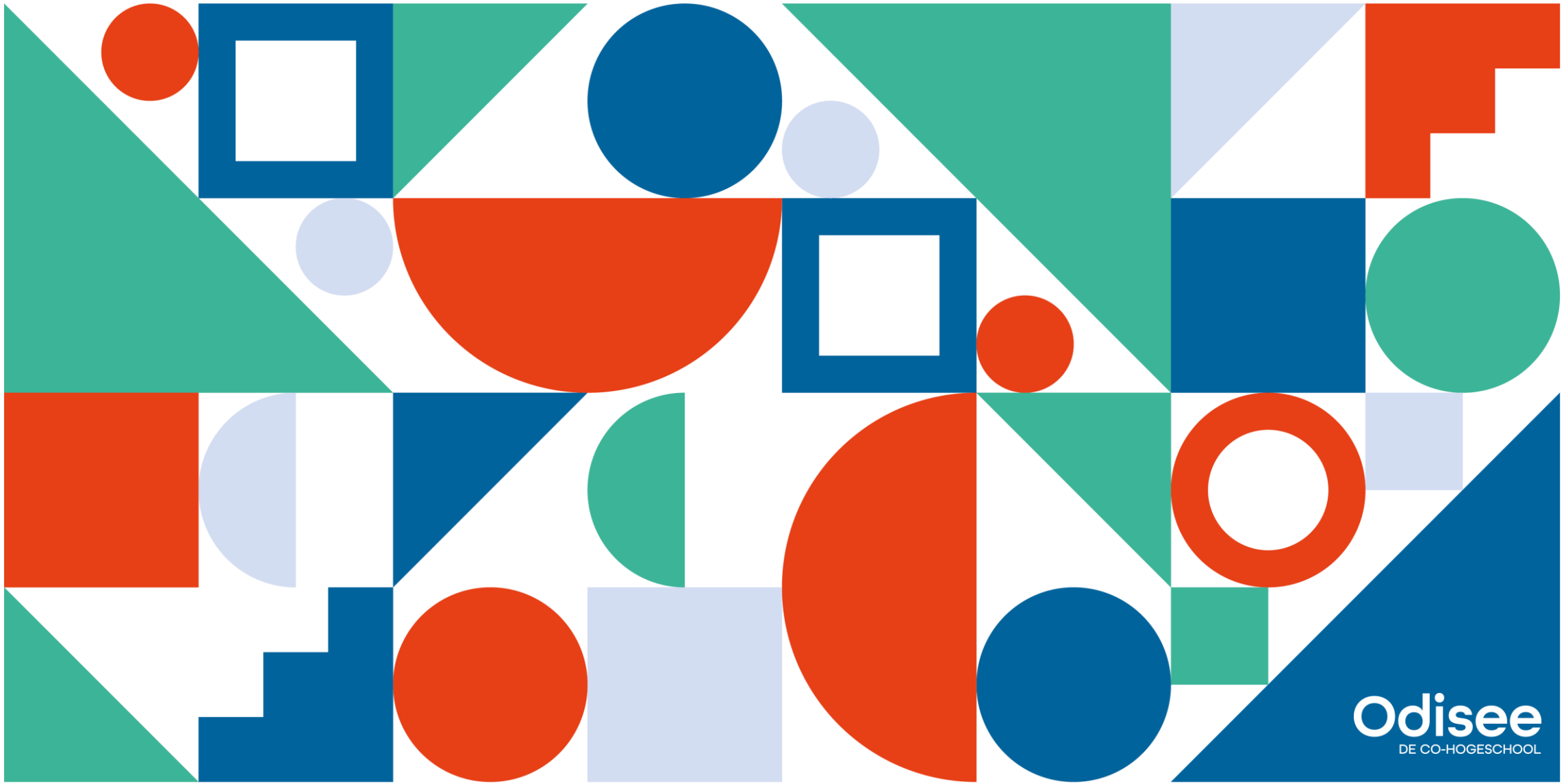

Odisee
DE CO-HOGESCHOOL

Software engineering fundamentals

Maarten Troost – Jens Baetens

Odisee
DE CO-HOGESCHOOL

Doelstellingen

Doelstellingen

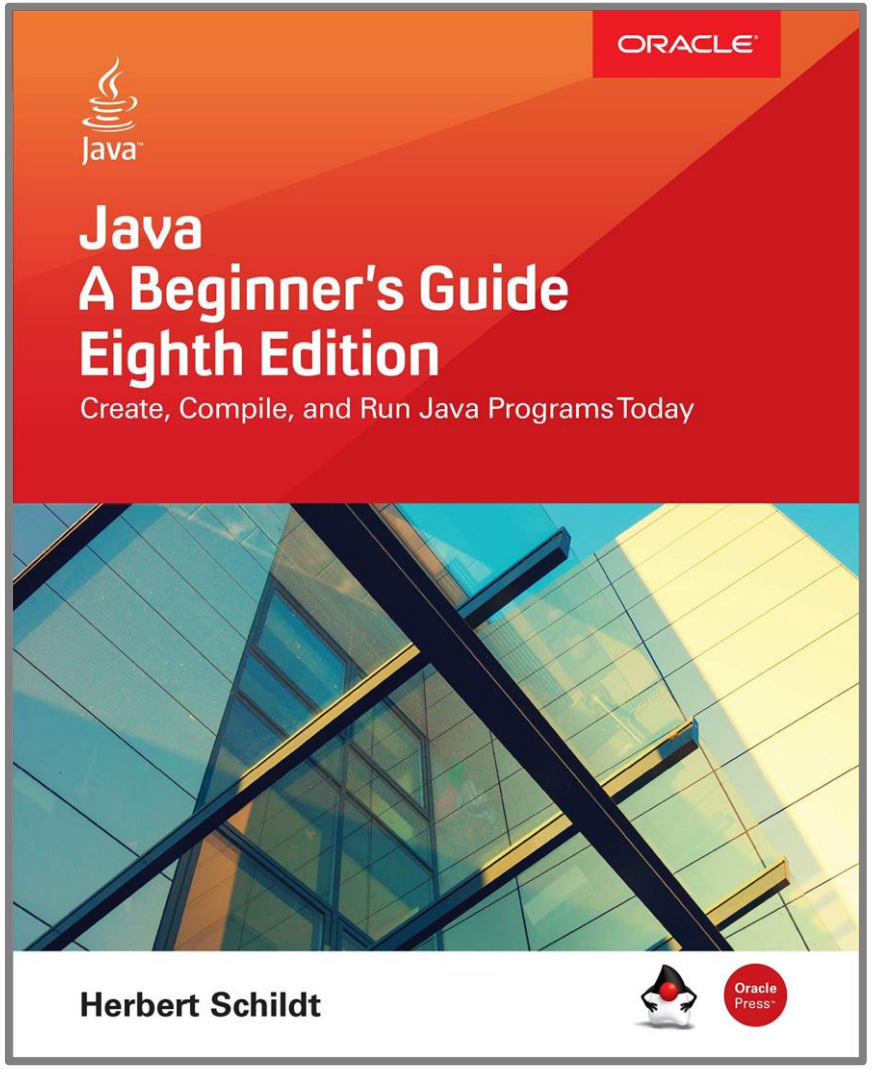
- Leer werken als een software engineer
  - Leer de Java-programmeertaal kennen en gebruiken in een Software Engineering context
  - Weet hoe u te werk gaat om vertrekkende van een probleemstelling op de juiste manier te komen tot efficiënte en productieve code
  - Leer object-georiënteerd werken
  - Leer hoe je software automatisch kunt testen
  - Leer hoe je code schrijft die lang kan gebruikt/herbruikt worden
  - Leer samenwerken aan code
  - Eerste voorbereiding op certificatie tot “Java Foundations Junior Associate”

