Github

Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

Docente:

Flávio Henrique Ferreira Couto

Turma 4

Rui Miguel Oliveira - up201000619@fe.up.pt

Sérgio Salgado - up201406136@fe.up.pt

Janeiro 2019

Índice

1. Descrição e Requerimentos do Sistema Informal

Para a unidade curricular de Métodos Formais em Engenharia de Software escolhemos o tema do Github para trabalho de grupo. O GitHub é uma plataforma de gestão de código-fonte com controlo de versão usando o Git. De modo a que este projecto não ficasse demasiado extenso, decidimos implementar parte das funcionalidades que o website permite.

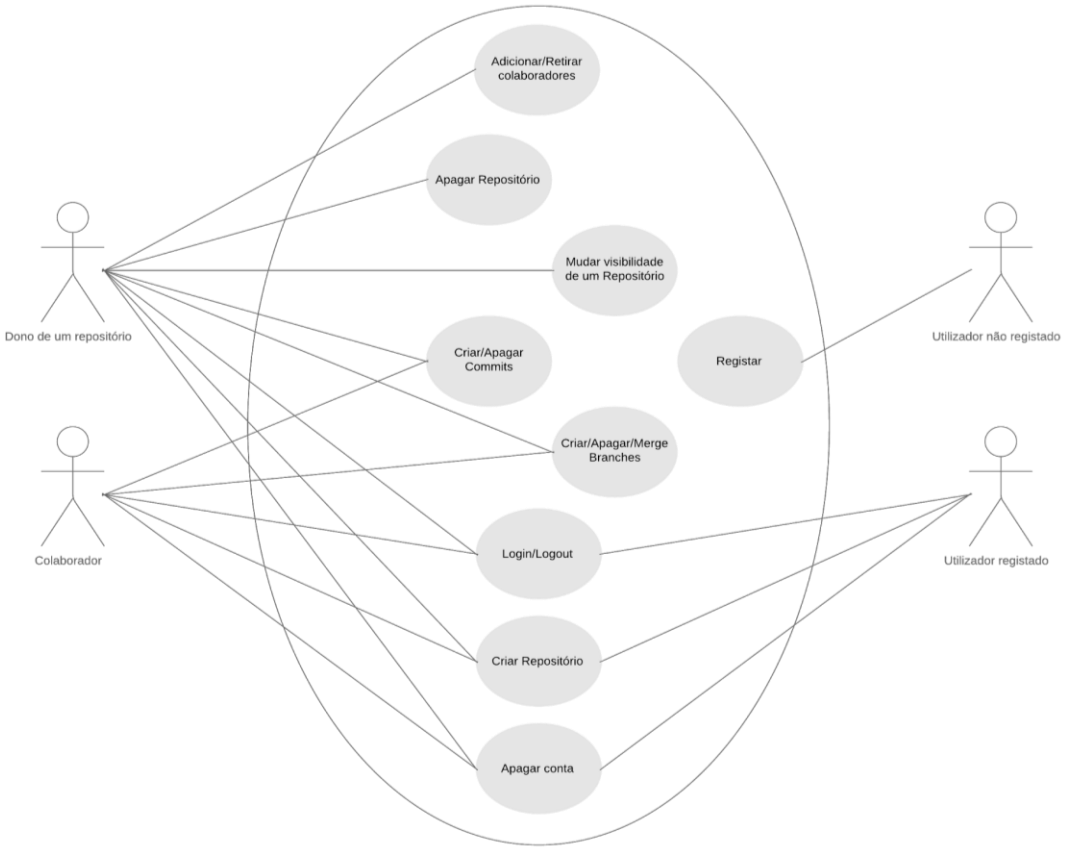
Lista de requisitos:

ID	**Prioridade**	**Descrição**
R1	Obrigatório	Permitir um utilizador registar-se e apagar a sua conta
R2	Obrigatório	Permitir um utilizador fazer login e logout
R3	Obrigatório	Permitir um utilizador criar e apagar repositórios
R4	Obrigatório	Permitir um utilizador adicionar e retirar colaboradores ao seu repositório
R5	Obrigatório	Permitir um utilizador mudar a visibilidade do seu repositório (público/privado)
R6	Obrigatório	Permitir um utilizador criar e apagar *branches* num repositório a que esteja associado
R7	Obrigatório	Permitir um utilizador fazer *commits* para um *branch*
R8	Obrigatório	Permitir um utilizador fazer *merge* de *branches* diferentes
R9	Obrigatório	Cada repositório tem de estar associado a pelo menos um utilizador (o dono do repositório)
R10	Obrigatório	Só o dono do repositório é que pode adicionar colaboradores
R11	Obrigatório	Cada repositório tem de ter um ID único
R12	Obrigatório	Cada utilizador só pode fazer *commits* para um repositório do qual seja dono ou colaborador
R13	Obrigatório	Cada *commit* tem de estar obrigatoriamente associado a um utilizador, a um *branch* e a um repositório
R14	Obrigatório	Cada utilizador só pode criar *branches* para um repositório do qual seja dono ou colaborador
R15	Obrigatório	Só se pode fazer *merge* de dois *branches* que estão no mesmo repositório

R16	Obrigatório	Quando um utilizador faz *merge*, o *branch* original é apagado e os seus *commits* são adicionados ao *branch* que permanece

2. Modelo Visual de UML

2.1. Diagrama de Casos de uso



De notar que o dono de um repositório e um colaborador também são utilizadores registados

Descrição dos casos de uso mais importantes:

Cenário	**Registar**
Descrição	Como utilizador não registado quero registar-me para usufruir do serviço Github
Pré-condições	1. Nenhum utilizador está com a sessão iniciada
Pós-condições	O utilizador fica registado no sistema
Passos	1. O utilizador escolhe a opção de registar 2. E insere as credenciais exigidas

