Eötvös Loránd Tudományegyetem
Informatikai Kar

Komputeralgebra Tanszék

Web alapú tesztelés és tesztautomatizálás

Témavezető:

Dr. Kovács Attila

Egyetemi docens ELTE

Készítette:

Huber Hajnalka

Programtervező informatikus

Budapest, 2018

Tartalomjegyzék

1. Bevezetés:

A mai gyorsan fejlődő világban egyre inkább szükséges, hogy a cégek egy megbízható terméket tudjanak a vásárlóiknak előállítani, lehetőleg minél hamarabb és minél jobb külső megjelenéssel. A szoftverfejlesztés esetében ez különösen igaz, hisz nagyon sokszor valami teljesen újat kell elkészíteni egy ötlet alapján. A StartUp vállalatok nagy kockázatot vállalva igyekeznek egy új, innovatív és jól működő szoftvert létrehozni. A megfelelő minőséget pedig a jól kiválasztott szoftverfejlesztési metodológia és tesztelési módszertanok alkalmazása biztosítja. A különböző minősített technikák lényege, hogy teljes mértékben személyre, cégre és csapatra szabhatóak. Mivel minden projekt különböző, és más képzettségű emberek dolgoznak egy adott szoftveren - ami sokszor nagyon komplex is -, ezért az elkészítés során hiba kerül a szoftverbe. Ezeket a hibákat hivatott a tesztelés megtalálni. Maga a tesztelés arra való, hogy a szoftver megbízhatóságát és annak stabilitását különböző irányozott tesztelési eszközökön, technikákon keresztül növelje. Korábban, amikor még nem léteztek tesztelést segítő alkalmazások, sőt a munkakör sem igen létezett, a szoftverfejlesztők ellenőrizték a kész programot, akik nem alkalmaztak szoftvertesztelőket. Ez viszont nem volt olyan hatékony, hiszen az ember hajlamos a saját hibáját nem észrevenni, ezért egy független, új szemszög és nézőpont sokkal nagyobb hatékonysággal és lefedettséggel tud dolgozni, így hibákat észrevenni és azokat megfelelő módon a fejlesztő felé átadni javításra. Idővel a cégek felfedezték az erre való igényt, és szoftverek, projektek méretének növekedésével kialakult az elvárás a gyors és alapos tesztelésre. Ennek köszönhetően fejlődtek ki a különböző teszttámogató eszközök, tesztelési módszertanok, stratégiák. Ezek a stratégiák ajánlást tesznek arra vonatkozóan, hogy hogyan, mikor, milyen eszközökkel érdemes, hasznos, vagy nem utolsó szempontként költséghatékony a tesztelés. A teszteléshez megfelelő környezetet létrehozhatja mag a tesztelő is, ám sokszor ez egy kisebb projekt a projektben, és olyan fejlesztői munkát kíván, mely akár nagyobb időt és ezzel együtt költséget is kíván [1]. Érdemes megjegyezni, hogy a tesztelés végigkíséri a szoftver teljes életciklusát, nem ér véget a szoftver elkészültével. A szoftver folyamatosan változik a kérések alapján, így azt folyamatosan karban kell tartani. Eleinte a legegyszerűbb módon, manuálisan keresték a tesztelők a hibákat, ám ez időigényes munka, aminek következtében a tesztelést végző személy is hibát követhet el. Mindez a tesztelés automatizálásához vezetett. A tesztelők egyre

inkább hasonló technikai ismeretekkel rendelkeznek, mint a fejlesztők és ez az automatizált tesztelést alkalmazó cégeknél ez egy fontos elvárás is. Nem csak programozói készség, de a logikus gondolkodásmód és minden más - amit fejlesztőktől megkövetelnek - szükséges egy tesztelői állás betöltéséhez.

Szakdolgozatomban az automatizált tesztelés előnyeire világítok rá egy konkrét példán keresztül. A jó megértéshez egy rövid ismertetést is írok, melyben kifejtem a tesztelésben használt kifejezéseket, hogy ezekre a dolgozat további részében már hivatkozhassak. Leírást adok a tesztelés szerepéről egy szoftverfejlesztési cikluson belül, valamint különböző lehetőségeket vázolok a tesztelés aspektusaira. Ezek ismeretében pedig felépítek egy rendszert, melyben az automatizált tesztesetek koordinálhatóak, az eredmények elemezhetőek és kezelhetőek. Munkám során web alapú tesztelést végzek [2]. Azért ezt a témát választottam, mert manapság egyre inkább terjednek az ilyen jellegű alkalmazások és jelenlegi munkakörömben is ilyen teszteléssel foglalkozom, ezért a dolgozat megírása során így tudok a leginkább hasznos új tudást szerezni, és azt felhasználni későbbi munkáim során.

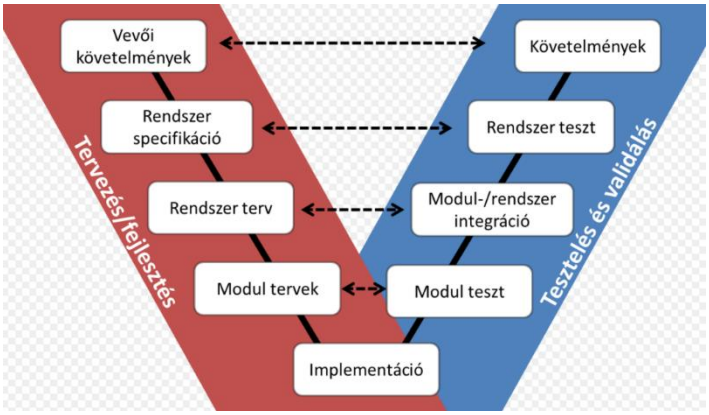
2. Szoftverfejlesztési életciklusok és a tesztelés

A módszertanok feladata, hogy meghatározzák, hogy a szoftver életciklus egyes lépései milyen sorrendben követik egymást, a tesztelés és fejlesztés hogyan kapcsolódik össze. Az életciklus-modellek leírják, hogy az egyes lépések során miket kell elkészíteni, legyen az egy terv egy specifikáció vagy maga a szoftver és annak tesztelése. Ezek a modellek, módszertanok ajánlást adhatnak a költséghatékony és jól szervezett szoftverfejlesztéshez. A továbbiakban két modellt ismertetek azok előnyével és hátrányaival.

2.1 V-modell

A V-modell a nevét onnan kapta, hogy két szára van és így egy V betűhöz hasonlít. Az egyik szára lényegében megegyezik a vízesés modellel. Ez a fejlesztési szár. A másik szára a létrejövő termékek tesztjeit tartalmazza. Ez a tesztelési szár.

A V-modell a korai modellek családjába tartozik, melyet a német védelmi minisztérium fejlesztett ki és eleinte a német hadsereg szoftverfejlesztéseiben vált használatossá. Az elnevezés nemcsak életciklus modellt, hanem egy teljes módszertant jelöl, aminek több elemét az ISO 12207 szabvány is átvette. A V-modell az egyes fázisok időbeli sorrendje mellett azt is definiálja, hogy az egyes fázisokban mely korábbi fázisok eredményeit kell használni; illetve az adott fázis tevékenységét és termékét mely korábbi fázisban leírt követelmények, illetve elkészített tervek alapján kell ellenőrizni. A V-modell használata főleg a biztonságkritikus számítógéprendszerek fejlesztése esetében a terjedt el [3].



1.Ábra: A V-modell fejlesztési és tesztelési ágai

Fejlesztési fázisok a V-modellben:

- Követelmények specifikálása: itt a fejlesztési folyamat kiindulási pontját képező követelmények feltárása, elemzése, majd specifikációja történik. A fázis eredménye egy dokumentum, amely részletes információt tartalmaz a rendszer szolgáltatásairól és megszorításairól.
- Kockázatok elemzése: célja a lehetséges veszélyhelyzetek meghatározása a rendszerben a megelőzés érdekében. Az analízisek elvégzéséhez különféle módszerek állnak rendelkezésre. A kockázatok elemzésére különféle technikák léteznek. Az elemzési folyamatok eredményeként létrehozandó a kockázati mátrix, és az elemeinek kezelését leíró dokumentáció.
- Teljes rendszer-specifikáció: a funkcionális követelmények, valamint a nem funkcionális, pl. biztonsági követelmények együttese alkotja. Mindezen specifikáció alapján megkezdhető a teljes rendszer konkrét tervezési folyamata.
- Architekturális tervezés: a teljes informatikai rendszer hardveres és szoftveres architektúrájának megtervezése. A tervezésnek ebben a fázisában többek között azt is el kell eldönteni, hogy mely funkciók legyenek megvalósítva hardver, és melyek szoftver által.
- A szoftver modulokra bontása: a fázisban a fejlesztési folyamatot további kisebb részekre, úgynevezett modulokra bontjuk fel a tervezési folyamat egyszerűsítése, áttekinthetőbbé

tétele végett. A tervezés eredményeként a szoftver modulok specifikációja, valamint a köztük levő kapcsolódási folyamatok terve készül el.

- A modulok elkészítése és tesztelése: a szakaszban egyes modulok teljes implementációja valósul meg, ezután az elkészült modulok önálló tesztelése következik. Célszerű a tesztelési folyamatokat szintén előzetesen megtervezni.
- Rendszerintegráció: ebben a fázisban az elkészült szoftver-modulok integrálása történik egy teljes rendszerré.
- Rendszer verifikáció: ezen fázis feladata annak az eldöntése, hogy rendszer megfelel-e a specifikációjának, funkcionálisan teljesíti-e az összes specifikációs pontot. Továbbá el kell dönteni, hogy a teljes rendszer megfelel-e minden további, nem funkcionális követelménynek. Ebbe beletartozik a biztonsági feltételek teljesítésének eldöntése is.
- Rendszer validáció: Ebben a fázisban az ügyfél igényeinek való megfelelés igazolása történik.
- A rendszer üzemeltetése: üzembe helyezés, üzemeltetés, karbantartás, elavulás, üzemeltetés megszüntetése [3].

Előnyei:

Ha a fejlesztés és tesztelés alatt nem változnak a követelmények, akkor ez egy nagyon jó, kiforrott, támogatott módszertan. Jó a dokumentáltsága, így későbbi fejlesztések során jól átlátható marad olyanok számára is, akik nem vettek részt a szoftver fejlesztésében. További előnye még, hogy egyszerűen menedzselhető, mert külön választja az egyes fázisokat.

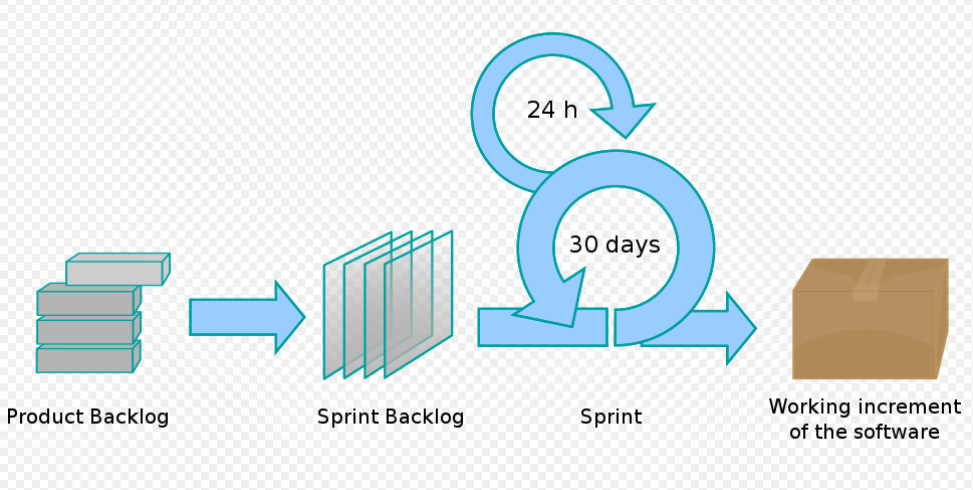
Hátránya:

Nagyon merev modell, kevéssé engedi meg a rugalmasságot a követelmények változtathatóságát illetően, továbbá viszonylag későn látja a vevő a termék első változatát, mely során sokszor kiderül, hogy nem is pontosan az került lefejlesztésre, amit a vevő megálmodott. Már a projekt elején megköveteli az ügyféltől, hogy véglegesítse a követelményeket mielőtt a tervezés elindulna, a tervezőtől pedig azt, hogy válasszon ki bizonyos tervezési stratégiákat az implementáció előtt.

