|  |  |
| --- | --- |
|  |  |
| Versiebeheer met Git |  |
|  |  |
|  | 07-12-2022  Advies versiebeheer Git |
|  | Stein Jonker |

Contents

[Probleem 3](#_Toc121316005)

[Oplossing 3](#_Toc121316006)

[Hoe werkt git? 4](#_Toc121316007)

[Repository 4](#_Toc121316008)

[Local & Remote 4](#_Toc121316009)

[Commit 5](#_Toc121316010)

[Merge 6](#_Toc121316011)

[Branches 6](#_Toc121316012)

[Git opstellen 7](#_Toc121316013)

[Branchemodel 7](#_Toc121316014)

# Probleem

Versiebeheer is belangrijk voor projecten. Het is namelijk uitermate frustrerend wanneer iets fout gaat en er dan ook nog ontdekt dat je nu veel werk opnieuw moet doen.

Om werkende versies vast te leggen wordt vaak een kopie van een map gemaakt en hernoemt naar “Project-v1”. De volgende krijgt v2, v3 etc.

Dit is niet de beste manier van versiebeheer, want:

* Het is lastig overzicht te bewaren over alle verschillende versies.

Wat was het verschil tussen v1 en v2? Kun je dat ergens terugvinden?

* Naarmate er meer versies zijn, gaat het meer opslag opnemen op je harde schijf. Je dupliceert namelijk elke keer alle bestanden in de map.
* Stel: je zit nu aan versie 5 te werken, maar je had in versie 2 een 3D model gebruikt, maar die in V3 verwijderd.

Hoe kun je nu makkelijk dit 3D model terugvinden en weer in versie 5 gebruiken? En wat als het model een bepaald gedrag had, zoals het rondrijden van een auto? Hoe vind je dan snel de code die het rondrijden bevat?

Alle code in V5 overschrijven met de code uit V2 is geen optie, want dan ben je de wijzigingen van V3, 4 en 5 kwijt.

* Als de computer kapotgaat, alle versies kwijt
* Vaak maak je weinig versies, die grote wijzigingen hebben. Met veel kleine versies gebruik je namelijk meer schijfruimte en wordt het overzicht steeds lastiger.

Het is dus een makkelijke manier van versiebeheer, maar wanneer je op een oude versie wilt teruggrijpen is dat eigenlijk best lastig.

# Oplossing

Git! Git is een versiebeheersysteem.

Het is uitgebracht in 2005 en ontwikkelt door Linus Torvalds. Linus is ook de hoofdontwikkelaar van Linux.

Het gebruik van git is een stuk complexer, maar tegenwoordig is het makkelijk te leren. Git is namelijk een programma zonder scherm: alles moet je met commando’s op de commandline doen.

Dit is voor beginners niet ideaal en soms ook gevaarlijk omdat er geen beschermingen ingebouwd zijn, waardoor het mogelijk is dat je versies per ongeluk verwijderd.

Gelukkig zijn er verschillende tools ontwikkelt die een gebruikersinterface hebben gemaakt, maar in de achtergrond gebruik maken van git.

Een aantal voorbeelden zijn GitKraken, Github (GH vanaf nu) Desktop en TurtoiseGit. Zelf gebruik ik Github Desktop.

## Hoe werkt git?

Een aantal belangrijke termen in git zijn:

* Repositories
* Local & Remote
* Commit
* Fetch / pull
* Push
* Merge
* Branche

Deze termen per hoofdstuk uitleggen.

### Repository

Een repository is map waarvoor versiebeheer wordt bijgehouden. Deze map is verder helemaal hetzelfde als elk andere map op een Windows of Linux machine.

Een repository kan je gebruiken om één project in te stoppen, of meerdere projecten in één.

Als voorbeeld, je kunt 2 repositories (2 losse mappen) maken voor de Monchyplein en het Kronenburg project.

Je kan er ook voor kiezen om deze in één map te stoppen, je krijgt dan een ‘monorepo’: één repository voor meerdere projecten.

### Local & Remote

Een repository maak je eerst aan op een andere machine, een server. Dit kan bijvoorbeeld github.com, bitbucket.com of gitlab.com. Met deze websites maak je gebruik van de servers die de website ter beschikking stelt, maar je kunt ook je eigen vergelijkbare website hosten.

De server wordt ook de remote genoemd. Local is de machine waarop je aan het werk bent, vaak je laptop of de standalone PC’s.

Graphical user interface, application

Description automatically generated

Figuur Clonen repository

Om de repository van de remote naar de local te krijgen, moeten we de repository ‘clonen’. Dit kan gemakkelijk met de Github Desktop tool. Onder file staat de knop Clone Repository. Hier kun je de repository opzoeken als hij op github staat, of een URL opgeven. Deze URL kun je op github/bitbucket/gitlab vinden. (Figuur 1)

A screenshot of a computer

Description automatically generated with medium confidenceWanneer je wijzigingen maakt worden deze in GH Desktop getoond. Aan de linkerkant is te zien dat settings.json is aangemaakt, want het heeft een groen plusje.

