Dokumentacja techniczna

Justyna Maciąg, Urszula Soboń

Styczeń 2018

1 Produkt

Moduł do pobierania danych do statystyk z wybranego zdalnego repozytorium.

2 Zastosowanie produktu

Produkt ma pobrać repozytorium z podanego przez użytkownika adresu URL jeśli jest połączenie z internetem oraz podany przez użytkownika adres URL jest poprawny i takie repozytorium istnieje. Dodatkowo produkt udostępnia ze sciągnietego repozytorium listę commitów na wszystkich branchach. Lista commitów zawiera informacje o dacie dodania commit'a, jego autorze, krótkiej wiadomości, nazwie brancha do którego należy oraz zawiera informacje o stanie wszystkich plików dla każdego z osobna(ilość dodanych i usunietych linii oraz kontent danego pliku).

3 Klasy

- Fetcher - klasa korzystając z gita pobiera listę wszystkich commitów dla danego repozytorium
- GitRevCommits - klasa pomocnicza do Fetchera, która dostara mu szczegółowych danych do listy informacji o commitach
- GitDownloader implementuje interfejs RepoDownloader, dzięki temu w przypadku zmian w sposobie pobierania informacji lub rozszerzeniu projektu na inne systemy
- URLReader - klasa odpowiedzialna za sprawdzenie czy podany przez użytkownika adres jest poprawny i czy połączenie z siecią jest
- FileDiffs - klasa definiująca strukturę pokazującą zmiany w pliku w danym commicie.
- CommitDetails - klasa definiująca zawartość danego commita.

4 Omówienie poszczególnych metod

4.1 Klasa Fetcher

```
public void prepareDownloader(String url, SimpleDoubleProperty progress)
    throws Exception {
        this.commitDetailsList = new ArrayList<>();
        this.git = gitDownloader.getRepository(url, progress);
    }
```

Funkcja dostaje adres URL, który podał użytkownik w GUI oraz SimpleDoubleProperty. Na tym etapie adres URL jest sprawdzony i na pewno poprawny. Funkcja tworzy nową listę dla commitów w danym repozytorium oraz wywołuje metodę z klasy GitDownloader która odpowiedzialna jest za pobranie repozytorium.

```
    public List<CommitDetails> getAllCommits() throws Exception {

        if (commitDetailsList.isEmpty()) {
            this.commitDetailsList = generateCommitDetailList();
        }
        return this.commitDetailsList;
    }
```

Funkcja zwraca listę commitów. Jeśli funkcja jest pierwszy raz wywoływana lista commitów jest pusta - bo nie została wcześniej wygenerowana więc zostaje wywołana funckja, która tworzy listę commitów. Jeśli lista commitów nie jest pusta funkcja zwraca już gotową i zapisaną w klasie liste commitów.

```
    public List<CommitDetails> getCommitsFromDateRange(DateTime
        startDate, DateTime endDate) throws Exception {
        return getAllCommits().stream().filter(d -> d.getCommitDate().
            isAfter(startDate)
                && d.getCommitDate().isBefore(endDate)).collect(
                    Collectors.toList());
    }
```

Funkcja zwraca listę commitów z danego przedziału dat.

```
    public List<CommitDetails> generateCommitDetailList() throws
        Exception {
        try {
            GitRevCommits revTmp = gitRevCommitsProvider.get();
            for (Git g : git) {
                for (RevCommit rev : revTmp.revCommitList(g)){
                    CommitDetails commit = commitDetailsProvider.get();
                    commit.setPrimaryInformation(new DateTime(rev.
                        getAuthorIdent().getWhen()),
                            rev.getAuthorIdent().getName(),
```

```
                                rev.getShortMessage(), g.getRepository().
                                    getBranch());

                        commit = revTmp.addDiffsToCommit(rev, commit, g);
                        commit = revTmp.addLinesForAllFiles(rev, commit, g);
                            //.getRepository());

                        this.commitDetailsList.add(commit);
                        }
                    }
                }
            catch (GitAPIException e) {
                throw new Exception("Problem occured during getting
                    RevCommits list.");
            }

            return commitDetailsList;
        }
```

Funkcja odpowiedzialna jest za dostarczenie listy commitów dla danego repozytorium które wcześniej zostało pobrane. Zwraca błąd w przypadku, gdy jest jakiś problemm z uzyskaniem potrzebnych danych.