Cenário	**Iniciar sessão (*Login*)**
Descrição	Como utilizador registado quero iniciar a sessão para usufruir do serviço Github
Pré-condições	1. Nenhum utilizador está com a sessão iniciada 2. O utilizador já está registado no sistema
Pós-condições	O utilizador fica com a sessão iniciada
Passos	1. O utilizador escolhe a opção de *Login* 2. E insere as credenciais exigidas

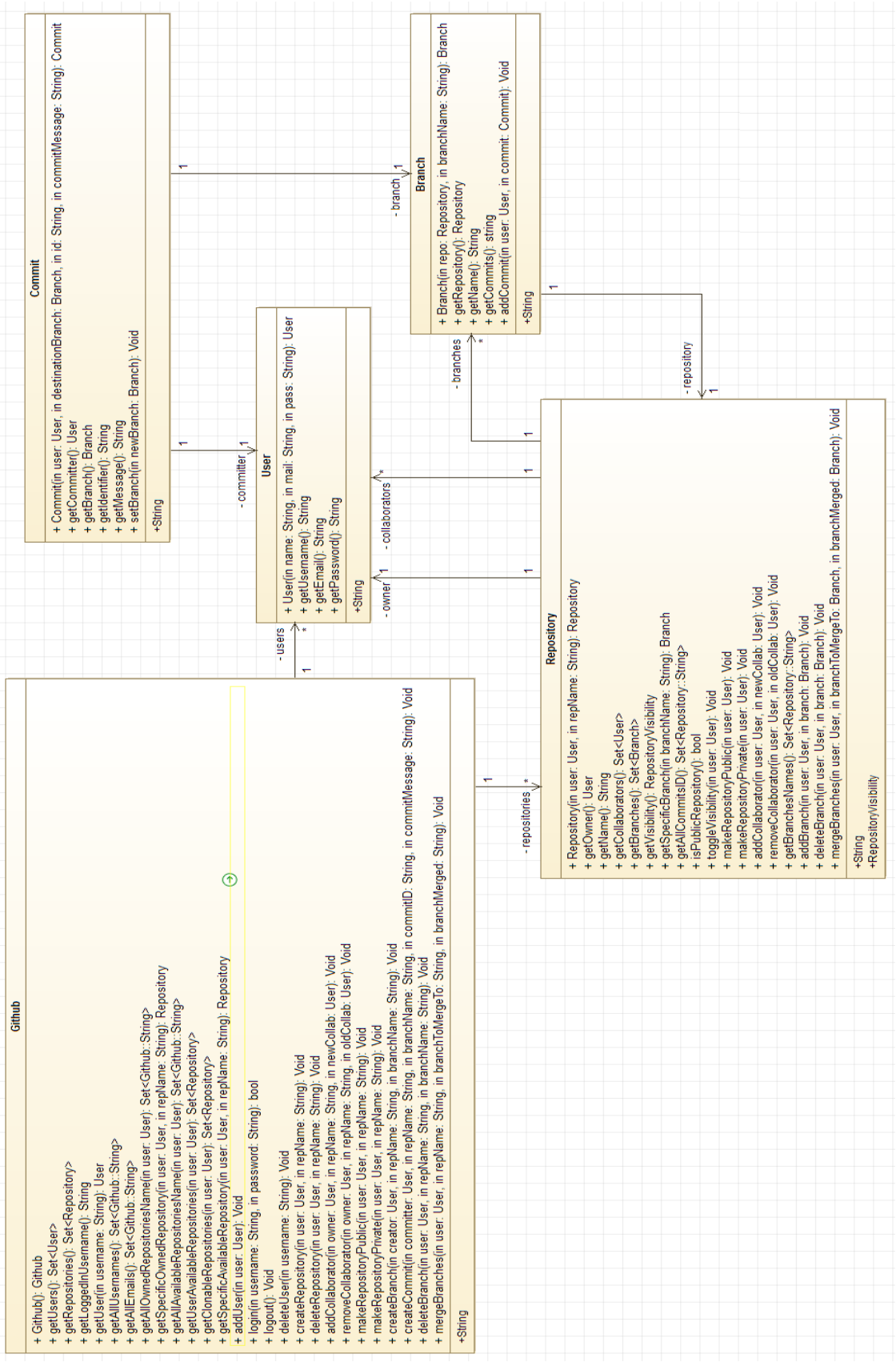
Cenário	**Criar Repositório**
Descrição	Como utilizador registado quero criar um repositório para guardar o meu código-fonte
Pré-condições	1. O utilizador tem a sessão iniciada
Pós-condições	O utilizador tem um repositório novo
Passos	1. O utilizador escolhe a opção de criar um repositório 2. Insere os dados exigidos

Cenário	**Adicionar colaboradores**
Descrição	Como utilizador registado quero adicionar outros utilizadores como colaboradores para poderem aceder ao meu repositório
Pré-condições	1. O utilizador tem a sessão iniciada 2. O utilizador é dono de um repositório público ou privado
Pós-condições	O utilizador fica com um colaborador adicionado
Passos	1. O utilizador escolhe a opção de adicionar um colaborador ao seu repositório 2. Insere os dados exigidos

Cenário	**Criar um *branch***
Descrição	Como utilizador registado quero criar um *branch* para dividir melhor a produção de código do meu repositório
Pré-condições	1. O utilizador tem a sessão iniciada 2. O utilizador é dono ou colaborador do repositório onde quer fazer o *branch*
Pós-condições	O utilizador fica com um novo branch no repositório
Passos	1. O utilizador escolhe a opção de criar um novo branch 2. Insere os dados exigidos

Cenário	**Fazer um *commit***
Descrição	Como utilizador registado quero fazer commit para um *branch* de um repositório
Pré-condições	1. O utilizador tem a sessão iniciada 2. O utilizador é dono ou colaborador do repositório onde quer fazer o *commit*
Pós-condições	O utilizador fica com o seu *commit* gravado
Passos	1. O utilizador escolhe a opção de fazer *commit* 2. Insere os dados exigidos