Te ontwikkelen competenties – niveau 1

1.3	Complexe informatie en IT-behoeften kritisch analyseren.
1.4	IT-oplossingen bedenken en modelleren.
2.1	De aangewezen techniek hanteren om de gewenste gegevens efficiënt te verzamelen.
3.1	Inzicht verwerven in de behoeften van verschillende stakeholders.
7.1	Een oplossing voor een (complex) IT probleem kunnen ontwerpen
7.2	Een oplossing kunnen implementeren, optimaal gebruik makend van standaard- of modeloplossingen
7.3	Een integrale oplossing structureel kunnen testen en de resultaten kunnen gebruiken om het ontwerp en/of de oplossing continu te verbeteren
7.4	Een oplossing kunnen documenteren zodat anderen de oplossing kunnen begrijpen, aanpassen of instaan voor het onderhoud
8.2	Deadlines en afspraken kunnen respecteren
8.3	Kritisch kunnen reflecteren over het functioneren van het team, de leden en zichzelf
8.4	Verantwoordelijkheden of leiderschap opnemen in een team en waken over het welbevinden van de teamleden

Interessant / Aanbevolen boeken

http://www.vanduurenmedia.nl/EAN/9789059403048/Handboek_Objectgeorienteerd_programmeren_2e_ed_



https://www.amazon.co.uk/Java-Beginners-Eighth-Herbert-Schildt/dp/1260440214/

Odisee
DE CO-HOGESCHOOL

ECTS – fiche evaluatie

- Permanente evaluatie
  - 20% Proces
  - 40% Werkproduct
  - 20% Documentatie
  - 20 % Presentatie
- Optioneel openboek mondeling examen met open vragen
  - Opt-in = je kan zelf kiezen of je deelneemt
  - Indien je deelneemt dan telt dit examen mee voor 40% (permanente evaluatie in totaal dan 60%)

Huistaken

- Opdrachten gemaakt via github classroom
- Individueel
- Let op goed gebruik van github
  - Maak meerdere commits
- Documentatie
- Codestructuur en naamgeving zijn belangrijk
- Let op de deadlines

Project

- Opgave
  - Kies een aantrekkelijke toepassing voor een bedrijf
  - Maak een oplossing voor het gekozen probleem op te lossen
  - Pas zoveel mogelijk van de geziene technieken uit de lessen van SEF toe
- Link naar de opdracht (zie ook Toledo)
  - TI 1-2: https://classroom.github.com/a/PT9ZWs_M
  - TI 3-4: https://classroom.github.com/a/3y4mH2o-
  - TI 5-6: https://classroom.github.com/a/h5fd53Gs
  - TI 7-8: https://classroom.github.com/a/Pi6UAZ5i

Deadlines

- 01/05 23u59: Maak groepen, lees de artikels op Toledo uit je klasgroep en kies je top 3 topics of stel er zelf 1 voor
- 07/05 23u59: Stel een aantrekkelijke toepassing voor geef 10 high-level requirements en maak wireframes -> voorbeeld op Toledo
- 26/05 23u59: Eerste oplossing indienen
- 09/06 23u59: Finale versie + presentatie indienen + peer reviews indienen
- Presenteren in de week van 12/6 tot 16/6

Presentatie project

- Presentatie van 10 minuten
  - Demonstreer het resultaat van uw teamwork
  - Toon de code en leg ze uit
  - Zorg dat elk teamlid aan het woord komt
  - Iedereen moet alle code kennen
- 5 minuten voor vragen

Peer evaluatie

- Individueel en vertrouwelijk
- In te dienen via Toledo
- Inhoud
  - Evaluatie door [uw naam]
  - Volgens mij heeft/is sinds vorige les [naam geëvalueerde]
    - [score] de afgesproken taken uitgevoerd
    - [score] bijgedragen tot ideeën en planning voor het werk
    - [score] bereikbaar/aanspreekbaar geweest voor overleg
    - [score] bijgedragen tot het succes van het project
  - Op te nemen in het verslag als feedback voor deze persoon [uw feedback in maximum een aantal regels]

1 = niet/nauwelijks
2 = af en toe / soms
3 = meestal / altijd

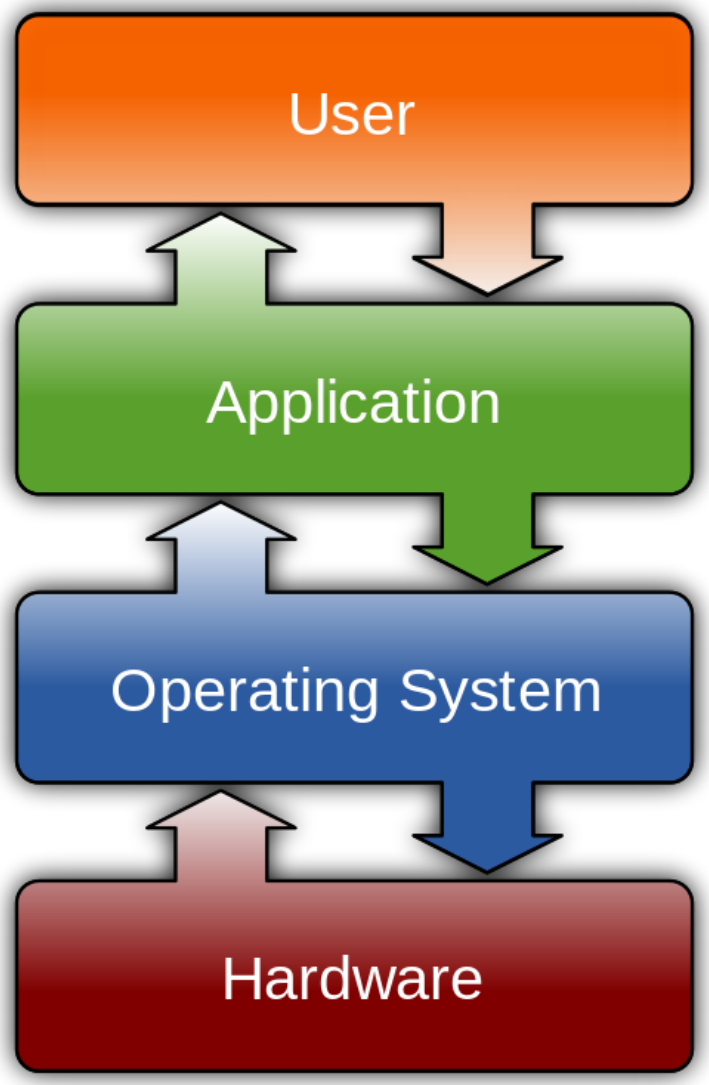
Vakinhoud

Overzicht van de SEF werkwijze

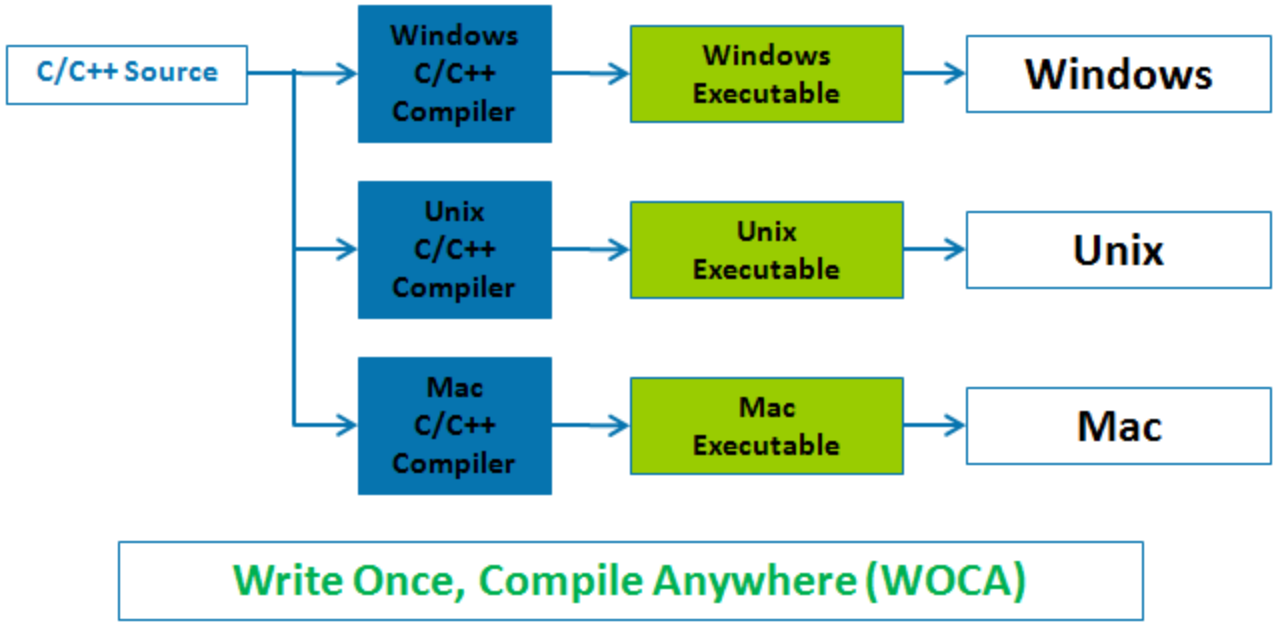
- Doorgronden van de probleemstelling
- Opstellen van lijst met te leveren publieke diensten
- Herhaal:
  - JUnit test maken (op basis van de lijst)
  - Schrijf code (eerst het skelet, daarna de details)
  - Refactor
  - Javadoc-documentatie schrijven
- En daarna … de GUI dus