2.2 Agilis szoftverfejlesztés

Az agilis szoftverfejlesztés a szoftverfejlesztési módszerek egy olyan csoportja, ahol a szoftverkövetelmények kezdetben csak részlegesen adottak és a megoldásokon keresztül válnak egyre pontosabbá, együtt fejlődik az önszerveződő és sokszor multifunkcionális csapatok támogatásával. Ez elősegíti az adaptív, rugalmas tervezést, az evolúciós fejlesztést, korai szállítást, folytonos továbbfejlesztést és lehetőséget biztosít a változásokra reagáló gyors és rugalmas válaszokra [4].

Az agilis szoftverfejlesztésben leggyakrabban inkrementális és iteratív modellek alapján dolgoznak. Ekkor a szoftver fejlesztésének folyamatát nem egy egyszerű lineáris folyamatként értelmezzük, hanem sokkal inkább egy önmagára visszamutató rendszeresen ismétlődő folyamataként. Ez azt jelenti, hogy ciklikus ismétlődések – nevezhetjük iterációknak a továbbiakban – során a rendszert mindig továbbdolgozzuk az igényelt változások szerint. Ezen megközelítés oka, hogy a nagy rendszerek esetében elkerülhetetlenek a fejlesztés során a változtatások. Változhatnak a követelmények, az üzletmenet, és a külső behatások [5]. Az egyik kedvelt agilis modell a **Scrum**:



2.Ábra: A Scrum fejlesztés ciklusai

A Scrum egy keretrendszer, amely magában foglal bizonyos tevékenységeket és meghatározott szerepeket. A Scrum főbb szerepkörei a „Scrum Master”, aki a folyamatot felügyeli és a projektmenedzserrel ellentétben, a csapat önálló munkavégzését támogatja, a „Product Owner” (magyarul terméktulajdonos), aki a projektben érdekelt döntéshozókat képviseli, és a „Csapat” (Team) ami a nagyjából 7 főből áll és végrehajtják az összes fejlesztési és tesztelési munkafolyamatot.

Minden „futam” (sprint) során - amely 2 és 4 hét közötti időtartamot jelent (a csapat döntésétől függően) - a csapat egy működő szoftveregységet hoz létre. A futam során megvalósítandó funkciók a „Product Backlog”-ból (termék teendőlistája) kerülnek ki, ami az elvégzendő munka magas szintű követelményeiből álló, fontossági sorrendbe állított lista. Hogy a futam során a lista melyik elemei kerülnek megvalósításra, azt a futam elején tartott „futamtervező” megbeszélés során választják ki. A megbeszélés során a „Product Owner” közli a csapattal, hogy a teendők listájából melyek azok, amiket leghamarabb szeretne, hogy elkészüljenek. Ezután a csapat eldönti, hogy ezek közül melyek azok, amelyeket a következő futam során meg tud valósítani, és ezek megvalósítására ígéretet tesz. A futam folyamán a „futam teendőlistáját” nem lehet megváltoztatni, a futam során elvégzett tevékenységek rögzítettek. Amint a futam a végéhez ért, a csapat bemutatja az elkészült funkciókat (demo).

Az önszerveződő csapatok kialakulásának elősegítése végett a *Scrum* arra ösztönöz, hogy a projekt résztvevői egy helyen dolgozzanak, és szóban kommunikáljanak egymással [6].

Előny:

Nagyon adaptív fejlesztési módszert tesz lehetővé. A tesztelés már a fejlesztési életciklus legelején megjelenik, így csökkentve a később lehetségesen előforduló hibák számát, amik nagyon nagy költségeket tudnak magukkal vonzani. Az ügyfél már az első hetekben is képek kaphat arról milyen lesz a végtermék, így nagyon hamar észreveheti, ha valami nem a terveinek megfelelően lett leimplementálva. A folyamatos kommunikáció pedig segít elkerülni a konfliktusokat és bizalmi légkört épít fel.

Hátrány:

A sok változtatás miatt a dokumentáció, sokszor hiányos vagy nem is létezik, így ha új ember kerül a csapatba, a tudásátadás legjobban tapasztalt kolléga által történhet meg; továbbá önállóan nehezebben lehet a rendszert megismerni és a későbbi karbantartási fázisban is gondot okozhat a dokumentáció hiánya.

3. Szoftvertesztelési folyamatok és módszerek

Mint már említettem, a Scrum modell előnyeinél, a szoftverfejlesztésben nagy kiadásokat lehet megspórolni azzal, ha az esetleges hibákat minél hamarabb észrevesszük. Ez adódhat félreértésből, melyet a követelmények elemzésénél már észrevehetünk és tisztázhatunk, de lehet olyan dolog is, ami egyszerűen nem került meghatározásra, holott fontos lett volna tudni. Például hogy egy oldal mennyi idő alatt töltődik be, melynek leírására a „gyorsan” nem elég, az ilyen jellegű nem funkcionális követelményeket is pontosan meg kell határozni. Ezekből kiindulva a tesztelési folyamatok már nagyon hamar bekapcsolódnak a szoftver életciklusába. Ezek a folyamatok nagyon sok követelményt is lefedhetnek, az igényeknek és a rendszer elvárásainak megfelelően.

A következő szempontokat, követelményeket érdemes alapul venni, hogy megállapítsuk milyen tesztelési stratégiára van szükségünk:

- Funkcionalitás és kapcsolódó nem-funkcionális követelmények (pl.: pontosság, biztonság)
- Használhatóság (pl.: tanulhatóság, érthetőség, üzemeltethetőség)
- Megbízhatóság (pl.: hiba tűrés, hibakezelés, visszaállíthatóság)
- Hatékonyság (pl.: több felhasználó, lekérdezés kezelése (load), válaszidők)
- Karbantarthatóság (pl.: szoftver rugalmassága, tesztelhetőség, analizálhatóság)
- Portabilitás (pl.: installálhatóság, helyettesíthetőség, együttműködés) [1].

3.1 Teszterv készítése

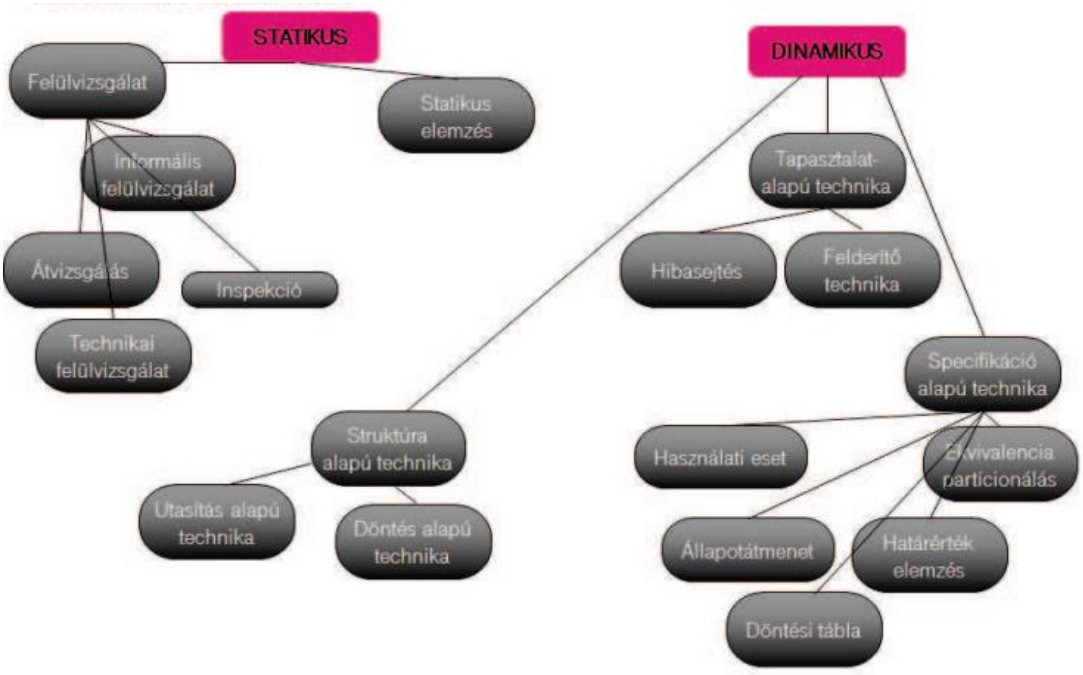
A teszterv készítésénél alapvetően meg kell határozni, hogy milyen alkalmazásról, szoftverről van szó. Az lehet biztonságkritikus pl. egy légi forgalom irányító rendszer, de lehet egy egyszerűbb alkalmazás is például egy hírportál. Nyilvánvalóan teljesen másként érdemes ezeket a rendszereket tesztelni. Az elsőnél a jó hibakezelés, pontos számítások elsődlegesek, míg nem olyan nagy baj az, ha szoftver nem olyan attraktív vagy éppen kicsit bonyolultabb és több tanulást igényel a jó használata. Míg egy hírportálnál, azonnal látnia kell a felhasználónak, hogy milyen lehetőségei vannak és fontos az is hogy hogyan reprezentáljuk a híreket.

A tesztelés tervezésénél, meg kell határozni, hogy a mely tesztek mely életciklusban hajtódnak végre és az ehhez tartozó környezetről és erőforrásokról is gondoskodni kell. Egy egy V-modell esetén elkülöníthetőbb, mint egy inkrementális modellnél, ahol az életciklus részei összemosódnak az iterációk során. Fontos továbbá meghatározni, hogy mikor nevezünk egy tesztelést késznek, mik a kilépési kritériumok.

Mivel munkám során egy web alapú oldalt fogok tesztelni, így a tervezésnél az adott oldal funkcionalitásának helyes működését fogom tesztelni, így leginkább a funkcionális és megbízhatósági szempontok alapján fogok tesztelni. A web alapú tesztelés nem alkalmas karbantarthatósági illetve hatékonysági tesztek kivitelezésére. Maga a tesztelés a teljes „rendszeren”, magán az oldalon fog zajlani. A web alapú tesztelésre jellemzően ez rendszertesztnek minősül, így az életciklus egy előrehaladottabb stádiumában léphet be, amikor az adott funkciók már stabilan elérhetők. Teszteszközöm pedig a Selenium webdriver lesz Eclipse fejlesztői környezetben. Mivel a munkám célja a web alapú tesztelés bemutatása, így itt kilépési kritériumot nem fogalmazok meg.

3.2. Tesztelési módszertanok

Számos különböző tesztelési módszer létezik, mindegyiknek van erőssége és gyengesége. A már korábban említett szempontok alapján, melyeket a tesztelési tervben már említettem megkülönböztetünk statikus és dinamikus tesztelési. Statikus módszer pl. egy dokumentum tesztelése, abban ellentmondások keresése. A dinamikusak közül bemutatom az alapvető tesztelési módszereket, hiszen ezt fogom én is alkalmazni. [7]



3.Ábra: Statikus és dinamikus technikák összefoglaló diagramja

3.2.1 Statikus tesztelési technikák

Statikus tesztelés alatt, ahogy azt a nevéből is ki lehet következtetni, olyan tesztelést értünk ahol nem történik kód futtatás, vagyis a futtatás nélkül végzünk tesztelést. Ez lehet dokumentum vagy kód analízis is. Ez a fajta tesztelés inkább alkalmas tévedések leheséges félreértelmezések, sem pedig hibák megtalálására. A folyamat során észrevehetünk

- Valamilyen szabálytól való eltérés
- Követelmény hiányossága
- Inkonzisztens interfész specifikáció
- Nehezen karbantartható részek detektálása

Ezen folyamat során észrevehetjük azokat a területeket is, amiket majd a dinamikus tesztelés során alaposabban kell tesztelni.

Statikus tesztelési technikák többek között:

- Adatfolyam elemzés
- Vezérlési folyam elemzés
- Felülvizsgálat (review)

3.2.2 Dinamikus tesztelési technikák

Dinamikus tesztelés esetén már olyan tesztelés is végezhető, amikor már fut a kód is és a specifikáció és a követelmények alapján végezzük a tesztelést. Ennek egy másik elnevezése a fekete doboz tesztelés, amikor magát a kódot nem látja a tesztelő, hanem az egyes folyamatok eredményét vizsgálja.

4.Ábra: A feketedoboz tesztelés sematikus diagramja

Első lépésként elő kell állítani azokat a tesztadatokat, amelyek feldolgozásra kerülnek, majd előállítjuk a végeredményt, amit a rendszernek is szolgáltatnia kell.

Tehát a módszer használatával a tesztelő arra összpontosít, hogy mit csinál a szoftver, nem pedig arra, hogyan teszi azt. Ugyanúgy végeznek funkcionális (egy adott funkció működik-e vagy sem) és nem funkcionális (egy adott funkció mennyire működik jól, pl. terhelés esetén) tesztelést is [8]. A tesztelés alkalmával ezt vagyis a fekete doboz módszert fogom alkalmazni.