2.2. Modelo de classes



3. Modelo Formal em VDM++

3.1. Github

```
class Github

types
-- TODO Define types here
	public String = Utilities`String;

values
-- TODO Define values here

instance variables
-- TODO Define instance variables here
	private users: set of User := {};
	private repositories: set of Repository := {};
	private loggedInUsername: String := "undef";

operations
-- TODO Define operations here

	-- GitHub Constructor
	public Github: () ==> Github
	Github() == (
		return self;
	)
	post users = {} and repositories = {};

	/*****************************************/
	/**************     GET INFO ***************/
	/*****************************************/

	-- Get all users
	public pure getUsers: () ==> set of User
	getUsers() == (
		return users;
	);

	-- Get all repositories
	public pure getRepositories: () ==> set of Repository
	getRepositories() == (
		return repositories;
	);

	-- Get currently logged in user username
	public pure getLoggedInUsername: () ==> String
	getLoggedInUsername () == (
		return loggedInUsername;
	);

	-- Get specific user, given the username
	public pure getUser: String ==> User
	getUser(username) == (
		for all u in set users do (
			if username = u.getUsername()
				then return u;
		);
```

```
		return new User();
	);

	-- Get all currently existant usernames
	public pure getAllUsernames: () ==> set of String
	getAllUsernames() == (
		dcl usernames: set of String := {};
		for all u in set users do
			usernames := usernames union {u.getUsername()};
		return usernames;
	);

	-- Get all currently existant emails
	public pure getAllEmails: () ==> set of String
	getAllEmails() == (
		dcl emails: set of String := {};
		for all u in set users do
			emails := emails union {u.getEmail()};
		return emails;
	);

	-- Get all current repositories owned by a specific user
	public pure getAllOwnedRepositoriesName: User ==> set of String
	getAllOwnedRepositoriesName(user) == (
		dcl userRepos: set of String := {};
		for all r in set repositories do
			if r.getOwner() = user
				then userRepos := userRepos union {r.getName()};
		return userRepos;
	)
	pre {user} inter users = {user}; -- utilizador esta no sistema;

	-- Get a specific repository owned by a user
	public pure getSpecificOwnedRepository: User * String ==> Repository
	getSpecificOwnedRepository(user, repName) == (
		dcl rep: Repository;
		for all r in set repositories do
			if r.getOwner() = user and r.getName() = repName
				then rep := r;
		return rep;
	)
	pre {user} inter users = {user} -- utilizador esta no sistema;
		and {repName} inter getAllOwnedRepositoriesName(user) = {repName}; --
o user e o dono do repositorio;

	-- Get all current repositories available for a specific user
	public pure getAllAvailableRepositoriesName: User ==> set of String
	getAllAvailableRepositoriesName(user) == (
		dcl userRepos: set of String := {};
		for all r in set repositories do
			if r.getOwner() = user or user in set r.getCollaborators()
				then userRepos := userRepos union {r.getName()};
		return userRepos;
	)
	pre {user} inter users = {user}; -- utilizador esta no sistema;

	-- Get all repositories where a user can commit and create branches
	public pure getUserAvailableRepositories: User ==> set of Repository
	getUserAvailableRepositories(user) == (
```

```
            dcl repos: set of Repository := {};
            for all r in set repositories do
                  if r.getOwner() = user or user in set r.getCollaborators()
                        then repos := repos union {r};
            return repos;
      )
      pre {user} inter users = {user}; -- utilizador esta no sistema;

      -- Get all clonable repositories by a user
      public pure getClonableRepositories: User ==> set of Repository
      getClonableRepositories(user) == (
            dcl repos: set of Repository := getUserAvailableRepositories(user);
            for all r in set repositories do
                  if r.isPublicRepository()
                        then repos := repos union {r};
            return repos;
      );

      -- Get a specific repository from all available repositories from a user
      public pure getSpecificAvailableRepository: User * String ==> Repository
      getSpecificAvailableRepository(user, repName) == (
            dcl availableRepos: set of Repository :=
getUserAvailableRepositories(user);
            for all r in set availableRepos do
                  if r.getName() = repName
                        then return r;
            return new Repository();
      )
      pre {user} inter users = {user} -- utilizador esta no sistema;
            and {repName} inter getAllAvailableRepositoriesName(user) =
{repName}; -- o nome introduzido existe na lista de repositorios disponivel

      /***************************************/
      /***************   USER ****************/
      /***************************************/

      -- Create a new user
      public addUser: User ==> ()
      addUser(user) == (
            users := users union {user}
      )
      pre {user.getUsername()} inter getAllUsernames() = {} -- nome do utilizador
nao esta a ser utilizado
            and {user.getEmail()} inter getAllEmails() = {} -- email do
utilizador nao esta a ser utilizado
            and {user} inter users = {} -- utilizador nao esta no sistema
      post {user.getUsername()} inter getAllUsernames() = {user.getUsername()}
            and {user.getEmail()} inter getAllEmails() = {user.getEmail()}
            and {user} inter users = {user}; -- utilizador esta no sistema

      -- User login
      public login: String * String ==> bool
      login(username, password) == (
            if getUser(username).getPassword() = password
                  then (
                        loggedInUsername := username;
                        return true
                  );
            return false;
```

```
	)
	pre {username} inter getAllUsernames() <> {} -- o user existe no sistema
		and len username > 0 and len password > 0; -- o input existe

	-- User logout
	public logout: () ==> ()
	logout() == (
		loggedInUsername := "undef";
	)
	pre {loggedInUsername} inter getAllUsernames() = {loggedInUsername}
	post loggedInUsername = "undef";

	-- Delete user and all owned repositories
	public deleteUser: String ==> ()
	deleteUser(username) == (
		if username = loggedInUsername
			then logout();

		for all r in set repositories do
			if r.getOwner() = getUser(username)
				then repositories := repositories \ {r};

		users := users \ {getUser(username)};
	)
	pre {getUser(username)} inter users = {getUser(username)}
	post {username} inter getAllUsernames() = {};

	/**************************************/
