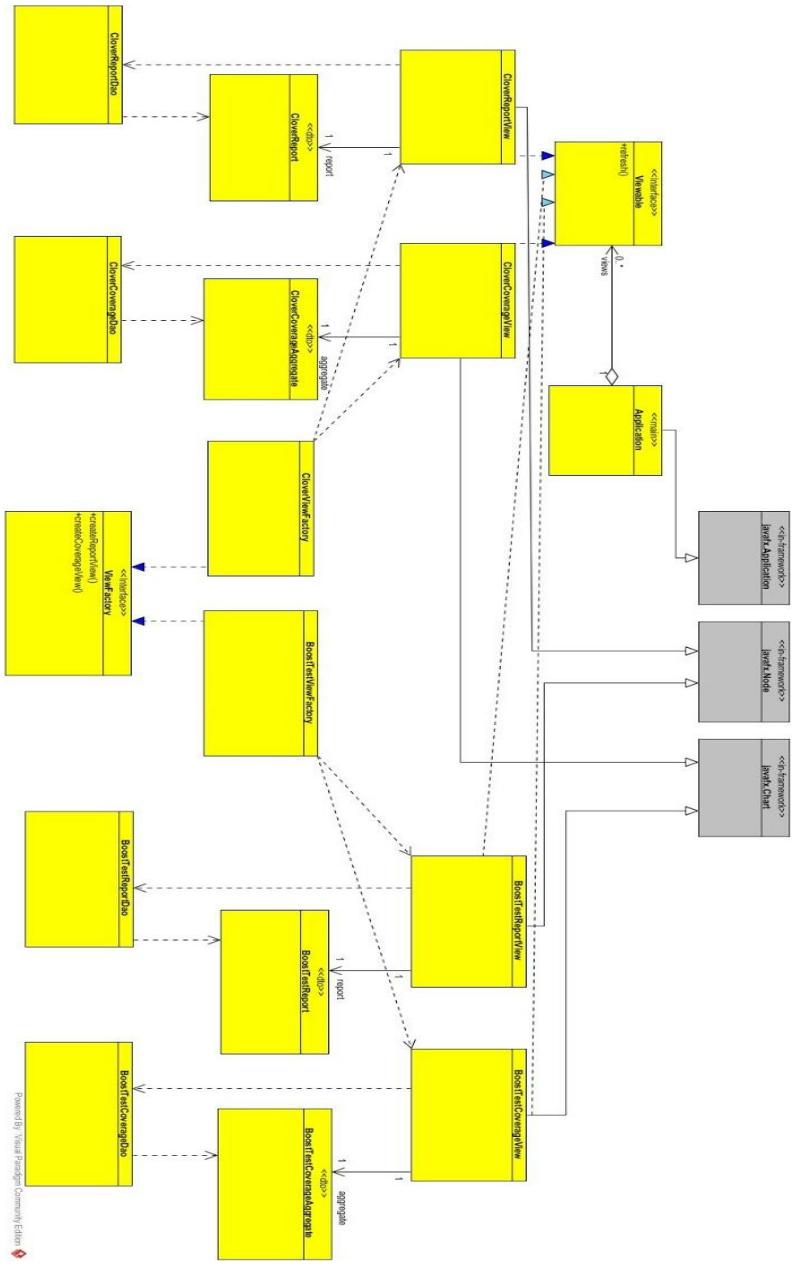
## Futtatási Platform

Az alkalmazás alepvetően a Java 8 Standard Edition szolgáltatásait, technológiát felhasználva épül fel, amely lehetővé teszi számára, hogy mind Linux, mind Windows környezetben is telepíthető, futtatható legyen. Az ablakozást a Java FX támogatásával alakítjuk ki. Az adatok historikus tárolására MongoDB-t használunk. A project build-elérésére és a függőségek (library-k például) menedzselésére pedig az Apache Maven eszközt. A változások hatásának minimalizálása és az egyes funkciók helyességének biztosítására a rendszer mellett annak egységtesztjei is elérhetők lesznek, amelyek a JUnit keretrendszer szolgáltatásaira támaszkodnak.

A közös, de más helyen végzendő munka megkönnyítése végett a Git (bitbucket-en tárolt bare repository-t használva) Source Code Management eszközt fogjuk használni. Az eszköz további előnye a haladás lövethetőség, változáskövetés és a visszaállíthatóság.

# Osztálydiagram

A rendszer objektumelvű szerkezetének leírására osztálydiagramot használtunk, mellyel az osztályok és azok tartalmát, viszonyát kívánjuk szemléltetni.



# Kiválasztott Minták

## I. Document-View

**Célja:** Adatok tárolása, menedzselése. Szokás modellnek is nevezni.

**Példa:** Egy dokumentumhoz több nézet rendelése. Pl.: táblázatos nézet, kördiagram...

**Előnyök:**

- Az architektúra több nézetet támogat egy azon dokumentumhoz

**Hátrányok:**

- Minimális többletköltség

**Felhasználás:** A programunk a projectek beolvasása után felhasználói beavatkozásra riportot generál. Ezeket a riportokat, jelen esetben a projectkhez tartozó tesztlefedettséget több nézetben is megjeleníthetjük. Erre fogjuk használni ezt a mintát.