4.2 GitRevCommit

```
    public CommitDetails addLinesForAllFiles(RevCommit rev, CommitDetails
        commit, Git git ) throws Exception {

        Repository repository = git.getRepository();
        TreeWalk treeWalk = new TreeWalk(repository);
        try {
            treeWalk.setRecursive(true);
            treeWalk.addTree(rev.getTree());

            while (treeWalk.next()) {
                String path = treeWalk.getPathString();

                ObjectId objectId = treeWalk.getObjectId(0);
                ObjectLoader loader = repository.open(objectId);

                ByteArrayOutputStream stream = new ByteArrayOutputStream
                    ();
                loader.copyTo(stream);

                int linesNumber = IOUtils.readLines(new
                    ByteArrayInputStream(
                        stream.toByteArray()), "UTF-8").size();
                FileDiffs fileDiffs = fileDiffsProvider.get();
```

```
                List<FileDiffs> fileDiffsList = commit.getFiles();
                boolean flag = false;
                for (FileDiffs f : fileDiffsList) {
                    if (f.getFileName().equals(path)) {
                        f.setLinesNumber(linesNumber);
                        if (rev.getParentCount() == 0)
                            f.setInsertions(linesNumber);
                        flag = true;
                    }
                }
                if (!flag) {
                    if (rev.getParentCount() == 0)
                        fileDiffs.setInformation(path, linesNumber, 0);
                    else fileDiffs.setInformation(path, 0, 0);
                    fileDiffs.setLinesNumber(linesNumber);
                    fileDiffs.setFileContent(stream);
                    commit.addFile(fileDiffs);
                }
                treeWalk.close();
            }
        } catch (IOException e) {
            throw new Exception("Problem occured during adding lines for
                all files.");
        }

        return commit;

    }
```

Funkcja dla wszystkich plików w danym commicie uzupełnia je o ich kontent.

```
    public CommitDetails addDiffsToCommit(RevCommit rev, CommitDetails
        commit, Git g) throws Exception {
        Repository repository = g.getRepository();
        if (rev.getParentCount() != 0) {
            List<DiffEntry> diffEntries = null;

            try {

                diffEntries = g.diff()
                        .setOldTree(getCanonicalTreeParser(rev.getParent
                            (0), repository))
                        .setNewTree(getCanonicalTreeParser(rev,
                            repository))
                        .call();
```

```
                for (DiffEntry diffEntry : diffEntries) {
                    int deletions;
                    int insertions;
                    DiffFormatter diffFormatter = new DiffFormatter(
                        DisabledOutputStream.INSTANCE);
                    diffFormatter.setRepository(repository);
                    diffFormatter.setContext(0);
                    EditList edits = diffFormatter.toFileHeader(
                        diffEntry).toEditList();
                    deletions = edits.stream().mapToInt(Edit::getLengthA
                        ).sum();
                    insertions = edits.stream().mapToInt(Edit::
                        getLengthB).sum();

                    FileDiffs fileDiffs = fileDiffsProvider.get();
                    fileDiffs.setInformation(diffEntry.getNewPath(),
                        insertions, deletions);
                    commit.addFile(fileDiffs);

                }
            } catch (IOException e) {
                throw new Exception("Problem occured during adding diffs
                    .");
            }
        }
        return commit;
    }
```

Funkcja dla danego commita uzupełnia informacje o wszystkich zmianach w plikach. Funkcja zwraca strukture commita uzupełnioną o listę zmian we wszystkich plikach czyli o ilości dodanych i usuniętych liniach.

```
    public CanonicalTreeParser getCanonicalTreeParser(ObjectId commitId,
        Repository repository) throws IOException {
        try( RevWalk walk = new RevWalk(repository)) {
            RevCommit commit = walk.parseCommit( commitId );
            ObjectId treeId = commit.getTree().getId();
            try( ObjectReader reader = repository.newObjectReader()) {
                return new CanonicalTreeParser( null, reader, treeId );
            }
        }
    }
```

Funkcja pomocnicza.