	/************REPOSITORY ***************/
	/**************************************/

	-- Create repository
	public createRepository: User * String ==> ()
	createRepository(user, repName) == (
		dcl newRepo: Repository := new Repository(user, repName);
		repositories := repositories union {newRepo};
	)
	pre {repName} inter getAllOwnedRepositoriesName(user) = {} -- repositorio
nao existe
		and user in set users -- user criador existe no sistema
		and len repName > 0 -- input existe
	post {repName} inter getAllOwnedRepositoriesName(user) = {repName}; --
repositorio esta no sistema

	-- Delete repository
	public deleteRepository: User * String ==> ()
	deleteRepository(user, repName) == (
		dcl repToDelete: Repository := getSpecificOwnedRepository(user,
repName);
		repositories := repositories \ {repToDelete};
	)
	pre	{user} inter users = {user} -- utilizador existe
		and {getSpecificOwnedRepository(user, repName)} inter repositories =
{getSpecificOwnedRepository(user, repName)} -- o repositorio existe
		and user = getSpecificOwnedRepository(user, repName).getOwner() -- o
utilizador é o owner deste repositorio
	post {repName} inter getAllOwnedRepositoriesName(user) = {}; -- o
repositorio nao existe
```

```
	-- Add collaborator to an owned repository
	public addCollaborator: User * String * User ==> ()
	addCollaborator(owner, repName, newCollab) == (
		dcl repo: Repository := getSpecificOwnedRepository(owner, repName);
		repo.addCollaborator(owner, newCollab);
	)
	pre {owner, newCollab} inter users = {owner, newCollab} -- quer o owner,
quer o novo colaborador estao no sistema
		and {repName} inter getAllOwnedRepositoriesName(owner) = {repName} --
o repositorio existe
		and getSpecificOwnedRepository(owner, repName).getCollaborators()
inter {newCollab} = {} -- o utilizador a ser introduzido ainda nao e colaborador
	post getSpecificOwnedRepository(owner, repName).getCollaborators() inter
{newCollab} = {newCollab};

	-- Remove collaborator from an owned repository
	public removeCollaborator: User * String * User ==> ()
	removeCollaborator(owner, repName, oldCollab) == (
		dcl repo: Repository := getSpecificOwnedRepository(owner, repName);
		repo.removeCollaborator(owner, oldCollab);
	)
	pre {owner, oldCollab} inter users = {owner, oldCollab} -- quer o owner,
quer o antigo colaborador estao no sistema
		and {repName} inter getAllOwnedRepositoriesName(owner) = {repName} --
o repositorio existe
		and getSpecificOwnedRepository(owner, repName).getCollaborators()
inter {oldCollab} = {oldCollab} -- o utilizador a ser removido era colaborador
	post getSpecificOwnedRepository(owner, repName).getCollaborators() inter
{oldCollab} = {}; -- o antigo colaborador nao existe

	-- Make a repository public
	public makeRepositoryPublic: User * String ==> ()
	makeRepositoryPublic(user, repName) == (
		dcl repo: Repository := getSpecificOwnedRepository(user, repName);
		repo.makeRepositoryPublic(user);
	)
	pre {user} inter users = {user} -- o owner esta no sistema
		and {repName} inter getAllOwnedRepositoriesName(user) = {repName} --
o repositorio existe
	post getSpecificOwnedRepository(user, repName).getVisibility() = <Public>;

	-- Make a repository private
	public makeRepositoryPrivate: User * String ==> ()
	makeRepositoryPrivate(user, repName) == (
		dcl repo: Repository := getSpecificOwnedRepository(user, repName);
		repo.makeRepositoryPrivate(user);
	)
	pre {user} inter users = {user} -- o owner esta no sistema
		and {repName} inter getAllOwnedRepositoriesName(user) = {repName} --
o repositorio existe
	post getSpecificOwnedRepository(user, repName).getVisibility() = <Private>;

	/**************************************/
	/**********  BRANCH AND COMMIT ***********/
	/**************************************/

	-- Create a new branch in a repository
	public createBranch: User * String * String ==> ()
	createBranch(creator, repName, branchName) == (
```

```
		dcl rep: Repository := getSpecificAvailableRepository(creator,
repName);
		dcl newBranch: Branch := new Branch(rep, branchName);
		rep.addBranch(creator, newBranch);
	)
	pre {creator} inter users = {creator} -- criador do branch existe no
sistema
		and {getSpecificAvailableRepository(creator, repName)} inter
getUserAvailableRepositories(creator) = {getSpecificAvailableRepository(creator,
repName)}; -- o repositorio onde esta a ser criado o branch esta disponivel para o
criador

	-- Create a new commit to a existant branch
	public createCommit: User * String * String * String * String ==> ()
	createCommit(committer, repName, branchName, commitID, commitMessage) == (
		dcl rep: Repository := getSpecificAvailableRepository(committer,
repName);
		dcl branch: Branch := rep.getSpecificBranch(branchName);
		dcl newCommit: Commit := new Commit(committer, branch, commitID,
commitMessage);
		branch.addCommit(committer, newCommit);
	)
	pre {committer} inter users = {committer} -- criador do commit existe no
sistema
		and {getSpecificAvailableRepository(committer, repName)} inter
getUserAvailableRepositories(committer) =
{getSpecificAvailableRepository(committer, repName)}; -- o repositorio onde esta a
ser criado o commit esta disponivel para o criador

	-- Delete an existant branch
	public deleteBranch: User * String * String ==> ()
	deleteBranch(user, repName, branchName) == (
		dcl rep: Repository := getSpecificAvailableRepository(user, repName);
		dcl branch: Branch := rep.getSpecificBranch(branchName);
		rep.deleteBranch(user, branch);
	)
	pre {user} inter users = {user} -- user existe no sistema
		and {getSpecificAvailableRepository(user, repName)} inter
getUserAvailableRepositories(user) = {getSpecificAvailableRepository(user,
repName)}; -- o repositorio esta disponivel para o user

	-- Merge two existant branches
	public mergeBranches: User * String * String * String ==> ()
	mergeBranches(user, repName, branchToMergeTo, branchMerged) == (
		dcl rep: Repository := getSpecificAvailableRepository(user, repName);
		dcl branch1: Branch := rep.getSpecificBranch(branchToMergeTo);
		dcl branch2: Branch := rep.getSpecificBranch(branchMerged);
		rep.mergeBranches(user, branch1, branch2);
	)
	pre {user} inter users = {user} -- user existe no sistema
		and {branchToMergeTo, branchMerged} inter
getSpecificAvailableRepository(user, repName).getBranchesNames() =
{branchToMergeTo, branchMerged} -- os branches existem
		and branchToMergeTo <> branchMerged -- os branches nao sao iguais
		and {getSpecificAvailableRepository(user, repName)} inter
getUserAvailableRepositories(user) = {getSpecificAvailableRepository(user,
repName)}; -- o repositorio esta disponivel para o user

functions
```

```
-- TODO Define functiones here

traces
-- TODO Define Combinatorial Test Traces here

end Github
```

