

TDK-dolgozat

2023.

Kovács Dániel Gergely
Pénzügyi és Számviteli Kar

**ALKALMAZÁS MIGRÁCIÓBÓL EREDENDŐ PROBLÉMÁK MEGOL-
DÁSA HATÉKONYAN, ERŐFORRÁS MENEDZSERI SZEMMEL**

**SOLVING ISSUES ARISING FROM APPLICATION MIGRATION EFFI-
CIENTLY WITH THE PERSPECTIVE OF A RESOURCE MANAGER**

dr. Dobák Dóra, Bükkösy Gergő

Kézirat lezárásának dátuma: 2023. november 07.

TARTALOM

Bevezetés	1
1. Tervezési fázis.....	3
1.1. A migrációból eredő probléma körüljárása	3
1.2. Megrendelői igény felmérése.....	4
1.3. Három lehetséges alternatíva a javításra	5
1.4. Alkalmazás kiválasztása az intézmény által is támogatott alkalmazások palettájából	7
2. Kód írása az automatizáláshoz.....	9
2.1. Mi az az IDE?.....	9
2.2. A robothoz szükséges elemek előkészítése	10
2.3. A menük kiírása fájlba.....	16
2.4. Mappa és fájl linkek lekérdezése	17
2.5. Fájlok beillesztése az elvárt igényeknek megfelelően a teszt szerverre.....	22
2.5.1. Lecserélendő link megkeresése	23
2.5.2. A fájl csatolása.....	24
2.6. Mappa csatolása és a robot éles futtatása	31
3. A robot sikeres futtatását követő adatok.....	33
3.1. Kivételkezelés és egyéb intézkedések a futtatáshoz	33
3.2. Statisztikák ismertetése és azok értelmezése	34
Összefoglalás.....	36

BEVEZETÉS

A TDK dolgozat célja a robotizálás alkalmazása, és annak népszerűsítése a kevésbé szokásos informatikát érintő területekre, ezzel kapacitást spórolva és hatékonyságot növelve egy cég alkalmazottjainak, és a nagy mennyiségű repetitív feladatok elvégzéséből eredendő hibázási lehetőség csökkentése.

A bankoknál jellemző, hogy jelenleg még kevés anyagi és humán erőforrás van fenntartva robotizálásnak, így azt csak limitált mértékben tudja kihasználni a cég. Az erőforrás hiány miatt jellemzően csak az adatbázis szinkronizálásokat oldják meg robotokkal, illetve a bankot közvetlenül érintő problémákat tudják csak automatizálni, így kihagyva a végfelhasználókat, akiknél a repetitív tevékenységek mennyisége lényegesen meghaladhatja az általánosabban robotizált folyamatok manuálisan történő elvégzésének idejét, és erőforrás igényét. Sok ismétlődő tevékenységet meg lehetne előzni, ha a felhasználó legalább lehetőséget kapna egy platformra, hogy a saját időigényes tevékenységeit automatizálni tudja.

A dolgozat mélységében elmerül, és körbe jár egy adott migrációs probléma megoldásának menetén. Sorra veszem a lépéseket, és intézkedéseket, amelyeket el kell végezni a folyamat sikeréhez, mint például probléma körüljárása, igény felmérés, kivételkezelés a programon belül, majd ezeket magyarázó ábrákkal mutatom be. A dolgozat rávilágít arra, hogy az ezen lehetőségek előnyösek a folyamat végrehajtójának, és mindenkinek, akiket a folyamat bármilyen formában is érinti, hiszen a robot teljesen párhuzamosan képes futni, amíg egy felhasználó használja a számítógépét, így a korábbi munkafolyamatok mellett tud segítséget nyújtani. A cégeknek is halmozottan előnyös, hiszen pontos munkavégzést kapnak a dolgozóiktól, melyet mindig időre elvégeznek, valamint, mint lentebb láthatjuk, akár ezeket mind nyílt forráskódú alkalmazásokkal is képesek véghez vinni, mely a cégek számára, így semmilyen addicionális költséget nem jelent.

A TDK dolgozat elkészítéséhez többek között a Selenium Webdriver dokumentációját, és egyéb internetes platformon elérhető tudásanyagban mélyedek el, mint szakirodalom, hiszen az alkalmazás elsajátítása nélkülözhetetlen a probléma megoldásában. A Selenium Webdriver egy olyan alkalmazás, ami szokványos API-okkal szemben, a felhasználói felületen mozog, így bármire képes, amire egy felhasználó is. Az alkalmazáshoz Java programozási nyelvet használok, hisz ez a legkézenfekvőbb tekintettel, a korábban folytatott tanulmányaimra az egyetemen.

A dolgozat során a hipotézisem, miszerint időben gyorsabb felvenni magunkra a szükséges tudást és megírni egy robotot egy nagyobb mennyiségű monoton probléma javításra, mint azt

manuálisan elvégezni. Ez a hipotézis magában foglalja azt is, hogy a robotizálás költséghatékonyabb is egy vállalat szempontjából, és értékes időt szabadít fel annak alkalmazóinak.

A dolgozat során kvantitatív kísérleti módszertant fogok használni a program elkészítésének leírása során, majd egy rövid költség-hatékonysági elemzéssel fogjuk megtudni a dolgozat kérdéseire a válaszokat.

1. TERVEZÉSI FÁZIS

1.1.A MIGRÁCIÓBÓL EREDŐ PROBLÉMA KÜRÖLJÁRÁSA

A világ bármely pontján vagyunk az informatikai rendszerek folyamatosan frissülnek, régi rendszerek elavulnak, új alternatívák jelennek meg a piacon. Bárhogyan is nézzük, időről időre minden rendszert le kell cserélni vagy fejleszteni, hogy a változó piaci elvárásoknak folyamatosan meg tudjanak felelni, és mind a cég munkatársainak, és az ügyfeleknek a legnagyobb kényelmet és biztonságot tudják adni. Egy alkalmazás lecserélésére, tehát több ok is elképzelhető, mint kiváltó ok, akár egy kívánatos funkció egy frissebb alkalmazásnál, lehet egy nem támogatott szoftver vagy akár egy kedvezőbb árazású rendszer, de bármi is váltja ki a változást, az ehhez kapcsolódó adatok migrációja, amennyiben fennáll, minden esetben nehézkes, és sohasem egy probléma mentes folyamat.

Ami a pénzügyintézeteket illeti, egy modern bank, mint tudjuk, egy komplex belső és külső auditál, és nagyon nehezen áttörhető biztonsági rendszerekkel rendelkező cég. A bankok is tehát kénytelenek nap mint nap újabb és újabb kihívásokkal szembe nézni, ami az informatikai rendszereket is illeti, és változásokat kezdeményezni, újabb, hatékonyabb, és biztonságosabb rendszerek felé. Ezekre kíván lehetőséget nyújtani az Atlassian a termékeivel.

Az Atlassian egy olyan vállalat, amely világszerte elismert a projektmenedzsment és együttműködési szoftverek terén. A vállalat az ügyfelek számára különféle eszközöket és alkalmazásokat kínál, amelyek segítenek a csapatoknak hatékonyabban együttműködni, projektjeiket követni és a munkafolyamatokat irányítani. Az Atlassian által kifejlesztett számos alkalmazás között két kiemelkedő platform emelkedik ki: a JIRA és a Confluence. Az Atlassian termékeit 2023-ra szinte az összes kisebb-nagyobb cég már használja a világban. A Jira egy feladat nyilvántartásra optimalizált alkalmazás, illetve a Confluence, amely egy tudásbázist képes fenntartani különböző cégek számára összekapcsolódva az Atlassian egyéb termékeivel. Az ezen alkalmazások által nyújtott lehetőségek több más alkalmazás kiváltására is képes, mely kellő körüljárást követően képes az intézményeknek pénzt és időt spórolni, valamint hatékonyságot növelni hosszú távon, hisz az alkalmazottakért fizetett licenz csak egy alkalmazást fog jelenteni, valamint csak egy alkalmazást kell karbantartani, így annak megtanulása egy egyetemes tudást nyújt a képességek elsajátítójának az összes szervezeten belüli rendszerhez.

A kedvező felételekből kifolyólag a bank fiókhálózatában dolgozó kollégák tudásbázisának migrálása már korábban megtörtént a Confluence alkalmazás korábbi, serveres platformjára, mely viszonylag gond nélkül lefolyt. Az Atlassian által serveres platformon nyújtott szolgáltatások azonban az Atlassian üzletpolitikájának változása miatt kivezetésre kerül, így a

management döntés alapján elindult a felhőbe költözés folyamata, mely alapos körüljárást követően is, a migrációt követően, már csak a tudásbázis méreteiből is adódóan, problémák jelentkeztek.

A fiókhálózat támogatására vonatkozó tudásbázisban egy részében felhasználói oldalról történő rossz gyakorlat miatt rengeteg lokális fájl csatolás volt, mely fájlokra, illetve mappákra mutatott, mely támogatottsága már 2012 óta a chromium alapú böngészőknél nem volt támogatott, így az elmúlt években is kerülő megoldásokkal próbáltunk meg segíteni a tudásbázis felhasználóinak (www.developer.chrome.com). Az eredeti használat akkor lenne lehetséges, ha hozzáférnénk, és módosítani tudnák az oldal forráskódját, de ehhez nincs lehetőségünk, hiszen csak végfelhasználói hozzáféréssel rendelkezünk a termékhez.

A migrációt követően a lokális fájl linkek csatolásának támogatottsága azonban végleg kudarca fulladt, az újabb chromium alapú böngészők tűzfal szabályai miatt, valamint a felhős adatbázisú Atlassian termékek sem támogatják az ekképpen történő hivatkozásokat, főként biztonsági okokra hivatkozva. Az adatok átvitelét REST API-al vitték véghez az erre szakosodott kollégák, viszont a migrációs folyamat közben lévő tesztelés során a tartalmi megfelelőséget figyelve a nem csatolódott hivatkozások figyelmen kívül maradtak, így azok nem kerültek becsatolásra. Az így, látszólag teljes egészében, gond nélkül bemigrált adat mennyiség jelentős értéke, használati szempontból nézve, elveszett. Minden esetben, ha egy probléma adódik egy bankon belül bármely alkalmazással, a megoldását az érintett alkalmazás felelősenek kell kezdeményeznie.

1.2.MEGRENDELŐI IGÉNY FELMÉRÉSE

Az így keletkezett károk felmérésének az első lépése, egy alkalmazás felelősenek, körbe járni, hogy hogyan tudna az alkalmazást használóinak minél jobban az igényeire szabott megoldást nyújtani, mely kimeríti minden munkához szükséges elvárásukat. A fiókhálózatban dolgozó kollégák számára a két megoldás a fájlok feltöltése a tudásbázisba, vagy a hivatkozási linkek szöveges módon történő csatolása adta.

Itt fontos megemlíteni, hogy a fiókhálózatban mozgó dokumentumok rendszeresfrissítése miatt volt a korábbiakban a mappa csatolása, hisz egy fájl így több felé is be lehetett csatolva, és csak egyszer kell, az adott mappában lecserélni, és a mappa, vagy fájl csatolásnak köszönhetően a tudásbázisban az összes, a témában érintett oldalon frissítve lettek a dokumentumok.

Mind a két megoldás a probléma megkerülésére szolgál, azonban nem tökéletes.

Ha az elérési útvonalakat csatoljuk, az azzal a pozitívummal jár, hogy az elérés úgymond, mindig automatikusan frissül, és a tudásbázis mérete is csak a beszúrt szöveg méretével nő, mely minden biztonsági mentés sebességét megnöveli, a felcsatolt dokumentumokkal szemben,

illetve a probléma megvalósításának pillanatától kezdve az összes jövőbeni migráció sebességét növelné, miközben a hibalehetőségeinek mennyiségét csökkentené. A két legnagyobb probléma, azonban az ily módon történő csatolásnak, hogy amennyiben a dokumentum neve egy betűvel is megváltozik, a csatolás szövegét módosítani kell. Ilyen lehet például bármely dokumentum, mely csak adott időszakokra vonatkoznak, és dátum szerepel a dokumentum nevében, mely minden frissítéssel változik, valamint a fájlok nevében is verzió kezelt dokumentumok. Amennyiben a csatolt mappa vagy fájl neve módosul, minden oldalon újra kell írni a csatolást, ami nagyban lecsökkenti az ezen lehetőség által nyújtott előnyöket.

A végfelhasználói élmény sem túlságosan kifinomult az ekképpen történő csatolás esetében, ahol minden alkalommal, amikor a csatolást szeretné használni a fiókhálózatban dolgozó kolléga, egy fájlkezelőt meg kell nyitnia, majd bemásolnia a dokumentum elérési útvonalát, ezzel fenntartva az emberi tényezőkből adódó hibák lehetőségét is, mely nagyban lassítja a kolléga munkavégzésének sebességét is, mindamellett, hogy érthető frusztrációt is okozhat hosszútávon. Az e módon történő csatolás, tehát egy gyakran frissített dokumentum esetében értelmét veszítheti a változó fájlnev miatt, a gyakran nem változó dokumentum esetében, pedig, a frissítés frekvenciája miatt, akár a későbbiekben a kézzel történő cseréje sem jelent nagy gondot.

A második opció a felcsatolt dokumentumok, mely két hátránya, hogy nem dinamikusan frissülnek a dokumentumok a lokális fájlokkal szemben, ámbár ezen funkció hosszútávon történő használata a fentiekből kifolyólag gyakorlatilag teljes mértékben figyelmen kívül hagyható. A másik probléma a fájlok feltöltésével a tudásbázis bitben mért mérete, mely így is a teljes bankhálózat tudásbázisának majdnem felét teszi ki, így még nagyobb méretekre számíthatunk a tudásbázis hosszú távú használatát és bővítését követően, az elkövetkezendő évek során.