## II. Observer

**Célja:** Objektumok egymást értesítik állapotuk megváltozásáról anélkül, hogy függőség lenne a konkrét osztályaiktól. Az Observer az egyik leggyakrabban használt minta.

**Példa:** MVC vagy Document-View architektúra - A felhasználó megváltoztatja az egyik nézeten az adatokat, hogyan frissítsük a többit?

- Egy objektum megváltoztatása maga után vonja más objektumok megváltoztatását, és nem tudjuk, hogy hány objektumról van szó
- Egy objektumnak értesítenie kell más objektumokat az értesítendő objektum szerkezetére vonatkozó feltételezések nélkül

**Előnyök:**

- Laza kapcsolat a Subject és az Observer között
- Üzenetszórás támogatása
- Könnyű kiterjeszthetőség

**Hátrányok:**

- Felesleges Update-ek
- Az egyszerű Update alapján az Subject összes adatát le kell kérdezni (bár a Subject az Object::Update meghívásakor megadhatja paraméterként, hogy mi változott meg).
- Az előbbi modellben a Subject ::Notifyt meghívó Observer-re is meghívódik az Update, holott az változtatta meg a ConcreteSubject állapotát

**Felhasználás:** Az alkalmazás a fedolgozott riportokat több nézetben is megjelenítheti. Elvárható működés az, hogy ezen nézetek új adat beérkezése esetén, vagyis mikor új riport generálódik, frissüljenek és az új eredményt mutassák. Ehhez a funkciónalitáshoz az Observer mintát fogjuk használni.

## III. (Abstract) Factory + Builder

**Célja:** A minta lehetővé teszi, hogy az új példány létrehozását a leszármazott osztályra bízzuk. Szokás virtuális konstruktornak is nevezni.

**Példa:** Ablakos rendszerek, GUI vezérlőelemek (ablak, nyomógomb, kiválasztógomb, stb.)

- A rendszernek függetlennek kell lennie az általa létrehozott dolgoktól ("termék" objektumok, pl. felhasználói felület elemek)
- A rendszernek több termékcsaláddal kell együttműködnie
- A rendszernek szorosan összetartozó "termék" objektumok adott családjával kell dolgoznia, és ezt akarjuk kényszeríteni a rendszerben

**Előnyök:**

- Elszigeteli a konkrét osztályokat
- A termékcsaládokat könnyű kicserélni
- Elősegíti a termékek közötti konzisztenciát

**Hátrányok:**

- Nehéz új termék hozzáadása. Ekkor az Abstract Factory egész hierarchiáját módosítani kell, mert az interfész rögzíti a létrehozható termékeket
- Megjegyzés: ezt bizonyos esetekben ki lehet kerülni

**Felhasználás:** A minta használatával a riporthoz tartozó nézetek példányosítását tesszük egyszerűbbé.

## IV. Data Access Object

**Cél:** Megkönnyíti az alkalmazás karbantartását, elválasztva az üzleti logikát az adatelérési logikától. Általában minden egyes entitáshoz (domain object) készül egy-egy DAO.

**Előnyök:**

- Üzleti logika és adatelérési logika szétválasztása
- Az adatelérési logika újrafelhasználható

**Hátrányok:**

- Hiányos absztrakció
- Kód duplikálás

**Felhasználás:** DAO minta felhasználásával megkönnyítjuk a projektek használatát és karbantartását, azáltal, hogy elkülönítjük az adatbázis specifikús kódot az alkalmazás kódjától.

### V. Bridge

**Cél:** Különválasztja az absztrakciót (interfészt) az implementációtól, hogy egymástól függetlenül lehessen őket változtatni

**Példa:** hordozható ablakozós rendszer XWindow és Presentation Manager alá

**Előnyök:**

- Az absztrakció és az implementáció különválasztása
- Az implementáció dinamikusan, akár futási időben is megváltoztatható
- Az implementációs részletek a klienstől teljesen elrejthetők
- Az implementációs hierarchia külön lefordított komponensbe tehető, így ha ez ritkán változik, nagy projektek esetén nagymértékben gyorsítható a fordítás/buildelés ideje
- Ugyanaz az implementációs objektum, több helyen is felhasználható

**Hátrányok:**

- Komplexitás növekedés, ami teljesítmény romlást is okozhat.

**Felhasználás:** A Bridge mintával az adatbázistól való függőséget szeretnénk minimalizálni. Használatával lehetőségünk lesz akár futásidőben változtatni az adatelérési logikát.

## Technológiai követelmények

- MongoDb
- Java jdk 8
- Java FX 2.x
- Maven 3.x
- JUnit 4.11
- Mockito 1.9.5