Labo 2 GitHub

In het eerste labo gingen we aan de slag met Git als middel om je eigen code te beheren en te publiceren op GitHub. Git laat echter ook toe om met meerdere gebruikers eenzelfde repository te beheren. In dit tweede labo gaan we eerst in op het achterliggende concept van een branch, waarna we zullen experimenteren met het gebruik van Git met meerdere gebruikers.

1 Branches

Tot nu toe was de geschiedenis van je code lineair: elke commit, behalve de nieuwste, had juist één opvolger. Dit hoeft echter niet het geval te zijn. Git laat toe om vanuit elke commit te vertakken naar alternatieve sporen in de geschiedenis. Een **branch** is een lijst met opeenvolgende commits doorheen de code.

Standaard bevind je je op de branch met als naam `master`. Elke repository heeft dan ook een branch met die naam. Je kwam die tijdens het eerste labo vast wel al eens tegen, bijvoorbeeld bij het pushen vanuit NetBeans.

1.1 Een branch maken

Laten we het aanmaken van branches even onder de loep nemen. Open Git Bash en voer volgende opdrachten uit:

```
cd /c/temp
git clone git@github.ugent.be:iii/so1-branch.git
cd so1-branch
```

Dit haalt de inhoud van de repository `so1-branch` af en plaatst hem in `C:\temp\so1-branch`. De laatste `cd`-opdracht wijzigt de werkdirectory naar de repository.

Tip: de foutmelding `The authenticity of host ... can't be established` kan je verhelpen door een nieuwe sleutelpaar aan te maken. Zie vorig labo; `ssh-keygen -t rsa`.

Als je met `git log` de geschiedenis van de code bekijkt, zul je merken dat er al enkele commits werden uitgevoerd. Deze zijn echter nog steeds lineair. Je kunt dit nagaan met de volgende opdracht:

```
gitk --all
```

Alle commits staan in `gitk` netjes onder elkaar. Er zijn dus nog geen vertakkingen gebeurd. (Sluit dit venster om verder te kunnen werken in GitBash.)

We zullen nu een nieuwe branch maken om een experimentele feature toe te voegen aan de applicatie, met als uiteindelijk doel om onze aanpassingen weer te integreren in `master`. In het bijzonder zullen we een eenvoudige grafische interface maken voor de applicatie. We noemen onze branch dan ook `gui`.

Om een nieuwe branch genaamd `gui` te maken vanuit de huidige commit, in ons geval de allernieuwste, voer je de volgende opdracht in:

```
git branch gui
```

Dit geeft geen uitvoer, maar zoals je intussen weet, is dit een goed teken. Git heeft nu immers een branch toegevoegd. Zoals je aan de prompt kunt zien, bevinden we ons echter nog op `master`. We moeten dus nog overschakelen op `gui`:

```
git checkout gui
```

Dit verplaatst de `HEAD`, ofwel de leeskop, naar de branch `gui`. Aan de bestanden in de repository is niets veranderd, want voorlopig heeft de nieuwe branch geen inhoud.

1.2 Commits op een branch

Laten we de branch dus wat inhoud geven. Plaats het bestand GuiUtil.java in de repository. Dit is een hulpklasse met twee functies om eenvoudige vensters weer te geven op het scherm.

De bedoeling is nu om de klasse `Greeter` zo te wijzigen dat ze gebruikmaakt van `GuiUtil`. Eerst voegen we echter `GuiUtil` alvast toe aan de repository:

```
git add GuiUtil.java
git commit -m "Adding GuiUtil"
```

(Vraagt GitBash '`Please tell me who you are`'? Haal er het vorige labo bij, en configureer GitBash.)

Onze eerste commit op de `gui`-branch is daarmee een feit. De `master`-branch en de `gui`-branch leven nu in vrede naast elkaar.

1.3 Terug naar 'master'

Alvorens we `Greeter.java` een grafische schil geven, gaan we terug naar `master` om een aanpassing uit te voeren. De opdracht om af te wisselen tussen branches kennen we reeds:

```
git checkout master
```

Nu is het zo dat `Greeter` toelaat dat een lege string wordt ingevoerd. Dit is natuurlijk niet de bedoeling. Voeg een `while`-lus toe die de naam van de gebruiker blijft vragen zolang de string uit nul karakters bestaat.

Je kunt de volgende opdracht gebruiken om de code in één beweging te compileren en, indien geslaagd, uit te voeren:

```
javac *.java && java Greeter
```

Tip: indien de java-compiler `javac` niet gevonden wordt, zoek je hem via de windows-verkenner op in de map `Program Files/Java/jdk_???/bin`. Als je hem gelokaliseerd hebt, voeg je het volledige pad toe. Je krijgt dan allicht zoiets:

```
/C/Program\ Files/Java/jdk1.8.0_101/bin/javac *.java && java Greeter
```

Werkt de code naar behoren? Dan kun je je werk opslaan in een commit:

```
git add Greeter.java
git commit -m "Disallowing empty names"
```

Onze codegeschiedenis is nu nog meer vertakt. Laten we nog eens een beroep doen op `gitk` om een overzicht te krijgen:

```
gitk --all
```

Begrijp je wat er wordt weergegeven?