3.2. User

```
class User

types
-- TODO Define types here
	public String = Utilities`String;

values
-- TODO Define values here

instance variables
-- TODO Define instance variables here
	private username: String;
	private email: String;
	private password: String;

	inv len username > 4 and len username < 20;
	inv len password > 5 and len password < 30;

operations
-- TODO Define operations here

	-- User constructor
	public User: String * String * String ==> User
	User(name, mail, pass) == (
		username := name;
		email := mail;
		password := pass;
		return self;
	);

	-- Get user's name
	public pure getUsername: () ==> String
	getUsername() == (
		return username;
	);

	public pure getEmail: () ==> String
	getEmail() == (
		return email;
	);

	public getPassword: () ==> String
	getPassword() == (
		return password;
	);

functions
-- TODO Define functiones here
```

```
traces
-- TODO Define Combinatorial Test Traces here

end User
```

3.3. Repository

```
class Repository

types
-- TODO Define types here
	public String = Utilities`String;
	public RepositoryVisibility = Utilities`RepositoryVisibility;

values
-- TODO Define values here

instance variables
-- TODO Define instance variables here
	private owner: User;
	private name: String;
	private collaborators: set of User := {};
	private branches: set of Branch := {};
	private visibility: RepositoryVisibility := <Public>;

	inv len name > 4 and len name < 25;
	inv card branches > 0; -- existe sempre um branch no repositorio

operations
-- TODO Define operations here

	-- Repository constructor
	public Repository: User * String ==> Repository
	Repository(user, repName) == (
		owner := user;
		name := repName;
		branches := branches union {new Branch(self, "master")};
		return self;
	)
	post collaborators = {} and visibility = <Public>;

	-- Get repository owner
	public pure getOwner: () ==> User
	getOwner() == (
		return owner;
	);

	-- Get repository name
	public pure getName: () ==> String
	getName() == (
		return name;
	);

	-- Get repository collaborators
	public pure getCollaborators: () ==> set of User
	getCollaborators() == (
		return collaborators;
	);
```

```
-- Get repository branches
public pure getBranches: () ==> set of Branch
getBranches() == (
	return branches;
);

public pure getVisibility: () ==> RepositoryVisibility
getVisibility() == (
	return visibility;
);

public pure getSpecificBranch: String ==> Branch
getSpecificBranch(branchName) == (
	for all b in set branches do
		if b.getName() = branchName
			then return b;
	return new Branch();
)
pre branchName in set getBranchesNames();

public pure getAllCommitsID: () ==> set of String
getAllCommitsID() == (
	dcl IDs: set of String := {};
	for all b in set branches do
		for all c in set (elems b.getCommits()) do
			IDs := IDs union {c.getIdentifier()};
	return IDs;
);

-- Get repository visibility
public pure isPublicRepository: () ==> bool
isPublicRepository() == (
	return visibility = <Public>;
);

public toggleVisibility: User ==> ()
toggleVisibility(user) == (
	if visibility = <Public>
		then visibility := <Private>
	else visibility := <Public>
)
pre owner = user;

public makeRepositoryPublic: User ==> ()
makeRepositoryPublic(user) == (
	visibility := <Public>;
)
pre owner = user
post visibility = <Public>;

public makeRepositoryPrivate: User ==> ()
makeRepositoryPrivate(user) == (
	visibility := <Private>;
)
pre owner = user
post visibility = <Private>;

public addCollaborator: User * User ==> ()
```

```
	addCollaborator(user, newCollab) == (
		collaborators := collaborators union {newCollab};
	)
	pre owner = user and {newCollab} inter collaborators = {};

	public removeCollaborator: User * User ==> ()
	removeCollaborator(user, oldCollab) == (
		collaborators := collaborators \ {oldCollab}
	)
	pre owner = user and {oldCollab} inter collaborators = {oldCollab}
	post {oldCollab} inter collaborators = {};

	public pure getBranchesNames: () ==> set of String
	getBranchesNames() == (
		dcl names: set of String := {};
		for all b in set branches do
			names := names union {b.getName()};
		return names;
	);

	public addBranch: User * Branch ==> ()
	addBranch(user, branch) == (
		branches := branches union {branch};
	)
	pre (owner = user or collaborators inter {user} = {user})  -- user tem
permissoes
		and {branch.getName()} inter getBranchesNames() = {}; -- nao ha
branches repetidos

	public deleteBranch: User * Branch ==> ()
	deleteBranch(user, branch) == (
		branches := branches \ {branch};
	)
	pre (owner = user or collaborators inter {user} = {user})  -- user tem
permissoes
		and {branch} inter getBranches() = {branch} -- o branch existe
	post {branch} inter getBranches() = {};

	public mergeBranches: User * Branch * Branch ==> ()
	mergeBranches(user, branchToMergeTo, branchMerged) == (
		deleteBranch(user, branchMerged);
		for all c in set (elems branchMerged.getCommits()) do
			(
				branchToMergeTo.addCommit(user, c);
				c.setBranch(branchToMergeTo);
			)
	)
	pre (owner = user or collaborators inter {user} = {user})  -- user tem
permissoes;
		and {branchToMergeTo, branchMerged} inter getBranches() =
{branchToMergeTo, branchMerged} -- os branches existe
	post {branchToMergeTo, branchMerged} inter getBranches() =
{branchToMergeTo}; -- o branch merged ja nao existe

functions
-- TODO Define functiones here

traces
-- TODO Define Combinatorial Test Traces here
```

```
end Repository
```