Hosszas egyeztetés után, mind két fél számára a legoptimálisabb megoldásnak az bizonyult, hogy minden dokumentumot fel kell tölteni a tudásbázisba, és a későbbiekben a tudásbázis szerkesztői minden egyes dokumentumot annak frissítésének idejében, a tudásbázisban felcsatolt példányát is aktualizálja, hogy a Confluence oldalt használók mindig a legfrissebb adatokkal tudják ellátni az ügyfeleket.

1.3.HÁROM LEHETSÉGES ALTERNATÍVA A JAVÍTÁSRA

Az elsősorban felmerült lehetőségek a

- manuális javítás
- hagyományos API-okkal
- tesztautomatával.

A felmerült probléma megoldása nem bonyolult, és manuálisan is teljes mértékben végig vihető, azonban a tudásbázis számosságát is figyelembe véve, nagyon nagy számokról van szó, mely több kollégának is több napjába telne, mire manuálisan javítani tudná az elérési útvonalakat érintő hibákat. Az efféle munkákra megfelelő erőforrás nagyon kevés van bármely cégnél, így az ez irányban történő elindulás lehetséges, de időigényes, mind erőforrás keresés szempontjából, mind a feladat végrehajtásának folyamatából.

A több ezer oldalas tudásbázis bármely pontján lehet egy hivatkozás, mely folyamatos összefésülése az eredeti felülettel időigényes, és a lehetséges hibalehetőségek mennyisége sem elhanyagolható, mely a manuális, emberi munka általános ismérve.

A hagyományos API-okkal történő javítás nem megvalósítható, hiszen a weboldalra megfelelő szövegrészek helyére kell behelyettesíteni a feltöltendő fájlokat, azonban a közismert REST API is csak oldalak aljára képes tartalmat felcsatolni (www.confluence.atlassian.com). A leg-egyszerűbb megoldás, így tehát el kell vetni, és esetleges egyéb, kerülő megoldások után kell nézni, amelyeket akár az intézményen belül is kivitelezni tudunk.

A REST API így kizárható mint alternatíva, és a megoldás csak a humán erőforrás alkalmazása, vagy pedig a folyamat lerobotizálása lehet. Robot alkalmazása esetén elég egy embernek foglalkoznia a robottal, és annak kódját jól megírni, ahhoz, hogy a robot gond nélkül le tudjon futni, így ezzel, ha időt nem is, de akár humán erőforrást tudunk spórolni.

Robotizálás terén a három lehetséges fő út van, a UI Path, mely nevéből is árulkodva felhasználási területéről, képes a felhasználói felületet használni, és azon egy low-code megoldással programozható kód sort létrehozva, mely egyesével lépteti majd az oldalakat, és cseréli le a linkeket, mintha egy felhasználó végezné azokat. A bankban azonban egy új program bevezetése nem lehetséges licenz vásárlása és megfelelő átvilágítási folyamat nélkül. A program licensze azonban nagyon drága, és beszerzése időigényes, mely nagyban elvenne a javításra szánt időkeretből, ezzel hátráltatva a fiókos dolgozók napi munkavégzését.

Két alkalmazás közül lehet választani, melyek közül az egyik a BluePrism, a másik pedig a java alapú Selenium. Mind a kettő alkalmazás webes tesztautomatizmusok létrehozására alkalmas. A BluePrism azonban a bankon belül rendszeresen történő adatok átvitelére alkalmazott, napi szinten, melyhez hozzáférést, a nem robotizálással foglalkozó kollégák nem kaphatnak. A Selenium részben és egészben jelen van a bank működésében, így ennek megigénylése bármely alkalmazott számára elérhető a bankon belül, és használható a saját folyamatainak robotizálására. A Selenium robot és a UI Path robot is tud futni a felhasználó számítógépén, így az külön

hardveres erőforrás kérést nem igényel, ezzel is csökkentve a folyamat végbeviteléhez szükséges időt.

Hosszas kutatást követően a Selenium robot látszott a legjárhatóbb útnak, így a robot programkódjának megírása volt már csak hátra.

A robot megírása némi programozási alapismeretet igényel, objektum orientált gondolkodásmódot, agilitást, és jó probléma megoldó készséget. Minden pénzintézet mára nagy létszámú informatikai részleggel rendelkezik, mely alkalmazottjai részben vagy egészben korábbi tanulmányaik során elsajátítottak már valamelyest programozási alapismereteket.

1.4.ALKALMAZÁS KIVÁLASZTÁSA AZ INTÉZMÉNY ÁLTAL IS TÁMOGATOTT ALKALMAZÁSOK PALETTÁJÁBÓL

A két alkalmazás, mely rendelkezésre áll tehát az intézményben, a BluePrism és a Selenium alapú WebDriver. A BluePrism, amelyhez való hozzáférhetőség korlátozott a nem e területen dolgozó munkatársak számára, nem mellesleg egy licenszköteles alkalmazás, így annak alkalmazása, főleg akár egy, mint a jelen esetben, egyszer lefuttatandó robot esetében nem éri meg a banknak, hogy licenszvásárlás mellett döntsön, valamint a licenszvásárlással való idővesztés sem elhanyagolható.

Egy másik lehetőség a Selenium, amely egy ingyenes, nyílt forráskódú eszköz, amelyet a webes alkalmazások tesztelésére használnak. A Selenium lehetővé teszi a fejlesztők számára, hogy automatizálják a böngészőkben végzett műveleteket, például űrlapok kitöltését, gombok megnyomását és az oldalak navigálását.

A Selenium WebDriver egy olyan interfész, amely lehetővé teszi a Java alkalmazásoknak a böngészőkön keresztül történő vezérlését és az ott végzett akciók automatizálását. A Selenium WebDriver segítségével könnyedén létrehozhatunk és futtathatunk automatizált teszteket, amelyek ellenőrzik az alkalmazások működését és stabilitását.

A Java programozási nyelv kiválóan illeszkedik a Seleniumhoz, mivel lehetővé teszi a kódolók számára a könnyű és hatékony tesztek írását. A Java segítségével könnyedén elérhetjük a Selenium WebDriver funkcionalitását, és használhatjuk a különböző metódusokat a böngészők vezérlésére és a tesztesetek végrehajtására.

Az alkalmazások fejlesztése során fontos, hogy rendszeresen teszteljük azokat, hogy biztosak lehessünk a kód minőségében és a funkcionalításban. A Selenium és a Java együttes alkalmazása lehetővé teszi a hatékony és megbízható tesztet tudjunk írni.

A Selenium WebDriver és a java nyelv különféle alosztályai képesek tökéletesen kiegészíteni egymást, abban az esetben is, ha kerülő megoldást kell találni egy automatizálandó problémára.

A Selenium tehát egy ingyenesen letölthető szoftver, így nem licenszköteles, és beszerzése is igen egyszerű. A tudása ugyanakkor tökéletesen megfelel a jelen automatizálandó folyamat kivitelezésére.

2. KÓD ÍRÁSA AZ AUTOMATIZÁLÁSHOZ

2.1.MI AZ AZ IDE?

Minden kód megírásához valamilyen szöveges felület alkalmazása elengedhetetlen. Az utóbbi időben elterjedtek a low-code és no-code alkalmazások, mint a fentebb is említett UIPath, amelyek pont ezen próbálnak módosítani, azonban ezek elterjedését nagyban gátolja az, hogyha az illető bármi komplexebbet szeretne készíteni az előre integrált elemeken felül, akkor kénytelen lesz visszalépni egyel, és kódoláshoz folyamodni. Nagyvállalati szinten sem működőképesek, hiszen a low-code és no-code alkalmazások nagyon nagy hardverkövetelményekkel rendelkeznek, amennyiben komplexebb folyamatokat szeretnénk bennük leprogramozni, és felhasználhatóságuk is limitált, így a hagyományos, high-code módszeren kell, hogy alapuljon a program. A program megírása bármilyen platform igénybevehető, mint például a Microsoft Word szövegszerkesztő alkalmazása, vagy egy egyszerű szöveges dokumentumban is meg tudjuk írni a kódot, különböző kerülő megoldásokkal, mint például Word-ben megírt kódot VBA makrók segítségével tudunk futtatni, közvetlenül a szerkesztőből (www.github.com, christophebedard). Ideális esetben bármely kód megírásához IDE-t használunk, amely az „Integrated Development Environment” rövidítése, és magyarra fordítva Integrált fejlesztői környezetnek felel meg. Ezek a kódszerkesztő alkalmazások rengeteg funkcióval bírnak minden programozó számára, mint a kód szerkesztése automatikus tördeléssel, szintaktikai hibák kiemelése, automatikus kitöltések, kódban lévő hibák keresése, és a különböző csomagok importálása a szerkesztőből (www.codecademy.com). Minden szerkesztőnek megvan a saját erőssége és gyengesége, és különböző bővítmények különböző szerkesztőkkel jobban működnek, de végső soron akár személyes preferencia, akár elérhetőség befolyásolhatja a döntésünket.

A robot megírásának nyelvének kiválasztása szinte teljes mértékben preferencia kérdése, de az egyszerűségéből és elterjedtségéből kifolyólag a legideálisabb a Java nyelv, még pedig azért, mert az interneten megtalálható oktatóanyagok képessé teszik a kód megíróját, hogy teljes mértékben önállóan elsajátíthassa a számára szükséges képességeket az automatizálás megírásához.

A robot kódjának megírásához a Java nyelv elsajátítása nélkülözhetetlen. A java könnyen tanulható, és könnyen bővíthető különböző plugin-ekkel, és rengetek jó kód szerkesztő alkalmazás támogatja, így kedvünkre választhatjuk ki a számunkra megfelelő IDE-t. A java rengeteg osztályt tartalmaz, melyek funkcionálisan kiegészítik a nyelv által nyújtott alap lehetőségeket. A Java nyelv is, a többi ismert, elterjedt programozási nyelvhez hasonlóan, képes „ArrayList”-eket kezelni, aminek funkciója elengedhetetlen egy automatizálásra használt webtesztelő alkalmazás futtatásához. A különféle adatok dinamikus gyűjteménye adhatja meg akár a robotunk

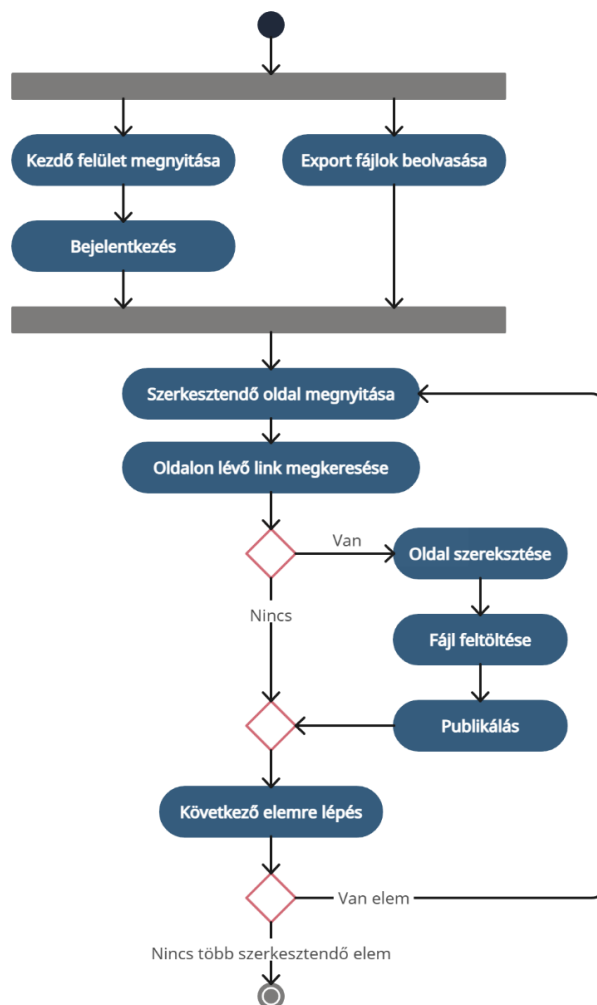
kimeneti adatainak összességét, de akár tárolhat átmeneti adatokat is, melyeket a programkód futtatása során hasznosít fel.

2.2.A ROBOTHOZ SZÜKSÉGES ELEMELŐKÉSZÍTÉSE

A robot megírásában az első fázis a problémás felület körüljárásával egyezik meg. A felhasználói felületen lévő gombok, beviteli mezők és egyéb felhasználói felületen lévő elemek a robot által manipulálható állapotba hozása elengedhetetlen egy tesztautomata megírásához.

A robot megírásához sorra kell venni, hogy mit végezne el egy ember a folyamat véghezvitele során, és azokat a lépéseket, amennyiben körbeírható és konkrét lépések, kell a robotot megírónak feltérképeznie, fejben strukturálnia, és megvalósítania. Amennyiben látszólag a folyamat nem diszkrét pontokból áll, abban az esetben valamilyen kerülő megoldással konkretizálni kell a lépéseket. A robot egyik fő lényege végtére is, hogy a monoton munkavégzést vegye le a robotot használó vállaról, így a nem beskatulyázható eseteket kézi megoldással is véghez tudjuk vinni, akár amíg a robot a nagy mennyiségű, ismétlődő eseteket hajtja végre.

1. sz. ábra. A teljes folyamat áttekintése



Forrás: Saját készítésű ábra

Az első fázis jelen esetünkben a Confluence tudásbázis banki oldalának megnyitása, ezt követően bejelentkezés, majd az összes oldalon történő szisztematikus léptetés, és összes link összehasonlítása a korábbi verzióval, és ahol eltérés van, tehát nincs becsatolva link, ott a megbeszéltek szerint a szükséges folyamat végrehajtása, majd az új, frissített tudásbázis oldal publikálása.

A korábbi szerveres verzió megnyitása szükségszerű volt, hogy a fiókhálózati munkatársak képesek legyenek elvégezni mindennapi munkájukat, amíg a fájlok feltöltése zajlik, illetve az eredeti, szerveres oldal az új, felhős tudásbázis felületével egyszerre történő összefésülése miatt is elhanyagolhatatlan volt, hiszen referencia nélkül az informatikai részlegen dolgozó alkalmazottak részéről, a tudásbázisban történő tartalom szakmai értelmezése nem elvárható.