1.4 Terug naar 'gui'

Tijd om onze klasse `Greeter` op te smukken. We bevinden ons nog op de `master`-branch, dus eerst moeten we terug naar `gui`:

```
git checkout gui
```

Vervang nu in `Greeter.java` alle uitvoer naar `System.out` door oproepen van `GuiUtil.showText`, en alle gebruikersinvoer door oproepen van `GuiUtil.askForText`. Opgelet: je mag de klasse `GuiUtil` niet wijzigen! Gebruik bovenstaande `javac`-opdracht om te testen.

Gelukt? Mooi zo! Met de opdracht `git diff` kun je een overzicht krijgen van de wijzigingen die je aanbracht. Ook deze kun je nu opslaan in een commit op de `gui`-branch:

```
git add Greeter.java
git commit -m "Making greeter graphical"
```

Voer nogmaals `gitk --all` uit om je ervan te vergewissen dat de `gui`-branch en de `master`-branch zijn gewijzigd zoals je verwachtte.

1.5 Merge

De tegenhanger van vertakken is samenvoegen, in het Engels **merge**. We zullen nu de wijzigingen die we aanbrachten op de `gui`-branch integreren in de (intussen gewijzigde) `master`-branch.

Dit is een courante manier van werken: je maakt een nieuwe branch voor experimentele aanpassingen, terwijl de 'stabiele' `master`-branch onafhankelijk verderleeft. Zodra de tijdelijke branch in een stabiele toestand is, hevel je de wijzigingen die je op die branch ontwikkelde over naar `master`.

We zullen `gui` integreren in `master` en moeten daarom allereerst overschakelen naar `master`:

```
git checkout master
```

Herinner je dat we `Greeter.java` aanpasten nadat we de branch-operatie uitvoerden. Wanneer we nu de twee branches samenvoegen, zal Git proberen om de wijzigingen die intussen gebeurden met elkaar te verenigen. Normaal gesproken loopt dit goed af. In geval van conflicten is echter een extra merge-stap vereist.

We wagen het erop! Laten we de wijzigingen van `gui` integreren in `master`:

```
git merge gui
```

Tenzij je `Greeter.java` heel efficiënt aanpaste, verschijnt er waarschijnlijk CONFLICT. (We geven toe dat we het een beetje met opzet deden.)

Het conflict oplossen is echter niet zo moeilijk. Als je `Greeter.java` opent, zul je merken dat er markeringen werden toegevoegd, om aan te geven welk(e) fragment(en) van het bestand op beide branches werden gewijzigd. Hoe je deze precies moet samenvoegen, is afhankelijk van je implementatie. De bedoeling is in elk geval om een stuk code te verkrijgen dat enerzijds de naam van de gebruiker blijft vragen tot deze niet leeg is (zoals op `master`) en anderzijds een beroep doet op `GuiUtil` om te interageren met de gebruiker (zoals op `gui`).

Is het conflict opgelost, dan moet je ten slotte Git nog op de hoogte brengen. De opdracht ken je al, want dit is niets anders dan een commit die de twee branches in elkaar laat lopen. Merk op dat `GuiUtil.java` een nieuw bestand is voor `master`, en je dus beide bronbestanden moet doorgeven aan `add`:

```
git add Greeter.java GuiUtil.java
git commit -m "Solved conflict in Greeter"
```

Voer een laatste keer `gitk --all` uit. Begrijp je wat er is gebeurd?

2 Samenwerken

Nu je begrijpt hoe branches werken, hoe je deze kunt samenvoegen en hoe je conflicten oplost, ben je zo goed als klaar om met twee of meer studenten aan één repository te sleutelen. Vanzelfsprekend zal dit handig van pas komen bij het project, waar je binnenkort aan begint.

Vanaf hier gaan we ervan uit dat je de oefening met z'n tweeën maakt. Bespreek vooraf wie van jullie beiden de rol van Student 1 bekleedt en wie die van Student 2. Het is sterk aan te raden om elk aan een eigen pc te werken.

2.1 Een repository delen

Om een repository te delen, moet je er natuurlijk eerst eentje aanmaken. Student 1 surft daarvoor naar de GitHub-pagina New Repository en maakt de publieke repository `so1-teamwork` aan.

Hoewel zowel Student 1 als Student 2 nu reeds een kloon zou kunnen maken van de publieke repository, zullen we er eerst voor zorgen dat Student 2 meteen ook schrijfrechten verkrijgt op de repository van Student 1.

