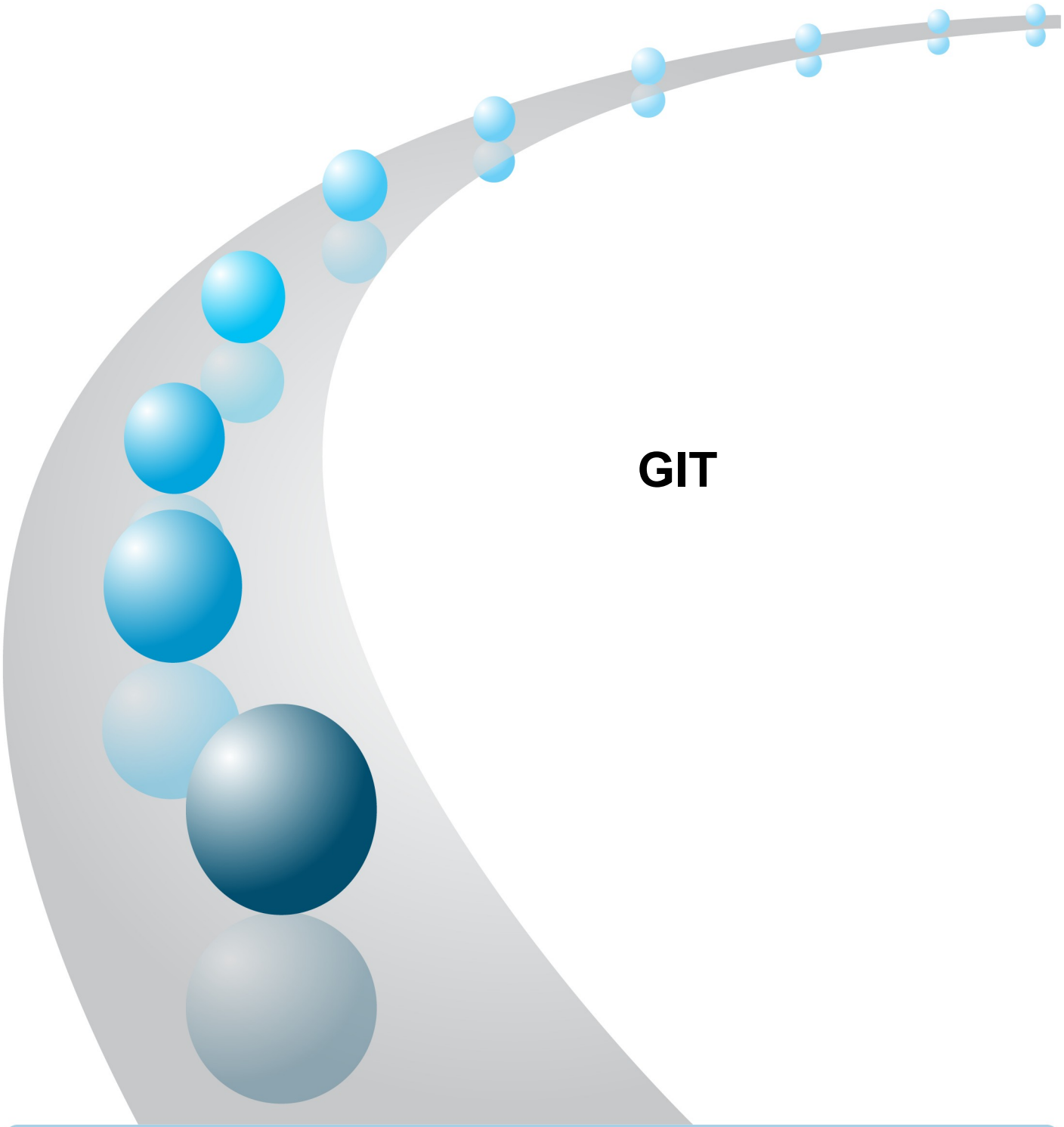
Noël Vaes
Java Trainer & Consultant



GIT

Roode Roosstraat 5
3500 Hasselt
België

+32 474 38 23 94
noel@noelvaes.eu
www.noelvaes.eu

Correspondentie inzake overnemen of reproductie kunt u richten aan:

Noël Vaes
Roode Roosstraat 5
3500 Hasselt
België

Tel: +32 474 38 23 94

noel@noelvaes.eu
www.noelvaes.eu

24/12/2020

Inhoudsopgave

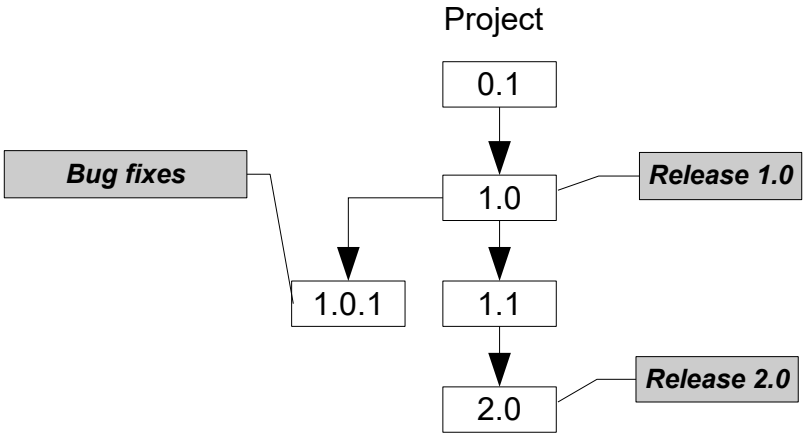
Hoofdstuk 1: GIT

1.1 Inleiding

Bij grotere programmeerprojecten waarbij tevens meerdere personen betrokken zijn, krijgt men te maken met een aantal typische problemen.

Zo doorloopt de ontwikkeling van de software **verschillende stadia**: er ontstaan verschillende versies van de software waaraan telkens wordt verder gebouwd. Op een bepaald moment doet men een *release* en wordt de software vrijgegeven voor uitvoerige testen en uiteindelijk wordt de software geschikt bevonden voor de eindklant. Na deze eerste versie komen er vaak nog aanpassingen die uiteindelijk via de nodige tussenliggende versies resulteren in een nieuwe officiële *release*.

Het eindproduct is de laatste versie in de rij van opeenvolgende versies waarbij telkens aanpassingen gedaan werden aan de bestaande versie.



Daar komt nog bij dat vaak op basis van een bestaande versie **parallel** gewerkt wordt aan kleine aanpassingen aan deze versie (*bug fixes*) en aan de voorbereiding van een nieuwe grote *release* met allerlei nieuwe functionaliteiten. Sommige delen van de code zijn hetzelfde, andere delen worden in het nieuwe traject helemaal herwerkt terwijl de oorspronkelijke code nog wordt onderhouden voor de ondersteuning van de oudere versie.

Verder wordt aan dergelijke grote programmeerprojecten meestal met **meerdere mensen** gewerkt. Samen delen ze dan dezelfde broncode waarbij ieder verantwoordelijk is voor een deel van die broncode. De broncode moet daarom op een of andere manier gedeeld kunnen worden zodat iedere programmeur er toegang toe heeft, de code kan compileren en uitvoeren en er ten slotte zijn eigen wijzigingen in kan aanbrengen. De andere programmeurs moeten nadien deze wijzigingen kunnen overnemen.

Vaak gebeurt het dat twee programmeurs tegelijkertijd bezig zijn met hetzelfde stuk broncode. Dan is het belangrijk dat de wijzigingen die door beide programmeurs gedaan werden, op correcte wijze geïntegreerd worden.

Bij de ontwikkeling van o*pen-source*-projecten worden deze problemen nog erger. Hier wordt door heel veel programmeurs gewerkt aan hetzelfde project dat meestal ook nog allerlei stadia heeft met heel wat vertakkingen.