Mivel a robot képtelen az önálló gondolkodásra, és csak az általunk megírt kód szerint halad, így a fentebb említett lépések elvégzésére különleges megoldások alkalmazására van szükség.

A robot a szisztematikus léptetésre az egymás alatt elhelyezkedő menüpontokra ily módon nem képes reagálni, így már a menüpontok léptetésénél egy kerülő megoldást kellett alkalmazni.

A szerveres és a felhős tudásbázis felületek összehasonlítására sem képes a robot, így erre is egy kézzel fogható, konkrét paraméterre van szükség, hogy tudja, hogy a robot, hogy mit keres.

Az első problémára a megoldás a Confluence oldalak menürendszerének letöltése jelenti, mely az alábbi lépésekből áll össze:

1. az aloldalak lenyitása
2. tudásbázis összes oldalának linkjének lehívása
3. adatok kimentése egy fájlba

Minden oldal alatt elképzelhető, hogy van egy aloldal, és nem tudni, hogy hány szinten vannak aloldalak, így a teljes menü látszólag történő kinyitása kell, hogy legyen az első lépés.

2. sz. ábra. A tudásbázis menürendszerének felépítésnek ábrázolása

- 6. Minta hitel
- ▼ 7. Minta oldal
 - ▼ 7.1. Minta aloldal
 - ▼ 7.1.1. Aloldalak mintája_v1
 - 7.1.1.1. Minta aloldalak_v2
 - 7.1.1.2. Aloldalak mintája_v2
 - 7.1.2. Aloldalak mintája_v3

Forrás: Saját készítésű ábra

Az oldalhoz tartozó szöveg, melyre kattintásra eljutnánk az általa képviselt oldalra, csak akkor jelenik meg, ha a szülő oldalának tartalma le van nyitva. Ha már le van nyitva, és megjelenik a gyerek oldalak szövege, akkor a HTML kódban is meg fognak jelenni a szülő, és gyerek oldalak hivatkozásának linkjei.

Következő lépés az oldalak linkjeinek kigyűjtése egy „ArrayList”-be, amit a későbbiekben ki fogunk írni egy txt fájlba, ami egy egyszerű szöveges dokumentum kiterjesztésű fájl, a robot futtatásának egyszerűsége érdekében.

A tervek megvalósításához három különböző felületen kell végig futtatni az oldal lekérdező robotot, a volt, serveres felületen, a felhős teszt felületen és az éles felhős felületen. A felhős felületek megegyeznek, azonban a robot teszten történő futtatásához nélkülözhetetlen a teszt felület oldalainak linkjeinek lekérdezése, hiszen minden nagyobb cégnél a megrendelőnek, jelen esetben a fiókhálózatban dolgozó, Confluence felületet szerkesztőknek jóvá kell hagynia egy teszt felületet, amin látják, hogy a robot sikeresen végig futott, és előre tudhatják, hogy mit fognak kapni az éles felületen, amely később munkavégzésük nélkülözhetetlen eleme lesz.

A serveres, régi felület és a felhős felület azonban eltér megjelenésre is, azonban, ami nagyobb gond a robot szempontjából, hogy HTML szinten is eltér, így kell külön írni egy robotot a serveres felületre és külön egy robotot a felhős felületre.

A serveres és a felhős felületen is kétféleképpen haladhatunk végig az aloldalak lenyitásához, és mind a kettő tökéletesen működőképes megoldás. A nyíl szimbólumokra kell kattintania a robotnak, amelyek az oldal címek előtt állnak, így bírják lenyitni az oldal alatt lévő gyerek

oldalakat. Ha a robot az oldal címére kattintana, akkor az oldal megnyitásra kerülne. A nyíl szimbólumok lenyitása nem frissíti az oldalt, így ezt a folyamatot minden egyéb DOM újra bekérés nélkül véghez tudjuk vinni.

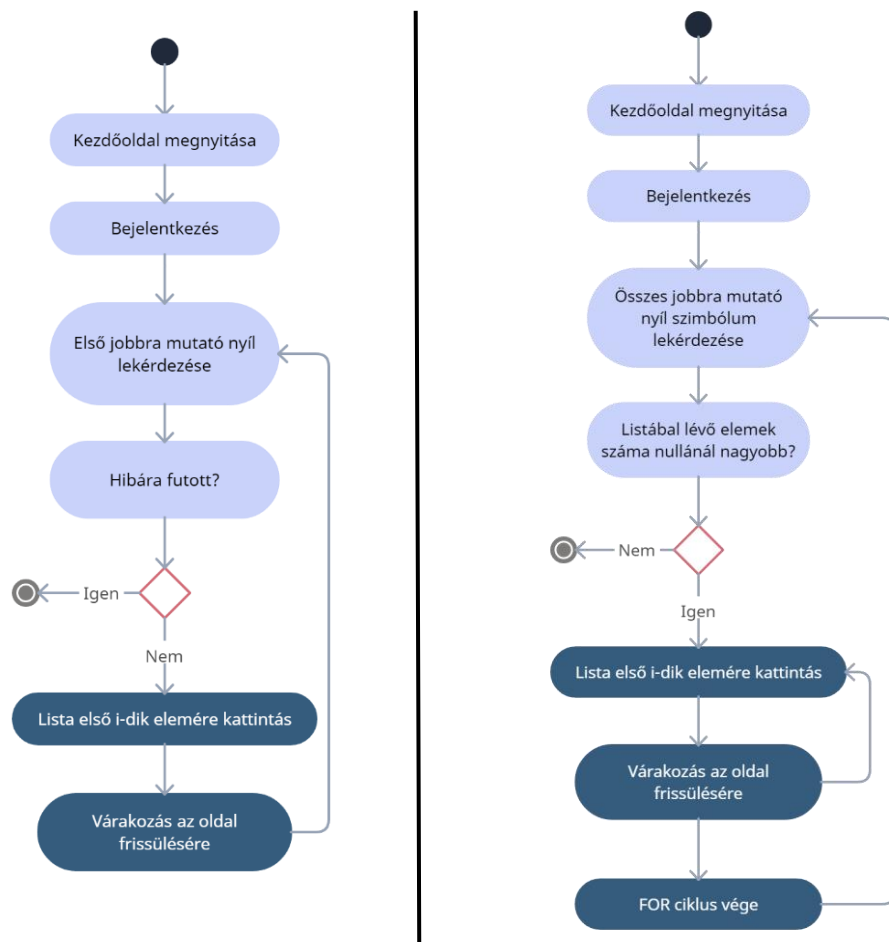
A DOM a „Document Object Model” rövidítése, magyarul Dokumentum Objektum Modell. A DOM-ot webfejlesztők használják ahhoz, hogy módosítsák egy weboldal struktúráját, stílusát és tartalmát. Így a DOM a Selenium WebDriver-ben fontos komponensnek számít a HTML5 és JavaScript használatával történő webfejlesztésben. (<https://www.guvi.in>, Lahari Chandana). Egyszerűbben fogalmazva a DOM egy olyan fastruktúra, amely az oldal tartalmát képezi le, és mindegyik elemnek van egy saját helye és tulajdonságai. Egy weboldalon, például, a DOM reprezentálja a szövegek, képek és linkek helyét és tulajdonságait, amelyeket egy böngésző megjelenít. Ebben a DOM-ban képes az általunk is használt Selenium beazonosítani egyértelműen elemeket, amelyeket később manipulálni tud, különböző módokon.

Az első lehetséges mód, az, ha a jobbra mutató, tehát még le nem nyitott nyíl osztályát kerestjük meg a HTML kódban a robottal minden kattintás előtt, majd arra közvetlenül az első megtalálását követően rá is kattintunk. Itt fontos kiemelni, hogy minden kattintást követően fontos, hogy a robot várjon egy kis időt, hiszen a robot nagy sebességgel tudja a kódsorban leírt feladatokat futtatni, azonban a felület nem tud olyan gyorsan reagálni a kattintásokra. Ezt a legkönnyebb ebben az esetben egy `Thread.Sleep()` metódussal tudjuk megtenni, ahol a zárójelben kell feltüntetni, hogy hány ezred másodpercre szeretnénk, ha a kód futtatása szünetelne. A kód így egy végtelen ciklusként tud futni, majd ha megfelelő idő után nem talál a keresésnek megfelelő elemet, akkor „`NoSuchElementException`”-t fog visszaadni hibaüzenetként, amit meg is tudunk fogni, ha az egész végtelen ciklust egy try-catch metódusban írjuk meg, ahol a catch részében kivételnek az előbb is említett lehetőséget használjuk, majd a végtelen ciklusból egy „break” funkcióval ki tudunk lépni, így megszakítva a végtelen ciklust hibaüzenet nélkül. Nagyon fontos, hogy hibaüzenet nélkül ki tudjuk nyitni az összes almenüt, hiszen, ha hibára fut a robot, akkor nem tudja teljesíteni a kód további részét, ami nélkülözhetetlen a tovább haladáshoz.

A másik lehetőség, hogyha a robot szintén egy végtelen ciklusban van, lekérdezi az összes megfelelő osztályú elemet, és egy „`WebElement`” típusú listába rakja őket, majd egy FOR ciklus végig kattintja az összes elemet egyesével, megfelelő időt kihagyva minden kattintás között, hogy a weboldal be tudja tölteni a megfelelő részleteket. Ez a változat szebb, annyival, hogy amennyiben nincsen egy elem sem a listában, tehát nincs több jobbra mutató nyíl osztályú elem,

a végtelen ciklust meg tudjuk állítani, és nem kell a hibaüzenetre hagyatkozni. Bármely lehetőséget választhatjuk, mind a kettő lehetőség járható és működőképes.

3. sz. ábra. Az almenük lenyitásának két lehetséges módjának folyamat ábrája. Bal oldalon az először felvezetett mód, jobb oldalon a második, „szebb” mód



Forrás: Saját készítésű ábra

A „WebElement” egy olyan objektum a Selenium tesztelési keretrendszerben, amely a weboldal HTML elemét reprezentálja, egy eleme a DOM-nak. A „WebElement” lehetővé teszi a webes elemek azonosítását és azokkal való interakciót, például kattintást vagy szövegbevitelt a tesztelés során. Ez az objektumtípus minden információt eltárol az adott objektumról, amit képvisel, ami a HTML kódban is megtalálható, így ezek az információk bármikor visszanyerhetők, csoportosíthatóak, és esetlegesen módosíthatóak is lehetnek (www.selenium.dev-A).

A nyílak lenyitását követően a következő lépés az, hogy a menü elemek nevét, ami jelen esetben az oldalak címe, és azok elérésének linkjeit egy kettő dimenziós listába írassuk. A mátrixban az első oszlop tartalmazza az oldalak megnevezését, a második oszlop az oldalak linkjeit. Mind a két információ nagyon fontos, mert a szerveres esetében nagyon fontos tudnunk, hogy mi a kiinduló oldalnak a címe, illetve a link, a szerveres esetében, a következő lépés miatt

nélkülözhetetlen, de a visszakereshetőség szempontjából is fontos. A felhős verziók esetében az oldalak címe adhatja majd meg az összehasonlítási paramétert, ami alapján majd tud haladni a robot, és célzottan tud majd érkezni a problémás oldalakra. A hozzá tartozó linkek, pedig a megnyitandó oldal URL kódját tartalmazza majd, ami szintén kulcs fontosságú elem.

Az oldalak címek feltöltése a mátrixba úgy történik, hogy a robot a HTML kódot nézi, és mivel a fejezetek listája, más néven a menüszerkezet, a kód elején szerepel, így, mivel a robot szisztematikusan halad, elég egy előtesztelő ciklust beállítani, olyan paraméterrel, hogy az utolsó menü elemnél leálljon. Ez jelen esetben úgy történt, hogy az utolsó elem, a tizennyolcadik pont volt, így a FOR ciklus addig ment, amíg a tizennyolcas menüpontot be nem írta a mátrixba, ezt követően halad tovább a kód további részére, kilépve a ciklusból. A ciklusban van egy függvény is, amely minden egyes, a mátrixba felveendő elemet megvizsgál, hogy a kellő kritériumoknak megfelel-e. A robot tehát felveszi az összes linket egy „WebElement” típusú listára, majd ezeket nézi át megfelelő kritériumok alapján.

E kritériumok a következők:

- a link szövege nem NULL, tehát nem üres
- a link szövegében van egy „.” karakter
- az link szövegének első karaktere egy szám
- a link szövegének első két karaktere szám, és kisebb vagy egyenlő, mint tizennyolc

A link szövegének lekérdezése a „WebElement”`getText()` metódusával kapható meg. A `getText` metódus már önmagában egy String-et ad visszatérési értéknek, amelyre az összes String osztályra vonatkozó metódus ugyanúgy alkalmazható. A „ha” függvényben a vizsgálandó feltételek, melyeket és kapcsolással kötünk össze, melynek jele a dupla angol „és” karakter, „&&”, melynek első feltétele a „.” tartalmazása. Az, hogy a soron következő felirat nem üres, azt a „isEmpty()” metódussal vizsgálhatjuk, melyet le kell tagadnunk egy felkiáltójellel a feltétel elején. A „.” karakter vizsgálata azért fontos, amennyiben a link szövegének első karaktere egy szám, elképzelhető, hogy nem sorszámot jelent. Egy oldal címe ebben a tudásbázisban így néz ki:

12.5.3. XYZ hitelezések

Amint a fenti példából is láthatjuk, az oldal, a 12. pont alatt, az 5. ponton belül a 3. fejezet. A pont karakter létezésének vizsgálatára menük címének szövegében, tudunk használni egy „charAt” metódust, mely visszatérési értéknek az első előfordulását adja vissza a paraméterként beírt karakternek. Amennyiben a karakter nem szerepel az adott szövegben, a visszatérési érték

-1 lesz, így abban az esetben, ha nagyobb mint -1 a „charAt” metódus eredménye, akkor tartalmaz a szöveg pontot.