3.4. Branch

```
class Branch

types
-- TODO Define types here
	public String = Utilities`String;

values
-- TODO Define values here

instance variables
-- TODO Define instance variables here
	private name: String;
	private repository: Repository;
	private commits: seq of Commit := [];

operations
-- TODO Define operations here
	public Branch: Repository * String ==> Branch
	Branch(repo, branchName) == (
		name := branchName;
		repository := repo;
		return self;
	);

	public pure getRepository: () ==> Repository
	getRepository() == (
		return repository;
	);

	public pure getName: () ==> String
	getName() == (
		return name;
	);

	public pure getCommits: () ==> seq of Commit
	getCommits() == (
		return commits;
	);

	public addCommit: User * Commit ==> ()
	addCommit(user, commit) == (
		commits := commits ^ [commit];
	)
	pre (user in set repository.getCollaborators() or user =
repository.getOwner())
		and {commit.getIdentifier()} inter repository.getAllCommitsID() = {};
-- o ID do commit e unico no respectivo repositorio

functions
-- TODO Define functiones here

traces
-- TODO Define Combinatorial Test Traces here
```

```
end Branch
```

3.5. Commit

```
class Commit

types
-- TODO Define types here
	public String = Utilities`String;

values
-- TODO Define values here

instance variables
-- TODO Define instance variables here
	private committer: User;
	private branch: Branch;
	private identifier: String;
	private message: String;

operations
-- TODO Define operations here
	public Commit: User * Branch * String * String ==> Commit
	Commit(user, destinationBranch, id, commitMessage) == (
		committer := user;
		branch := destinationBranch;
		identifier := id;
		message := commitMessage;
		return self;
	);

	public pure getCommitter: () ==> User
	getCommitter() == (
		return committer;
	);

	public pure getBranch: () ==> Branch
	getBranch() == (
		return branch;
	);

	public pure getIdentifier: () ==> String
	getIdentifier() == (
		return identifier;
	);

	public pure getMessage: () ==> String
	getMessage() == (
		return message;
	);

	-- Useful for merges
	public setBranch: Branch ==> ()
	setBranch(newBranch) == (
		branch := newBranch;
	);
```

```
functions
-- TODO Define functiones here

traces
-- TODO Define Combinatorial Test Traces here

end Commit
```

3.6. Utilities

```
class Utilities
types
-- TODO Define types here
	public String = seq1 of char;
	public RepositoryVisibility = <Public> | <Private>;

values
-- TODO Define values here

instance variables
-- TODO Define instance variables here

operations
-- TODO Define operations here

	-- Constructor
	public Utilities: () ==> Utilities
	Utilities() == (
		return self;
	);

functions
-- TODO Define functiones here

traces
-- TODO Define Combinatorial Test Traces here

end Utilities
```