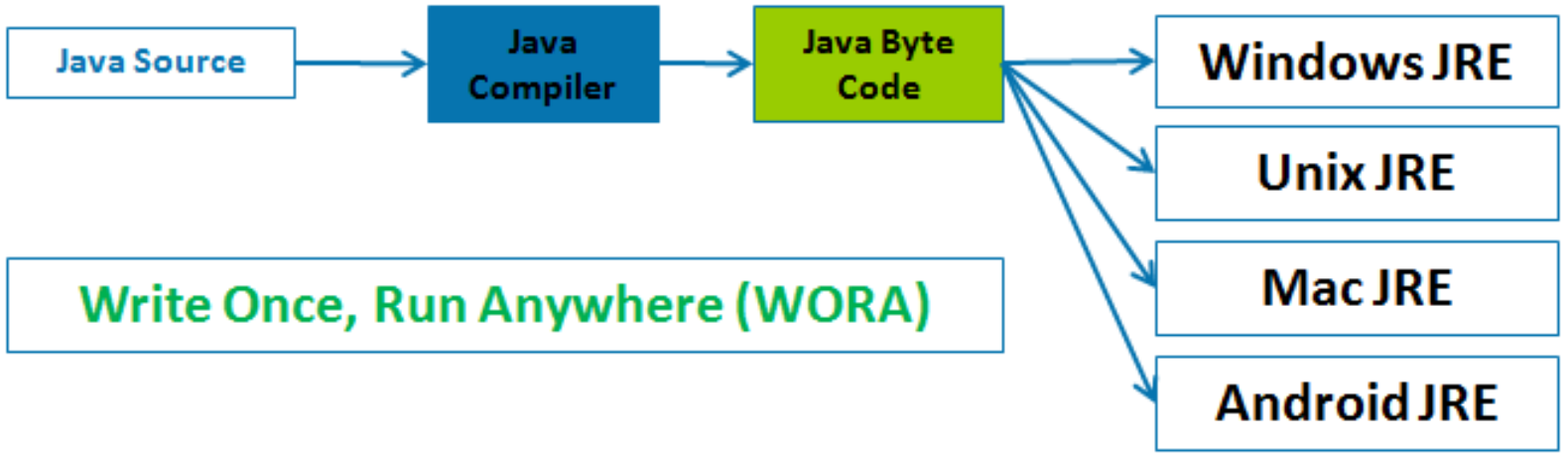
JAVA

- Virtueel platform
- Hogere programmeertaal (gelijkaardig aan C#)
- Gecompileerd naar virtueel platform
  - Dit platform wordt aangeboden door OS
- Platformonafhankelijk



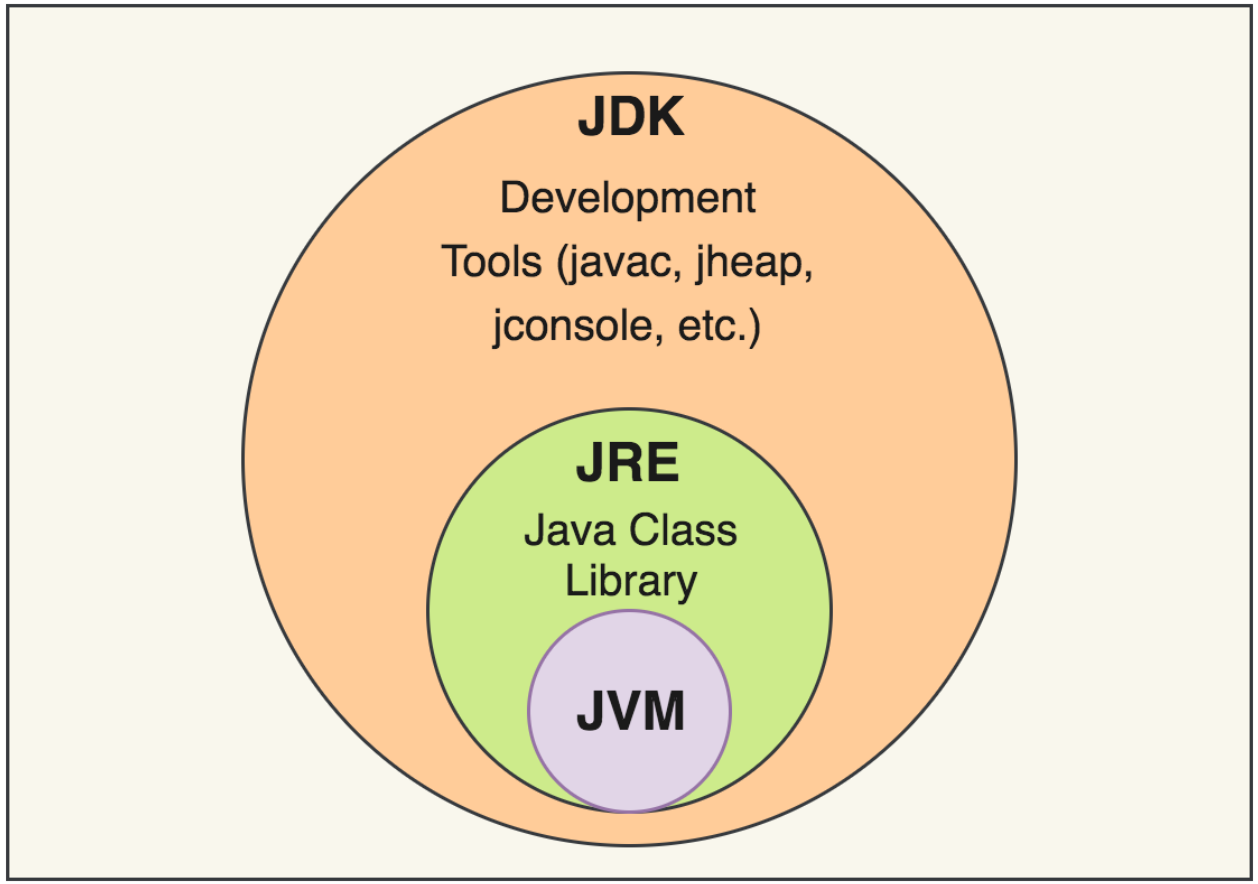
Java is platformonafhankelijk door





JRE vs JVM vs JDK

- JRE
  - JVM
  - standaard software componenten
    - OS-afhankelijk (dialogs, …)
- JDK
  - Plus compiler, javadoc, jar-tool, …
- Types
  - SE -> standard edition
  - ME -> mobile edition (oude gsm's)
  - EE -> enterprise edition (servers)



Geschiedenis?