3.3 Manuális és automatizált tesztelés összehasonlítása

Egy szoftver manuálisan és automatizálva is letesztelhető, de hogy melyiket választjuk, az attól függ, hogy a befektetett költség hogyan térül meg jobban.

Automata tesztelésnél egyes előre meghatározott lépések elvégzése történik újra és újra miközben összehasonlítjuk az elvárt és a tényleges kimeneti értékeket. Manuális tesztelés esetén pedig az adott folyamatot a tesztelő hajtja végre személyesen, mintha egy felhasználója lenne annak a szoftvernek, amit éppen ellenőriz.

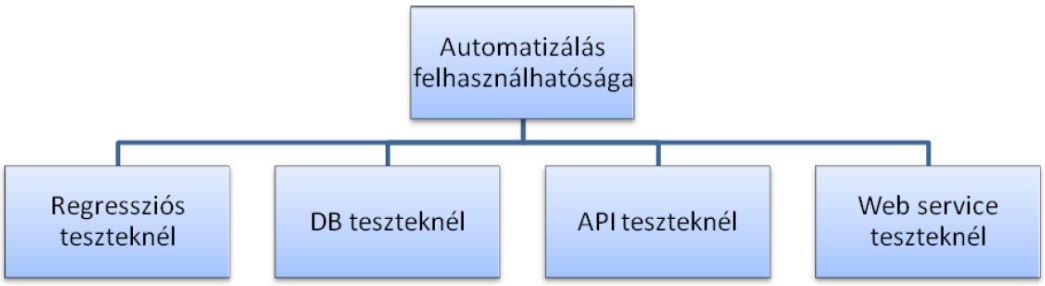
Manuális tesztelés előnyei és hátrányai [9]:

Előny	**Hátrány**
Kisebb projekten költséghatékony	Sok olyan teszt van, ami nem tesztelhető manuálisan pl.: performancia
Képes olyat észrevenni, amit automatizálva nem lehet: ember végzi.	Monoton-nagyobb a hibalehetőség
Gyorsabb reakciót tud biztosítani	Nehezebben reprodukálható, sebesség, környezet változhat.

Automatizált tesztelés előnyei és hátrányai [9]:

Előny	**Hátrány**
Tesztesetek futtatása gyors és hatékony	Drága a rendszert összerakni
Hosszú távon költséghatékonyabb	Hiba esetén időigényes lehet az újra futtatás
Érdekesebb munka ezért a hibázás lehetősége is kisebb	Az eszközöknek vannak korlátai, nem minden tesztelhető automatikusan
Bárki láthatja az eredményeket	Programozói tudást igényel
Automatikus riportolást tesz lehetővé	
Könnyen reprodukálható	

Az automatikus tesztelés alapvetően az ábrán látható területeken jelenik meg.



5.Ábra: Automatizált tesztelés lehetséges területei

4. Tesztkörnyezetben használt eszközök kiválasztása

Az eszközök kiválasztásánál figyelnünk kell arra, hogy milyen célra szeretnénk felhasználni.

Rengeteg lehetőség kínálkozik:

- Kód analízis
- Performacia teszt
- Web alapú teszt
- API tesztelés
- Model alapú tesztelés
- Teszt management eszközök

Az eszköz kiválasztása során figyelembe kell vennünk azt is, hogy az eszköz együtt tudjon működni más rendszerekkel is szükség esetén. Lehetőség szerint pedig az ezekben tárolt adatok, ha több eszközt is használunk egyszerre, jó, ha az adatok nem duplikálódnak és automatikusan kerülnek be a lehető legkevesebb manuális tevékenységgel. [1]

4.1 Automatizált tesztelést támogató eszközök

Munkám során egy web alapú tesztelést valósítok meg, mely egy internetes oldal működését teszteli, tehát ez egy ún. funkcionális teszt. Fontos szempont, hogy mindezt úgy valósítsam meg, hogy az elkészült esetek sokáig használhatók legyenek a lehető legkevesebb karbantartással és könnyen bővíthetők legyenek. **A tesztelés célja egy webes felület ellenőrzése megadott szempontok alapján**, ehhez további applikációk vagy eszközök tesztelése nem szükséges csakis a **webes felület funkcionalitásai lesznek a teszt célcsoportja**. Tehát egy olyan eszközt szerettem volna találni, ami a webes felületek minél alaposabb tesztelését teszi lehetővé, úgy hogy az minél közelebb álljon egy valódi felhasználóhoz.

A keresés során több lehetőséget és eszközt is vizsgáltam.

A **SOAP UI** egy széles körben elterjedt program, mellyel hatékonyan végezhetők funkcionális tesztek. Előnye továbbá, hogy ez egy szabad forrású eszköz, tehát nem kell megvenni és jól használható web service hívások és válaszok tesztelésére. Amiért mégis elvetettem, az az ok volt,

hogy alapvetően háttérből látjuk az eseményeket általa, tehát nem felhasználó felületen, hanem hívás-válasz interakciókon keresztül tesztelhetünk általa [10].

Ezek után már olyan eszközöket kerestem, melyek a felhasználó felületen keresztül képesek automatikus tesztek megvalósítására. Vizsgálatom során utána néztem az eggPlant, QuickTest Professional (QTP) és Selenium programoknak.

Az **eggPlan**t szoftver egy két-rendszer modellt valósít meg. Az egyik az ahol az eggPlant fut a másit pedig rendszer, amit tesztet (SUT- System under test). Előnye, hogy nem csak webes felülete tesztjét képes megvalósítani, hanem szükségszerűen más applikációk működését is ellenőrizni tudja. Hátránya, hogy vizuális szempontok alapján elemzi a képernyőn megjelenő adatokat, tehát az ott elfoglalt pozíciót kell megadni, hogy megtaláljon egy-egy gombot vagy szöveget. Ez azt jelenti, hogy későbbi fejlesztés során, ha egy szöveg vagy gomb máshol jelenik meg, akkor azt módosítani kell a tesztben is így itt a karbantartási költségek miatt döntöttem úgy, hogy nem ezt a szoftvert használom. [11]

A következő lépésben a **QTP** szoftvert is figyelembe vettem, mint lehetséges szoftvert. Ez nagyon hasonló a Seleniumhoz és a talált leírások alapján könnyebben használható. További előnye a Seleniumhoz képest, hogy ez is, mint az eggPlant nem csak webes felületek funkcionális tesztjére alkalmas. Egyetlen komoly hátránya, hogy nem ingyenes, tehát használata komoly költségeket von maga után, amik csak nagyon nagy léptékű tesztelés esetén térülne meg. Mivel nekem csak webes felületű teszt megvalósítására van szükségem és a tesztek mennyisége sem indokolja ezért ezt az eszközt is elvetettem. [12]

Ezek után már tisztán látszott, hogy a **Selenium** által nyújtott lehetőségek lesznek a leginkább megfelelőek számomra. Egyetlen hátránya, hogy komolyabb fejlesztői tudást igényel a tesztelőktől, és csak webes eszközök felhasználói felületének tesztelésére alkalmas. Nagy előnye viszont, hogy jól karban tartható a széleskörű lokalizációs lehetőségeknek köszönhetően valamint ingyenesen lehet használni és népszerűsége miatt sok segítség érhető el az interneten így külön tanfolyamköltségekre sem kell számítani, ha ezt az eszközt választom.

Első lépésben egy olyan Selenium által támogatott eszközt használok, mely egyszerű, de épp ezért elég merev lehetőségei korlátozottak. Ez az eszköz hasznos lehet egy kisebb projektben, automatizálást valósít meg, olyan módon hogy az adott lépéseket felveszi és ezen lépésekhez

adatokat rendelhetünk ezzel bővítve a tesztesetek számát. Majd ahogy elméletben nő a projekt, egy sokkal komplexebb rendszert valósítok meg, mely nem csak tesztek futtatásáért felelős sokkal rugalmasabb módon, de az eredmények kezelését is hatékonyabban valósítja meg. Ekkor használom a Selenium WebDriver-t és annak architektúráját, mellyel már programozható a tesztelés, hatásos megoldásokat nyújtva. Így a tesztelési folyamat teljes körűen támogatottá válik. [13]

4.1.1. Record and replay eszközök

GUI record & replay eszközök azon célból lettek kifejlesztve, hogy grafikus interfésszel rendelkező applikációkat lehessen tesztelni velük. Használva az eszközt a tesztelő felveheti a lépéseket, amiket az grafikus felületen végrehajt. A szkript ekkor rögzít minden felhasználói lépést beleértve az egér mozgatását is, majd később képes pontosan ugyanezt visszajátszani ahányszor csak az szükséges. Ezek az eszközök így lehetőséget nyújtanak egy teljes regressziós tesztre grafikus interfészek esetén.

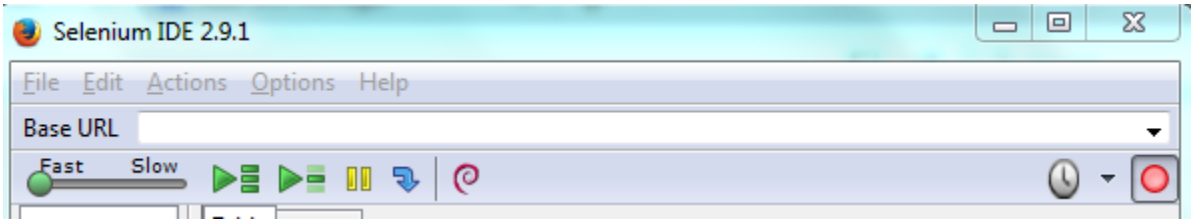
Hogy egy ilyen eszközt is bemutassak kiválasztottam egy honlapot, melyet A Selenium IDE plugin segítségével fogok tesztelni. Így be tudom mutatni azt, hogy mik ezeknek az eszközöknek az előnye és a hátránya.

Selenium IDE:

Selenium IDE egy beépített fejlesztői környezet Selenium szkriptekhez. A Firefox bővítményeként használható és tesztek felvételét, módosítását és debugolását teszi lehetővé. Selenium IDE tartalmazza a teljes Selenium Coret, elősegítve ezzel a könnyű és gyors teszt felvételt és visszajátszást az aktuális környezetben, ahol az futott.

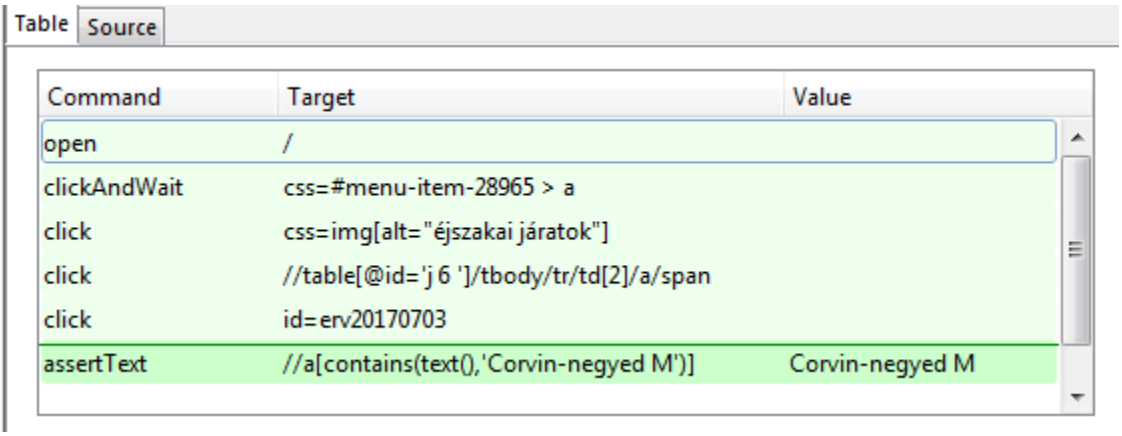
Azért ezt az eszközt választom, mert az esetek felvétele mellett képes vagyok adatokat is az esetekhez rendelni és mivel később is Selenium eszközt használok, majd a tesztek automatizálására jó előkészítésnek tartom.

Selenium IDE használatával nem csak felvenni, lejátszani tudok eseteket, de a lejátszási sebességen is változtathatok. A futtatás során mindig egy új ablak jelenik meg, amelyen nyomom követhetjük, hogy mi történik. Nem csak egy-egy esetet, hanem teljes teszt sorozatot is lefuttathatunk.