Bovenstaande problemen maken een systeem noodzakelijk dat de broncode van een programmeerproject beheert en de verschillende versies van de software bijhoudt en toegankelijk maakt. Dit noemt men een systeem voor versiebeheer (*version control system of VCS*).

1.2 Centraal versus gedistribueerd

Systemen voor versiecontrole kunnen ingedeeld worden in twee groepen. Zo zijn er de systemen die de bestanden centraal beheren en waarbij iedere deelnemer van het project een (gedeeltelijke) lokale kopie hiervan maakt, vervolgens wijzigingen aanbrengt en nadien deze wijzigingen teruggeplaatst op de centrale locatie zodat ze beschikbaar zijn voor de andere deelnemers. We spreken over *Centralized Version Control Systems* of CVCS.

Het nadeel van dergelijke centrale systemen is evenwel dat gebruikers steeds *online* moeten zijn om de volledige functionaliteit te kunnen benutten.

Gekende gecentraliseerde systemen zijn *ClearCase, PVCS, SourceSafe* enzovoort. In de *Open-source*-wereld hebben we het inmiddels verouderde ***Concurrent Versions System*** (CVS) en diens opvolger ***Subversion****.*

Naast gecentraliseerde systemen zijn er ook gedistribueerde systemen waarbij de bestanden niet centraal bewaard worden maar op verschillende plaatsen. Men noemt dit daarom *Distributed Version Control Systems* of DVCS.

GIT is een dergelijk gedistribueerd systeem dat momenteel erg populair is. In deze cursus zullen we leren hoe GIT werkt.

1.3 Locking

Om de werking van versiecontrolesystemen te begrijpen moeten we eerst de problematiek van gedeelde bestanden verder bekijken. Indien twee gebruikers van hetzelfde bestand gebruik willen maken en dit bestand willen aanpassen, kunnen er conflicten optreden.

Gebruiker A kan een bestand uit het versiecontrolesysteem halen, dit aanpassen en het gewijzigde bestand opnieuw wegschrijven in het systeem.

Gebruiker B kan dat zelfde bestand ook lezen en aanpassen. Indien hij echter zijn wijzigingen iets later dan A wegschrijft, gaan de wijzigingen van A verloren.

Om dit probleem op te lossen zijn er twee mechanismen:

1.3.1 Lock-modify-unlock (pessimistic locking)

Een eerste mechanisme is het *lock-modify-unlock*-mechanisme. Men noemt dit ook wel *pessimistic locking.* Hierbij zal gebruiker A het bestand lezen en onmiddellijk vergrendelen. Nadat hij wijzigingen heeft aangebracht kan hij het nieuwe bestand opnieuw wegschrijven en wordt het bestand ontgrendeld. Zolang het bestand vergrendeld is, kan een andere gebruiker het bestand niet wijzigen. Alleen degene die het bestand vergrendeld heeft, kan wijzigingen aanbrengen.
Gebruiker B moet dus wachten totdat het bestand ontgrendeld is vooraleer hij een wijzigbare versie van het bestand kan bemachtigen.

Dit mechanisme heeft echter een aantal nadelen:

1. Indien gebruiker A **vergeet** het bestand opnieuw te ontgrendelen kan gebruiker B niet verder werken. De beheerder moet dan het bestand manueel opnieuw ontgrendelen.

2. Gebruiker B moet altijd wachten totdat het bestand ontgrendeld is, ook al zijn de wijzigingen die hij maakt **niet** noodzakelijk **conflicterend** met de wijzigingen die gebruiker A maakt. Denken we bijvoorbeeld aan een broncode-bestand waarbij de verschillende programmeurs op verschillende plaatsen wijzigingen aan het aanbrengen zijn, zonder dat die invloed hebben op elkaar. Er is dus heel wat vertraging bij het samenwerken aan hetzelfde bestand.
3. Het vergrendelen van een bestand kan de **valse illusie** wekken dat er niets mis kan gaan. Het kan evenwel gebeuren dat een bestand afhankelijk is van een ander bestand dat niet vergrendeld is en dus wel gewijzigd kan worden door een andere gebruiker. Het uiteindelijke resultaat kan dan toch nog foutief zijn.