- Gemaakt door Sun
- Sun is in 2009 overgenomen door Oracle
- Nu elk jaar minstens 1 update
- Java gebruikt in grote bedrijven
  - Banken, Verzekeringen, …
  - Backwards-compatibility is belangrijk!
- C# eerder in kleinere bedrijven

Belangrijke terminologie

- U leert *user software* te maken.
- Een Java-compiler produceert bytecode. Dit is platformonafhankelijke code die door een interpreter wordt uitgevoerd.
- Java-programma's worden uitgevoerd op een Java Virtual Machine (JVM). Dat is een operationele computer waaraan een Java Runtime Environment (JRE) is toegevoegd.
- Om zelf Java-programma's te maken, hebt u een Java Software Development Kit (SDK) nodig.

Nieuwe termen en begrippen

User software	Bytecode
Operating System	Java Runtime Environment (JRE)
Virtuele machine	Java Virtual Machine (JVM)
Platform	Software Development Kit (SDK)
Platform(on)afhankelijkheid	

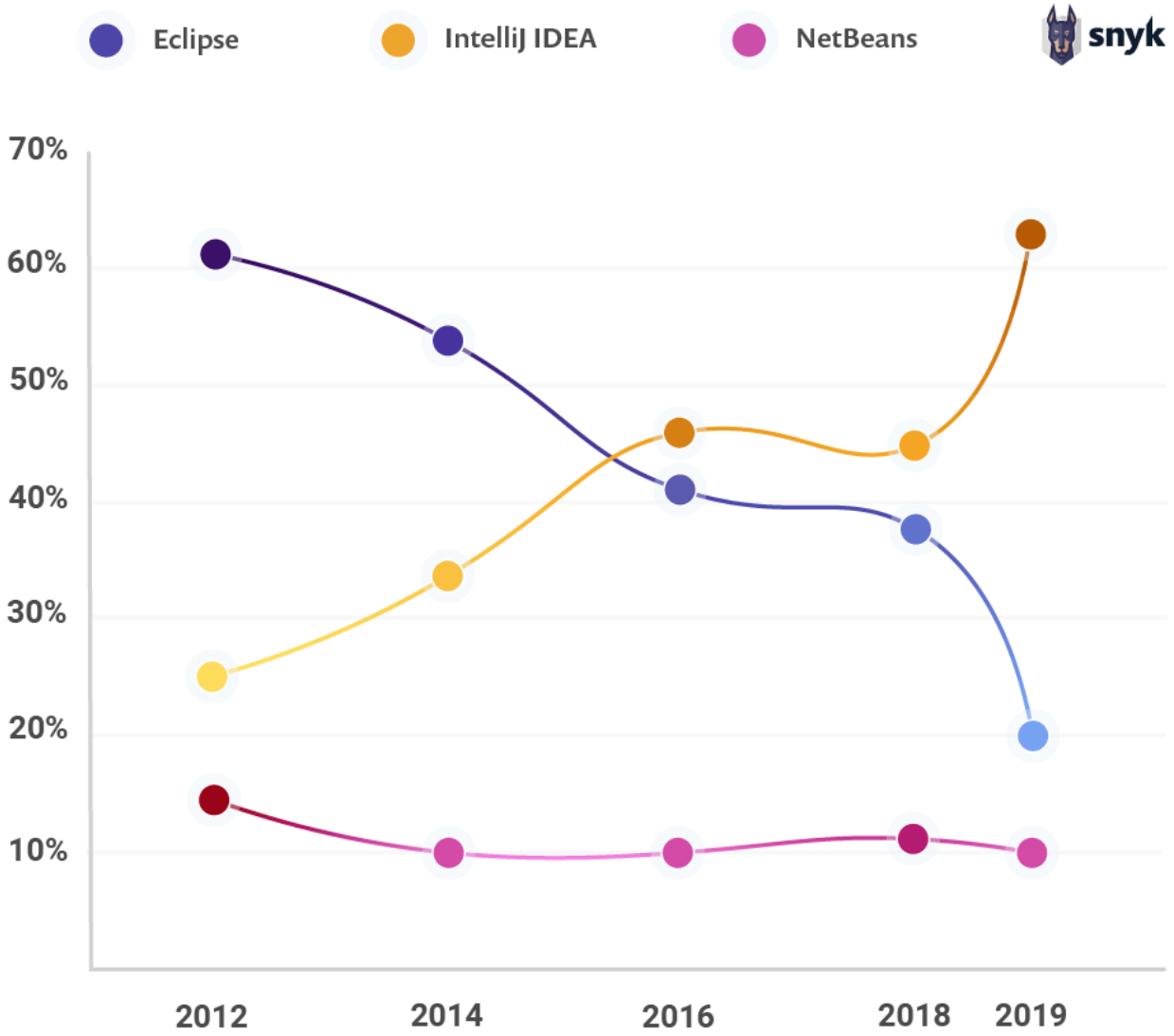
IDE
Integrated Developer Environment

Hoe Java programmeren?

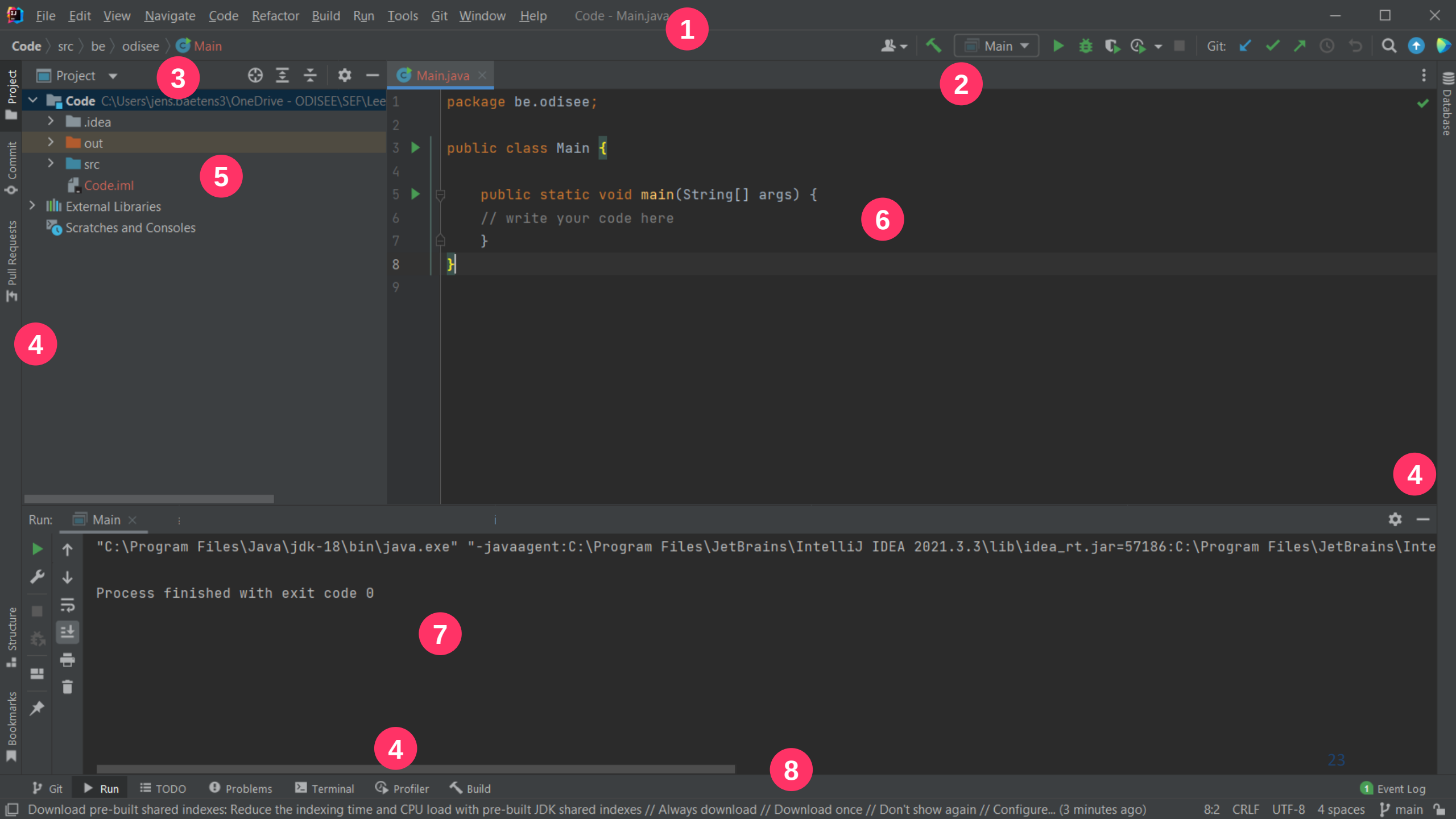
- Java programma's bestaan uit .java bestanden
- Deze kunnen geopend en bewerkt worden door een standaard tekst editor
  - Of Notepad++
  - Of Atom
  - Of Visual Studio Code
- Functionaliteiten beperkt
  - Maak gebruik van een IDE voor programmeren (zoals Visual Studio)

Welke IDE gebruiken voor Java?