6.Ábra: Selenium Ide főképernyő részlet

Az eset első felvétele után, könnyen lehet, hogy változtatni kell az egyes lépéseken, vagy az web alkalmazás elemeinek elérését kell pontosítani, hogy azt majd később megtaláljuk. Az alkalmazás ehhez is lehetőséget ad kétféle nézetben is: Table és Source



7.Ábra: Selenium IDE teszteset kódja

A felvett eseteket később karban is kell tartani. Ez a record-and-replay alkalmazások hátránya, hiszen nagyon törékeny merev kód jön létre, ha egy kezdeti lépés, például bejelentkezés megváltozik, akkor azt minden egyes esetben meg kell majd később változtatni, ami nagyon időigényes, ezért ezt az eszközt kezdetben kisebb feladatok tesztelésére szokták használni, illetve akár adatelőkészítésre is alkalmas lehet a sokszor ismétlődő egyszerű lépéseket kell csak végrehajtani. [14]

4.1.2. Selenium általános bemutatása

Selenium elnevezés használata sokszor nem egyértelmű, megkülönböztetünk Selenium 1 és Selenium 2-t, melyek a Selenium projekt verziói. A Selenium 1 még nem volt böngésző független, az egyes böngészőkkel a Selenium Remote Control (RC) kommunikált, így lehetséges volt, hogy az egyik böngészőn működött a megírt kód a másikon viszont már nem. A Selenium 2 már webdrivert használ, mint alapértelmezett API-t. Ennek használatával a parancsok a kliens gépről közvetlenül a böngészőbe továbbítódnak driverek segítségével.

8.Ábra: Selenium Webdriver architektúra

Munkám során a Selenium 2-t fogom használni, mivel sokkal rugalmasabb és jobban használható a korábbi verziónál, valamint nagyon sok forrás áll rendelkezésre, hogy a tesztek automatizálását megfelelően és hatékonyan tudjam leimplementálni.

4.2 Teszteszközök bemutatása

A tesztek előkészítéséhez és implementálához, valamint az eredmények riportolásához az alábbi eszközöket használtam:

1. **Selenium Webdriver API** – a web alapú automatizáláshoz
2. **Eclipse** – a tesztek implementálásához
3. **Git** – az implementáció nyomon követéséhez
4. **TestLink** – tesztek követhetőségéhez, riportozáshoz

Selenium Webdriver API

Mint azt a *8. Ábrán* is láthattuk a Selenium Webdriver API egyfajta köztes csatornát biztosít a különböző kódnyelven megírt scriptek és a különböző Browserek között. A különböző browserek használatánál viszont továbbra is kihívást jelenthet azok mérete ezzel együtt az egyes elemek láthatósága. Maga a browser gyorsasága és a cookie-k használata is.

A Selenium Webdriver API segítségével lehet a különböző oldalakon és azok közt **navigáln**i valamint az oldalakon az egyes adatokra keresni vagy módosításokat végezni.

Az alapvető műveletek [15]:

- Navigálni egy url-hez
    - driver.navigate().back()
    - driver.navigate().forward()

- Frissíteni az oldalt
    - driver.navigate().refresh()
- Bezárni az oldalt
    - driver.close()
- Lekérni az oldal adatait (cím, url, tartalom)
    - driver.getCurrentUrl()
    - driver.getTitle()
    - driver.getPageSource
    - driver.findElement(By…).getText()
- Beírni az oldalra
    - driver.sendKeys()

A findBy függvényben különböző paraméterek segítségével tudjuk **megadni az egyes elemek helyé**t. Ezt a html oldal forrásának elemzése után könnyen meg tudjuk tenni. Magát a html forrást az egyes browserek fejlesztői nézetében tudjuk megnézni. Itt aztán megtalálhatjuk az egyes elemek lokációját [16]:

- id
- className
- linkText
- name
- cssSelector
- xpath

CssSelector és Xpath esetében relatív és abszolút megadási lehetőséget is vannak. Itt a relatív útvonal használata a leginkább javasolt, mert ha később változtatnak az oldal struktúráján, akkor lehet, hogy az abszolút útvonal már nem lesz használható.

Egy-egy példa ezekre:

Abszolút útvonal cssSelectorral és Xpathal:

- driver.findElement(By.cssSelector(”html>body>div>p>input”));
- driver.findElement(By.xpath(”html/body/p/input”));

Relativ útvonal cssSelectorral:

- driver.findElement(By.cssSelector(”button[name=,cancel’]”));
- driver.findElement(By.xpath(”//input[@id=’username’]”));

A Selenium Webdriver lehetőséget ad **műveletek kezelésére** is. Ez lehet egér vagy billentyű művelet is, ezzel is szimulálva a valódi felhasználói tevékenységet, mely a tesztelés során kiemelkedően fontos.

Néhány példa ezekre is:

- element.click, ahol az element lehet egy gomb, link vagy bármely más elem
- New Actions(driver).keyDown(Keys.SHIFT).click(element). keyUp(Keys.SHIFT).build().perform(); mely során egy shift+click művelet valósul meg
- newActions(driver).doubleClick().build().perform(); ekkor a kétszeri egérkattintást szimuláljuk

További lehetőségek is rendelkezésre állnak, hogy az automatizálás minél jobban tudja tükrözni egy valós felhasználó gondolkodását, így tudunk várni az oldal betöltődésére, mielőtt azon bármit is keresni kezdenénk, képesek vagyunk változtatni a képernyő méretét, stb.

Eclipse

Az Eclipse Oxygen csomagot, mint integrált fejlesztői környezetet (IDE) azért választottam, mert könnyen bővíthető plug-in kezelése van, így támogatja a Selenium Webdriver a JUnit használatát, valamint jól integrálható a TestLink ingyenes tesztmanagement eszközzel.

Továbbá hasznosnak tartom, hogy már a kód megírása során is ellenőrzi azt a beépített statikus elemző eszközével és jól használhatóak a billentyűműveletek is, amik meggyorsítják az implementációt.

Git

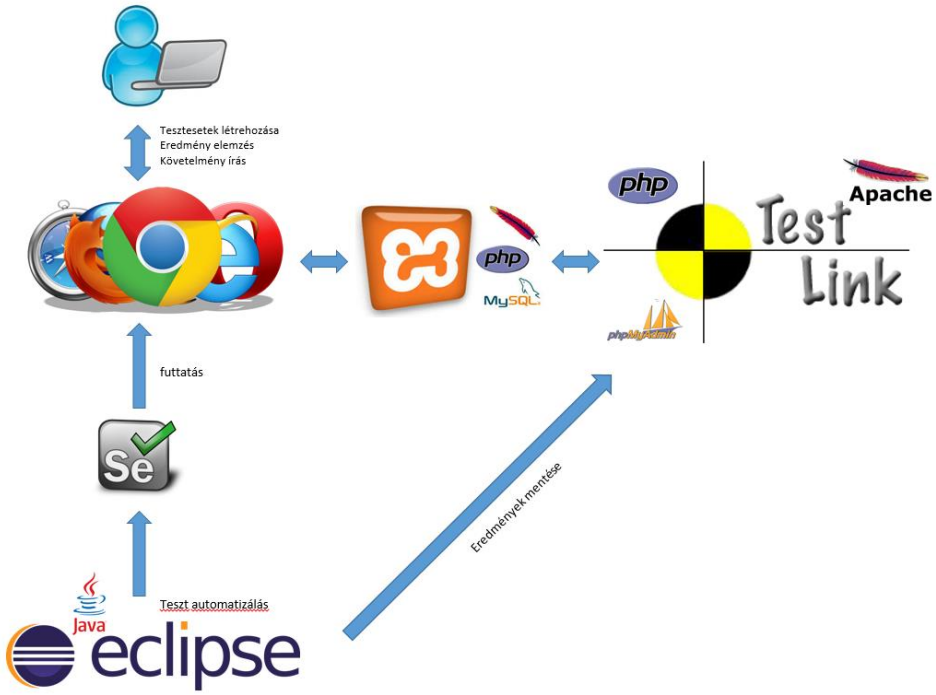
A Git verzió követő rendszer segít a fejlesztés során, hogy minden egyes módosítás követhető legyen. Több ember együttműködése esetén nagy szerepe van abban, hogy a feladatokat egymástól függetlenül (branch-eken) tudják végezni a fejlesztők, majd azok végeztével egy egésszé állhasson össze a szoftver (master branch).

A verziók és fejlesztői ágak között könnyen lehet navigálni, így a munka során, ha valami nem sikerült, akkor könnyen vissza lehet állni egy még stabil, működő verzióra, valamint tudjuk, hogy az egyes frissítéseket ki és miért tette be, így ha valamit nem értünk, gyorsabban tudunk a problémára vagy kérdésre választ találni. [17]

TestLink

A TestLink egy ingyenes web-alapú tesztmanagement eszköz [18]. Ez a felület támogatja a tesztesetek, teszttervek, követelmények riportok és statisztikák kezelését is. Azért ezt az eszközt választottam, mert a riportok legszélesebb körben ebből az eszközből importálhatóak és jól együtt működik a Selenium Webdriver által elkészített tesztesetekkel [19]. Az eszköz segítségével mindent egy helyen tudtam létrehozni és kezelni, így az információk duplikálása nem volt szükség, így a hibák lehetőségét nagyban csökkenteni lehet. Mivel a dolgozat megírásánál egy igazi tesztprojekt életét és működését is szerettem volna bemutatni, ezért ezt az eszközt alkalmasnak találtam arra, hogy szemléltessem a tesztelés összetettségét management szinten is. Továbbá használatára és felépítésére jól használható leírások vannak, így a megvalósítás során tudtam ezekre támaszkodni.

A létrehozott rendszer struktúráját a 9. Ábra jól szemléltetni és a [20] forrás leírása alapján integrálható össze.



9.Ábra: Tesztprojekt struktúrája

5. Automatikus tesztek tervezése és megvalósítása

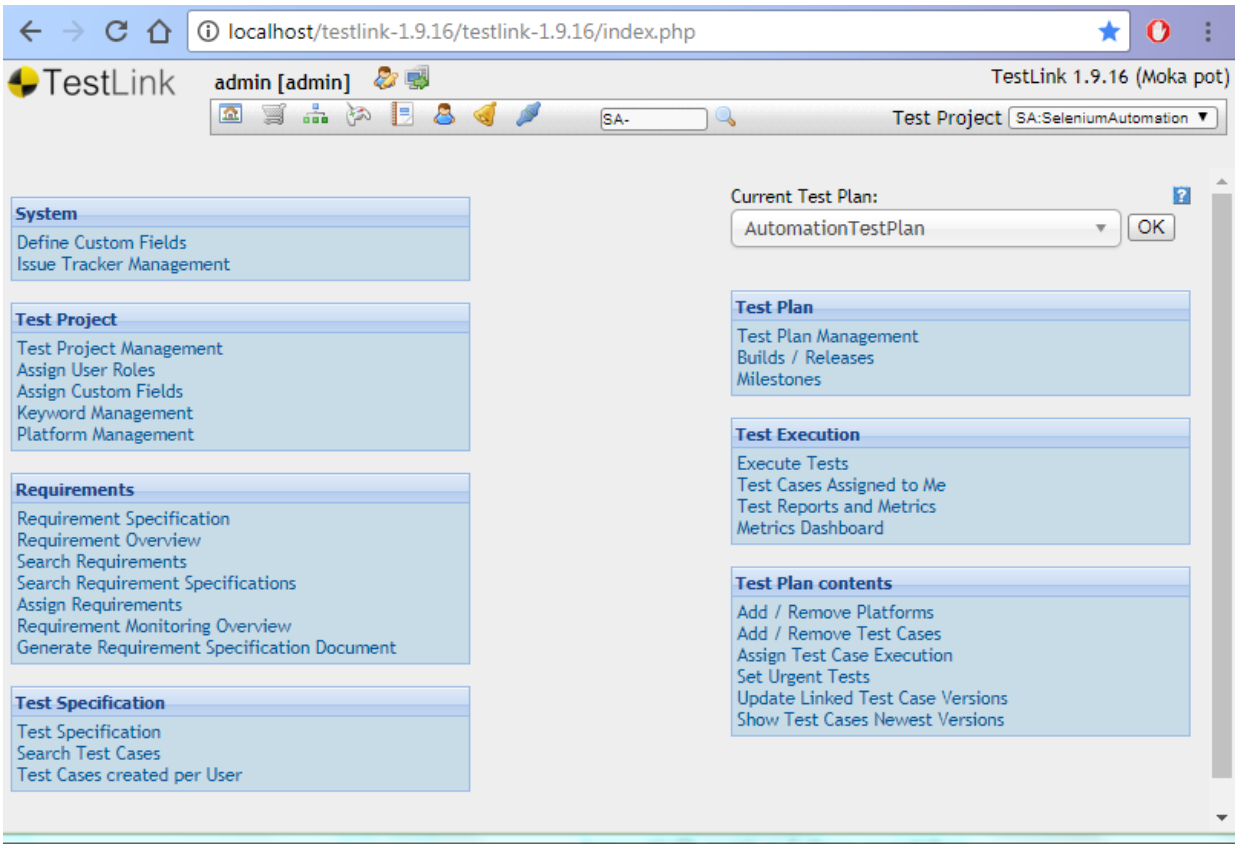
A legtöbb esetben az automatizált tesztek elkészítése a manuális tesztek lefuttatása során történik meg. Ez azért szükséges, hogy lássuk, hogy az adott funkció már elég stabil-e, ahhoz, hogy automatizáljuk. Amennyiben még sok változtatás várható, akkor az automatizált esetet nagyon sokszor át kéne írni, ami nem hatékony. Ahhoz, hogy tudjuk mit érdemes automatizálni, egy jól megtervezett tesztelési stratégia szükséges. Tudni kell, mik azok a kritériumok, amiket az eseteknek mindenképpen le kell fednie és mik azok, amik kisebb prioritással kell tesztelni. Az

automatikus esetek később azért lesznek felelősek, hogy regressziós tesztfuttatások során felfedjék az esetlegesen szoftverbe bevitt hibákat.