Hiertoe surft Student 1 naar zijn/haar repository `so1-teamwork` op GitHub en klikt hij/zij vervolgens op Settings (rechtsboven). Onder Collaborators kan vervolgens de gebruikersnaam van Student 2 worden ingevoerd om schrijfrechten toe te kennen.

2.2 Een eerste commit

Student 2 krijgt nu de eer om de eerste commit aan te maken. Daarvoor moet hij/zij wel eerst over een lokale kopie van de repository beschikken. Dit kopiëren gebeurt, zoals we weten, met de `clone`-opdracht (kopieer het juiste adres vanop GitHub):

```
cd /c/temp
git clone git@github.ugent.be:student1/so1-teamwork.git
cd so1-teamwork
```

Merk op dat de gebruikersnaam van Student 1 moet worden ingevuld en dus niet die van Student 2. Eerstgenoemde blijft immers eigenaar van de repository. Wel kan eender wie een eigen kopie maken van de repository en zo zijn eigen versie van de applicatie verder ontwikkelen. Deze bewerking heet een fork, maar wordt in deze labo's niet behandeld.

Vaak bevat de eerste commit een readme-bestand, met daarin wat uitleg over de repository. Student 2 maakt daarom het bestand `README.md` aan. De extensie `.md` duidt op de opmaaktaal Markdown, een eenvoudig alternatief voor HTML. Geef het bestand deze inhoud:

```
Teamwork
========

Een experiment met teamwork in Git.
```

Daarnaast voegt Student 2 nog een tweede bestand toe, genaamd `authors.txt`. Hierin komen de namen van iedereen die heeft meegewerkt aan de inhoud van de repository. Voorlopig is dat dus enkel Student 2.

Nu kan Student 2 de twee bestanden opnemen in de eerste commit. We weten inmiddels hoe hij deze op GitHub kan plaatsen:

```
git add README.md authors.txt
git commit -m "Initial commit"
git push -u origin master
```

Surf naar de repository `so1-teamwork` op GitHub. Wat is er met het readme-bestand gebeurd?

2.3 Meerdere auteurs

Nu haalt ook Student 1 de repository-inhoud af met `clone`. Voer dus ook op de andere pc de `clone`-opdracht uit. Meteen krijg je zo de eerste commit van Student 2 mee binnen.

Student 1 wil natuurlijk ook vermeld worden in de lijst met auteurs. Daarom past hij/zij de inhoud van `authors.txt` aan. Sla de wijziging op in een commit en push deze naar GitHub:

```
notepad authors.txt
git add authors.txt
git commit -m "Adding author"
git push
```

2.4 Synchronisatie

Doordat Student 1 een wijziging pushte naar GitHub, bevat de repository op de pc van Student 2 nu verouderde informatie. Hij/zij kan echter makkelijk de nieuwste inhoud afhalen met de tegenhanger van `push`, die logischerwijs `pull` heet:

```
git pull
```

Op deze manier zijn beide pc's weer up-to-date. Nu ken je reeds alle opdrachten die nodig zijn om samen aan een repository te sleutelen.

Maar wat indien de auteurs tegelijk wijzigingen aanbrengen? We proberen dit even uit.

2.5 Simultane wijzigingen

Voeg **allebei** wat inhoud toe aan `README.md`, elk op je eigen pc. Voor meer informatie over het Markdown-formaat kun je op deze pagina terecht, maar de eigenlijke inhoud van het bestand is in dit geval niet zo belangrijk.

Het readme-bestand wijzigen op je eigen pc vormt natuurlijk geen probleem. Ook één of meerdere commits uitvoeren kan prima; je werkt immers gewoon met je lokale repository. Zorg ervoor dat je beiden minstens één nieuwe commit hebt waarin je de inhoud van `README.md` wijzigt. Pas dus `README.md` aan en sla je wijzigingen op in een commit.

Probeer de commit(s) nu te pushen naar GitHub. Wie deze opdracht als eerste las, boft: voor hem/haar verandert er helemaal niets. Zijn/haar lokale versie van de repository was up-to-date, zodat de wijzigingen probleemloos werden doorgegeven aan GitHub.

De andere student heeft daarentegen minder geluk. De opdracht `git push` slaagt immers enkel wanneer beide repository's gesynchroniseerd zijn. Voor je `git push` kunt uitvoeren, moet je daarom eerst met `git pull` de nieuwste commits binnenhalen.

Soms kun je daarna meteen `git push` uitvoeren, soms niet. De reden ken je eigenlijk al: er gebeurt een merge-bewerking. Het enige verschil met onze merge tussen `master` en `gui` van eerder, is dat we dit keer de lokale branch `master` samenvoegen met `origin/master`, die we afhaalden van GitHub. Opnieuw doet Git dus zijn best om automatisch de twee samen te voegen. Zijn er geen conflicten, dan duikt automatisch een commit op die de twee branches samenvoegt; anders moet je, net als in deel 1, eerst manueel de inhoud samenvoegen. Wanneer het samenvoegen is voltooid, kun je dan (eindelijk) `git push` uitvoeren.