1.3.2 Copy-modify-merge (optimistic locking)

Een ander mechanisme is *copy-modify-merge* waarbij de gebruiker een lokale kopie neemt van het bestand en hierin aanpassingen doet. Nadien zal hij het gewijzigde bestand terug proberen te plaatsen in de *repository*. Indien het centrale bestand intussen door iemand anders gewijzigd is, wordt getracht de wijzigingen te versmelten (*merge*). Indien dit niet lukt, moet het conflict opgelost worden door de gebruiker.

Dit systeem is veel flexibeler omdat hier geen bestanden vergrendeld worden en gebruikers tegelijkertijd wijzigingen kunnen aanbrengen. De wijzigingen worden indien nodig automatisch samengevoegd en enkel indien er een conflict optreedt, zal de gebruiker deze samenvoeging manueel moeten doen.

Men noemt dit ook wel *optimistic locking* omdat men er van uit gaat dat er geen conflicten zullen zijn (*optimistisch*) en als ze er toch zijn, moeten ze maar opgelost worden.

GIT heeft geen mogelijkheid tot het vergendelen van bestanden maar maakt steeds gebruik van deze *copy-modify-merge*.

1.4 Installatie en configuratie van GIT

GIT is een *open-source*-project dat beschikbaar is op de volgende website http://git-scm.com. Daar kan men de versie afhalen overeenkomstig het besturingssysteem. Na installatie kan men GIT gebruiken in een commandovenster.

De meeste IDE's zoals *Eclipse*, *IntelliJ* of *NetBeans* beschikken over grafische mogelijkheden om te werken met GIT. De vormgeving en mogelijkheden verschillen naargelang het product en de versie. In deze cursus zullen we leren werken met GIT via de commandolijn omdat deze benadering het best de werking van GIT uitlegt. Vanuit deze kennis kan men vervolgens makkelijk de grafische mogelijkheden van allerlei andere grafische toepassingen voor GIT begrijpen.

Opdracht 1: Installatie van GIT
In deze opdracht gaan we GIT installeren en configureren op ons systeem. We geven de mogelijkheden voor verschillende besturingssystemen:

- ***Windows***: haal GIT af op de volgende website: https://git-scm.com/downloads. Kies hierbij de versie overeenkomstig je besturingssysteem. Voer het installatieprogramma uit en kies hier bij telkens de voorgeselecteerde opties.
- ***Linux***: installeer GIT met een van de volgende commando's:
  ```
  apt-get install git
  yum install git
  ```
- Open een consolevenster en voer het volgende commando uit:
  ```
  git --version
  ```

Voor we beginnen met het gebruik van GIT, moeten we een aantal instellingen doen. GIT bewaart instellingen op drie niveaus:

1. **Systeem (*system)***: deze instellingen gelden voor alle gebruikers op het systeem. Ze worden opgeslagen in het bestand ***/etc/gitconfig.*** Bij *Windows* bevindt deze map zich in een submap van de installatiemap van GIT.
2. **Globaal (*global*)**: deze instellingen gelden voor een bepaalde gebruiker. Ze worden opgeslagen in het bestand ***.gitconfig*** in de *home directory* van de gebruiker.
3. **Lokaal (*local)***: deze instellingen gelden voor een specifiek project. Ze worden opgeslagen in de het bestand ***.git/config*** van het project.

Er bestaat een hiërarchie tussen de instellingen waarbij de instellingen van een lager niveau de instellingen van een hoger niveau kunnen overstemmen. Zo zal een instelling voor de gebruiker (*global)* de overhand krijgen over een systeeminstelling en zal een instelling voor een project (*local)* de overhand krijgen over zowel de gebruiker als het systeem.