A következő feltétel a „ha” függvényben az első karakter vizsgálata, mely egy, a „Character” osztály „isDigit” igaz/hamis értéket visszaadó metódusával is vizsgálható. A metódusnak oda kell adni a link szövegének első karakterét, ami a „serverLinkek.get(i).getText().charAt(0)” kóddal nyerhető vissza, ahol a „serverLinkek” a „WebElement” típusú kettő dimenziós lista, az „i” változó, pedig a FOR ciklus számlálója. Ha eddig az összes feltételnek megfelelt a vizsgált link szövege, akkor a negyedik feltételnek is meg kell felelnie, ahhoz, hogy a vizsgáltban lévő aktuális elem bekerülhessen a String típusú kétdimenziós listába. A negyedik feltétel pedig, hogy az első szám a link szövegének elején nem nagyobb mint 18. Az első két karakter vizsgálata helytelen eredményt adna vissza, az oka pedig a következő.

Az első két karakter közül jelen esetben már az első biztosan szám, a harmadik feltételnek köszönhetően. A második karakter ez esetben lehet egy betű, egy szám vagy egy írásjel. Ha nem szám szerepel a második karakter helyén, akkor az Integer-ré konvertálás során jóval nagyobb értéket kapnánk, mint az általunk határértéknek ítélt 18, és így helytelenül hamis ágra futna a „ha” függvényünk. A helyes megoldás tehát, ha egy „substring” metódust alkalmazunk, és lefejtjük az első számot a link szövegéről. A „substring” metódus azért kell, mert amennyiben egy jegyű a menüelem száma, akkor egy karaktert kell lefejtenünk, és Integer-ré konvertálnunk, azonban, ha már két számjegyű menü elemnél járunk, akkor már két számjegy lesz a konvertálandó érték. A „substring” metódus első paramétere természetesen 0 lesz, mivel mindig a karakterlánc elején kell kezdenünk a lefejtést, azonban a második paraméter már a „,” karakter helyzetének sorszáma a vizsgált String karakterei között. A konvertálás igen egyszerű, az Integer osztály „parseInt” metódusát alkalmazzuk hozzá, majd egy alapvető relációs jellel megvizsgáljuk az így kapott Integer értékét.

Az így kapott, a kritériumoknak megfelelő értékeket hozzáadjuk egy kettő dimenziós String típusú listához, melynek első oszlopa tartalmazza a címeket, a második oszlopa a hivatkozásokat.

2.3.A MENÜK KIÍRÁSA FÁJLBA

A kód a menü lekérdezése és kiolvasása után végig fut az összes soron, majd egy „FileWriter” metódust használva kiírja egy txt fájlba, a hibák elkerülése végett. Nagymennyiségű adat írása esetében a fájl író memóriája megtelhet, és félbehagyhatja a fájl írását, mindenféle hibajelzés nélkül, amelyre nincs is külön beépített funkcionalitással, amely a puffer megtelését jelzi vissza, így a fájlba kimentett export hiányos lesz, és használhatatlan. A probléma megoldására mielőtt

megtelik a fájl író metódus memóriája, ki kell azt üríteni, amire megoldás lehet, ha minden sor írása előtt töröljük a memóriáját egy „`filewriter.flush()`” metódussal. A metódus kiírja azt, ami éppen a memóriájában van, az aktuális fájlba, így helyet hagyva a soron következő elemeknek. A „flush” metódust meg tudjuk hívni a fájlba történő írás előtt is, így a korábban betárazott, még fájlba sikeresen be nem írt adatokat erőltetve be tudjuk íratni, vagy a fájl írás után, így azonnal lezárva a fájl írásának folyamatát (www.digitalocean.com, Pankaj). A fájl exportálást érdemes soronként kiírni, mint ahogyan strukturálisan el van rendezve a kettő dimenziós listán belül is. Az oszlopok elválasztására egy olyan karaktert kell használnunk, amit el tud menteni a txt, azaz egy egyszerű szöveges fájl később könnyen olvasható, tagolható általa a szöveg, és egyedi soronként, tehát véletlenül sem található meg a mindennapi szövegben, ami jelen esetben a fejezet címek és az URL hivatkozások. Az általam választott karakter a pontosvessző lett, mert ez minden előbb felsorolt kritériumnak megfelelt. A fájl kiírása egy FOR ciklussal történik, amely végig lépteti a menüt tartalmazó listákat, és soronként beírja a fájlba. A „FileWriter” osztály „write” metódusa alapértelmezetten nem sorokat ír, hanem az elhelyezett kurzornál folytatja a fájl írását, így össze kel fűzni a sor első elemét, egy pontos vessző szimbólumot, majd a hivatkozást, és egy vezérlősorozattal kell zárni, ami megkezdí az új sort. A fentebb is említett okokból minden sor írása előtt vagy után, törölni kell a fájl író memóriáját is.

A kódot háromszor futtatjuk le, annak érdekében, hogy könnyebben monitorozható legyen a kód haladása, és elakadás esetén is, csak egy adott fájl írását kelljen újraindítani. A kód háromszori lefuttatása után, mely jelen esetben tartalmazza lefuttatásonként a szülő oldalak kibontását, az oldalak lekérdezését és a fájl írását, kapunk három fájl, amelyek már egy beolvasható fájl tartalmaznak, és szükségesek lesznek a tovább haladáshoz és a robot léptetéséhez.

2.4.MAPPA ÉS FÁJL LINKEK LEKÉRDEZÉSE

A második legnagyobb fázis a probléma robotizált megoldásának menetében, amikor a tudásbázis oldalait szisztematikusan végig léptetjük, és az oldalakon található fájl hivatkozásokat szintén, egy kettő dimenziós „ArrayList”-be helyezzük, majd szintén, a fentebb is említett könnyebbségi okoknál fogva, kiírjuk ezeket egy fájlba, amely később beolvasható lesz. A fájl strukturálása tehát, olyan kell, hogy legyen, ami alapján majd a robot egyértelműen tudja majd azonosítani a behelyettesítendő szövegeket, a megfelelő helyen, és a megfelelő oldalon.

A lekérdezés indításához a már korábban is használt „get()” metódust használjuk, mely szisztematikusan, az oldalak számával megegyező ismétlődéssel végrehajtunk egy FOR ciklust, amely végig tudja léptetni az immáron beolvasott oldalakat. A fájl beolvasására egy „FileReader” metódusba ágyazott „BufferedReader” osztályt használunk, ami vissza olvassa a fájl

megírása előtti helyzethez hasonlatosan egy két dimenziós „ArrayList”-be a fájlba írt menük nevét, és URL hivatkozását. A két külön osztály alkalmazása azért fontos, mert így a program sokkal gyorsabban tud működni, és azért érdemes akár hosszútávon is, mert a „BufferedReader”, mint ahogy arról neve is árulkodik, rendelkezik egy alapértelmezett 8 kilobyte-os memóriával, így nem byte-ról byte-ra olvassa a fájlt, hanem egyszerre egy nagyobb szeletet is képes kezelni (www.scaler.com, Jai Juneja). A jelenlegi robot számára az oldalak elnevezése teljes mértékben elhanyagolható, és csak majd az új fájl megírásában lesz csak szerepe.

A FOR ciklust a fentebb említett „get()” metódussal kell kezdeni, majd egy „explicit wait” megnevezésű elemmel kell folytatnunk. Az „wait” osztály „until()” metódusa egy úgy nevezett „explicit wait”, ami képes egy folyamatot egy maximálisan, általunk meghatározott ideig váratni, amíg egy bizonyos kritérium nem teljesül. Ez a várakoztatás leginkább a Selenium használata során tud számunkra hasznos lenni, mivel így tudjuk a robot által ellenőriztetni, hogy egy esetleges oldal frissítést követően az új oldal megnyílt-e már, megfelelő módon (www.selenium.dev-B). A várakoztatás jelen esetben nélkülözhetetlen, hiszen, ha a ciklusunk tovább haladna mindenféle várakozás nélkül, és úgy próbálna a weboldal „DOM”-jához, akkor „StaleElementException”-t dobna vissza hibaüzenetként, amely a leggyakoribb hibaüzenet bármely Selenium által futtatott tesztautomatizmus során (www.linkedin.com, Serhat Uçar). A „StaleElementException” egy olyan hibaüzenet, amely azt takarja, hogy az általunk keresett „WebElement” nem található meg a korábban beolvasott „DOM”-on belül. A hibaüzenet kikerülése végett, minden esetben, amikor a robot frissít oldalt, vagy bármely olyan tevékenységet végez, mely végeredménye egy új oldal betöltése lenne, a „DOM” és azon belül lévő elemek újra beolvasása nélkülözhetetlen (www.stackoverflow.com). Az oldal betöltését és a várakozást követően a linkek beolvasása kezdetét veheti. A beolvasáshoz egy listát hozunk létre, mely a beolvasandó elemek típusával megegyező elemeket képes tárolni, tehát „WebElement” típusú lesz. Ezt a listát mindenképpen egy rugalmas „ArrayList”-ként kell deklarálni, mivel a linkek mennyisége oldalanként változhat. A linkek lekérdezésének legegyszerűbb módja, ha az összes linket az „a” HTML címkére hivatkozva tesszük, melyet a weboldal szerkezetét leíró, oldal mögött futó HTML kód alapján is ki tudunk keresni. Ezt a címkét egy „By.tagName” metódusba kell beírni. A jelen esetben az oldalon lévő összes link lekérdezése és listába írása az alábbi módon néz ki:

```
„List<WebElement> linkLista = driver.findElement(By.tagName(„a”));”
```

Az összes link azonban tartalmazza az összes gomb, menü elem és tényleges hivatkozás linkjét is, így ezeket tovább kell szűrni. A linkek szűrését a listába írás előtt is meg tudjuk ejteni,

azonban jelen esetben ez a lista elkészülte után történik meg. Általunk, a szűrés kritériuma a linkek hivatkozásának szövegének első pár karaktere fogja adni. A mappa és fájl hivatkozások eleje minden esetben kétféleképpen kezdődik, vagy úgy, hogy „file:///P:” vagy úgy, hogy „file://”, amely már a hivatkozás kezdetén a meghajtóra hivatkozik. A szűrést tehát a lista elkészültét követően egy másik FOR ciklusba ágyazott, ha függvénnyel tudjuk megtenni, amely képes lesz az általunk kiválasztott elemek kiszűrésére, és azok egy új listába történő elhelyezésére. A ha függvény feltétele, tehát a „linkLista.get(i).getAttribute(„href”)” kódrészletével történik, amely már önmagában egy Stringet ad vissza, mint visszatérési érték. Ezt a visszatérési értéket kell még megtoldanunk egy „startsWith()” metódussal, amely már egy String osztálybeli metódus, és az így kapott kód már egy igaz-hamis értéket fog visszaadni. A HA függvény három kritériumból fog állni, elsőnek fontos meggyőződnünk, hogy a „getAttribute” metódus nem ad vissza NULL értéket, amely egy üres String-et jelképez, amely abban az esetben, ha NULL értéket adna vissza, a kódot hibára futtatná. A NULL érték kiszűrése után egy ÉS kapcsolattal, tehát kettő angol „és” szimbólum segítségével a második kritériumot utána fűzzük, amely a „file:///P:” kezdetű link hivatkozásokat szűrjük, majd egy „vagy” kapcsolattal, tehát két „|” karakterrel a „file://” kezdetű hivatkozásokat szűrjük.

A HA függvény feltétele tehát az alábbi módon épül fel:

```
HA( NULL! && ((„file://”) || („file:///P:”)))
```

Amennyiben az adott link megfelel a kritériumoknak és az igaz ágra fut, akkor a program felvesz egy új elemet még a kezdeti FOR ciklus előtt deklarált kettő dimenziós String listába, melynek első oszlopa tartalmazza a link oldalának címét, második oszlopa az elérési linkjét, harmadik oszlopa a link felírtát, más néven a link szövegét, melyet a „getText()” metódussal nyerünk vissza a „WebElement”-ből. A negyedik oszlop pedig, maga a link fájl, vagy mappa hivatkozását fogja tartalmazni.

A kér FOR ciklus lefuttatását követően a robot készen fog állni arra, hogy az így kinyert információkat ki tudja menteni egy fájlba, amit szintén egy „txt” kiterjesztésű egyszerű dokumentumba mentünk el, melyben az oszlopok pontos vesszővel lesznek elválasztva.

A fenti, link lefejtő kód, nagyjából oldalanként 3-6 másodpercet vesz igénybe, amely az összes, 588 oldallal számolva 45 percet vesz igénybe. A robot futtatásának sebessége természetesen nagyban függ a robotot futtató számítógép teljesítményétől, a futtató gép internetének sávszélességétől, és a robot által megnyitandó oldalt biztosító szerverek sebességétől.

A robot teljes mértékben tud futni, úgy, hogy a robotot futtató számítógépen, a programmal párhuzamosan a felhasználó egyéb tevékenységeket is tud futtatni, azonban a sávszélesség

kihasználása, redukálhatja a robot teljesítésének idejét. A robot lefutásának ideje, leginkább az új oldalak betöltésével tud megcsúszni, ahol figyelniünk kell, hogy a várakoztatást elegendő idő intervallumra adjuk meg, de ne túl sokra, abban az esetben, ha a robotot futtató számítógép esetében a kapcsolat nem tud létrejönni a megnyitandó oldallal. Ebben az esetben hagyni kell, hogy a robot hibára fusson, és a robot újra indítása elkerülhetetlen.

A mappa és fájl linkek meglétének ellenőrzése nélkülözhetetlen, és nagyban megkönnyíti a robot futtatását, majd a későbbi fázisokban. A linkek tudásbázisba történő felcsatolásuk során, és a linkek mögött lévő mappa struktúra megváltozása folyamán nincs rá mód, hogy a tudásbázis automatikusan monitorozza és esetlegesen visszajelezzon, amennyiben a link már nem létező helyre mutat, így ezt a lépést is nekünk kell megtennünk. Ezt a lépést be lehet ágyazni magába a link behelyettesítő robot részébe, azonban ez a kódsor teljes mértékben tud a háttérben futni, csendesen, így ennek a folyamatnak a kivétele a robotot egyszerűsíti, és az esetleges hibalehetőségeket redukálja.