- Er zijn drie veelgebruikte
  - Eclipse
  - Netbeans
  - IntelliJ



https://snyk.io/blog/intellij-idea-dominates-the-ide-market-with-62-adoption-among-jvm-developers/


File Edit View Navigate Code Refactor Build Run Tools Git Window Help
Code - Main.java
1
Code src be odisee Main
Main
Git:
2
Project
3
Main.java
Code C:\Users\jens.baetens3\OneDrive - ODISEE\SEF\Lee
.idea
out
src
Code.iml
External Libraries
Scratches and Consoles
5
package be.odisee;
public class Main {
public static void main(String[] args) {
// write your code here
}
}
6
4
Run: Main
"C:\Program Files\Java\jdk-18\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2021.3.3\lib\idea_rt.jar=57186:C:\Program Files\JetBrains\Inte
Process finished with exit code 0
7
4
8
Git Run TODO Problems Terminal Profiler Build
Event Log
Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built JDK shared indexes // Always download // Download once // Don't show again // Configure... (3 minutes ago)
8:2 CRLF UTF-8 4 spaces main
Project Commit Pull Requests Structure Bookmarks Database

1. Hoofdmenu -> projecten beheren, refactoring, intellij aanpassen
2. Hoofdtoolbar -> Knoppen om taken uit te voeren binnen het project
3. Navigatiebar -> Welk betand actief is + pad in het project
4. Toolbladbladen -> Links, onderaan, rechts met tools binnen het project.
   Kunnen ook geopend worden via view
5. Toolbladvenster -> Vensters die opengaan bij klikken op een tool
6. Editor -> Venster met de code van de actieve file
7. Run-scherm -> Belangrijk scherm voor console applicatie voor input/ouput
8. Statusbar -> Huidige toestand van de IDE

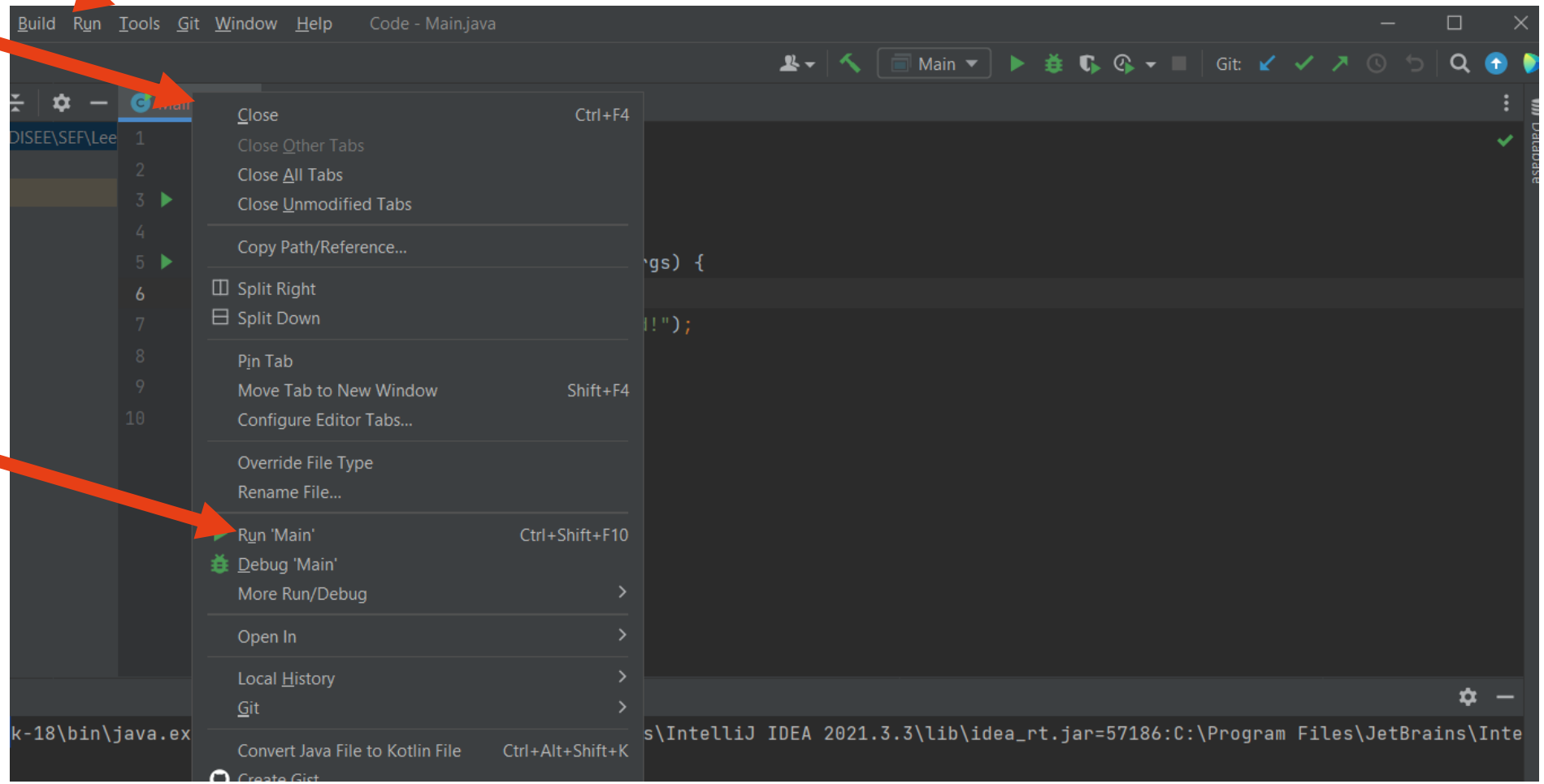
Hello world applicatie

- Voeg een lijn toe dat output print naar console
  - Tip: type “sout” en kijk wat de intellisense aanbeveelt
  - Andere handige afkorting psvm
    - Dit maakt de main functie aan

```
1  package be.odisee;
2
3  public class Main {
4
5      public static void main(String[] args) {
6      // write your code here
7      }
8  }
9
```