Op dit moment moeten we alvast de volgende instellingen doen:

- De **naam** van de gebruiker.
- Het **e-mail-adres** van de gebruiker.

GIT gebruikt namelijk deze informatie bij het bewaren van wijzigingen. Deze instellingen gebeuren dus op het niveau van de gebruiker.

De instellingen van GIT kunnen we wijzigen en opvragen met het volgende commando:

```
git config
```

Dit commando kan gevolgd worden door allerlei opties. Om een volledige uitleg te krijgen over dit commando gebruik je de optie `--help`.

```
git config --help
```

Deze optie `--help` kunnen we overigens ook gebruiken voor alle andere commando's. Zo krijgen we snel de uitgebreide uitleg over deze commando's.

Om een overzicht te krijgen over de huidige instelling gebruiken we:

```
git config --list
```

Hierbij worden de instellingen van alle niveaus weergegeven. Dat is het standaardgedrag.

We kunnen eventueel ook expliciet het niveau aangeven:

```
git config --system --list (systeem)
git config --global --list (globaal)
git config --local  --list (lokaal)
```

Hiermee krijgen we enkel de instellingen op dat niveau te zien.

Instellingen kunnen we toevoegen met de optie `--add`. Standaard worden deze toegevoegd aan het lokale niveau tenzij we het expliciet anders bepalen met de optie `--system`, `--global` of `--local` (de standaard).

De naam en het e-mail-adres kunnen we als volgt instellen:

```
git config --global --add user.name "Noel Vaes"
git config --global --add user.email noel@noelvaes.eu
```

Dit zijn globale instellingen omdat ze specifiek zijn voor de gebruiker.

GIT zal voor het invoeren van boodschappen gebruikmaken van de standaard tekstverwerker. Bij *Windows* is dat bijvoorbeeld *Notepad*. Ontwikkelaars gebruiker vaak de betere variant *Notepad++*.

Indien deze toepassing geïnstalleerd is, kunnen we een extra instelling toevoegen zodat GIT *Notepad++* zal gebruiken:

```
git config --global --add core.editor "'C:\Program Files (x86)\
Notepad++\notepad++.exe' -multiInst -notabbar -nosession"
```

Opdracht 2: GIT configureren

In deze opdracht gaan we GIT configureren. We zullen onder andere onze eigen naam en e-mail-adres instellen. Tevens verkennen we de instellingen op de verschillende niveaus.

- Open een commandovenster.
- Voer de volgende commando's uit:
  ```
  git config --help
  git config --list
  git config --global --add user.name naam
  git config --global --add user.email e-mail
  git config --list
  git config --system --list
  git config --global --list
  git config --local --list
  ```
- Optioneel: installeer ***Notepad++***.
- Optioneel: voeg de volgende configuratie toe:
  ```
  git config --global --add core.editor "'C:\Program Files\
  Notepad++\notepad++.exe' -multiInst -notabbar -nosession"
  ```

1.5 Lokale *repositories*

Versiecontrolesystemen maken gebruik van *repositories*. Een *repository* is een opslagplaats waar bestanden en hun verschillende versies bewaard worden. Bij centrale systemen (CVCS) bevindt deze *repository* zich op een centrale plaats die door alle deelnemers toegankelijk is. De toegang gebeurt hierbij via een of ander netwerkprotocol.

Bij gedistribueerde systemen zoals GIT kan zo'n *repository* zich op verschillende plaatsen bevinden: dit kan op het lokale systeem van de ontwikkelaar zijn maar ook en tegelijkertijd op een of meerdere centraal toegankelijke plaatsen.

In deze paragraaf gaan we eerst werken met een lokale *repository* die zich op ons eigen systeem bevindt. Zodra we de basishandelingen met betrekking tot versiebeheer onder de knie hebben, gaan we zien hoe we gebruik kunnen maken van *remote repositories* en hoe we die met onze lokale *repository* kunnen synchroniseren.