A felcsatolt linkek 21%-ban már nem voltak aktívak a robot ezen részének lefuttatásának pillanatában sem, így ezek kivétele a linkeket tartalmazó fájl listából, vagy jelen esetben egy új fájl írása szükséges a létező, aktív linkekkel.

A fájlok és mappák és fájlok válogatására a „Files” osztály „isDirectory” metódusa tud segíteni (www.tutorialspoint.com – A, karthikeya Boyini). A metódusnak csak „Path” típusú változót tudunk odaadni, így azt először deklarálni kell, majd a String típusú linkjeinket át kell konvertálni az elvárt változótípusba. A konvertálás előtt, azonban egy egyedi problémát kell megoldanunk, ami a fájl linkek lementése folyamán adódott. A linkek magyar karaktereket és szóközöket is tartalmaznak, melyek URL kódoláson esnek át, ahol helyettesítő karakterek kihelyettesítették azokat, amit nekünk manuálisan, a linkek lementését követően vissza kell helyettesítenünk.

4. sz. ábra. Képernyőkép, a magyar karakterek és URL kódjai betöltve egy kettő dimenziós listába

```
List<List<String>> szimbolumok = new ArrayList<>();
szimbolumok.add(Arrays.asList("%C3%A1", "á"));
szimbolumok.add(Arrays.asList("%C3%81", "Á"));
szimbolumok.add(Arrays.asList("%C3%A9", "é"));
szimbolumok.add(Arrays.asList("%C3%89", "É"));
szimbolumok.add(Arrays.asList("%C3%AD", "í"));
szimbolumok.add(Arrays.asList("%C3%BD", "Í"));
szimbolumok.add(Arrays.asList("%C3%B3", "ó"));
szimbolumok.add(Arrays.asList("%C3%93", "Ó"));
szimbolumok.add(Arrays.asList("%C3%B6", "ö"));
szimbolumok.add(Arrays.asList("%C3%96", "Ö"));
szimbolumok.add(Arrays.asList("%C5%91", "ő"));
szimbolumok.add(Arrays.asList("%C5%90", "Ő"));
szimbolumok.add(Arrays.asList("%C3%BA", "ú"));
szimbolumok.add(Arrays.asList("%C3%A", "Ú"));
szimbolumok.add(Arrays.asList("%C3%BC", "ü"));
szimbolumok.add(Arrays.asList("%C3%9C", "Ü"));
szimbolumok.add(Arrays.asList("%C5%B1", "ű"));
szimbolumok.add(Arrays.asList("%C5%B0", "Ű"));
szimbolumok.add(Arrays.asList("%20", " "));
```

Forrás: Saját készítésű ábra

A megoldás lehet, egy lista létrehozása, majd feltöltése a karakterekkel, majd egy FOR ciklussal, és a String osztály „replace” metódusával, ahol az metódus első paramétere a kettő dimenziós lista első oszlopának „i”-ik eleme, a második paramétere pedig a második oszlop „i”-ik eleme.

A változó típus a meghajtó nevével vagy a szerver nevével kell kezdődjön, ahhoz, hogy érvényes elérési útvonal típusú lehessen, így a „file:///P:/” kezdetű elérési útvonalak esetében az első 8 karaktert le kell vágni az elejétől, amit a String osztály „substring” metódusával tudunk megtenni. Amennyiben nem felel meg a link a fent jelzett kritériumnak, akkor az elérési útvonal a link első 5 karaktere után lesz értelmezendő, mivel a „file:///” kezdetű hivatkozások folytatása már a szerver nevét tartalmazza, tehát az első 5 karaktert az elejétől le kell vágnunk a konverálás során, így a „Path” típusú változó márt értelmezhető lesz a kód számára. Mindkét esetben, ha fájl és ha mappa link, akkor is fontos, hogy az „exists” metódussal meggyőződjünk a fájl, vagy mappa létezéséről, majd az „isDirectory” metódus tudja eldönteni, hogy az elbírálásra küldött elérési útvonal fájl, vagy mappa.

Ebben a lépésben külön is tudjuk venni a fájl linkeket, és azok vizsgálatát, valamint a mappa linkeket, és azok vizsgálatát, így a behelyettesítés pillanatában elég lesz kisebb szekciókat

indítani a robotból, így az esetleges hibaforrásokra tudunk majd visszajelzést kapni, amennyiben nem az elvárt végeredményt kapjuk majd a robotnál.

2.5.FÁJLOK BEILLESZTÉSE AZ ELVÁRT IGÉNYEKNEK MEGFELELŐEN A TESZT SZERVERRE

A teszt szerver, mint ahogyan a neve is utal rá, a folyamatok tesztelésére szolgál, mielőtt azokat az éles szerveren implementálni tudjuk. A robot a teszt szerveren már a folyamat megkezdése óta folytattunk teszteléseket, melyeket a Confluence hibátlan verziókezelésének segítségével minden gond nélkül vissza is tudunk állítani a robot futtatása előtti verzióra, így a folyamatos tesztelések gond nélkül futtathatók.

A teszt szerverre a robotot szinte teljes mértékben megegyezik a majd éles felületen futó robottal, és nagy módosításokat nem kell végrehajtani a felületek azonos felépítéséből kifolyólag.

Az első módosító robot feladata a fájlok felcsatolása, ahol az eredeti, szerveres tudásbázisban is fájlokra mutató linkek voltak felcsatolva. A robotnak a humán link lecseréléssel szemben, nincs szüksége a két oldal folyamatos összefűzésére, hiszen a korábbi fázisokban már az összes, a robot számára hasznos tartalom kiexportálása megtörtént. Ebben a fázisban elég lesz, ha a robot, csak a létező fájlok és mappák listáját végig görgetve halad, és illeszti be azokat.

A robot először beolvassa a létező fájl linkek listáját, majd a teszt felhős tudásbázis menüjének címeit és URL linkjeit tartalmazó fájl, majd a beolvasást követően a tudásbázis megnyitása, bejelentkezés, majd egy nagy FOR ciklus segítségével halad végig az összes feltöltendő fájlra.

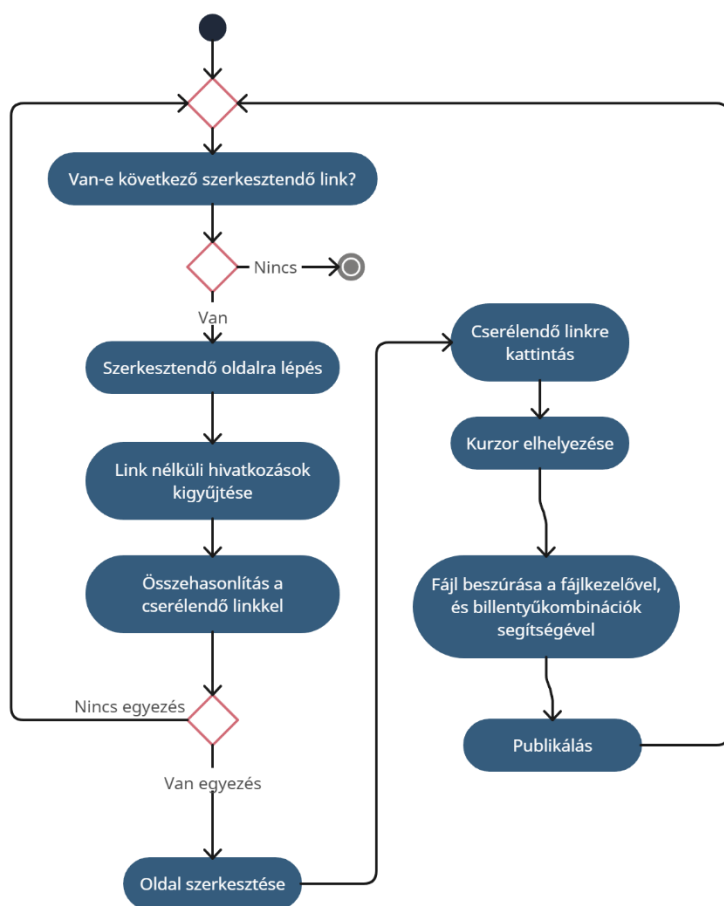
A beolvasást követően megnyitja a tudásbázis kezdőoldalát, egy „dirver.get” metódussal, majd be kell jelentkeznie egy olyan felhasználóval, akinek van szerkesztési jogosultsága az egész módosítandó tudásbázis összes oldalára. A bejelentkezést úgy tudjuk végrehajtani, hogy a bejelentkezés gombra várunk, amíg meg nem jelenik egy „explicit wait”-el, majd a gombot megkerestetjük a robottal, melyhez a gomb „XPath” hivatkozását használjuk, mely a weboldal HTML szintjén történő elhelyezésével azonosítja be az elemet, ahol használhatunk relatív helyzetet vagy abszolút helyzetet (www.guru99.com, Krishna Rungta).

A bejelentkezési gomb megnyomása után, újra be kell olvasatni az elemeket, mivel egy bejelentkező képernyő fog minket fogadni, mely korábban nem volt a része az oldalnak HTML szinten sem. A beolvasott elemekből 3 elemre lesz szükség, a bejelentkezési email cím bemeneti mezője, melynek oda tudjuk adni a felhasználó email címét, és a jelszó mező, amelynek pedig a felhasználó jelszavát. A bejelentkezési adatok beírását a „sendKeys” metódussal tudjuk megtenni, mely egy String-et kér tőlünk, amit be fog nekünk gépelni a bemeneti mező „WebElement”-be, amelyre a metódus meghívásra került. Az adatok beírását követően érdemes egy

„Thread.sleep” metódust meghívni, amely egy statikus várakozás, amellyel meg tudunk szakítani bármilyen folyamatot egy általunk meghatározott ideig. Ez a metódus alkalmas lehet visszaszámlálók készítésére is, de jelen esetben egy pár ezred másodperc elég ahhoz, hogy az autentikációs adatok beírása véghez tudjon menni, és ezt követően a bejelentkezési gomb gond nélkül megnyomható legyen. A bejelentkezést követően az oldalon a szerkesztés gomb lesz a kritérium az újabb „explicit wait”-ben, mely csak a bejelentkezést követően jelenik meg, így megfelelő támpont lehet a várakozáshoz. Ha minden gond nélkül eddig lezajlott, akkor kezdődhet a fájl linkek javítása, és a fájlok feltöltése.

2.5.1. Lecserélendő link megkeresése

5. sz. ábra. A fájl linkek javítását leíró, átfogó ábra



Forrás: Saját készítésű ábra

A fájlok feltöltése egy FOR ciklussal történik, melynek első pontja, hogy az első feltöltendő lista oldalának címénél megáll, majd végig görgeti az összes teszt menüt, és megnézi, hogy ahol van egyezés a két menü neve között, akkor ott megáll, és a „driver.get” metódussal megnyitja az adott oldalt, amelyet a korábban beolvasott teszt tudásbázis oldalainak elérési útvonalával

tud megtenni. Az oldal megnyitása után meg kell keresnie, hogy léteznek-e még az eredeti linkek szövegei, ahol eredetileg fájl link volt becsatolva a szöveg mögé.

A tudásbázis HTML kódja úgy néz ki, hogy ahol korábban fájl vagy mappa link volt, ott a migrált felületen továbbra is az link, tehát az „a” címkét kapta meg az adott szövegrészlet, de mögötte nincsen már ténylegesen hivatkozás, tehát a „href” címke üres. A robot lekérdezi az összes „a” címkéjű linket, majd végig nézi, hogy hol ad vissza NULL értéket a „href” címke értéke, amit a „getAttribute” metódussal tudunk lekérdezni. Amennyiben a HA függvényeken túl jut az adott „WebElement”, akkor megnézi a robot, hogy milyen értéket ad vissza a „getText” metódus, és ha megegyezik a lecserélendő link feliratával, akkor elindul a szerkesztési fázis.

2.5.2. A fájl csatolása

A fájl csatolásának első lépése, hogy a robot megnyomja a szerkesztés gombot, amelyet a gomb „XPath” elérésének kimásolásával kaphatunk meg. A szerkesztő felület a hagyományos bemeneti mezőkkel ellentétben a Confluence szerkesztő felülete egy beágyazott HTML oldal, amely betöltését egy „explicit wait” metódussal várunk meg. A szerkesztő felület betöltését követően el kell végezni azt a folyamatot, amit egy szerkesztő felhasználó is végezne a jelen esetben.

Egy felhasználó a link javítását az alábbi módon vinné végbe:

1. aktuális hiányos link megkeresése
2. a link szövege után kattintás, a kurzor elhelyezése végett
3. beszúrás gomb megnyomása
4. fájl csatolása gomb megnyomása
5. fájl kiválasztása fájlkeresőn belül
6. becsatolás gomb megnyomása
7. eredeti szöveg kivágása
8. „csatolmány megjelenítés linkként”
9. csatolmány módosítása
10. helyettesítő szöveg átírása az eredeti szövegre
11. „OK”
12. közzététel gomb megnyomása

A folyamatot tehát a kattintással, és a kurzor elhelyezésével fogjuk kezdeni. A folyamat ezen pontján már nagyon pontosan tudnunk kell, hogy milyen lehetséges helyeken lehetnek a linkek, ahhoz, hogy jól futó robotot tudjunk hozzá írni, hiszen az összes lehetséges lehetőségre nem lehet semmilyen robotot felkészíteni, kifejezetten a javításra szánt időkorlát betartásával nem. Minden robotnak az aktuális problémákra kell megoldást adjon, és jelen esetben egyszeri

hibátlan lefutása a cél, és nem egy periodikusan ismétlődő események automatizálására szolgál, így a speciális eseteket mind külön kell kezelni, és nincs mód az esetek uniformizálására.