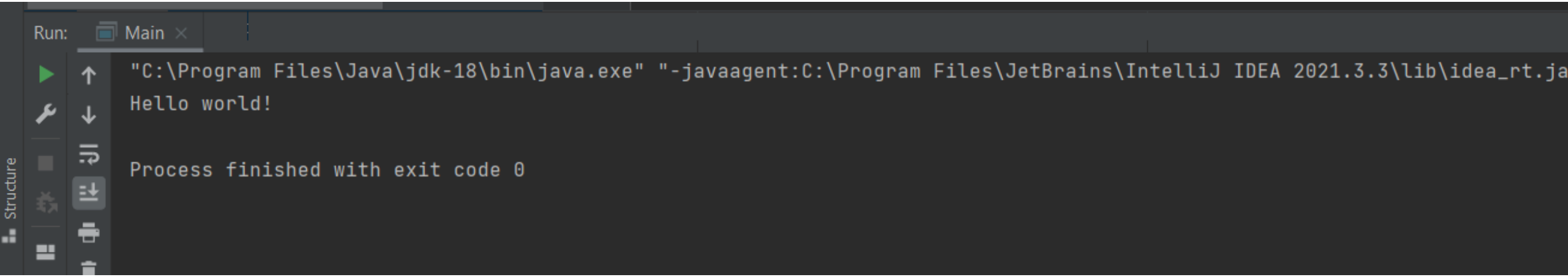
Uitvoeren van de applicatie

OF



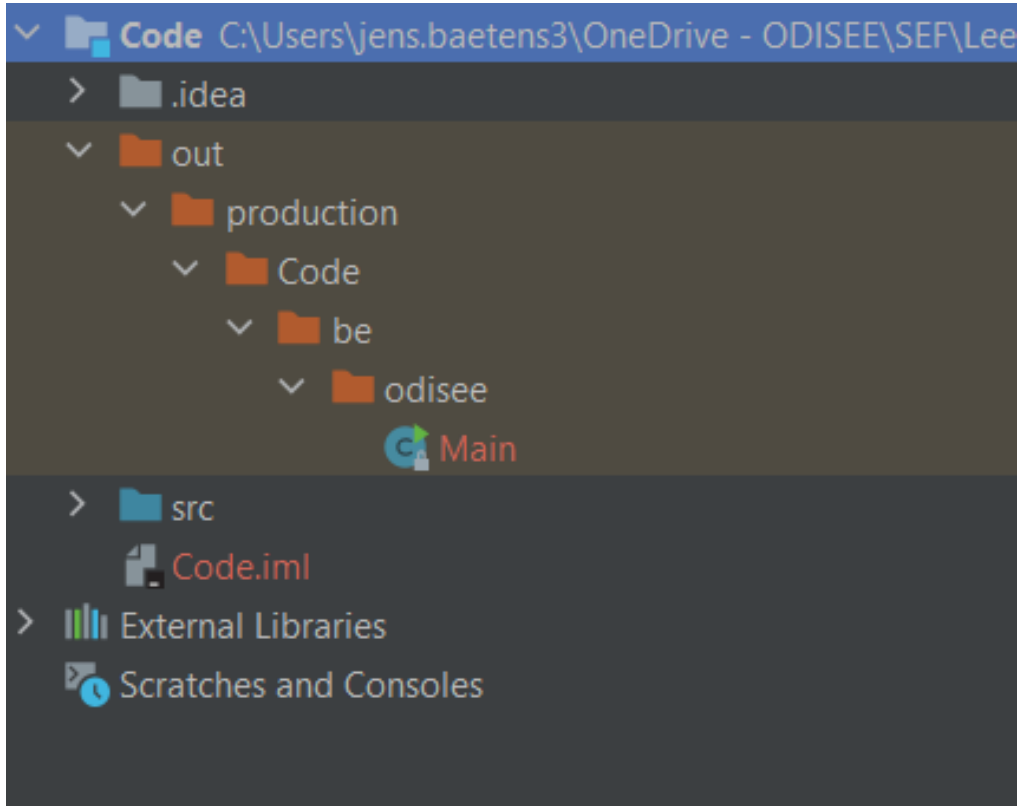
Uitvoeren van de applicatie

- In de achtergrond doet IntelliJ de commando's
  - javac
  - java
- Met als resultaat in het run-view



Plaats bytecode

- Al de .class files aangemaakt door javac worden in een aparte folder geplaatst
  - Standaard in de “out” folder



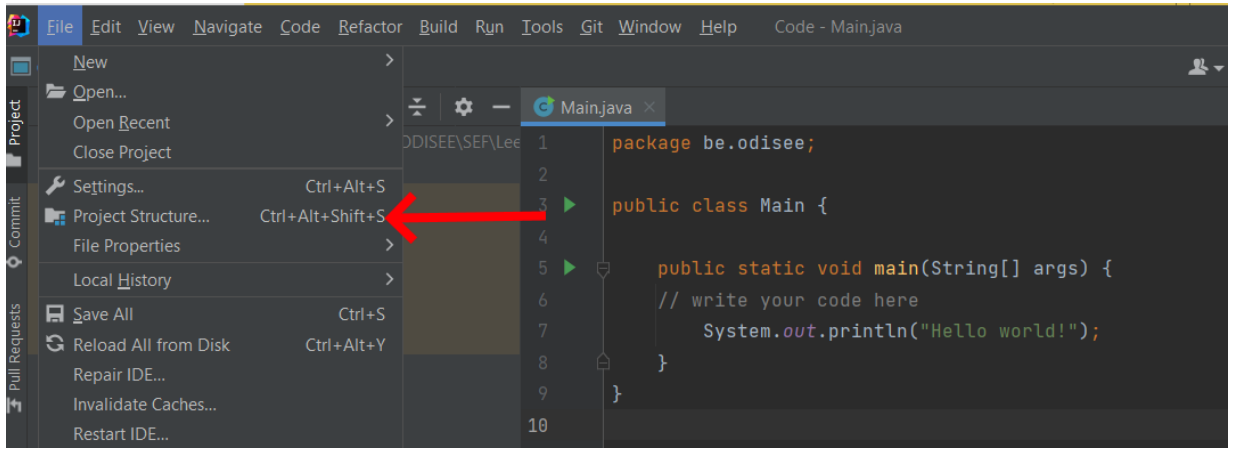
Wat gebeurt er bij het uitvoeren van de applicatie via IntelliJ

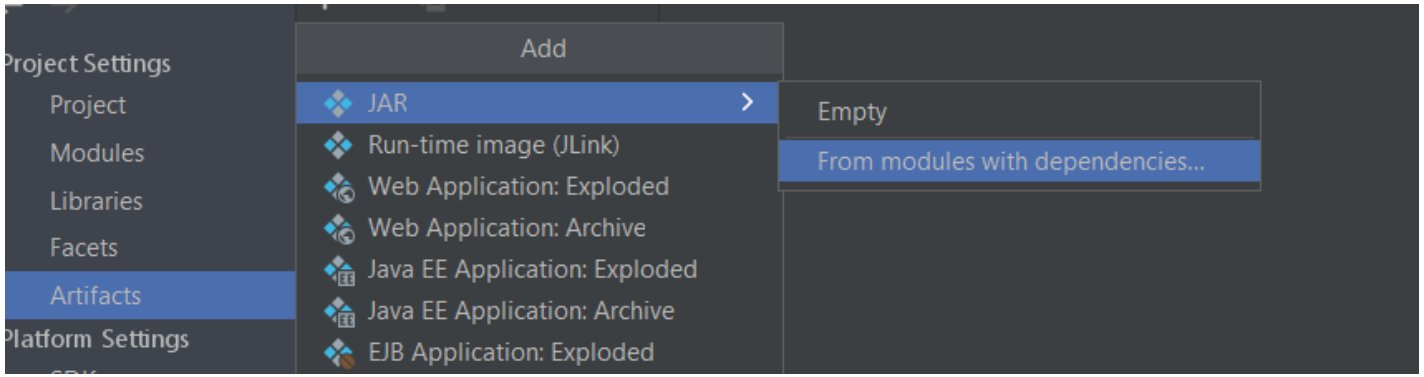
- Alle files worden bewaard
- Alle .java files die veranderd zijn worden gecompileerd
- De .class file dat de main functie bevat wordt uitgevoerd