Szakdolgozatomban a Gmail levelezőrendszert fogom letesztelni. Azért ezt a rendszert választottam, mert ez egy stabil webes alkalmazás, melyet sokan használnak és széles körű tesztelési lehetőséget ad, így jól be tudom mutatni a Selenium Webdriver adta lehetőségeket a tesztautomatizálás során. Ezeket a részleteket majd a későbbi fejezetben ki is fejtem.

Első lépésben bekonfiguráltam a TestLink teszt management eszközt, ill. azzal a nehézséggel szembesültem, hogy a Testlink v1.9.16-ban az eredmények szinkronizálása az Execute Tests felületen nem működik, így az erre vonatkozó .php file-t le kellett cserélnem. Ehhez az XAMMP alkalmazás Apache és MySQL modulját is használtam, hogy a saját gépemet localhostként használva legyen egy phpmyadmin felületem és maga a tool is localhoston keresztül működik és kapcsolódik majd az Eclipse projekthez.

Ebben aztán leírom a követelményeket. A követelményeket egy igazi projekt során persze nem a tesztelő adja meg, hanem a Test Manageren keresztül, vagy közvetlenül az ügyféltől érkezik. Ezután fel tudom majd venni a manuális teszteseteket lépésenként és TestPlanhez és userhez tudom rendelni őket. Ha több ember is dolgozna a projekten, akkor a munkát így jól meg lehet osztani, mindenki tudja mi a feladata és mivel készült már el. Elsőnek manuális tesztként veszem fel az eseteket, majd az implementáció előrehaladásával állítom át őket automatikus esetekké. Az eredmények és a hibák riportozása is lehetséges ezen eszközön keresztül.



9.Ábra: TestLink kezelő felülete

Az alábbi sematikus terv szerint fogom majd megadni a követelményeket és a tervet.

Tesztkörnyezet: Gmail levelezőrendszer

Felhasználó: seleniumTesztUser

Tesztcsoportok (testsuit)

<u>Bejelentkezés tesztje:</u>
1. pozitív- jó jelszó és felhasználónév
2. rossz jelszó
3. rossz felhasználó

<u>Beérkező levelek funkció tesztje:</u>
1. A betöltött felület validálása
2. A felületen látható ikonok tesztje
3. Keresés tárgy alapján
4. Keresés tartalom alapján

Kimenő levelek funkció tesztje
1. A betöltött felület validálása
2. A felületen nem látható ikonok tesztje
3. Keresés feladó alapján
4. Keresés nem létező tartalomra

Általános felület tesztje (smoke teszt)
1. Jobb oldali ikonok tesztje
2. Új címke létrehozása és törlése
3. Levél áthelyezése
4. Levél törlése
5. Háttérképi téma módosítása

5.1 Az implementáció elkészítése során figyelembe vett szempontok

A projekt tervezésénél elsődleges szempont volt, hogy az később jól **karbantartha**tó legyen, hiszen egy projektnél, főleg ha már több ezer automatizált eset létezik, fontos hogy azokban a változtatások minél kevesebb időt és energiát vegyenek igénybe, jól átlátható legyen a kód, hogy azt az új munkatársak is könnyen megérthessék.

A másik szempont, mely szintén főleg nagyszámú eset esetén válik igazán kritikussá az a **futás gyorsaság**a. Ebben nagyon sokat számít, hogy a webelemek keresésénél, milyen lokátorokat használunk és hogy a browser válaszidejét minél rugalmasabban, függőségek által tudjuk kezelni, nem pedig hardkódolva adjuk meg a szükséges várakozási időket.

Localizátorok

Az elemek megtalálásához lokalizátorok széles tárháza áll a fejlesztő rendelkezésére. Tudni kell viszont, hogy melyik lokalizátor mennyire gyors és mennyire ajánlott a használata. [21]

Az elemek azonosításánál a Selenium Webdriver által használható By class elemeit használjuk. Ezek a lokalizátorok.

Legegyszerűbb és hatékony megoldás **id**-ra, **className**-re vagy **Name**-re keresni, ezek egyediek, de sokszor generáltak vagy változhatnak egy-egy oldal betöltése során, vagy ha változtatnak az oldal felépítésén. Ha a fejlesztés során ügyelnek rá, hogy az adott alkalmazás jól automatizálható legyen, akkor ezeket a legcélszerűbb használni.

A **CSS (Cascading Style Sheets) selector** a leginkább browser független, könnyű karban tartani és hatékony keresést eredményez, ami igen gyors. Itt is a relatív útvonal megadása ajánlott az abszolút helyett, hogy később, ha változik az oldal strukturális felépítése, akkor is működjön. [22]

Hasonló, bár kevésbé hatékony az **Xpath** a DOM struktúrájától függő elem, ellentétben a CSS selectortól. Itt is van mód relatív és abszolút útvonalak megadására. Xpath és CSS selector esetén is tudunk az attribútumokra keresni, ami nagypontosságú keresésre ad lehetőséget.

Link text egy olyan paraméter a html oldalon, mely felhasználói felületen is látható. Erre könnyű keresni, de más nyelvbeállítás esetén már más nyelvű lesz a megjelenés is, így már nem működne a lokátor.

Figyelembe kell venni azt is, hogy bár fejlesztői (inspect) nézetben mi látunk elemeket, de az nem biztos, hogy látható (not visible, enabled) vagy használható (not clickable) a felhasználói felületen is. Ezért a keresésnél érdemes figyelembe venni ezeket a paramétereket is.

Várakozási idő kezelése

A várakozások kezelése a szinkronizáció témakörbe tartozik. Nem csak a browserek de az egyes gépek teljesítménye között is különbségek vannak. Erre a szinkronizációra ad lehetőséget az explicit is az implicit várakozások beépítése a kódba. [23]

Az implicit várakozás (implicit wait) azt jelenti, hogy a WebDriver addig a maximális ideig keresi az elemet az oldalon, amíg az általánosan megadott timeout le nem jár, ezután Exception-t dob:

*driver.manage().timeouts().implicitlyWait(10, TimeUnit.**SECONDS**);*

Ezzel szemben az explicit várakozás (explicit wait) egy feltétel bekövetkezésére vár, de itt is van maximális várakozási idő. Akkor használjuk, amikor a végrehajtás előtt várnunk kell néhány feltétel előállására:

```
WebElement element = (new WebDriverWait(driver,10)).
until(ExpectedConditions.presenceOfElementLocated(By.id(elem)));
```

Page Objekt metodológia

A Page Objekt metodológia egy olyan ún. Design Pattern ami azért vált nagyon kedveltté, mert jól karbantartható kódot eredményez. A Page objekt egy objektum orientált osztály, mely egyfajta interfészként szolgál az egyes weboldalakhoz az automatizáció során. Előnye hogy a tesztesetek és a mögöttük lévő oldalstruktúra elkülönítődik, így változtatás során nem kell csak egy helyen módosítani a programot, mely után az összes olyan eset, mely felhasználta az adott elemet, megfelelően fog futni. A lokalizátorok kezelése is elkülönül a tesztektől, így ha szükséges, szintén nem kell végigmenni az összes teszteseten, hanem csak egy helyen elég elvégezni a frissítést.[21]

5.2. Teszt szkriptek elkészítése, futtatása

Azért, hogy az kód írása során az osztályokat csoportosítsam három packege-t hoztam létre. A base csomag tartalmazza azt a két osztályt, amiben webdriver inicializálása történik és amelyik olyan általános funkciókat tartalmaz, amiket aztán az összeg PageObject osztályban fel tudok használni. Ilyen funkciók például az egyes webelemekre történő dinamikus várakozás (10. Ábra).

```java
/**
 * This function waits for element to be clickable
 *
 * @param driver
 * @param locator
 * @param timeout
 */

public static WebElement waitUntilElementIsClickable(WebDriver driver, By locator, int timeout) {
    WebDriverWait wait = new WebDriverWait(driver, timeout);
    WebElement element=null;
    wait.until(ExpectedConditions.elementToBeClickable(locator));
    element=driver.findElement(locator);

    return element;
}
```

10.Ábra: Kódrészlet explicit wait alkalmazására

A pageobject csomagba, ahogyan azt a nevéből is ki lehet következtetni, az egyes page osztályok kerültek. Ezekben az osztályokban már magában a konstruktorban megtörténik a weboldalak validálása azok <title> értéke alapján. (11. Ábra)

```java
/**
 * Constructor -validate page and initialize the check boxes on the page for further usage
 * @param driver
 * @throws Exception
 */

public IncomingMailsPage(WebDriver driver) throws Exception {
    this.driver = driver;
    TestFunctions.validatePage(this.driver, "Inbox");
    checkboxes=driver.findElements(By.cssSelector(checkBoxesPath));
    System.out.println("Inbox page was loaded");
}
```

11.Ábra: Kódrészlet page object construktorra

Fontos, hogy az oldalakhoz tartozó webelemek elérési útvonala, lokátorai is itt találhatók, így ha bármit módosítani kell, akkor könnyen megtaláljuk a kódban a frissítendő változót és csak egy helyen kell a módosítást elvégezni. Az oldalakhoz tartozó műveletek is az osztályokban kerültek

leimplementálásra és a műveletek eredményét az Assert osztály által nyújtott függvényekkel könnyen le is lehet ellenőrizni (12. Ábra)

```
/**
 * Function teszt the existence of the icons, which should be shown on the page
 * @param iconNames - list of the expected names of the icons
 * @throws Exception
 */

public void checkIcons(String[] iconNames) throws Exception{
    for(String iconName:iconNames) {
        WebElement icon=TestFunctions.waitUntilElementIsClickable(driver, By.cssSelector("a[title*='"+iconName+"']"), 5);
        Assert.assertTrue("Check of the icon: "+iconName, icon.getText().contains(iconName));
        System.out.println(iconName+" was found");
    }
}
```

12.Ábra: Kódrészlet page object függvényre

Több helyen is úgy hoztam létre a függvényeket, hogy azok később bővíthetők legyenek. Ez azért volt szükséges, mert egy-egy menüponton belül listaszerűen lehetett kiválasztani az opciókat így a kattintásokhoz nem kellett külön-külön függvényeket készíteni, elég volt egyet lekódolni. Ennek a megvalósításához a switch-case elágazást használtam, mellyen későbbiekben tovább lehet bővíteni a funkciót (13. Ábra)

```
/**
 * Function to simulate the mouse movement to see hidden parts of the page
 * The function is extensible
 * @param text
 * @throws Exception
 */

public void moveMouseTo(String text) throws Exception {
     Actions action = new Actions(driver);

        switch(text) {
        case "More":
              action.moveToElement(driver.findElement(By.cssSelector(moreOptionsPath)));
              action.perform();
              break;
        default:
            System.out.println("move the mosue to "+text+" is not possible");
            throw new Exception("move the mosue to "+text+" is not possible");
        }

}
```

13.Ábra: Kódrészlet bővíthető page object függvényre

A page object osztályok felhasználásával hoztam létre aztán a teszteseteket is a testlink packageben. Az esetekben példányosítom ezeket az osztályokat és a hozzájuk tartozó funkciók és a webdriver segítségével már könnyen tudtam szimulálni egy valódi felhasználó viselkedését. A teszteseteket szcenáriókra osztottam, függően attól, hogy éppen melyik funkcionalitást kell tesztelni. Jelen esetben az 5. fejezet alapján, melyben megadtam, hogy milyen tesztesetek elkészítését tervezem. Azért ezeket az eseteket választottam, mert ezek elkészítésével jól be tudom mutatni a Selenium adta lehetőségeket és jó bevezetése lehet ez a néhány eset egy igazi komolyabb több száz vagy ezer esetből álló projektnek. Természetesen minél több a tesztesetek száma, annál több emberre van szükség azok elemzésére és a karbantartásukra is.