A linkek elhelyezkedhetnek egy szövegben, felsorolásban szerepelhetnek egymás alatt a fájl linkek, vagy pedig felsorolásban a szöveg részében tartalmazzák csak a linkeket.

6. sz. ábra. Linkek különböző lehetséges elhelyezkedése

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt [A link átnyúl új sorba](#) ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea [Szöveg közepén helyezkedik el](#) commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

- [Felsorolás szerűen](#)
- Felsorolás [részeként](#)

Forrás: Saját készítésű ábra

A linkek szövegnek hossza pedig egészen terjed a 3 betűs szavaktól a 238 karakter hosszú szövegekig. A kurzor elhelyezése, sajnos a Confluence szerkesztő sajátosságából kifolyólag nem megoldható a „JavaScriptExecutor” osztállyal, mivel a kód futtató osztály csak a bemeneti mező típusú mezők tartalmát képes manipulálni, és nem az egymásba ágyazott HTML felületekét, így kerülő megoldáshoz kell folyamodni.

Az általunk választott megoldás, hogyha a szerkesztő felületén bekérjük újra az összes linket, egy „WebElement” típusú listába, majd a korábban is ismertetett kritériumokon átszűrt elemre történő kattintás, akkor, ha a link szövege megegyezik az általunk keresett aktuális feltöltendő fájl szövegével. A kattintást a Selenium „click()” módszerével tudjuk megoldani, amely minden esetben az adott „WebElement” közepére kattint. Ez a működés nagyban kifizetődő, amennyiben a kattintandó objektum egy gomb. Amennyiben azonban a kattintandó objektum egy szöveg, akkor is a közepére kattint, abban az esetben is, ha a szöveg átfolyik egy új sorba, így a kattintás egy teljesen véletlenszerű pontban történik. Az ilyen hosszú link szövegek száma elhanyagolható azonban, és ezek javítása utólag kézzel is könnyen megoldható, így a kerülő megoldás kitalálása ezen esetekre elhanyagolható, és a ráfordított idő nem hozná vissza a manuális javítással megspórolt időt.

A kattintás a szöveg közepére történik, még pedig a pixel pontos közepére. Például, ha egy szó páratlan karakterből áll, akkor a karakterszámából kivonunk egyet, majd megfelezzük, majd hozzáadunk egyet, és a kurzor az így kapott karakter után lesz elhelyezve. Figyelembe kell azonban venni, a szélességében kisebb karaktereket és szélesebb karaktereket. A kattintást, természetesen csak a belső HTML oldalra történő váltás után tudjuk kivitelezni, amelyet a

„driver.switchTo()” metódussal tudunk véghez vinni. A „switchTo” metódus elvárja, hogy adjunk meg neki, egy „WebElement”-et, amely az új „iFrame”-et tartalmazza. Ezt a Confluence oldal HTML oldalának vizsgálat módjából tudjuk ismét kiolvasni. A kapott elemet többek között az ID-je alapján is be tudjuk azonosítani, amely jelen esetben az „wysiwygTextarea_ifr”. Ez az „iFrame” a szerkesztő felület beágyazott felülete, így ezt elég egyszer még az első váltás előtt deklarálni, mint változó, és ezt követően csak a deklaráláskor elnevezett változóra meghívni a „switchTo().frame” metódust, melynek oda tudjuk adni az említett „WebElement” típusú változót. Amennyiben a későbbiekben a felületen lévő navigációs, vagy formázó gombokat szeretnénk használni, vagy bármely elemet, amely az „iFrame”-en kívül helyezkedik el, a ChromeDriver osztály „switchTo().defaultContent()” metódusával vissza tudjuk állítani a fókuszban lévő felületet, így aktívak lesznek újra a gombok, és a robot manipulálni tudja majd őket újra.

Miután a kattintás megtörtént a robot által, a kurzor a keresett link szövegének viszonylagos közepén van. A kurzor eljuttatása a szöveg végére azonban egy külön folyamat. A legegyszerűbb megoldás a fentebb is említett „JavaScriptExecutor” osztály alkalmazása, és a JavaScript programozási nyelv egy szinten történő elsajátítása adná, azonban a két szintes HTML szerkezet nem engedi meg a JavaScript-ben ismeretes „Selection” metódus alkalmazását, amellyel képesek lennénk akár teljes intervallumokat kijelölni, vagy csak a kurzort elhelyezni, abban az esetben, ha a kijelölés kezdeti és végző dátumát azonos értékkel töltjük fel. A kerülő megoldás egy képlet, mely képes a szöveg közepén elhelyezett kurzort a szöveg végére terelni. A megvalósításhoz a „Robot” osztályt alkalmazzuk, amely egy alap eszköze a Java nyelvnek. Az „java.awt” a Java Abstract Window Toolkit (AWT) része, amely grafikus felhasználói felületek létrehozásához és kezeléséhez használt csomag. Az „Robot” osztály lehetővé teszi a szimulált egérmozgatást és billentyűzetműveleteket grafikus felhasználói felületeken belül, vagy bárhol, ahol lehet billentyű kombinációkkal vezérelni alkalmazásokat, vagy akár makró funkciók futtatására is alkalmas.

A „Robot” osztály által képesek vagyunk megfelelő billentyűkombinációk segítségével a keresett szöveg végére helyezni a kurzort, azonban a sokszínű lehetőségek miatt a képlet komplexitása is nő. A képlet pedig a következő:

A keresett szöveg hosszából kivonunk kettőt, akkor, ha a szöveg páros karakterszámú, és egyet, ha páratlan, abban az esetben, ha a keresett szöveg karakterinek hossza több mint három. Az így kapott számot elosztjuk kettővel, majd kivonunk kettőt, és az így kapott összeget bevezetjük egy FOR ciklusba, ahol a számmal megegyező mennyiségben végrehajtunk a „Robot”

osztályon belül található „`keyPress(KeyEvent.VK_RIGHT)`” művelettel jobbra léptetjük a kurzort. A leütött billentyű felengedése kötelező mindig, a „Robot” osztály alkalmazásánál, és ennek kihagyása a kód hibára futásához vezethet. A gomb felengedéséhez a „`keyRelease()`” metódust kell alkalmazni, ahol a felengedendő gomb azonosítóját kell feltüntetni. A jelenlegi ponton a kurzor négy pozícióban lehet:

- pont a szöveg végén
- utolsó karakter előtt
- utolsó előtti karakter előtt
- utolsó 3 karakter előtt

A fent említett összes lehetőség közül nagyon kicsi a lehetősége annak, hogy amennyiben az utolsó három karakter előtt áll a kurzor, akkor az utolsó három karakter önmagában egy szóból áll, tehát tartalmaz a három karakter szóközt. A tovább haladáshoz a „Robot” osztállyal kiadunk egy kontrol és jobbra nyíl gomb billentyűkombinációt, ahol a nyomvatartandó gomb lenyomása és felengedése közé kell zárunk a billentyűkombináció második elemét. Így a kurzor az utolsó szó után lesz, azonban itt három lehetőség állhat fent:

- a keresett link egy szövegen belül helyezkedik el, így a kurzor a kontrol és jobbra nyíl lenyomása után a következő szó elejére fog ugrani
- a kurzor jó helyen, a sor végén fog állni
- a kurzor megáll egy esetleges zárójelen belül

Mivel csak billentyűkombinációkkal tudjuk navigálni a kurzort a szerkesztő felületen, így megbizonyosodni arról, hogy a kurzor a fájl beszúrása előtt a megfelelő helyen van-e karakter szinten, csak a vágólapot tudjuk használni. A lentebb írt kódrészlet létrehoz egy „`clp`” nevű változót, amely segítségével egy String típusú visszatérési értékű metódussal vissza tudjuk majd nyerni a vágólapra másolt elemeket.

```
„Clipboard clp = Toolkit.getDefaultToolkit().getSystemClipboard()”
```

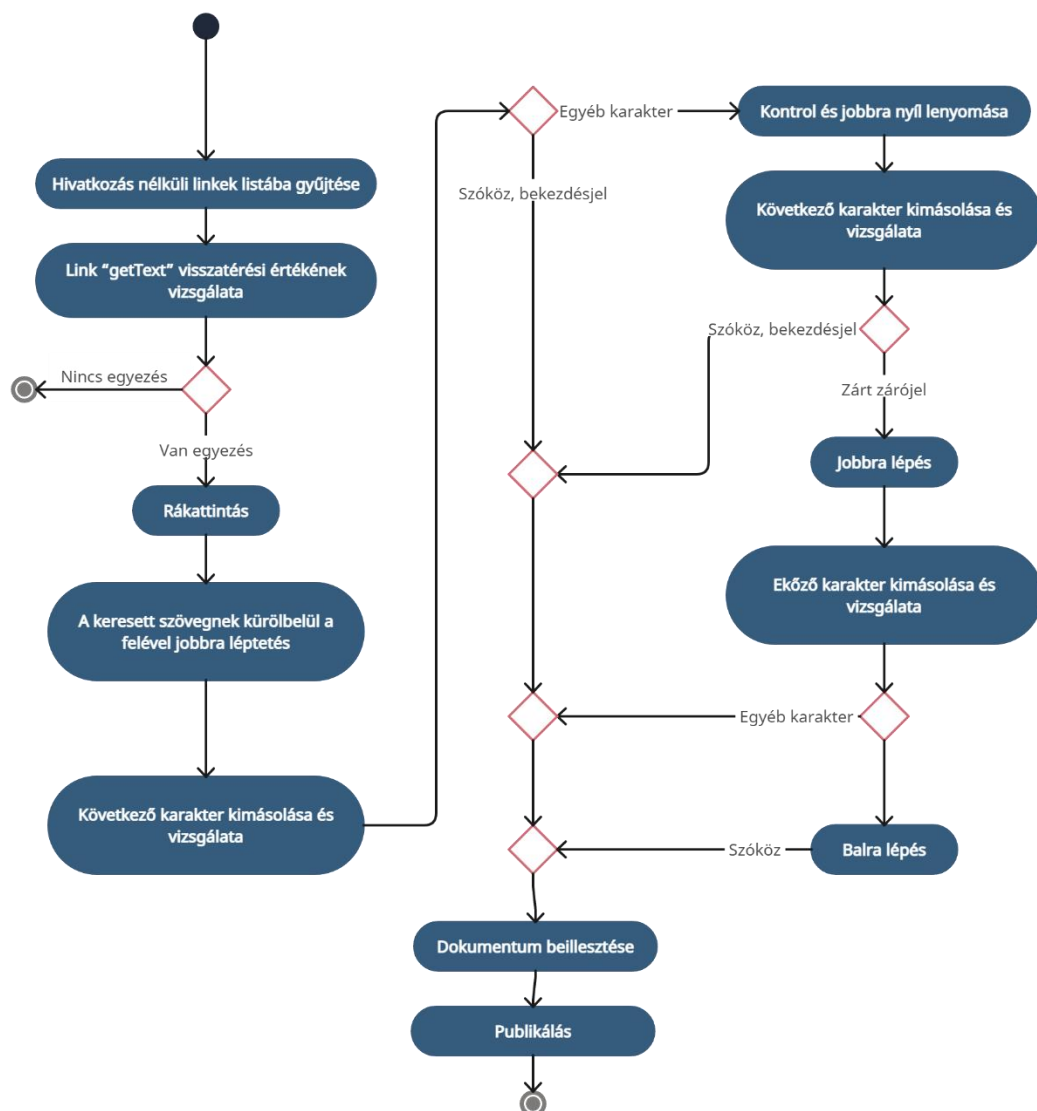
A változó helyes deklarálását követően, ahol természetesen a változó neve kedv szerint módosítható, a vágólap tartalmát bármikor be tudjuk a kóddal olvastatni, a „`getContents(DataFlavor.stringFlavor)`” metódussal. Fontos kikötni, hogy a vágólapra másolt elem visszafejtése során szinte minden esetben „try-catch” kapcsolat használata szükséges, ahol a hiba elkapása során az „`UnsupportedFlavorException`”-re kell hivatkoznunk, hiszen könnyen előfordulhat, hogy a vágólap nem tartalmaz semmi, képet tartalmaz, fájl, vagy bármely más, nem szöveges tartalmat tartalmaz, amelyre a kikötött hibaüzenet érkezne válaszként a lekérdezés során. A három eset vizsgálata hasonlóképpen zajlik:

1. billentyűkombinációkkal a karakter kijelölése
2. a karakter kimásolása
3. HA függvénnyel a karakter vizsgálata
4. a HA függvényvizsgálatából adódó eseménysorozat végrehajtása

A robotnak először, tehát, ki kell jelölnie a karaktert, amelyet vizsgálni szeretne, ez elsősorban a kurzor után lévő karakter. A karakter kijelölése a SHIFT és jobbra nyíl gombok kombinációjával lehetséges. Ezt követően ki kell másolni a karaktert, amelyet a CTRL és a „C” gomb lenyomásával tehet meg a robot, majd a kurzort vissza kell helyeznie az eredeti pozícióba, majd onnan várnia a következő utasításokat, amelyet egy balra nyíl gomb lenyomásával tud megtenni. A HA függvény ezt követően megvizsgálja a vágólapra másolt elemet, és amennyiben nem NULL, tehát üres, akkor megvizsgálja, hogy egy új bekezdés karakter-e vagy egy szóköz-e, amely vizsgálatát a Java nyelvben használatos escape sorozattal tehetünk meg, a „\n” értékkel és a szóköz esetében pedig a „ ” karakterrel. Az escape sorozatok különböző speciális karaktereket képesek megjeleníteni, amelyeknek elsődleges felhasználási területe a konzolon történő felíratok megjelenítése, rendszerezése. Az escape sorozatok közé tartozik az új sort jelképező „\n” karakter, a „\t” escape karakter, amely a tabulátor karaktert jelképezi. Escape karakterek a megoldás akkor is, ha szeretnénk Stringként kezelni különböző szintaktikai elemeket, mint az egyszeres és kétszeres idézőjel vagy a hátulsó perjel.