4.3 GitDownloader

```
    @Override
    public List<Git> getRepository(String repoUrl, SimpleDoubleProperty
        progress) throws Exception {
        Git git;
        List<String> branches = getBranchesToClone(repoUrl);
        final int branchesCount = branches.size();
        List<Git> gits = new ArrayList<>();
        try{
            for(int i = 0; i < branchesCount; i++){
                File file = Files.createTempDir();
                git = Git.cloneRepository()
                        .setURI(repoUrl)
                        .setDirectory(file)
                        .setBranch(branches.get(i))
                        .call();
                git.getRepository().close();
                git.close();
                gits.add(git);
                final int ind = i;
                Platform.runLater(()-> progress.set(1.0*(ind+1)/
                    branchesCount));
            }
        }
        catch (GitAPIException e) {
            throw new Exception("Problem with cloning remote repository.
                ");
        }
        return gits;
    }
```

Funkcja z każdej branchy znajdującej się na githubie pod podanym adresem URL pobiera repozytorium i zapisuje do listy odniesienie do tego pobranego repozytorium. Funkcja zwraca tę listę do klasy Fetcher. Na podstawie tych repozytoriów pobierane są dane do analizy. Funkcja zwraca błąd, gdy wystąpił jakiś problem przy pobieraniu repozytorium.

```
    private List<String> getBranchesToClone(String url  ) throws
        Exception{
        List<String> branches = new ArrayList<>();
        Collection<Ref> refs;
        try {
            refs = Git.lsRemoteRepository()
                    .setHeads(true)
                    .setRemote(url)
                    .call();
            for (Ref ref : refs) {
                branches.add(ref.getName().substring(ref.getName().
                    lastIndexOf("/") + 1, ref.getName().length()));
```

```
            }
        } catch (Exception e) {
            throw new Exception("Error during getBranchToClone.");
        }
        return branches;
    }
```

Funkcja pomocnicza,ktora dla danego repozytorium pobiera nazwy branchy, które trzeba sklonować.

4.4 URLReader

```
    public static boolean checkIfExistsRemote(String repoUrl) throws
        Exception {
        boolean result;
        InputStream ins = null;
        try {
            URLConnection conn = new URL(repoUrl).openConnection();
            ins = conn.getInputStream();
            result = true;
        }
        catch(MalformedURLException e ){
            result = false;
        }
        catch (IOException e) {
            throw new Exception("Connection problem");
        }
        finally {
            try {
                if(ins != null)
                    ins.close();
            }
            catch (NullPointerException | IOException e) {
                System.err.println("Error during closing connection.");
            }
        }
        return result;
    }
```

Funkcja nawiązuje połaczenie z siecią i sprawdza czy podany przez użytkownika adres URL jest poprawny. Zwraca true jesli udało nawiązać się połączenie z siecią i istnieje takie repozytorium , false gdy nie istnieje dane repozytorium oraz błąd gdy nastąpił problem z połączeniem z siecią.

5 Testy

5.1 TestRepository

```
    private Provider<GitRevCommits> gitRevCommitsProvider = new Provider
        <GitRevCommits>() {
         @Override
         public GitRevCommits get() {
             return gitRevCommits;
         }
    };
    private Provider<CommitDetails> commitDetailsProvider = new Provider
        <CommitDetails>() {
         @Override
         public CommitDetails get() {
             return commitDetails;
         }
    };
    private Provider<FileDiffs> fileDiffsProvider = new Provider<
        FileDiffs>() {
         @Override
         public FileDiffs get() {
             return fileDiffs;
         }
    };
    private GitRevCommits gitRevCommits = new GitRevCommits(
        commitDetailsProvider, fileDiffsProvider);
    private static CommitDetails commitDetails = new CommitDetails();
    private static FileDiffs fileDiffs = new FileDiffs();
    @Mock
    private RepoDownloader repoDownloader;
    @Mock
    private SimpleDoubleProperty progress;
    private Fetcher fetcher;
    private List<Git> gitList = new ArrayList();
    private Git git;

    @Before
    public void setup() throws Exception {
         MockitoAnnotations.initMocks(FetcherTest.class);
         git = Git.init().setDirectory(new File("C:\\tmpRepo")).call();
         fetcher = new Fetcher(repoDownloader, commitDetailsProvider,
             fileDiffsProvider, gitRevCommitsProvider);
         Mockito.when(repoDownloader.getRepository("mama", progress)).
             thenReturn(gitList);
         fetcher.prepareDownloader("mama", progress);
         gitList.add(git);
    }
```

```
@After
public void clean() {
    git.getRepository().close();
    deleteFolder(new File("C:\\tmpRepo"));
}
public static void deleteFolder(File folder) {
    File[] files = folder.listFiles();
    if(files!=null) {
        for(File f: files) {
            if(f.isDirectory()) {
                deleteFolder(f);
            } else {
                f.delete();
            }
        }
    }
    folder.delete();
}
```