A TestLink inicializálása is a testlink csomagban történik meg, mert a tesztesetek futtatása után ezen osztályon keresztül kerülnek az eredmények be a TestLink rendszerbe (14. Ábra).

```
public class TestLinkIntegration {

    public static final String TESTLINK_KEY="e871f93999cca57bdf9961d68fc343e5";
    public static final String TESTLINK_URL="http://localhost//testlink-1.9.16/testlink-1.9.16/lib/api/xmlrpc/v1/xmlrpc.php";
    public static final String TEST_PROJEKT_NAME="SeleniumAutomation";
    public static final String TEST_PLAN_NAME="AutomationTestPlan";
    public static final String TEST_CASE_NAME="ValidLogin";
    public static final String BUILD_NAME="Sprint1Build";

    public static void updateResults(String testCaseName, String exception, String results) throws TestLinkAPIException {
        TestLinkAPIClient testlink=new TestLinkAPIClient(TESTLINK_KEY, TESTLINK_URL);
        testlink.reportTestCaseResult(TEST_PROJEKT_NAME, TEST_PLAN_NAME, testCaseName, BUILD_NAME, exception, results);

    }
}
```

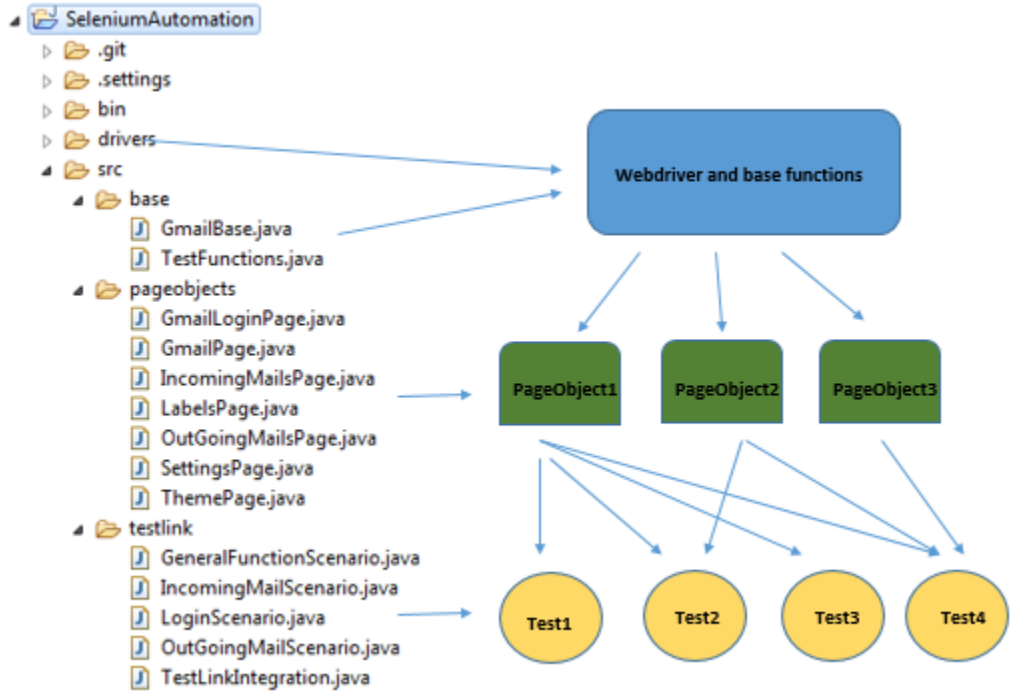
14.Ábra: TestLink integrálása Eclipse projekttel

A tesztesetek futtatása JUnit tesztesetekként történik. A JUnit annotációit jól fel tudtam használni az előfeltételek kezelésére. Így minden egyes alkalommal megtörténik a webdriver inicializálása és login tesztek kivételével a bejelentkezést is előfeltételnek tudtam megadni a @Before annotációval. Az egyes esetek lefutása során pedig mindig bezárom a webdrivert az @After annotáció segítségével. JUnit estek lévén az egyes esetek között nincs összefüggés, azok önálló külön esetekként is lefuttathatók, ami nagyban segíti az elemzést, mert egyesével is újra lehet futtatni az eseteket (14. Ábra).

```
18  public class LoginScenario {
19      GmailLoginPage gmailLogin;
20
21      @Before
22      public void initializeLoginPage() throws Exception{
23          gmailLogin=new GmailLoginPage();
24      }
25
27      public void validLogin() throws Exception {..
37
39      public void invalidUserName() throws Exception {..
48
50      public void invalidPassword() throws Exception {..
59
60
61      @After
62      public void closeBrowser() {
63          GmailLoginPage.driver.close();
64      }
65
```

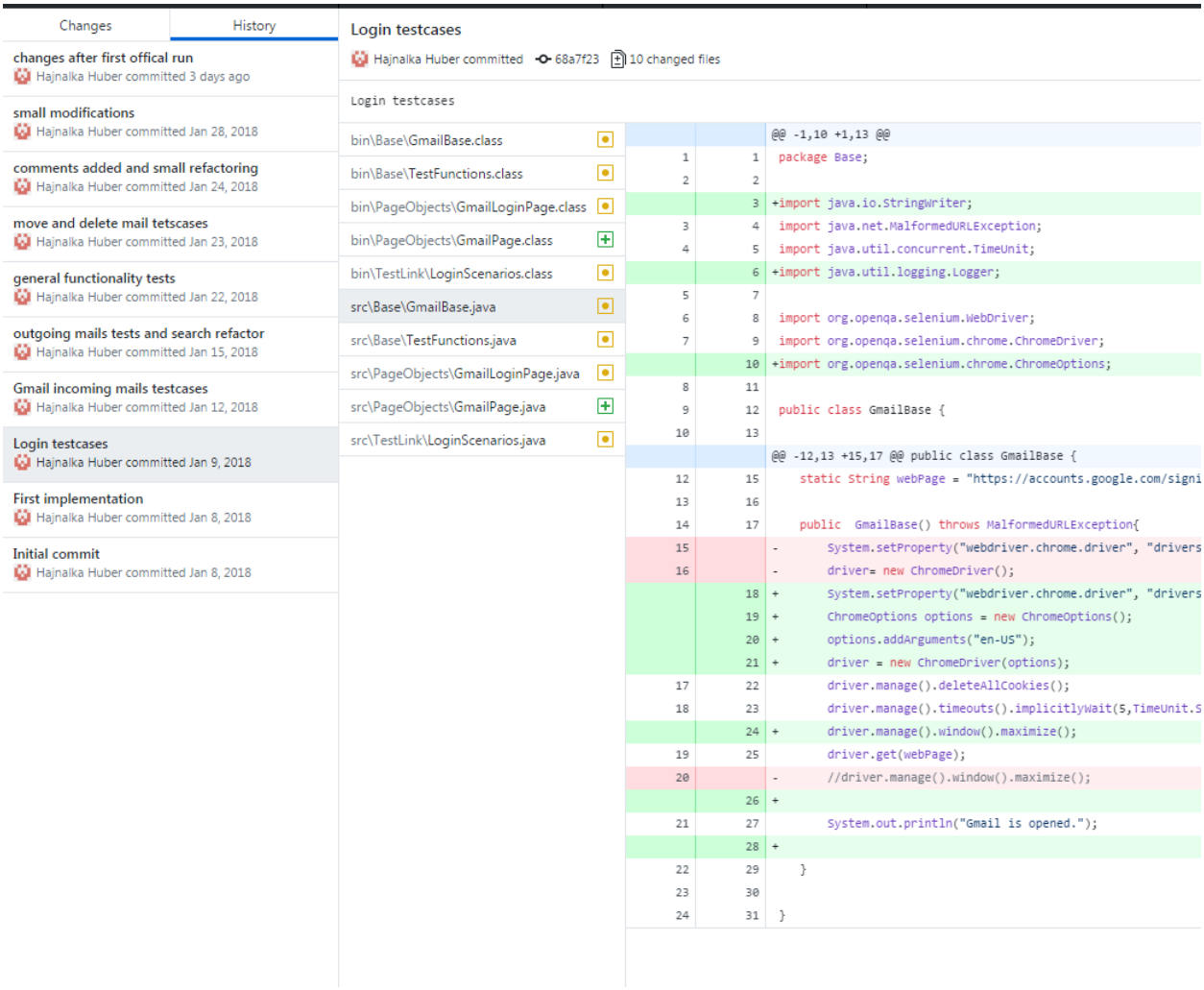
14.Ábra: LoginScenario JUnit tesztjei

A 15. ábra diagramja mutatja be a teljes struktúra felépítését.



15.Ábra: Framework diagram

A fejlesztés teljes időtartama alatt az egyes lépéseket a Git rendszerébe feltöltöttem, hogy a munkám során történt változtatásokat követni tudjam, ha szükséges vissza tudjak állni egy korábbi verzióra is. (16.Ábra)



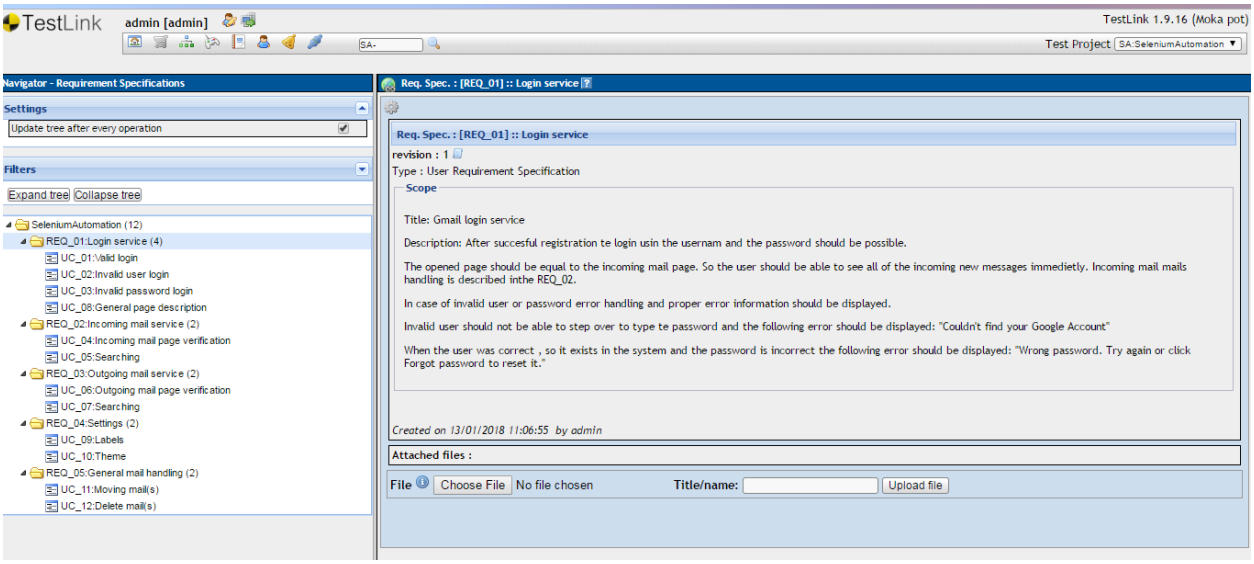
16.Ábra: Git verzió követő rendszer használata

5.3. A projekt és az eredmények a TestLinken keresztül

Ez előző fejezetben ismertettem a projektet a leimplementált kód szempontjából, fejlesztői szemszögből. Bemutattam a struktúráját és az alapvető technikákat, amiket használtam. A futtatás során létrejövő eredményeket és az implementáció alapját adó követelményeket a TestLink környezetben elehet elérni.