Hoe verspreiden van de applicatie

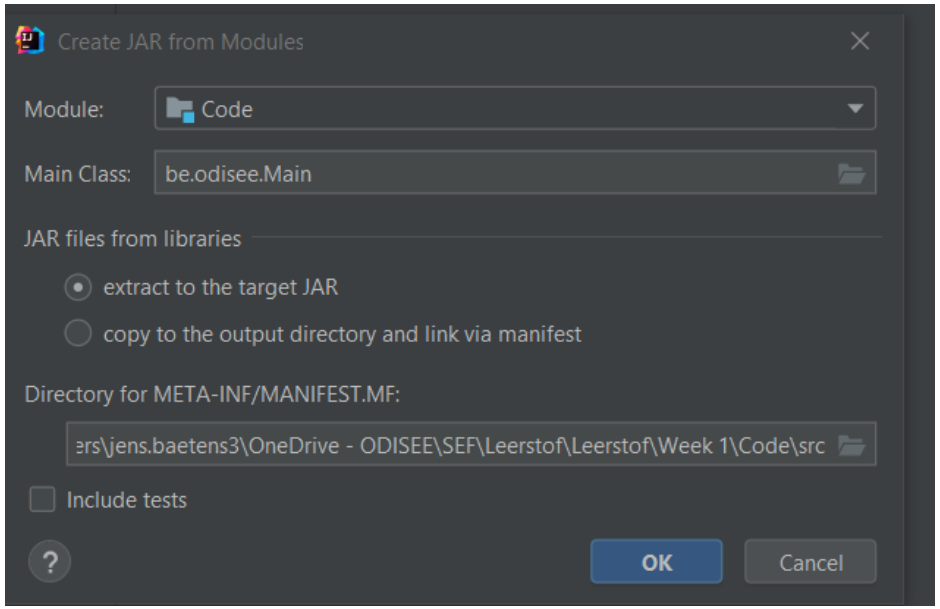
- Aan de hand van een Jar file
  - Java Archive
  - Een package/zip met alle .class files
- In IntelliJ zijn de volgende stappen nodig
  - Maak een artifact aan
  - Build de artifact
  - https://www.jetbrains.com/help/idea/compiling-applications.html#package_into_jar

Maken van een artifact

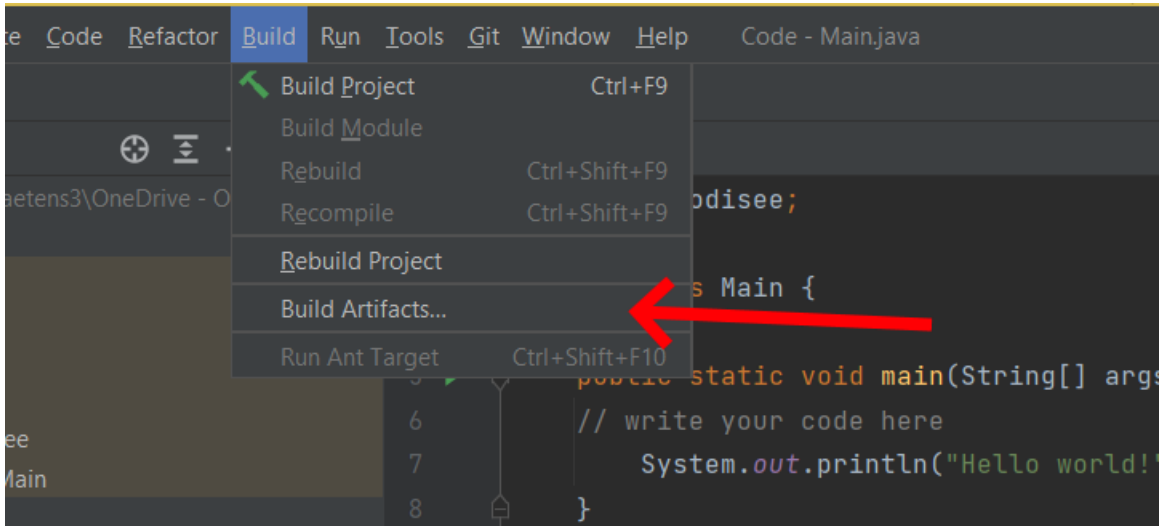


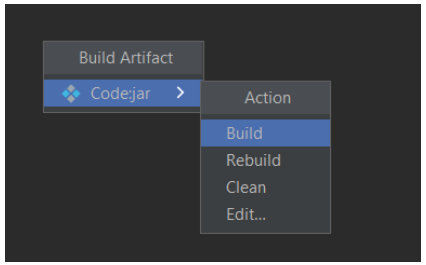


Maken van een artifact



Builden van een artifact





Uitvoeren van de jar

- Deze jar file kan verplaatst worden naar elke computer beschikkende over een JRE
- Kan uitgevoerd worden door middel van het java –jar commando

```
C:\Users\jens.baetens3\OneDrive - ODISEE\SEF\Leerstof\Leerstof\Week 1\Code\out\artifacts\Code_jar>java -jar Code.jar
Hello world!
```

Aantal handige tips

- Dubbelklik op de bestandsnaam om het editor-scherm zo groot mogelijk te maken
- Rechtsklik op de bestandsnaam om meerdere bestanden naast elkaar te zetten
  - Verticaal: split-down
  - Horizontaal: split-right

Uitgebreider voorbeeld

- We maken een programma welke:
  - een getal vraagt als input via commandline (stdin)
  - het kwadraat en vierkantswortel berekent
  - Het resultaat van de berekeningen op standard output print

- Welk klasses hebben we hiervoor nodig? Welke verantwoordelijkheden heeft elke klasse?

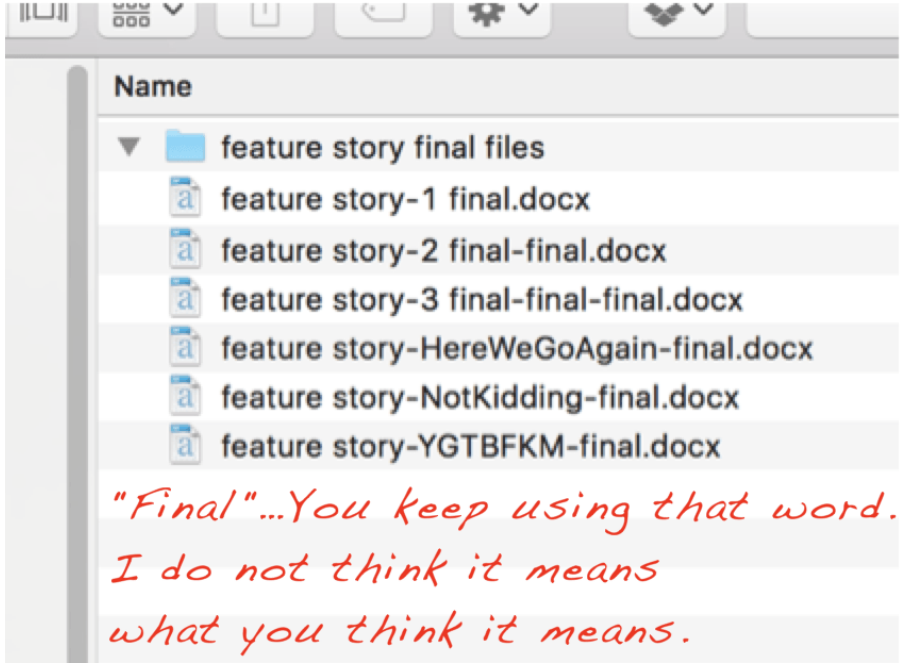
Uitgebreider voorbeeld

- Maak twee klassen aan:
  - SquareMath (verantwoordelijkheid: wiskundige berekeningen)
    - Bevat twee functies
      - square -> berekent en returned het kwadraat
      - squareRoot -> berekent en returned de vierkantswortel
  - Calculator (verantwoordelijkheid: interactie user)
    - Bevat de main functie
    - Vraagt een getal als input via de standard input van de command line
    - Berekent het kwadraat en vierkantswortel met behulp van de SquareMath klasse
    - Print alle berekende gegevens uit naar standard output van de command line

Git

Version control system

- Een VCS is een systeem dat er voor zorgt dat er op een manier een versiebeheer is van je files en dit ook volledig beheert.
  - Voordeel: zelf geen verschillende versies bijhouden
- Verschillende systemen:
  - GIT (distributed) - Opgelet: dit is niet github
  - SVN (centralized)
  - SCCS (local)



Odisee
DE CO-HOGESCHOOL

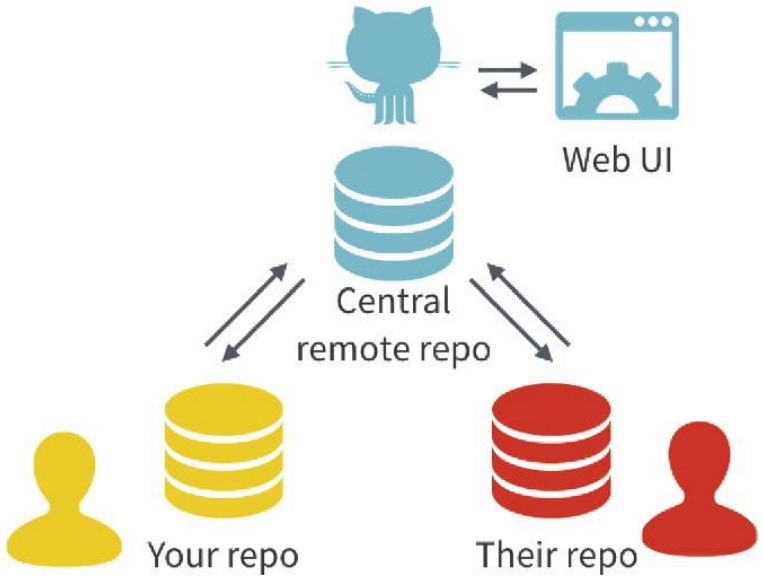
Wat is Git?