Przygotowanie środowiska do testowania funkcją setup wykonywaną zawsze przed wykonaniem testu. W funkcji setup tworzymy repozytorium lokalne, musimy odpowiednio zadbać o to, aby po wykonaniu testów repozytorium zostało usunięte. Tę funkcję spełnia metoda clean, wywoływana po teście.

```
@Test
public void testRealRepo() throws Exception {
    Repository repository = git.getRepository();
    File myFile = new File(repository.getDirectory().getParent(), "
        testFile");
    if(!myFile.createNewFile()) {
        throw new IOException("Could not create file " + myFile);
    }

    git.add().addFilepattern("testFile").call();
    git.commit().setMessage("Commit1-Adding testFile").call();

    Path file = new File("C:\\tmpRepo\\testFile2").toPath();
    byte[] buf = "testLine\n testLine2".getBytes();
    Files.write(file, buf);

    git.add().addFilepattern("testFile2").call();
    git.commit().setMessage("Commit2-Adding testFile2").call();

    buf = "testLine\n testLine2\n adddedLine".getBytes();
    Files.write(file, buf);

    git.add().addFilepattern("testFile2").call();
    git.commit().setMessage("Commit3-Changing testFile2").call();
```

```
        List<CommitDetails> result = fetcher.getAllCommits();
        assertEquals(result.size(), 3);

    }
```

Jest to właściwa funkcja testująca. Jej zadaniem jest stworzenie 3 commitów. Następnie sprawdzamy czy zaimplementowana przez nas metoda getAllCommits w klasie Fetcher spełnia swoje zadanie, metoda ta powinna zwrócić 3 commity, tak też się dzieje.

5.2 TestRepositoryPieces

```
    @Rule
    public TemporaryFolder tempFolder = new TemporaryFolder();


    private final static Provider<GitRevCommits> gitRevCommitsProvider =
        new Provider<GitRevCommits>() {
        @Override
        public GitRevCommits get() {
            return gitRevCommits;
        }
    };
    private final static Provider<CommitDetails> commitDetailsProvider =
        new Provider<CommitDetails>() {
        @Override
        public CommitDetails get() {
            return commitDetails;
        }
    };
    private final static Provider<FileDiffs> fileDiffsProvider = new
       Provider<FileDiffs>() {
        @Override
        public FileDiffs get() {
            return fileDiffs;
        }
    };
    private final static GitRevCommits gitRevCommits = new GitRevCommits
       (commitDetailsProvider, fileDiffsProvider);
    private final static CommitDetails commitDetails = new CommitDetails
       ();
    private final static FileDiffs fileDiffs = new FileDiffs();
    @Mock
    private RepoDownloader repoDownloader;
    @Mock
    private SimpleDoubleProperty progress;
    private Fetcher fetcher;
    private Repository repository;
```

```
    private List<Git> gitList = new ArrayList<>();
    private Git git;
    @Before
    public void setup() throws Exception {
        MockitoAnnotations.initMocks(FetcherTest.class);
        git = Git.init().setDirectory(tempFolder.getRoot()).call();
        fetcher = new Fetcher(repoDownloader, commitDetailsProvider,
            fileDiffsProvider, gitRevCommitsProvider);
        repository = git.getRepository();
        gitList.add(git);
        Mockito.when(repoDownloader.getRepository("mama", progress)).
            thenReturn(gitList);
        fetcher.prepareDownloader("mama", progress);
    }

    @After
    public void clean(){
        git.getRepository().close();
    }
```