Amennyiben a vágólapra másolt érték egy bekezdés karakter vagy egy szóköz, úgy nincs szükség több korrekcióra, és haladhat tovább a robot a fájl beillesztésére. Amennyiben azonban nem az, a vizsgálat tovább folytatódik. Abban az esetben, ha bármely más karakter szerepel a vágólapon, akkor a Robot osztályt alkalmazva a fentebb is említett, kontrol és jobbra nyíl billentyűkombinációt kell végrehajtanunk, így a szó végére érve, vagy a következő szó elejére. A kurzor pozíciójának újonnan meghatározására ismét a vágólapra történő karakter vizsgálatát használjuk, így kimásoljuk a kurzor utáni karaktert, majd vizsgálat alá vetjük, hogy az ismét egy szóköz, vagy enter, amikor más teendőre nincs szükségünk, ha zárójel, akkor egyel jobbra kell lépnünk. A kurzor előtt lévő karakter vizsgálata is fontos ezen a ponton, hiszen elképzelhető, hogy a kurzor a következő szó elejére ugrott, így amennyiben a kurzor előtt lévő karakter egy szóköz, egyel balra kell léptetni a kurzort, egy balra nyíl gomb lenyomását imitálva a Robot osztállyal. A kurzor jó helyen van ezennel, és lehet beszúrni a fájlt.

7. sz. ábra. A kurzor elhelyezését bemutató folyamatábra



Forrás: Saját készítésű ábra

A fájl feltöltése egyszerű, ámbar egy kerülő megoldást kell majd alkalmazni. A feltöltést először is megkezdjük azzal, hogy „iFrame”-et váltunk, hiszen a beszúrás gomb is már az eredeti, nem beágyazott felületen van. A hozzáadás, beszúrás gomb megnyomására használhatunk „XPath” hivatkozást is, hiszen ez minden oldalon azonos lesz, valamint a robot csak egyszer lesz lefuttatva, így a későbbi frissítésekre nem kell jelen esetben felkészíteni a dinamikus elem azonosítással. A gomb megnyomása tehát az alábbi módon néz ki:

```
„driver.findElement(By.xpath(„//*[@id=’rte-button-insert’]“)).click()”
```

A pluszjel után a fájl hozzáadása gomb megnyomása jön, amely előtt érdemes egy „explicit wait” beszúrása, hogy meg tudja várni a robot dinamikusan, amíg megjelenik a kattintandó elem. Az így megjelenő párbeszéd panelen lévő fájl feltöltése gomb, azonban nem a szokványos

bemeneti mező típusú, hanem csak egy „label”, azaz címke típusú gomb, ami egy új párbeszéd panelben hozza fel a fájl beszúrására alkalmas fájlkezelőt. A bemeneti mező típusú gomb esetében a fájl beszúrását a „sendKeys” metódussal tehetünk meg, amelynek odaadnánk String-ként a fájl elérési útvonalát, és az automatikusan a fájl feltöltését kezdeményezné (www.soft-waretestinghelp.com, Sonali Satpute). A jelen esetben azonban itt is egy kerülő megoldást alkalmazunk. A megoldás a már használt Robot osztály, és a vágólap manipulálására használatos „Toolkit” osztály „getDefaultToolkit().getSystemClipboard().setContents” metódusa, amelynek jelen esetben oda kell adnunk az elérési útvonalat, mint első elem, és a vágólap tulajdonosát tölthetjük ki, vagy egy üres értéként is ki tudjuk kerülni ezt a lépést, amelyet egy „null” érték beírásával érhetünk el. A fájl elérési útvonalát a beolvasott, és aktuálisan beszúrandó kettő dimenziós listának a 3. elemének visszanyerésével tehetünk meg. A vezérlés innentől kezdve megint billentyűkombinációk segítségével történik. A fájl beszúrás gomb megnyomását követően a kurzor a fájl nevének beillesztésére szolgáló mezőben fog elhelyezkedni, így oda elég lesz a fájl elérésének útvonalát, és a fájl nevét, valamint kiterjesztését tartalmazó String bemásolásával az általunk kívánatos fájl beszúrható lesz, bárhol is helyezkedjen az el mappa szinten. A beillesztéshez egy billentyű kombináció lenyomására lesz szükség, amelyet a kontrol és a „v” lenyomásával érhetünk el, majd egy enter lenyomását is kezdeményeznünk kell a roboton keresztül, hiszen így nyomjuk meg a „megnyitás” gombot a fájlkezelő ablakában. Ezt követően a fájl felöltése megkezdődik, majd egy „explicit wait” segítségével, a beszúrás gomb megjelenését megvárva, az elem lenyomásával folytatja a robot. A beszúrás ezennel megtörtént. A robot a későbbi tömeges manuális formázás elkerülése végett a felcsatolt fájl formázását is elvégzi. Az elvégzéshez szükséges három lépés, az eredeti link szövegének kitörlése, a fájl megjelenítése linkként, majd a link szövegének módosítása. Az eredeti szöveg kitörléséhez a robot egyel balra lépteti a kurzort, majd egy FOR ciklust kezdeményez, amelyet a helyettesítendő szöveg karakterszámával megegyező számmal megismétel, amelyben a shift gomb nyomva tartása mellett a balra mutató nyíl gomb megnyomását ismétli. A kijelölést befejezve felengedi a shift gombot, majd a törlés gomb megnyomásával az eredeti link szövegét kitörli a tudásbázisból, helyet adva a beszúrandó fájl linkjének. A fájl egy „img”-ként, azaz képként jelenik meg a szerkesztő felületen. A robotnak feladatnak kiadni, hogy nyomjon meg egy „img” címkéjű elemet azonban nem megoldás, hiszen a tudásbázis tartalmazhat tényleges kép elemeket is, amelyek jelen esetben linkként történő megjelenítését kezdeményeznénk. A számunkra megfelelő címkék mellett fontos azon elemek szűrése is. A robot tehát első sorban visszavált a szerkesztő „iFrame”-jére, majd az összes kép kikeresése után végig futtat egy keresést a listába betöltött képeken, és ahol a „data-macro-parameters” attribútuma a „WebElement”-nek megegyezik az

imént becsatolt fájl elérési útvonalával, akkor arra a robot kattint, majd újra „iFrame”-et kell váltania, mivel a felugró ablak már a szerkesztő felületen kívül jelenik meg kód szinten, így azt a robot nem látná. A sikeres keret módosítása után a robot megnyomhatja a „Show as link” gombot, amely a gomb szövegének megfelelően a becsatolt tartalmat egy kis előnézet helyett szöveggé fogja megjeleníteni, még hozzá minden esetben a becsatolt fájl nevének és kiterjesztésének összefűzésekként. Mind a linkként történő megjelenítést, mind az ehhez tartozó szerkesztés gombját „XPath” segítségével tudjuk beazonosítani, mivel szerencsére a gombok minden esetben ugyanott helyezkednek el az oldal forráskódjában. A szerkesztés gomb megnyomása után egy felugró ablak jelenik meg, ahol van egy bemeneti mező, amire akár „sendKeys” metódussal is be tudjuk írni a fájl megjelenő szövegét, vagy egy kattintással a mezőre, majd egy kontrol és „a” billentyűkombinációval a mezőben lévő eredeti feliratot kijelöljük, majd kitöröljük egy törlés gomb szimulálásával, ezt követően a vágólapot feltöltjük a megfelelő elérési útvonalat tartalmazó String szöveggel és megnyomjuk beillesztésre alkalmas billentyűkombinációt, amelynek segítségével be tudjuk illeszteni a megjelenő szöveget. A fájl ezennel be van illesztve és a szövege is az eredeti szöveghez hasonlatos, hiszen a szöveg után beszúrt fájl megtartja az eredeti formázási beállításokat is. Az oldal ezennel közzé tehető, amely előtt meg kell bizonyosodni, hogy az oldalt figyelők nem kapnak értesítést a módosításokról, amit a régi szerkesztő felületen egy pipa jelentett. A pipa bejelölt állapota jelenti, hogy az értesítő email ki lesz küldve. A pipa állapotáról a Selenium beépített funkciója, az „isSelected” funkció tud információt adni (www.browserstack.com – A, Sonal Dwivedi), amely egy igaz/hamis visszatérési értékkel szolgál számunkra. A visszatérési értéket magában már be tudjuk helyezni egy HA függvénybe, így annak lefutása csak akkor zajlik le, ha a pipa be van jelölve. Ha a függvény lefut, akkor a vizsgált elemre küldeni kell egy kattintást, amely így kikapcsolja az értesítendő személyeket. Az is előfordulhat a publikálás pillanatában, hogy egy üzenet ablak ugrik fel, amely azt mondja, hogy vannak más szerkesztők által végzett publikálatlan módosítások, amelyek minden esetben publikálhatóak lesznek, így ez az ablak felugrása esetén a „Publish anyway” gombra kell nyomnunk, mely a figyelmeztetés ellenére is publikálni fogja az oldalt.

2.6.MAPPA CSATOLÁSA ÉS A ROBOT ÉLES FUTTATÁSA

A mappák csatolása nagy részben megegyezik a fájlok csatolásával, magát a csatolás részt leszámítva. A robot a hivatkozás nélküli linkeket keresi minden szerkesztés alkalmával, így azt a jelenlegi állapotban is ki kell törölnünk, és felülírunk. Amíg a fájlok csatolásánál az eredeti felirat kitörlésre kerül, addig a mappa hivatkozásnál ez nem lehetséges, hiszen a szöveget vissza kéne írunk, azonban a link címke kitörlése létfontosságú a robot működésének szempontjából. A szerkesztő felületre eljutásig a robot teljes mértékben megegyezik az előző folyamattal, a

különbség a csatoláson, és a link migrált címének cseréjében van. A linkek minden esetben egy bizonyos tároló elemen belül helyezkednek el, így a legegyszerűbb, ha egy „JavaScriptExecutor” osztályt alkalmazunk, és lefuttatunk egy kódot, amely az adott elem megkeresése után kitörli az elem szülőjén belül lévő összes gyerek elemet, majd a belső HTML elem szövegébe beleírja az eredeti link szövegét (www.browserstack.com – B, Mohammad Waseem).

8. sz. ábra. „JavaScriptExecutor”-al lefutott kódrészlet, melyben ki van paraméterezve a link szövege

```
String szo = filelink.get(i).get(2);
JavascriptExecutor js = (JavascriptExecutor) driver;
js.executeScript("document.querySelectorAll('a').forEach(element => {\n" +
    "if (element.getAttribute('href') === null && element.text === '"+ szo +"') {\n" +
    "console.log(element.parentElement);\n" +
    "const parent = element.parentElement;\n" +
    "parent.removeChild(parent.firstChild);\n" +
    "parent.innerHTML = '" + szo + "';\n}\n}");
```

Forrás: Saját készítésű ábra

A link szövegének beírása az elérési útvonal vágólapra történő másolásával, majd annak billentyűkombinációkkal a szerkesztő felületre történő helyezése jelenti. A kurzor mozgatásával a mappa linkek behelyezése során nem kell foglalkozni, mert amikor a JavaScript kód lefut akkor a szerkesztő automatikusan az új szövegrészlet után teszi a kurzort. A dolgunk már csak egy szóköz billentyű leütése és az elérési útvonal vágólapra töltése és beillesztése a felületre, majd az oldal készen van, és publikálható. A teljes folyamat ezennel le van fedve és készen áll az éles, felhős szerveren történő futtatásra.

Az éles szerveren a kutatásba fordított idő miatt a robot futtatásában módosításokat kell végrehajtani, amelyek a tudásbázis frissüléséből adódik. A kapcsolatok, amelyeket a menük nevével tudunk csak létrehozni, kritikusan függ a robot sikerességével. A módosított oldal címek nevét nekünk is módosítanunk kell az éles tudásbázis linkjeit tartalmazó oldalak esetében, és a linkek esetében is, amelyre tudunk használni kódot is, de kézzel is minden gond nélkül megoldható. A robot végig fogja olvasni az összes kicserélendő linket, és oldal címek egyezőségének hiányából adódóan azok nem kerülnek majd beillesztésre. Az oldalak elérési URL-je nem változik az oldal címének módosításával, így azokat nem kell módosítani.

A teszt szerveren sikeres lefutást követően az éles futtatás több szakaszban történik, ezzel kedvezve az alkalmazás felelőseinek, abban az esetben, ha valahol vétene a robot, nem az összes probléma javítását kell kivitelezni egy adott napon. A robot 3 nap, munkaidő után sikeresen végig tud futni, így a problémák javítása minden probléma nélkül megoldásra került.

3. A ROBOT SIKERES FUTTATÁSÁT KÖVETŐ ADATOK

3.1.KIVÉTELKEZELÉS ÉS EGYÉB INTÉZKEDÉSEK A FUTTATÁSHOZ

Kivételkezeléskor fontos megemlíteni a különböző osztályoknál használt kikötendő kivételeket, amelyeket egyaránt egy „try-catch” metódusban is fel tudunk venni, vagy osztály szinten is kiköthetjük, az osztály által bekért változók után a „throws” után is, amely így figyelmen kívül hagyja, ha a program az utána beírt kivételeket vagy hibaüzeneteket dobna. Ezek az osztályok, amelyek a program során megjelentek, és kivételkezelésük szükséges, a következők:

1. táblázat. A kikötendő kivételek csomagja, osztálya, és a kivétel megnevezése

Csomag	Osztály, metódus	Kivétel/kikötés
java.io	FileReader, FileWriter	IOException
org.openqa.*	driver.get(), driver.findElement()	UnhandledAlertException
java.awt.*	Robot	AWTException
java.lang	Thread.sleep()	InterruptedException
java.awt.datatransfer.*	Toolkit.getDefaultToolkit().getSystemClipboard()	UnsupportedFlavorException

Forrás: Saját készítésű táblázat

Az osztályok alkalmazása során a kivétel kikötése azért fontos, mert ezek a leggyakoribb hibák, amiket a használt metódus dobhat, így azok kezelése fontos lehet a programozónak. Ebben segítségre van a kódszerkesztő is, amely automatikusan hibaként is jelzi, ha nincs az adott, leggyakoribb hiba lekezelve. A lekezelése a tipikus hibaüzeneteknek rendkívül fontos, és az így, hogy hogyan tudjuk vagy szeretnénk lekezelni. A legtöbb hibaüzenet figyelmen kívül hagyható, amelyek a táblázatban is szerepelnek, azonban a Selenium általános „StaleElementException”, tehát nem található elem kivétele rendkívül fontos, hogy helyesen legyen lekezelve a program egészének során, mindig a megfelelő helyzethez alkalmazva.