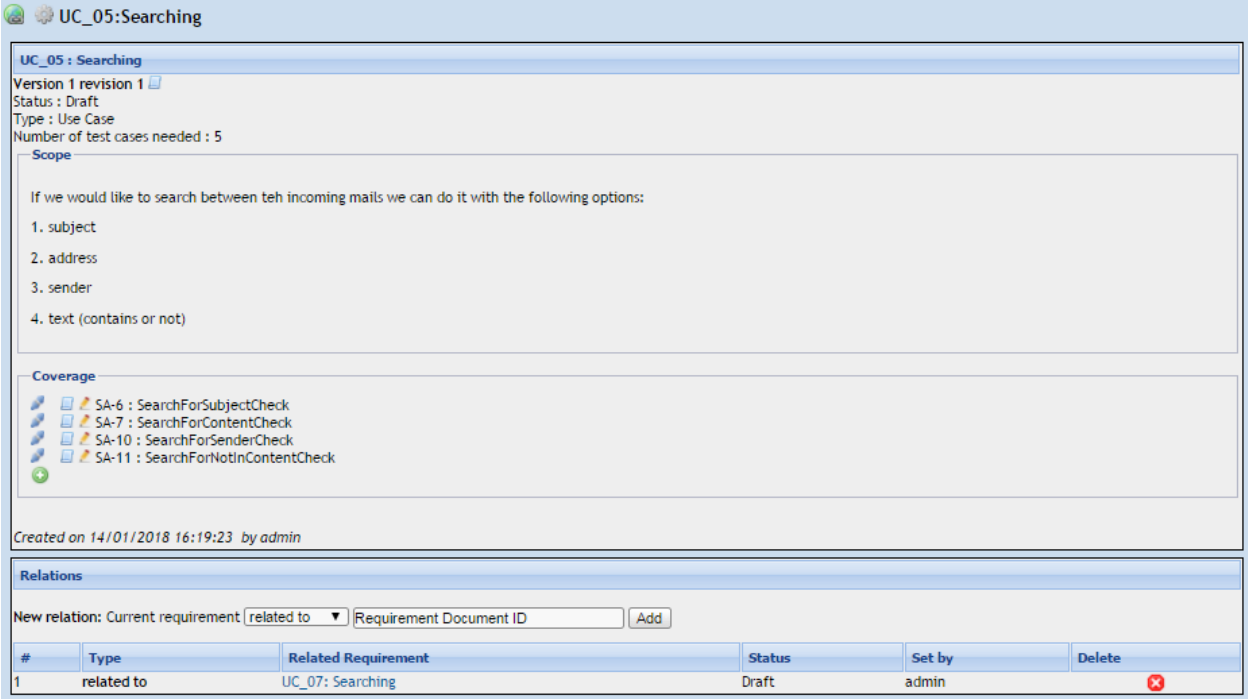
A létrehozott projektben teszttterveket ahhoz pedig a követelményeket tesztesetekhez lehet rendelni, amiket aztán pedig tesztelőkhöz lehet rendelni.

A követelményeket a funkcionalitások alapján csoportosítottam (17. Ábra)



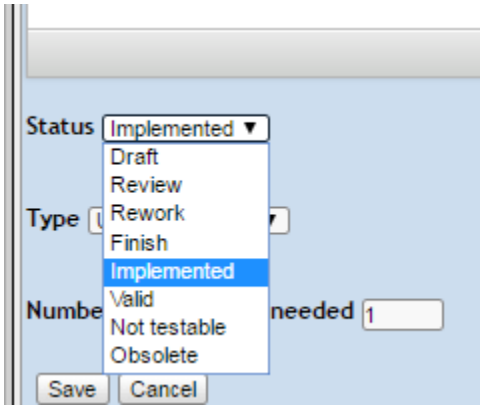
17.Ábra: TestLink követelmények

A követelmények között összefüggéseket is be lehet állítani. Például a keresés mind a kimenő mind a bejövő e-mailek esetében azonosan működik, így nem kell kétszer leírni azt. Elég egy hivatkozást létrehozni (18.Ábra).



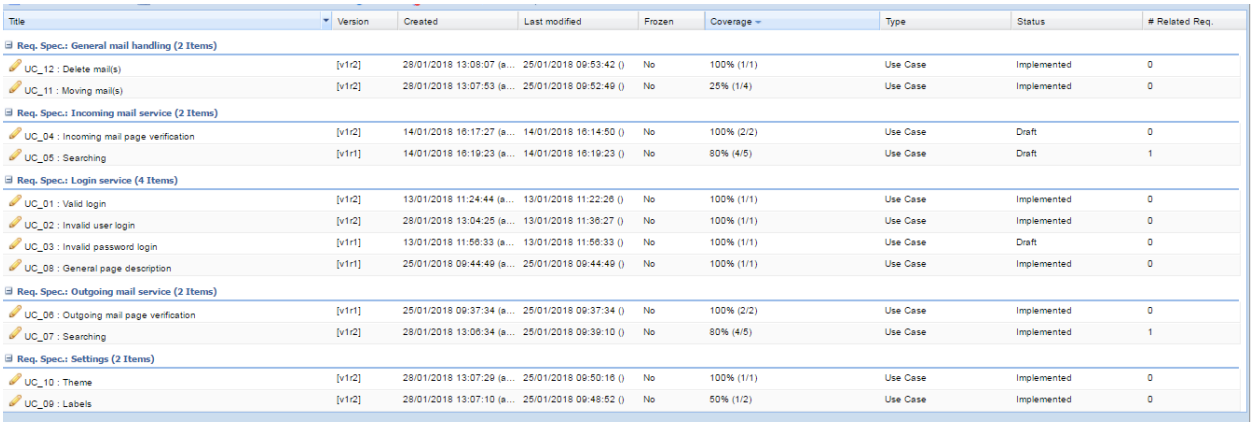
18.Ábra: UC_05 kapcsolata UC_07 követelménnyel

A követelmények státuszát is meg lehet adni, így lehet tudni, hogy melyik milyen állapotban van éppen és várható-e változás még azzal kapcsolatban (19.Ábra).



19.Ábra: Követelmények lehetséges státuszai

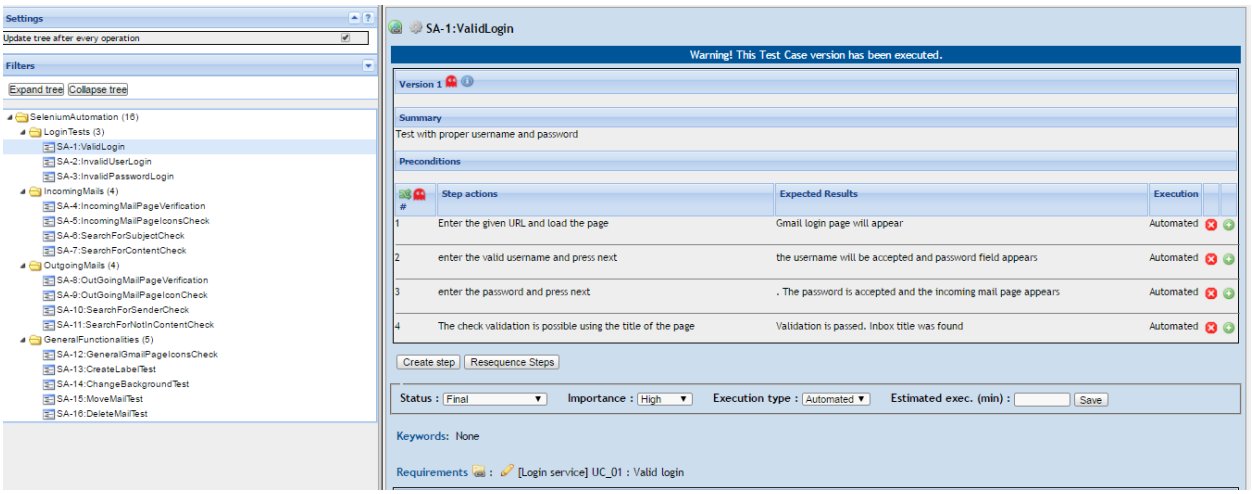
A követelmények állapotát és azt, hogy hol hiányzik m ég teszteset az Requiremnent Overview oldalon táblázatos formában is meg lehet nézni (20.Ábra).



Title	Version	Created	Last modified	Frozen	Coverage	Type	Status	# Related Req.
Req. Spec.: General mail handling (2 Items)								
UC_12 : Delete mail(s)	[v1r2]	28/01/2018 13:08:07 (a...	25/01/2018 09:53:42 ()	No	100% (1/1)	Use Case	Implemented	0
UC_11 : Moving mail(s)	[v1r2]	28/01/2018 13:07:53 (a...	25/01/2018 09:52:49 ()	No	25% (1/4)	Use Case	Implemented	0
Req. Spec.: Incoming mail service (2 Items)								
UC_04 : Incoming mail page verification	[v1r2]	14/01/2018 16:17:27 (a...	14/01/2018 16:14:50 ()	No	100% (2/2)	Use Case	Draft	0
UC_05 : Searching	[v1r1]	14/01/2018 16:19:23 (a...	14/01/2018 16:19:23 ()	No	80% (4/5)	Use Case	Draft	1
Req. Spec.: Login service (4 Items)								
UC_01 : Valid login	[v1r2]	13/01/2018 11:24:44 (a...	13/01/2018 11:22:26 ()	No	100% (1/1)	Use Case	Implemented	0
UC_02 : Invalid user login	[v1r2]	28/01/2018 13:04:25 (a...	13/01/2018 11:36:27 ()	No	100% (1/1)	Use Case	Implemented	0
UC_03 : Invalid password login	[v1r1]	13/01/2018 11:56:33 (a...	13/01/2018 11:56:33 ()	No	100% (1/1)	Use Case	Draft	0
UC_08 : General page description	[v1r1]	25/01/2018 09:44:49 (a...	25/01/2018 09:44:49 ()	No	100% (1/1)	Use Case	Implemented	0
Req. Spec.: Outgoing mail service (2 Items)								
UC_06 : Outgoing mail page verification	[v1r1]	25/01/2018 09:37:34 (a...	25/01/2018 09:37:34 ()	No	100% (2/2)	Use Case	Implemented	0
UC_07 : Searching	[v1r2]	28/01/2018 13:06:34 (a...	25/01/2018 09:39:10 ()	No	80% (4/5)	Use Case	Implemented	1
Req. Spec.: Settings (2 Items)								
UC_10 : Theme	[v1r2]	28/01/2018 13:07:29 (a...	25/01/2018 09:50:16 ()	No	100% (1/1)	Use Case	Implemented	0
UC_09 : Labels	[v1r2]	28/01/2018 13:07:10 (a...	25/01/2018 09:48:52 ()	No	50% (1/2)	Use Case	Implemented	0

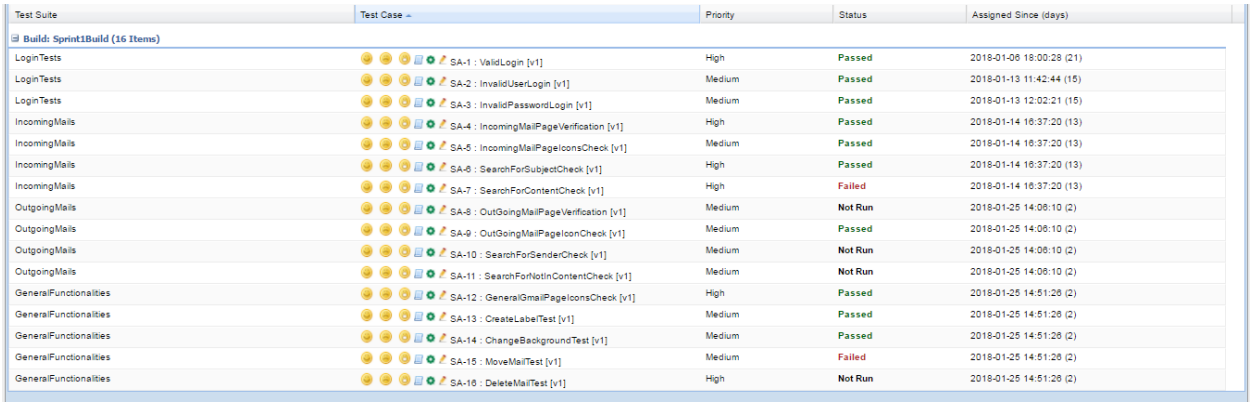
20.Ábra: Követelmények összefoglaló táblázata

A projekten én csak automatikus eseteket valósítottam meg, így a tesztesetek elrendezése megegyezik az Eclipse projektbeli elrendezéssel. Az esetekhez a lépéseket is jól meg lehet adni, ezzel is segítve az implementációt, hiszen lépésről lépésre le van írva, hogy mit kell tennie a programnak (21. Ábra).