Probeer dit een aantal keer uit. Spreek vooraf af wie als eerste `git push` zal uitvoeren en tracht te voorspellen of er conflicten zullen optreden.

3 Via NetBeans

In het eerste labo maakten we al kennis met de Git-ondersteuning van NetBeans. Ook branches worden als fundament van Git natuurlijk ondersteund. Bijgevolg kun je ook met meerdere gebruikers aan een gedeelde Git-repository werken zonder de NetBeans-omgeving te verlaten.

Indien je het voorgaande volledig begrijpt, zal je moeiteloos overweg kunnen met de nog niet besproken Git-functionaliteit van NetBeans. Zo zul je de optie Pull… snel terugvinden in de menu's, want in het eerste labo leerde je haar tegenhanger Push… al kennen. Ook de ondersteuning voor branches is niet meer dan een grafische schil rond de reeds besproken opdrachten.

Laten we om te beginnen onze `so1-teamwork`-repository importeren in NetBeans. **Beide studenten** kiezen daarvoor in het menu Team voor Git en vervolgens Clone. Vul de GitHub-URL van de gedeelde `so1-teamwork`-repository in en verwijs NetBeans opnieuw naar het bestand `id_rsa`. Klik tweemaal op Next. Vervolgens wordt gevraagd waar je de kloon wilt plaatsen. NetBeans stelt de map NetBeansProjects voor, wat prima is. Klik dus op Finish.

Dan wordt de repository gekloond. NetBeans gaat vervolgens op zoek naar NetBeans-projecten in de repository, maar helaas zijn die spoorloos. Daarom stelt hij voor om een NetBeans-project te initialiseren. Dat is een goed idee, dus klik op Create Project…

Welk soort project willen we aanmaken? We kiezen onder Java voor de eerste optie, Java Application. Klik op Next, vul als naam `HelloWorld` in en klik op Finish.

NetBeans zet de klasse `HelloWorld` klaar. Student 1 plaatst in de `main`-methode volgende code:

```
System.out.println("Hello, world!");
```

Student 2 daarentegen implementeert de klasse als volgt:

```
System.out.println("Hallo, wereld!");
```

Tijd om het werk van **beide studenten** op GitHub te publiceren. Eerst moeten we daarvoor lokaal een commit aanmaken. Rechtsklik dus op het project en kies onder Git voor Commit… Vul een duidelijke boodschap in en kies vervolgens onder Remote voor Push to Upstream.

Als je deel 1 en 2 van dit labo begreep, dan weet je ook waarom dit niet zomaar lukt: er treedt opnieuw een conflict op. Tracht dit nu op te lossen met behulp van NetBeans. De bewerkingen zijn dezelfde als via de opdrachtlijn, maar uiteraard moet je ze nu via de menu's van NetBeans aanspreken.

Heb je **allebei** een push uitgevoerd? Dan heb je ook Git in NetBeans helemaal onder de knie!

4 Tot slot

Daarmee zijn we aan het einde gekomen van deze labo's Git. Zoals al een paar keer vermeld, hebben we hiermee zeker niet alle functionaliteit besproken. Met hetgeen je nu weet, kun je echter wel al op professionele wijze omspringen met Git en GitHub, en je project aanvatten.

Voor nog meer documentatie zijn de voornaamste bronnen het gratis boek Pro Git en de hulppagina's van GitHub. Daarnaast vind je op git ready een heleboel nuttige tips, gerangschikt volgens moeilijkheidsgraad.

De beste manier om de kneepjes van Git te leren, is en blijft echter door oefening. Verplicht jezelf om bij zo veel mogelijk vakken en/of hobbyprojectjes code en andere documenten op GitHub te plaatsen. Dit heeft immers niets dan voordelen! Bovendien ontstaan zo spontaan realistische situaties waarin je handig gebruik kunt maken van de andere mogelijkheden van Git, zoals de courante opdrachten `diff`, `revert` en `rebase`. Op termijn kun je dan zelfs aan de slag met onder andere `tag`, `stash` en `cherry-pick`.

Vroeg of laat zal je ongetwijfeld vast komen te zitten. Dat is hoegenaamd geen schande. Sterker nog: je bent vast niet de eerste met je probleem. Vaak loont het dan ook de moeite om Google af te schuimen. Vragen stellen aan de docenten kan natuurlijk ook, maar de échte Git-experten zijn bij bosjes te vinden op onder andere Stack Overflow.

We hopen in elk geval dat deze labo's je hebben overtuigd van de kracht van Git. Veel succes!