Przygotowanie środowiska, analogicznie jak poprzednio, jednakże w tym przypadku całe repozytorium jest tworzone w katalogu tymczasowym. Poniżej przedstawiono kilka przykładowych testów:

```
    @Test
    public void testSimpleCommit() throws Exception {
        createRevCommit("exampleFile.txt", "Line1\n Line2\n Line3\n", "
            commit1");
        List<CommitDetails> result = fetcher.getAllCommits();
        assertEquals(result.size(), 1);
        assertEquals(result.get(0).getCommitMessage(), "commit1");
        assertEquals(result.get(0).getBranch(), "master");
        assertEquals(result.get(0).getFiles().get(0).getFileName(), "
            exampleFile.txt");
        assertEquals(result.get(0).getFiles().get(0).getLinesNumber(),3)
            ;
        assertEquals(result.get(0).getFiles().get(0).getDeletions(),0);
        assertEquals(result.get(0).getFiles().get(0).getInsertions(),3);
    }
```

Test, w którym dla jednego stworzonego commita sprawdzamy, czy metoda getAllCommits w Fetcherze zwraca poprawne wyniki.

```
    @Test
    public void testCountLines() throws Exception {
        RevCommit commit = createRevCommit("exampleFile", "Line1\n Line2
            \n Line3\n", "commit1");
        CommitDetails commitDetailsNew = new CommitDetails();
        CommitDetails commitDetails = gitRevCommits.addLinesForAllFiles(
            commit, commitDetailsNew, git);
```

```
        assertEquals(commitDetails.getFiles().get(0).getLinesNumber(),
            3);

    }
```

Powyższy test ma za zadanie sprawdzić poprawność działania metody pośrednio wywoływanej w fetcherze - metody addLinesForNumbers. Metoda ta zwraca wyniki dotyczące ilości wszystkich linii w danym pliku w danym commicie.

```
    @Test
    public void testDiffs() throws Exception {
        RevCommit commitOld = createRevCommit("exampleFile1", "Line1\n
            Line2\n Line3\n Line4\n", "commitFirst");
        RevCommit commitNew = createRevCommit("exampleFile1", "Line1\n
            Line2\n LineCompletlyNew\n LineNew\n LineNew\n", "
            commitSeconf");
        CommitDetails commitDetailsNew = new CommitDetails();
        CommitDetails commitDetails = gitRevCommits.addDiffsToCommit(
            commitNew, commitDetailsNew, git);
        assertEquals(commitDetails.getFiles().get(0).getInsertions(), 3)
            ;
        assertEquals(commitDetails.getFiles().get(0).getDeletions(), 2);
    }
```

Test ten ma na celu sprawdzenie czy dla danego commita ilość linii dodanych i usuniętych wzgledem poprzedniego zgadza się w stosunku do tego co otrzymamy wywołując metodę addDiffsToCommit.

```
    private RevCommit createRevCommit(String name, String content,
        String commitMessage) throws IOException, GitAPIException {
        createFile(name, content);
        git.add().addFilepattern(name).call();
        return git.commit().setMessage(commitMessage).call();

    }

    private File createFile(String name, String content) throws
        IOException {
        File file = new File(git.getRepository().getWorkTree(), name);
        try(FileOutputStream outputStream = new FileOutputStream(file)){
            outputStream.write(content.getBytes());
        }
        return file;
    }
```

Są to metody pomocnicze, które ułatwiają stworzenie pliku i dodanie go nowego commita w repozytorium.