21.Ábra: TestLink tesztesetek

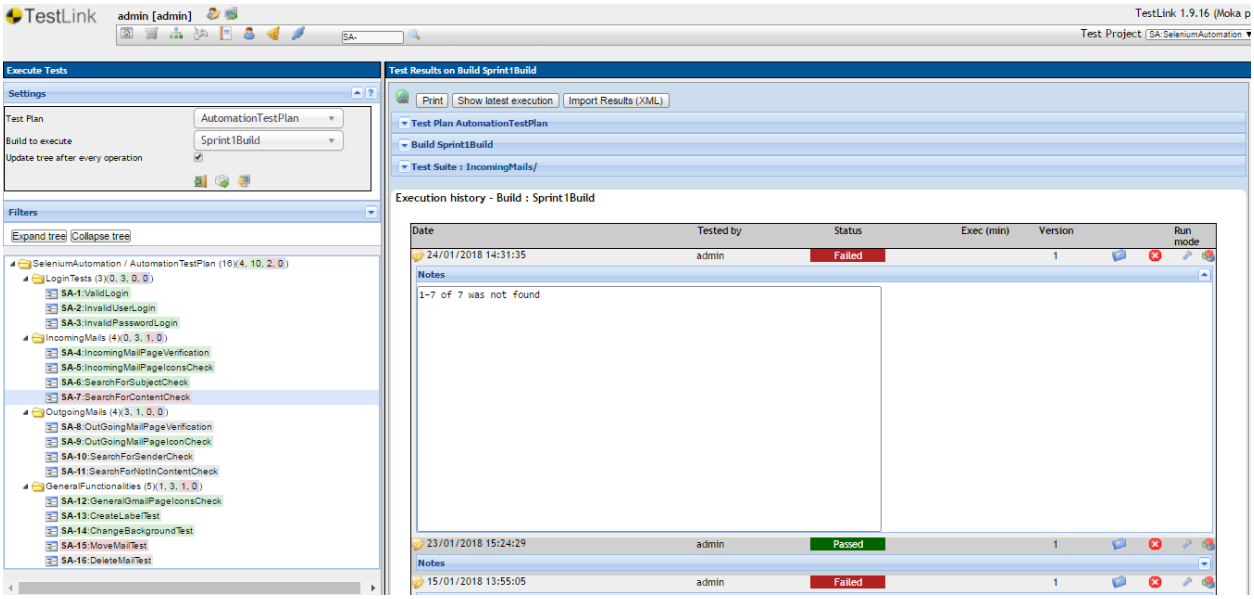
A teszteseteket jelen esetben egy userhez rendeltem, hiszen egyedül én készítettem őket. Van lehetőségem arra, hogy lássam azokat összesítve akár korábban a követelményeket. Láthatom, hogy melyik futott és az milyen eredménnyel zárult (22.Ábra).



Test Suite	Test Case	Priority	Status	Assigned Since (days)
Build: Sprint1Build (16 Items)				
LoginTests	SA-1 : ValidLogin [v1]	High	Passed	2018-01-06 18:00:28 (21)
LoginTests	SA-2 : InvalidUserLogin [v1]	Medium	Passed	2018-01-13 11:42:44 (15)
LoginTests	SA-3 : InvalidPasswordLogin [v1]	Medium	Passed	2018-01-13 12:02:21 (15)
IncomingMails	SA-4 : IncomingMailPageVerification [v1]	High	Passed	2018-01-14 16:37:20 (13)
IncomingMails	SA-5 : IncomingMailPageIconsCheck [v1]	Medium	Passed	2018-01-14 16:37:20 (13)
IncomingMails	SA-6 : SearchForSubjectCheck [v1]	High	Passed	2018-01-14 16:37:20 (13)
IncomingMails	SA-7 : SearchForContentCheck [v1]	High	Failed	2018-01-14 16:37:20 (13)
OutgoingMails	SA-8 : OutGoingMailPageVerification [v1]	Medium	Not Run	2018-01-25 14:08:10 (2)
OutgoingMails	SA-9 : OutGoingMailPageIconCheck [v1]	Medium	Passed	2018-01-25 14:08:10 (2)
OutgoingMails	SA-10 : SearchForSenderCheck [v1]	Medium	Not Run	2018-01-25 14:08:10 (2)
OutgoingMails	SA-11 : SearchForNotInContentCheck [v1]	Medium	Not Run	2018-01-25 14:08:10 (2)
GeneralFunctionalities	SA-12 : GeneralGmailPageIconsCheck [v1]	High	Passed	2018-01-25 14:51:26 (2)
GeneralFunctionalities	SA-13 : CreateLabelTest [v1]	Medium	Passed	2018-01-25 14:51:26 (2)
GeneralFunctionalities	SA-14 : ChangeBackgroundTest [v1]	Medium	Passed	2018-01-25 14:51:26 (2)
GeneralFunctionalities	SA-15 : MoveMailTest [v1]	Medium	Failed	2018-01-25 14:51:26 (2)
GeneralFunctionalities	SA-16 : DeleteMailTest [v1]	High	Not Run	2018-01-25 14:51:26 (2)

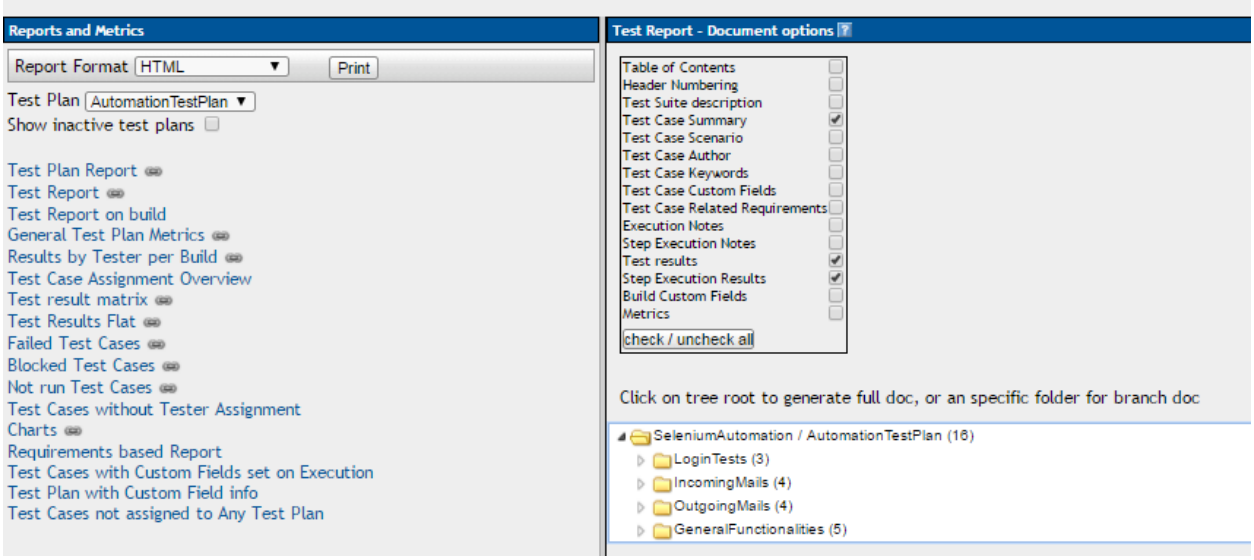
22.Ábra: Tesztesetek összegző táblázata

Az Execute Test menüpontban pedig láthatom azt is, hogy az egyes eset miért nem futott le sikeresen. Ennek az információnak a birtokában pedig ellenőrizhetem manuálisan a kérdéses lépést, majd szükség szerint javítom a teszt kódját vagy hibariportot nyitok (23.Ábra).



23.Ábra: Execute Test menüpont

Végezetül az teszteredményeket különböző riportok szerint tudjuk lekérdezni. Az alapvető beépített riportokban pedig meg tudjuk adni, hogy milyen információkat tartalmazzon (24. Ábra).



24.Ábra: TestLink riportok

A riportokat le lehet menteni, így azokat tovább lehet küldeni, nem kell feltétlenül felhasználónak lenni a programban és később is felhasználhatjuk az így lementett adatokat.

6. Összefoglalás

Szakdolgozatomban a Gmail **web applikáció automatizált teszt**jét valósítottam meg. **Célom** volt a fejlesztés során, hogy a tesztestek minél inkább egy **valódi felhasználó lépéseit modellezzék** és az elkészült projekt jól **karbantartható** és könnyen **bővíthető** legyen.

A szakdolgozatom első részében ismertettem a **tesztelés szempontjából fontos szoftverfejlesztési modelleket**, a kezdeti modellektől kezdve a napjainkban használt agilis módszertanokig bezárólag. Ezek előnyeit, hátrányait ütköztettem, majd leírtam, hogy milyen eszközök állnak rendelkezésre kezdetben majd a projekt előrehaladásával milyen **bonyolultabb módszertanok alkalmazhatók.**

A további fejezetekben megfogalmaztam a szakdolgozatom célját és az **automatizált regressziós célú tesztek** előnyeit. Ezek után **leírást adtam a kiválasztott eszközökről és indokoltam,** hogy miért ezen eszközök mellett döntöttem. Célomnak leginkább a **Selenium Webdriver** által nyújtott lehetőségek **Eclipseben** történő megvalósítása felelt meg. A kódban történt változtatások figyelemmel kísérése **Git** verzió követő rendszert használtam. A kész projekt futtatása után kapott eredményeket pedig a **TestLink** management eszközön keresztül lehet monitorozni. Leírtam a webes alapú automatizált tesztelésnél **használható módszereket**, amiket az implementáció során alkalmaztam, így a tesztek fejlesztése a **Page Object minta** alapján készült el.

A dolgozat utolsó fejezeteiben részleteztem a **projekt kialakítását és működését**. Ábrákkal és kódrészletekkel igyekeztem **informatív**vá tenni a **leírások**at. Kifejtettem, hogy a **kiválasztott eszközök hogyan működnek együtt** és hogyan használható egy teszt management eszköz a különböző információk gyűjtésére, karbantartására és rendszerezésére. Az eredményekről készült heti riportok a szakdolgozathoz csatolva megtalálhatóak.

További lehetőségként tekintek egy **hibakezelő platform** létrehozására is, melyet szintén integrálni lehet a TestLink eszközzel, ezzel is segítve az információk helyes kezelését a tesztelési-fejlesztési folyamat során.

Köszönetnyilvánítás

Munkám elkészítéséhez sok segítséget, valamint útmutatást nyújtott témavezetőm Dr. Kovács Attila.

Köszönetet mondok Neki a tervezés során adott útmutatásáért, a dolgozat megírása közben kapott sok hasznos tanácsokért, és a szakdolgozat véglegesítéséhez szükséges észrevételeiért, amelyek mind nagymértékben elősegítették munkámat a dolgozat elkészítése során. Hálás vagyok továbbá, állhatatos munkájáért, aminek köszönhetően észrevette a dolgozatom írása során elkövetett hibákat, ezzel is javítva annak minőségét és teljességét.

Örömmel dolgoztam együtt a Tanár Úrral, és hiszem, hogy az általa adott tanácsok a jövőben is a fejlődésemet szolgálják majd.

Források:

1. http://iso25000.com/index.php/en/iso-25000-standards/iso-25010 [2017.11.10]

2. https://en.wikipedia.org/wiki/Web_testing [2017.11.10]

3. http://moodle.autolab.uni-pannon.hu/Mecha_tananyag/autoipari_beagyazott_rendszerek/ch07.html [2017.11.11]

4. https://hu.wikipedia.org/wiki/Agilis_szoftverfejleszt%C3%A9s [2017.11.21]

5. http://www.tankonyvtar.hu/hu/tartalom/tamop425/0046_szoftverfejlesztes/ch02s04.html [2017.11.21]

6. https://hu.wikipedia.org/wiki/Scrum [2017.11.21]

7. http://rs1.szif.hu/~heckenas/okt/SQA06.pdf [2017.11.15]

8. Dorothy Graham, Erik Van Veenendaal, Isabel Evans, Rex Black: A szoftvertesztelés alapjai (Alvicom Kft., 2010, ISBN 978-963-06-9858-0)

9. http://www.base36.com/2013/03/automated-vs-manual-testing-the-pros-and-cons-of-each/ [2017.12.10]

10. https://en.wikipedia.org/wiki/SoapUI [2018.01.22]

11. http://searchsoftwarequality.techtarget.com/feature/Examining-the-eggPlant-Functional-app-testing-tool [2018.01.22]

12. https://www.quora.com/Which-automation-tool-is-better-in-software-industry-egg-plant-selenium-and-QTP [2018.01.22]

13. https://www.seleniumhq.org/projects/ide/plugins.jsp[2018.01.22]

14. http://www.seleniumhq.org/projects/ide/[2018.01.22]

15. http://toolsqa.com/selenium-webdriver/browser-navigation-commands/ [2018.02.11]

16. https://www.webperformance.com/load-testing-tools/blog/real-browser-manual/building-a-testcase/how-locate-element-the-page/type-element-locators/ [2018.02.11]

17. https://git-scm.com/about [2018.02.11]

18. https://en.wikipedia.org/wiki/TestLink [2018.02.11]

19. https://blogs.perficient.com/delivery/blog/2015/08/17/comparison-of-test-management-tools/ [2018.02.11]

20. https://blogs.perficient.com/delivery/blog/2017/08/04/integration-of-selenium-webdriver-with-testlink/ [2018.02.26]

21. http://elementalselenium.com/tips/22-locator-strategy [2018.02.26]

22. http://toolsqa.com/selenium-webdriver/implicit-explicit-n-fluent-wait/ [2018.02.26]

23. http://www.seleniumhq.org/docs/06_test_design_considerations.jsp[2018.02.26]