4. Validação do Modelo

4.1. Classe de Testes (GithubTest)

```
class GithubTest
types
-- TODO Define types here
values
-- TODO Define values here
instance variables
  g : Github := new Github();
  u : Utilities := new Utilities();
  u1 : User := new User("user1", "mail1", "password1");
  u2 : User := new User("user2", "mail2", "password2");
  u3 : User := new User("user3", "mail3", "password3");
  rep : Repository;
  rep2 : Repository;
  b: Branch;
  c: Commit;
operations
  private assertTrue: bool ==> ()
  assertTrue(cond) == return
  pre cond;

       -- Create user test
  private createUserTest: () ==> ()
  createUserTest() ==
  (
       -- There are no users in the initial state
       assertTrue({} = g.getUsers());
       -- We add a couple of users
       g.addUser(u1);
       g.addUser(u2);
       -- They can be reached within Github class
       assertTrue("user1" in set g.getAllUsernames());
       assertTrue("user2" in set g.getAllUsernames());
       assertTrue(u1 = g.getUser("user1"));
       assertTrue(u2 = g.getUser("user2"));
       assertTrue({u1, u2} = g.getUsers());
       );

       -- Login Test
  private loginTest: () ==> ()
  loginTest() ==
  (
       -- No logged in user in the initial state
       assertTrue("undef" = g.getLoggedInUsername());
       -- Cannot login with wrong password
       assertTrue(not g.login("user1", "wrong password"));
       -- Can login with right password
       assertTrue(g.login("user1", "password1"));
       -- Confirm user is logged in
       assertTrue("user1" = g.getLoggedInUsername());
  );
```

```
    -- Create Repository Test
private createRepositoryTest: () ==> ()
createRepositoryTest() ==
(
    -- No repositories were created in initial state
    assertTrue({} = g.getRepositories());
    -- Create one repository
    g.createRepository(u1, "repName");
    -- Confirm said repository is created
    rep := g.getSpecificOwnedRepository(u1, "repName");
    assertTrue({rep} = g.getUserAvailableRepositories(u1));
    assertTrue(rep = g.getSpecificAvailableRepository(u1, "repName"));
    assertTrue({"repName"} = g.getAllAvailableRepositoriesName(u1));
    -- Confirm the creator is the owner of the repository
    assertTrue(u1 = rep.getOwner());
    -- Confirm master branch was automatically created
    assertTrue({"master"} = rep.getBranchesNames());
    -- Confirm repository is automatically public
    assertTrue(rep.isPublicRepository());
);

    -- Add Colaborator Test
private addCollaboratorTest: () ==> ()
addCollaboratorTest() ==
(
    -- Confirm no colaborator was added
    assertTrue({} = rep.getCollaborators());
    -- Add a colaborator
    g.addCollaborator(u1, "repName", u2);
    -- Confirm colaborator was added
    assertTrue({u2} = rep.getCollaborators());
    assertTrue({"repName"} = g.getAllAvailableRepositoriesName(u2));
    assertTrue({rep} = g.getUserAvailableRepositories(u2));
);

    -- Repository Visibility Tests
private repositoryVisibilityTests: () ==> ()
repositoryVisibilityTests() ==
(
    -- Confirm Repository is public by default
    assertTrue(rep.isPublicRepository());
    -- Toggle Visibility
    rep.toggleVisibility(u1);
    -- Confirm change to Private
    assertTrue(not rep.isPublicRepository());
    -- Toggle Visibility
    rep.toggleVisibility(u1);
    -- Confirm change to Public
    assertTrue(rep.isPublicRepository());
    -- Make it Private
    rep.makeRepositoryPrivate(u1);
    -- Confirm change to Private
    assertTrue(<Private> = rep.getVisibility());
    -- Make it Public
    rep.makeRepositoryPublic(u1);
    -- Confirm change to Public
    assertTrue(<Public> = rep.getVisibility());
);
```

```
    -- Clonable Repositories Test
private getClonableRepositoriesTest: () ==> ()
getClonableRepositoriesTest() ==
(
    -- Create new user
    g.addUser(u3);
    -- Repository cannot be cloned by others because it is private
    g.makeRepositoryPrivate(u1, "repName");
    assertTrue(<Private> = rep.getVisibility());
    assertTrue({rep} = g.getClonableRepositories(u1));
    assertTrue({rep} = g.getClonableRepositories(u2));
    assertTrue({} = g.getClonableRepositories(u3));
    -- Repository can be cloned because it is public
    g.makeRepositoryPublic(u1, "repName");
    assertTrue(<Public> = rep.getVisibility());
    assertTrue({rep} = g.getClonableRepositories(u1));
    assertTrue({rep} = g.getClonableRepositories(u2));
    assertTrue({rep} = g.getClonableRepositories(u3));
);

    -- Create Branch Test
private createBranchTest: () ==> ()
createBranchTest() ==
(
    -- Create a branch
    g.createBranch(u2, "repName", "branchName");
    -- Confirm branch was created
    assertTrue({"master", "branchName"} = rep.getBranchesNames());
    assertTrue({rep.getSpecificBranch("master"),
rep.getSpecificBranch("branchName")} = rep.getBranches());
    -- Confirm branch is associated with the right repository
    b := rep.getSpecificBranch("branchName");
    assertTrue(rep = b.getRepository());
    -- Confirm branch has no commits yet
    assertTrue([] = b.getCommits());
);

    -- Delete Branch Test
private deleteBranchTest: () ==> ()
deleteBranchTest() ==
(
    -- Create a branch
    g.createBranch(u2, "repName", "deleteBranchTest");
    -- Confirm creation
    assertTrue("deleteBranchTest" in set rep.getBranchesNames());
    -- Delete a branch
    g.deleteBranch(u2, "repName", "deleteBranchTest");
    -- Confirm deletion
    assertTrue("deleteBranchTest" not in set rep.getBranchesNames());
);

    -- Create Commit Test
private createCommitTest: () ==> ()
createCommitTest() ==
(
        -- Create a commit
    g.createCommit(u2, "repName", "branchName", "1commitID", "commitMessage1");
    -- Confirm creation
```

```
        assertTrue(1 = len b.getCommits());
        -- Confirm commit data
        c := b.getCommits()(1);
        assertTrue(u2 = c.getCommitter());
        assertTrue(b = c.getBranch());
        assertTrue("commitMessage1" = c.getMessage());
                -- Create another commit
        g.createCommit(u1, "repName", "branchName", "2commitID", "commitMessage2");
                -- Confirm creation
        assertTrue(2 = len b.getCommits());
    );

        -- Merge Branches Test
    private mergeBranchesTest: () ==> ()
    mergeBranchesTest() ==
    (
        -- Create a new branch
        g.createBranch(u1, "repName", "newBranchName");
                -- Create a commit for said branch
        g.createCommit(u1, "repName", "newBranchName", "3commitID",
"commitMessage3");
                -- Merge branches
        g.mergeBranches(u1, "repName", "branchName", "newBranchName");
                -- Confirm merged branch was deleted
        assertTrue("newBranchName" not in set rep.getBranchesNames());
                -- Confirm commits were merged
        assertTrue(3 = len b.getCommits());
        -- Same test but with a collaborator
        g.createBranch(u2, "repName", "newBranchName2");
        g.createCommit(u2, "repName", "newBranchName2", "4commitID",
"commitMessage4");
        g.mergeBranches(u2, "repName", "branchName", "newBranchName2");
        assertTrue("newBranchName2" not in set rep.getBranchesNames());
        assertTrue(4 = len b.getCommits());
    );

        -- Remove Collaborator Test
    private removeCollaboratorTest: () ==> ()
    removeCollaboratorTest() ==
    (
        -- Confirm collaborator exists
        assertTrue({u2} = rep.getCollaborators());
        -- Remove Collaborator
        g.removeCollaborator(u1, "repName", u2);
        -- Confirm removal
        assertTrue({} = rep.getCollaborators());
    );

        -- Logout Test
    private logoutTest: () ==> ()
    logoutTest() ==
    (
        -- Confirm user is logged in
        assertTrue("user1" = g.getLoggedInUsername());
        -- Logout
        g.logout();
        -- Confirm user is logged out
        assertTrue("undef" = g.getLoggedInUsername());
    );
```

```
    -- Delete User Test
private deleteUserTest: () ==> ()
deleteUserTest() ==
(
    -- Delete a collaborator
    -- Confirm user exists
    assertTrue({u1, u2, u3} = g.getUsers());
    -- Confirm user is logged in
    assertTrue(g.login("user2", "password2"));
    -- Delete user
    g.deleteUser("user2");
    -- Confirm user is logged out
    assertTrue("undef" = g.getLoggedInUsername());
    -- Confirm user does not exist
    assertTrue({u1, u3} = g.getUsers());

    -- Delete an owner of a repository
    -- Create a repository for u3
    g.createRepository(u3, "repNameD");
    -- Confirm creation
    rep2 := g.getSpecificOwnedRepository(u3, "repNameD");
    assertTrue({rep, rep2} = g.getClonableRepositories(u1));
    -- Delete user
    g.deleteUser("user3");
    -- Confirm user does not exist
    assertTrue({u1} = g.getUsers());
    -- Confirm repository does not exist
    assertTrue({rep} = g.getClonableRepositories(u1));
);

-- Delete Repository Test
private deleteRepositoryTest: () ==> ()
deleteRepositoryTest() ==
(
    -- Confirm repository exists
    assertTrue({rep} = g.getRepositories());
    assertTrue({rep} = g.getUserAvailableRepositories(u1));
    -- Log in with repository owner
    assertTrue(g.login("user1", "password1"));
    -- Delete repository
    g.deleteRepository(u1, "repName");
    -- Confirm repository was deleted
    assertTrue({} = g.getRepositories());
    assertTrue({} = g.getUserAvailableRepositories(u1));
);

private doTests: () ==> ()
doTests() ==
(
    createUserTest();
    loginTest();
    createRepositoryTest();
        addCollaboratorTest();
        repositoryVisibilityTests();
        getClonableRepositoriesTest();
        createBranchTest();
        deleteBranchTest();
        createCommitTest();
```

```
                mergeBranchesTest();
                removeCollaboratorTest();
                logoutTest();
                deleteUserTest();
                deleteRepositoryTest();
    );

    public static main: () ==> ()
    main() ==
    (
      new GithubTest().doTests();
    );
functions
-- TODO Define functiones here
traces
-- TODO Define Combinatorial Test Traces here
        end GithubTest
```