5.3 FetcherTest

Klasa ta, podobnie jak poprzednie testuje zachowanie Fetchera, różnicą jest tutaj sposób testowania. W tym przypadku nie generujemy właściwego repozytorium, a zamiast tego mockujemy wszystkie obiekty, których używamy w klasie Fetcher.

```
@Test
public void prepareDownloader() throws Exception {
    Mockito.when(repoDownloader.getRepository("mama", progress)).
        thenReturn(git);
    f.prepareDownloader("mama", progress);
    assertEquals(f.getGit(), git);
}
```

Test ma za zadanie sprawdzenie poprawności działania metody prepareDownloader w Fetcherze.

```
@Test
public void testGenerateCommitDetailsList() throws Exception {

    commitDetailsNotMock.setPrimaryInformation(new DateTime(2000,
        11, 11, 11, 11, 11, 11), "auth1", "commess", "branch1");
    commitDetailsList.add(commitDetailsNotMock);
    commitDetailsListNotMocked.add(commitDetailsNotMock);
    commitDetailsNotMock.setPrimaryInformation(new DateTime(2000,
        11, 11, 11, 11, 11, 11), "auth2", "commess2", "branch2");
    commitDetailsList.add(commitDetailsNotMock);
    commitDetailsListNotMocked.add(commitDetailsNotMock);
    Mockito.when(com.get()).thenReturn(commitDetailsObject);
    revCommitMock1 = createDetailedCommit("Author1", "ComMess1", new
         Date(2000, 11, 11, 11, 11, 11));
    revCommitMock2 = createDetailedCommit("author2", "ComMess1", new
         Date(2010, 11, 11, 11 ,1));
    revCommits.add(revCommitMock1);
    revCommits.add(revCommitMock2);

    Mockito.when(repository.getBranch()).thenReturn("Branch1");
    Mockito.when(gitObject.getRepository()).thenReturn(repository);
    git.add(gitObject);
    Mockito.when(repoDownloader.getRepository("mama", progress)).
        thenReturn(git);
    f.prepareDownloader("mama", progress);
    Mockito.when(gitRevCommits.revCommitList(git.get(0))).thenReturn
        (revCommits);
    Mockito.when(gitRevCommits.addDiffsToCommit(revCommitMock1,
        commitDetailsObject, gitObject)).thenReturn(
        commitDetailsNotMock);
    Mockito.when(gitRevCommits.addDiffsToCommit(revCommitMock2,
        commitDetailsObject, gitObject)).thenReturn(
        commitDetailsNotMock);
```

```
        Mockito.when(gitRevCommits.addLinesForAllFiles(revCommitMock1,
            commitDetailsNotMock, gitObject)).thenReturn(
            commitDetailsNotMock);
        Mockito.when(gitRevCommits.addLinesForAllFiles(revCommitMock2,
            commitDetailsNotMock, gitObject)).thenReturn(
            commitDetailsNotMock);
        Mockito.when(gitRevCommitsProvider.get()).thenReturn(
            gitRevCommits);

        assertEquals(f.getAllCommits(), commitDetailsListNotMocked);

    }
```

Test ten kompleksowo sprawdza poprawność działania metody getAllCommits z klasy Fetcher. Jak widać nie są tworzone żadne instancje obiektów. Wywołania niektórych metod jak i obiekty użyte pośrednio są odpowiednio mockowane.

```
    private RevCommit createDetailedCommit(String author, String
        commitMessage, Date date) {
        StringBuilder parents = new StringBuilder();
        int numParents = 1;
        for (; numParents > 0; numParents--) {
            parents.append(String.format("parent %040x\n", new java.util
                .Random().nextLong()));
        }

        String commitData = String.format("tree %040x\n" +
                parents +
                        "author %s\n" +
                        "committer %s %d +0100\n\n" +
                        "%s",
                new Random().nextLong(),
                author,
                author,
                date.getTime(),
                commitMessage);

        return RevCommit.parse(commitData.getBytes());

    }
```

Metoda pomocnicza, tworząca spersonalizowany mock klasy RevCommit.

5.4 URLReaderTest

```
    @Test
    public void checkIfExistsRemote() throws Exception {
        assertEquals(URLReader.checkIfExistsRemote("mama"), false);
    }
```

Test sprawdzający, że dane repozytorium zdalne nie istnieje.

6 Podsumowanie

Dzięki danym które projekt udostępnia można stworzyć wiele ciekawych statystyk, które pozwolą lepiej zrozumieć przyzwyczajenia i potrzeby pracowników i sprawią, że praca bedzie efaktywniejsza.