1.5.1 Een *repository* aanmaken

In tegenstelling tot andere versiecontrolesystemen is het bij GIT gebruikelijk een afzonderlijke *repository* aan te maken voor ieder afzonderlijk project. Er is namelijk ook geen centrale plaats waar alle bestanden van al onze projecten bewaard worden.

Een GIT-*repository* is gewoon een map in ons bestandssysteem waar alle versies van een project bewaard worden. Deze map heeft steeds de naam ***.git*** en is een submap van het project. In deze map bevinden zich allerlei bestanden en submappen die door GIT beheerd worden.

Deze *repository* kunnen we aanmaken door het volgende commando uit te voeren in de projectmap:

```
git init
```

De submap ***.git*** wordt dan gecreëerd en daarin worden allerlei administratieve bestanden geplaatst.

Opdracht 3: Een nieuw project maken en een *repository* toevoegen

In deze opdracht maken we een eenvoudig project en voegen we vervolgens een *repository* toe.

- Maak een nieuwe map voor een nieuw project. Bijvoorbeeld ***C:\GIT\HelloWorld***
- Voeg in deze map een nieuwe bestand ***HelloWorld.txt*** toe met de volgende tekst "Hello World".
- Open een commandovenster in de projectmap (of navigeer naar de projectmap) en voer het volgende commando uit:

  ```
  git init
  ```

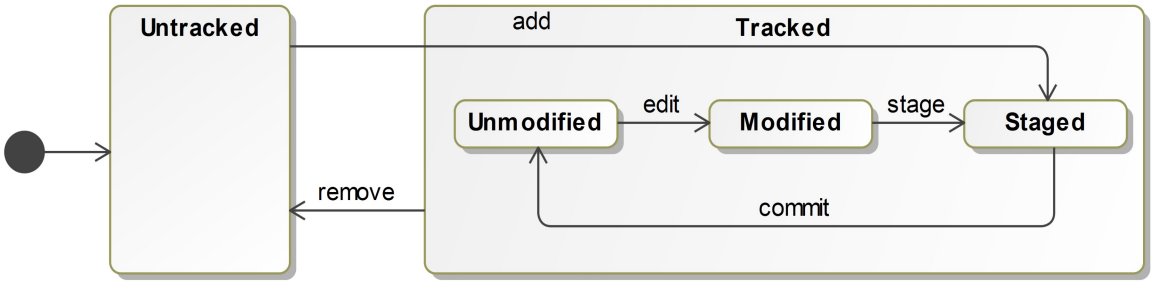
- Ga na of de map ***.git*** aangemaakt werd en wat de inhoud ervan is.

1.5.2 Toestanden van een bestand

Op dit moment hebben we een project met een bestand. We noemen dit de *working directory*. In deze map kunnen we dus werken met de bestanden van het project.

Daarnaast hebben we ook een GIT *repository.* Hierin zullen we de geschiedenis van het project bewaren. Op dit moment zitten er nog geen bestanden in de *repository*. We moeten het namelijk eerst hebben over de verschillende toestanden die een bestand kan hebben in GIT.

In het onderstaande schema worden de verschillende toestanden weergegeven van de bestanden in de *working* directory en tevens zie we de overgangen tussen deze toestanden:



Afbeelding 1: Toestanden van bestanden in de working directory.

Untracked: Bestanden in deze toestand worden door GIT niet gevolgd (*tracked*). Dit wil gewoon zeggen dat deze bestanden niet worden opgenomen in de *repository*. Alle nieuwe bestanden die we toevoegen in de *working directory* komen automatisch in deze toestand terecht en zullen we dus eerst expliciet moeten toevoegen vooraleer ze in rekening worden gebracht.

***Tracked**:* Bestanden die wel door GIT gevolgd worden, bevinden zich in deze toestand. Dit is evenwel een *superstate* die meerdere andere toestanden van het bestand bevat. Bestanden uit de *untracked* toestand kunnen worden toegevoegd en komen dan in de toestand *Staged* terecht. Hierover dadelijk meer. Bestanden kunnen deze supertoestand verlaten door ze te verwijderen. Ze worden dan niet langer meer door GIT gevolgd maar blijven wel aanwezig in de *working directory*.