5. Verificação do Modelo

5.1. Exemplo de verificação de domínio

Uma das *Proof Obligations* geradas pelo Overture é:

No.	PO Name	Type
85	Repository`makeRepositoryPublic(User)	operation establishes postcondition

O código sob análise é o seguinte:

```
public makeRepositoryPublic: User ==> ()
makeRepositoryPublic(user) == (
	visibility := <Public>;
)
pre owner = user
post visibility = <Public>;
```

Esta operação altera a visibilidade do repositório para pública alterando a variável *visibility* da classe *Repository* para *<Public>*. Neste caso, a pós-condição obriga a *visibility* a ficar pública.

Da *proof obligation view* temos então:

```
(forall user:User & ((owner = user) => (<Public> = <Public>)))
```

5.2. Exemplo de verificação de invariante

Outra das *Proof Obligations* geradas pelo Overture é:

No.	PO Name	Type
92	Repository`addBranch(User, Branch)	state invariant holds

O código sob análise é o seguinte:

```
public addBranch: User * Branch ==> ()
addBranch(user, branch) == (
	branches := branches union {branch};
)
pre (owner = user or collaborators inter {user} = {user})  -- user tem permissoes
	and {branch.getName()} inter getBranchesNames() = {}; -- nao ha branches repetidos
```

Esta operação adiciona um novo *branch* ao repositório adicionando uma nova entrada no *set branches* da classe *Repository* com um novo elemento da class *Branch*.

Da *proof obligation view* temos então:

```
(forall user:User, branch:Branch & ((((owner = user) or ((collaborators
inter {user}) = {user})) and (({(branch.getName)()} inter getBranchesNames()) =
{})) => (((((len name) > 4) and ((len name) < 25)) and ((card branches) > 0)) =>
((((len name) > 4) and ((len name) < 25)) and ((card (branches union {branch})) >
0)))))
```

6. Geração de código

Para simular a aplicação Github com o modelo que foi criado, gerou-se código Java utilizando a ferramenta do *Overture*. Este código, que se encontra presente em "**generated/java/src/model**", foi incorporado numa aplicação Java que se encontra na pasta "**generated/java**" do ficheiro ZIP submetido, sendo também criada uma pasta de backup em "**interface**" caso ocorra um overwrite por parte do *Overture*. Foi criada uma interface de linha de comandos, criada usando o *Eclipse* e importando o projeto criado pela geração de código do *Overture*.

7. Conclusões

O modelo desenvolvido cobre todos os requisitos estipulados pelo grupo. Infelizmente não implementámos ficheiros, apenas conceitualizamos o seu papel no processo todo do Github.

Este projecto demorou cerca de 18 horas por pessoa para concluir. Perto de um terço deste tempo foi dedicado a fazer o relatório final e o restante a fazer código.

A divisão de carga de trabalho é sempre difícil de quantificar, mas se tiver de tender para algum membro então foi ligeiramente superior para o Sérgio Salgado uma vez que foi mais proactivo na divisão de tarefas.

8. Referências

P. G. Larsen and B. S. Hansen and H. Brunn and N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others. Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language. December 1996.

C.B. Jones. The Meta-Language: A Reference Manual. In The Vienna Development Method: The Meta-Language, Springer-Verlag, 1978.