A robot futtatásához szükséges állományokat folyamatosan tudjuk pótolni, amennyiben alkalmazásuk szükséges lesz a folyamatunk során. A fenti robot elkészítéséhez a Selenium saját oldalán is megtalálható „ChromeDriver” letöltése fontos, amelyet az alábbi módon tudunk rendszerváltozóként rögzíteni:

„System.setProperty(“webdriver.chrome.driver”, [a letöltött állomány elérési útvonala]);”

Természetesen, az általunk automatizálni kívánt böngésző WebDriver-jét kell letöltenünk, ami jelen esetben a Google Chrome volt (www.tutorialspoint.com - B).

A Selenium oldaláról értelemszerűen le kell töltenünk még a közvetlen komponenseit, amelyet szerverként is le tudunk tölteni vagy „jar”, azaz „Java Archive” rövidítését kiterjesztésként használó forrásfájlokat (www.docs.oracle.com) is, amelyeket utána a projektünk struktúrájában hozzá kell adnunk. Több hibakódot is dobhat a kódunk futtatást követően, azonban nyílt forráskódú szoftverhez mérten bármilyen állomány az internetről bármikor letölthető és konfigurálható, ezzel pótolva a hiányosságokat.

3.2.STATISZTIKÁK ISMERTETÉSE ÉS AZOK ÉRTELMEZÉSE

A robothoz szükséges szoftverek kiválasztása, beszerzése, valamint a hozzájuk tartozó licen-szek beszerzése a cég méretéből és a szigorú auditból kifolyólag 2 hétbe tellett. A robot megírásához szükséges alapismeretek elsajátítása és a robot tesztelése szintén másfél hetet vett igénybe, amely órákban számolva 80 munkaórának felel meg. Az egyeztetések a megrendelővel, jelen esetben a fiókhálózatban dolgozó kollégákkal másfél hetet vettek igénybe, de ez az időszak egymással átfedésben volt, mind a program megírásának időtartama alatt, mind az alkalmazás igénylés idejére vetítve. A teljes megvalósításhoz szükséges idő, tehát közel egy hónapot vett igénybe a három nap futtatással töltött idővel együtt, amely naponta másfél órát vett igénybe, úgy, hogy a robot lassú, otthoni hálózaton volt futtatva, így a várakozási idők nagyban megnövekedtek.

A manuális feltöltés átlagosan egy oldal esetében 5-8 percet vesz igénybe, amely az összes oldal feltöltése esetében 49 emberórától kezdve egészen 78-80 órát is igénybe vehet, feltéve, hogy az illető nem dolgozik máson, és megállás nélkül a kijelölt feladattal foglalatoskodik, nem beszélve arról, hogy manuális helyettesítés esetében, ha a megrendelő módosítja az eredeti megállapodást a feladat lefolyásával kapcsolatban, akkor a folyamat újra kezdhető és az idő sokszorosodni kezd. A jelenlegi esetben 2 igény módosítás történt a behelyettesítés végkimenetével kapcsolatban, amelyek javítása a robot esetében csak pár kód részlet frissítését vette igénybe, azonban a manuális feltöltés esetében vissza kell mennie a szerkesztő személynek, és javítania a problémás linket. Természetesen, ez a személy lehet az javítást végrehajtó személy is, vagy a tudásbázist használó és szerkesztő fiókhálózati munkatárs, viszont a javítással foglalkozó személye nem változtat a cselekménybe fektetett idő mennyiségén. A kézzel történő javítás ugyanakkor nem feltétlenül jó, ha munkaidőben történik, hiszen, ha egyszerre szerkeszti az oldalt két személy is, esetlegesen, akkor a módosítása összeakadhat a később publikáló személynek az időben korábban publikáló személyével, abban az esetben, ha az egyik személy szerkesztésének megkezdése után a második személy is elkezd a szerkesztést, majd a később

kezdett személy publikálást végez. Az így inkább munkaidő után végzendő feladat azonban túlórának számít a cég számára, és plusz költséget jelent.

A robotizálás azonban befektetés a kollégákba, a folyamat megértésébe, és a jövőbe. A cégen belüli folyamatokat egyszer kell kijárnia valakinek, és utána már sokkal hamarabb juthatnak hozzá mások a szükséges erőforrásokhoz, így gyorsítva bármely jövőbeli fejlesztést. Mivel, mint jól láthatjuk a számok a robotizálás ellen szólnak, ugyanakkor nem szabad megfeledkeznünk a lehetőségek tárházáról. A robotizálás bármely részzszakaszban megállítható, és akár darabokban is alkalmazható, tehát ha csak egy lekérdezést szeretnénk futtatni, arra is alkalmas lehet. A fájlok helyének lekérdezése többek között igen hasznos első lépés lehet akár akkor is, ha szeretnénk magát a fájlt feltöltést manuálisan végezni. Elvégezhető például a fájlok validációja is, pár perc leforgása alatt, így aki kézzel javít, már csak a létező fájlokat próbálja meg feltölteni, ezzel is időt spórolva magának, és a céltudatos munkavégzésnek köszönhetően a hibalehetőségeket is minimalizálni tudja. A program használható az adatok kiexportálására, az adatok manipulálására, mint jelen esetben a dekódolás magyar karakterekre, vagy éppen használható lehet a tényleges elemek lecserélésére. Elképzelhető megoldás lehet, hogy valamilyen módon kiírjuk a lecserélendő elemeket a konzolra, valamint megnyitjuk az oldalt, ahol az elemeket kell cserélni, de azokat manuálisan kezeljük, és valamilyen inputot várunk a program szerkesztő alkalmazás oldalára a felhasználótól, például a „Scanner” osztály alkalmazásával, amíg a program vár, és amit megkapta a várt bemenetet a program folytatódik, és kiírja az új adat lecserélendő fájlt, és megnyitja a szerkesztendő oldalt.

A kutatásba vetett idő azonban nem volt nagy kihatással az egyébként érintett, megrendelői részlegekre nézve, hiszen a régi felülethez történő hozzáféréssel minden korábbi fájl elérését lehet biztosítani, amely ad némi időt a robot finomításához.

Még egy fontos pont, hogy az emberi erőforrás megteremtése a humán módon elvégzett javításhoz, úgyszintén időbe telik, amíg rendelkezésünkre áll, és sok egyéb, a humán erőforrásból akadó probléma megnyújthatja az ekképpen történő javítást, mint például: betegség, hiányos vagy hibás munkavégzés.

ÖSSZEFOGLALÁS

A dolgozat hipotézisét alátámasztva elfogadjuk, hiszen kellő bizonyítékunk van annak megalapozottságára, amely tapasztalati, és számszerű ismérveink alapján is azonosan, pozitív irányba billentik döntésünket.

A Seleniumnal történő robotizálás, tehát egy minden oldalról előnyös befektetés kollégák számára, hiszen az első alkalommal kell csak az alábbi folyamatokon átesni:

- Selenium megigénylése
- Selenium bank biztonsággal történő elfogadtatása
- ChromeDriver igénylése
- ChromeDriver bank biztonsággal történő elfogadtatása
- Licenz igénylése a kódszerkesztő alkalmazáshoz
- Az összes alkalmazás telepítése és konfigurálása
- El lett indítva a licenz nélküli kód szerkesztő átnézetése a bank biztonsággal
- Az alkalmazások megtanulása:
 - Selenium és általános robotizálás bármilyen korábbi tudás nélkül
 - Java nyelv felfrissítése az Egyetemi tudás alapján
 - Java osztályok (Robot, JavascriptExecutor, FileWriter, FileReader) kiismerése
- Confluence felület HTML kódjának kellő kiismerése és átlátása
- HTML kód általános átlátása, és JavaScripttel történő manipulálásának elsajátítása
- Adatszerkezetek meg strukturálása
- Kommunikáció a bankon belüli folyamatok és lehetőségek átlátása végett mind a megrendelő oldallal, mind a megoldó csoportokkal
- Alkalmazásüzemeltetők tudásbázisába egy összefűzött export megosztása és a folyamat leírása, a későbbi hibakezelések elősegítésére

A saját rendszerének ismerete mindig jó bármely rendszer felelőségének, és azon belüli problémák esetleges automatizálásának lehetősége is könnyebbséget tud jelenteni. A befektetett plusz időt befektetésnek is fel lehet fogni, hiszen minden bankon vagy vállalatban belüli folyamat egyszer már végig lett csinálva, így a következő személy, aki hasonló folyamaton szeretne végig menni, az előkészületekkel töltendő idő hatványozottan lecsökkenhet. Nem is beszélve az egyén plusz hozzáértését és izgalmasabb munkavégzését, amely a programozás által lehetséges, hiszen nem a monoton feladatot kell csinálnia, hanem meg tudja ismerni az alkalmazást, amin a javítást végzezi, illetve a későbbiekben neki már minden lehetőség adott más alkalmazásokon belüli folyamatok automatizálására is.

A dolgozat során átvettem a szükséges folyamatokat, és leírtam a hozzá szükséges ismereteket, amelyek a robot megvalósításához nélkülözhetetlenek voltak. A folyamat során teljes mértékben önálló munkavégzés volt, mind a kutatómunka részéről, mind a kódírás, és futtatás folyamán, így a kapott eredményekben nem voltak külső befolyásoló tényezők, és valós eredményeket kaptunk. A javítást követően semmilyen negatív visszajelzés nem érkezett a megrendelői oldalról, ami azt bizonyítja, hogy a megoldás a való életben is tökéletesen működött, és az esetleges hibákat a végfelhasználók is tudták javítani, amennyiben volt. A folyamat, tehát teljes mértékben megtanulható önállóan is, valamint a felhasznált kód és tudás könnyen átadható annak, aki arra igényt tart és valamilyen egyéb folyamatot szeretne ugyanezen a felületen automatizálni. A Selenium nagyon beszédes osztály neveket alkalmaz, így a megírt kód olvasása sem megterhelő, még annak sem, aki újonnan csatlakozna a programozás világába. UI Path-el ellentétben a Selenium sokkal könnyebben tudja kezelni az adatokat, ami a Java nyelvből is adódik, hiszen úgy formáljuk, tároljuk és olvassuk vissza az adatokat, ahogy ahhoz kedvünk tartja. A két oszlopos táblázatokat meg lehetett volna oldani MAP-ekkel is, amelyek egy egyértelmű kulcs után bármilyen értéket tudnak tárolni. Ez azért lehetett volna járható út az oldalak neveinél, mert az oldalak címei minden esetben különböznek, így egyértelműen beazonosíthatóak lettek volna az egyes sorok úgy is. A több oszlopos táblázatok esetében már indokolt az általunk is használt két szintű lista, amely esetében használhattunk volna egy olyan listát, amelynek sorai csak tömbök, hiszen az oszlopok száma nem változó, csak a sorok száma fontos, hogy dinamikusan tudjon növekedni. A dinamikus listák hátránya a nagyobb lemez igény, és a visszakereshetőség sebessége, amely jelen esetben nem jelentett gondot, azonban egy olyan robotnál, amely rendszeresen lesz futtatva, ott célszerű a lehető legjobban optimalizálni az adat-szerkezeteket.

A robot részben és egészben is alkalmazható, mint a fentebb is említett export készítés, fájlok írása, manipulálása és fájlok összefésülése. Kellő tudással akár minden javítás alkalmával egy létrehozott és összefűzött export automatikusan feltölthető lehet a tudásbázisba, hogy az illetékesek tudják, hogy mi történt a legutóbbi javítás alkalmával.

A robotizálás tehát egy nagyon sok oldalú lehetőséget kínál a felhasználóinak, főleg amíg az egy ilyen módon, nyílt forráskóddal bír, és egy programozási nyelv elemeivel is tudjuk kombinálni. A funkcionalitása a Java nyelv és sok más a nyelvbe beépített osztályainak és a Selenium, vagy egyéb más akár ingyenesen letölthető csomag kombinációjával olyan nagy, hogy annak, aki azt átlátja, bármit képes megoldani az eszközök ismeretével. A befektetett energia minden

esetben a jövőbeli problémák megoldásának gyorsaságát növeli, és platformot ad a produktivitás növelésére.

IRODALOMJEGYZÉK

www.developer.chrome.com/docs/apps/app_external/

www.confluence.atlassian.com/confkb/using-the-confluence-rest-api-to-upload-an-attachment-to-one-or-more-pages-1014274390.html

www.github.com/christophebedard/JavaWord

www.codecademy.com/article/what-is-an-ide

www.guvi.in/blog/what-is-dom-in-selenium/

A - www.selenium.dev/selenium/docs/api/py/webdriver_remote/selenium.webdriver.remote.webelement.html

www.digitalocean.com/community/tutorials/java-filewriter-example

www.scaler.com/topics/bufferedReader-java/

B - www.selenium.dev/documentation/webdriver/waits/

www.linkedin.com/pulse/how-handle-stale-element-exception-selenium-while-testing-serhatu%C3%A7ar

www.stackoverflow.com/questions/52087471/selenium-wont-find-element-on-page-after-page-has-reloaded

A - www.tutorialspoint.com/check-whether-a-file-is-a-directory-in-java

www.guru99.com/xpath-selenium.html

www.softwaretestinghelp.com/file-upload-in-selenium

A - www.browserstack.com/guide/checkbox-in-selenium

B - www.browserstack.com/guide/javascriptexecutor-in-selenium

B - www.tutorialspoint.com/java/lang/system_setproperty.htm

www.docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html