De *superstate Tracked* heeft de volgende *substates:*

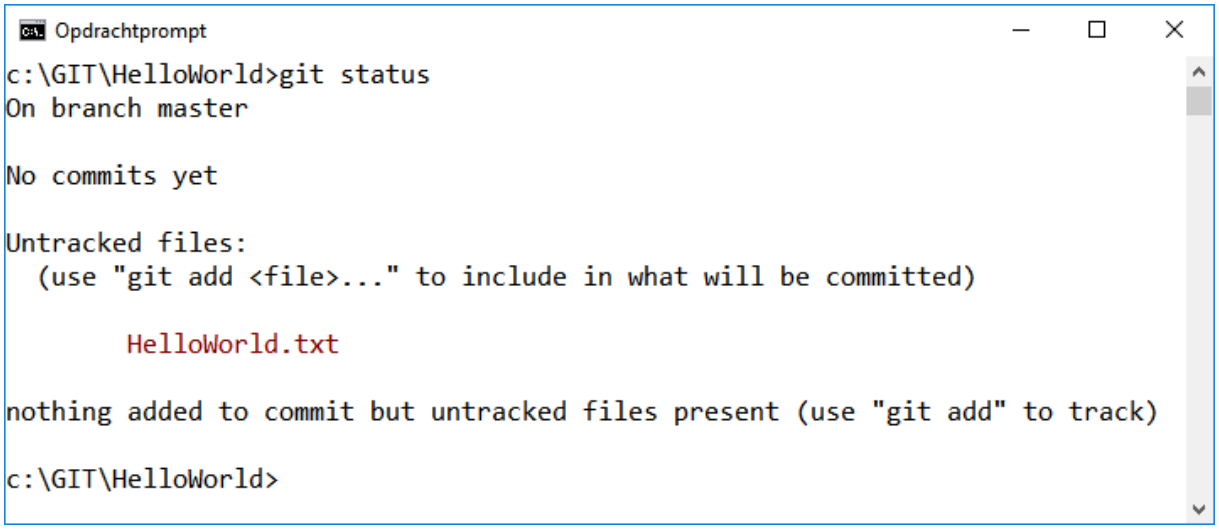
Unmodified: Bestanden in deze toestand worden door GIT gevolgd. Ze zijn evenwel niet gewijzigd ten opzichte van de toestand in de *repository*. Bestanden die uit de *repository* gehaald worden (*checkout)* komen spontaan in deze toestand terecht.

Modified: Zodra bestanden gewijzigd worden en dus verschillen van hun oorspronkelijke toestand in de *repository,* komen ze in deze toestand terecht.

***Staged**:* Vooraleer we gewijzigde bestanden terug in de *repository* kunnen plaatsen (*commit)* komen ze in de toestand *staged* terecht. Dit wil zeggen dat ze klaargezet worden voor een *commit* naar de *repository*. Deze bestanden worden uit de *working directory* gehaald en in een soort voorportaal van de *repository* geplaatst. Dit wordt soms ook de *index* genoemd. Het is dus mogelijk dat er in de *working directory* nadien opnieuw wijzigingen worden aangebracht die niet worden meegenomen bij de eerstvolgende *commit*. Het zijn namelijk de bestanden in de toestand *staged* die gecommit worden en niet de bestanden in de *working directory*. Het is evenwel mogelijk het *stagen* en *committen* in een beweging te doen.

De toestand van de bestanden in de *working directory* kunnen we opvragen met het volgende commando:

```
git status
```



1.5.3 Indexeren (*staging*) van bestanden

Om een nieuw bestand in de *repository* te plaatsen, moeten we het eerst in de toestand *staged* brengen. Dat wil dus zeggen dat het bestand bij de eerstvolgende *commit* mee in rekening zal worden genomen. Dit doen we met het volgende commando:

```
git add HelloWorld.txt
```