Figuur Wijzigingen

Het gele icoontje met een stip betekent dat de inhoud van het bestand is veranderd en een blauwe pijl betekent dat een bestand hernoemt of verplaatst is.

### Commit

A screenshot of a phone

Description automatically generated with medium confidenceOm deze wijzigingen vast te leggen als een versie, committen we. Dit kan ook in GH Desktop, dit is onderin te zien in Figuur 3.

Figuur Committen

We committen dan naar de repository op de local (dus de eigen machine)

Deze versie is daarom nog niet te zien voor anderen, want het staat nog niet op de server. Daarvoor moet je eerst nog “pushen”, hier is ook een knop voor in GH Desktop.

Het voordeel van deze aanpak is dat er soms een situatie is die je wel wilt vastleggen (committen) maar nog niet wilt delen met anderen omdat het nog niet helemaal af is.

Bijvoorbeeld als je projectafhankelijkheden aan het updaten bent, de update is nog niet af maar je wilt wel vastleggen dat deze versie werkt.

Het kan zijn dat iemand anders iets heeft gepushed terwijl jij aan het werk was.

Je moet dan eerst fetchen, voordat je kan pushen. Ook hier helpt GH desktop.

### Merge

Git is best wel goed in het samenvoegen van wijzigingen.

Neem de volgende situatie:

1. Je fetched de repository
2. A.txt heeft vier regels tekst
3. Op regel 3 staat een typefout, deze herstel je.
4. Iemand anders heeft in de tussentijd een vijfde regel toegevoegd
5. Je wilt committen maar omdat er nieuwe commits op de remote aanwezig zijn moet je eerst updaten: fetchen.
6. Git detecteert dat de remote en de local A.txt wil aanpassen. Hij past daarom “git merge” toe waardoor hij de wijzigingen probeert samen te voegen. Dat lukt in dit geval omdat de wijzigingen op verschillende regels zijn. Wanneer een wijziging op dezelfde regels worden gemaakt ontstaat er vaak een “git merge conflict”. Je moet dan handmatig het bestand gaan samenvoegen. Ook hier zijn tools voor, misschien zelfs in Unity ingebouwd.

### Branches

Samenwerken wordt best vervelend als je elke keer git conflicten moet oplossen. Deze conflicten ontstaan namelijk omdat twee of meer mensen op dezelfde versie wijzingen maken, waardoor de versie van één persoon steeds ‘oud’ is.

Daarom kun je in een team het beste samenwerken met “branches”. Je kunt het letterlijk zien als een tak.

Je hebt een stam, de “main” (vroeger master) branche. Vanaf de stam maak je een nieuwe branche. Deze geef je een naam van de feature of scenario die je aan het bouwen bent.

Vervolgens kun je in GH desktop naar deze branche wisselen. Dit is een kopie van de master branche op dat moment. Je maakt je wijzigingen en commit / pushed deze zoals gebruikelijk.

Vervolgens kun je de branche terug naar master mergen d.m.v. een pull-request. Met een pull-request krijg je een overzicht van alle bestanden die gewijzigd zijn. Met pull-requesten kun je ook een review proces afdwingen: je kunt instellen dat er reviewers toestemming moeten geven voordat de feature naar master mag.

Branches zijn schematisch weergegeven in Figuur 4



Figuur Branches

## Git opstellen

Ik adviseer om op een website zoals Github of Bitbucket één repository aan te maken: een monorepo voor alle projecten uitgevoerd door afdeling 3D.

Gemeente Amsterdam en gemeente Delft zijn op Github al te vinden!

Een mappenstructuur die kan werken is bijvoorbeeld:

Arnhem3D/

├─ Monchyplein/

│ ├─ models/

│ ├─ .../

├─ Kronenburg/

│ ├─ models/

│ ├─ ..../

├─ .gitignore

├─ README.md

Onder Monchyplein en Kronenburg staat dus de Unity code.

### Branchemodel

De master-branche moet je zien als een branche waar werkende, demoklare applicaties staan. De master branche wordt vaak gezien als de productiebranche. De laatste versie die de gebruiker kan downloaden, is met de source code in de master branche gemaakt.

Daarom raad ik aan om de master branche te beschermen tegen pushes: niemand kan direct op de master branche pushen.

Dit zorgt ervoor dat je een review-proces kan waarborgen en je ook niet iets per ongeluk kan kapot maken omdat je op de verkeerde branche werkte.

Daarnaast raad ik tijdelijke “feature” branches aan. Het ontwikkelen van een nieuwe feature doe je in een feature-branche. Wanneer de feature productie-klaar is, maak je een pull-request naar main aan. Hier kan de review plaatsvinden.

Als er meerdere personen aan één project werken is het verstandig om een branche voor dat project aan te maken. De feature-branches maken dan geen pull-requests meer aan naar main, maar naar deze branche.

Dit geeft de mogelijkheid om voltooide pull-requests in de project-branche te testen. Als blijkt dat er iets niet goed is in de samenwerking tussen twee los ontwikkelde features kan dit nog worden hersteld voordat alles naar main wordt gezet.