- GIT is een distributed version control system
- Een systeem om aan versie controle te doen door middel van een gedistribueerde structuur
  - In principe is er geen server nodig
  - Dit wordt meestal wel gedaan om een single source of truth te hebben
    - Deze server moet niet persé online zijn om te kunnen werken. (bij een gecentraliseerd systeem is dit wel nodig)
  - Dit kan op een eigen server of op een van de onderstaande platformen:
    - Github
    - Bitbucket
    - Azure repos
    - GitLab

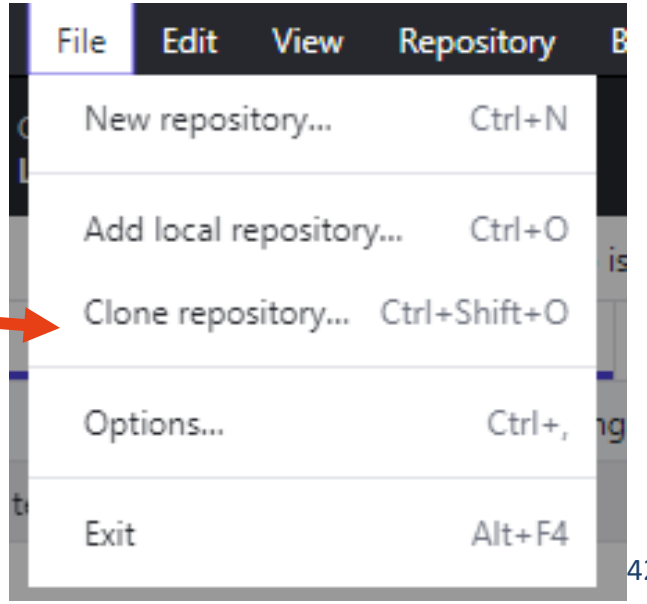
Git Basics

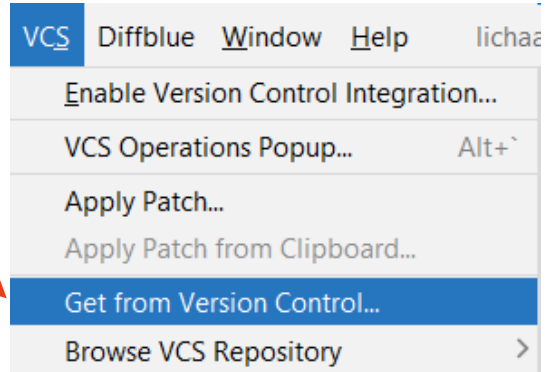
- Hoe je met git werkt, is je eigen keuze.
- Zoals je wellicht reeds wist zijn er verschillende tools om met git te werken
  - Commandline (https://git-scm.com/)
  - Github desktop (https://desktop.github.com/)
  - Fork (https://git-fork.com/)
  - Visual studio git (via visual studio installer)
  - …
- In **geen** geval is het nog de bedoeling dat je manueel upload naar github.

Git init



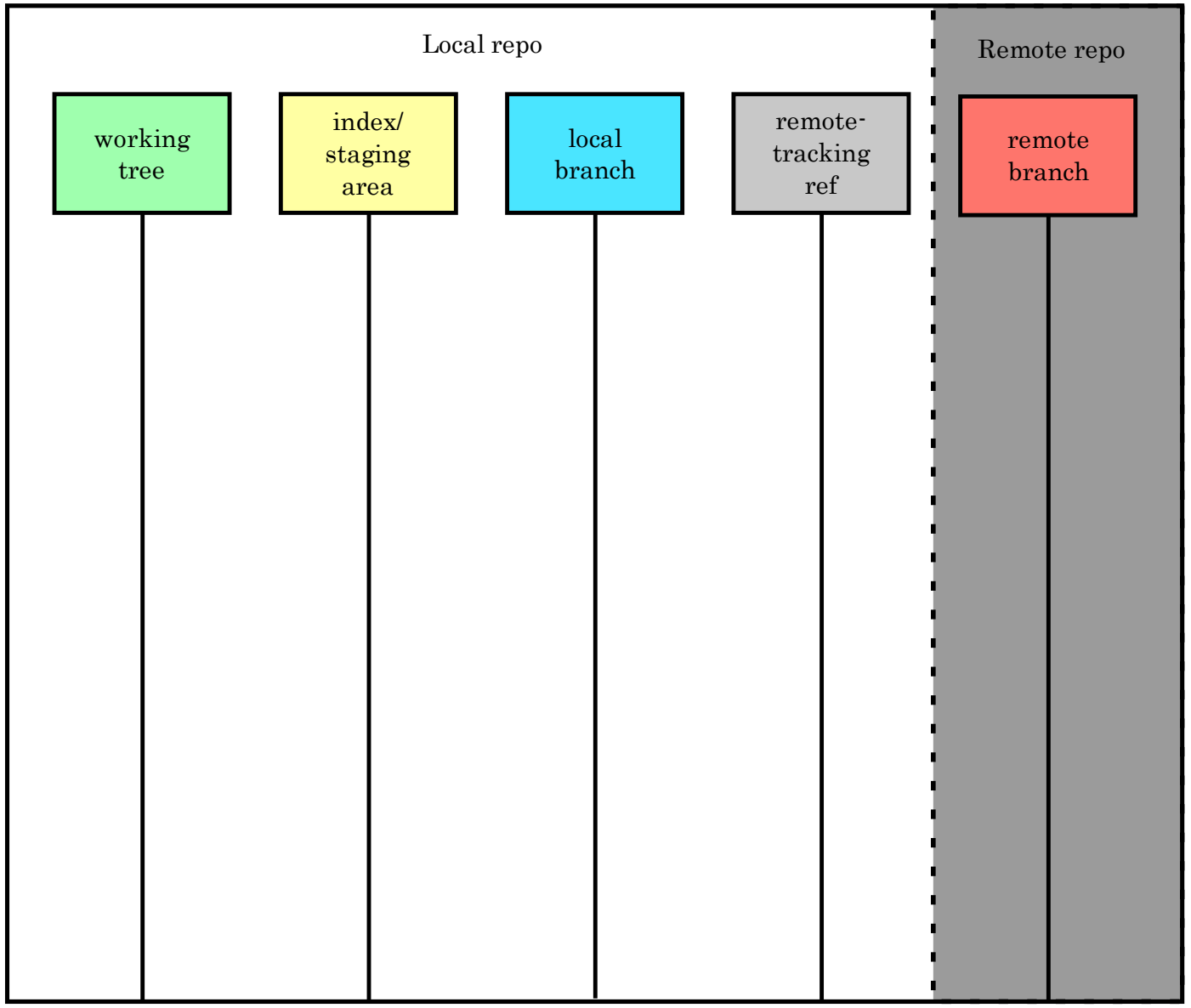
- Om te kunnen starten met git hebben we een eigen git repo nodig (op onze lokale machine). We gebruiken github om een repo te maken en maken hiervan een lokale kopij. Ieder werkt op een eigen kopij.
  - We maken een clone van een bestaande repo
    - Github Desktop
    - IntelliJ





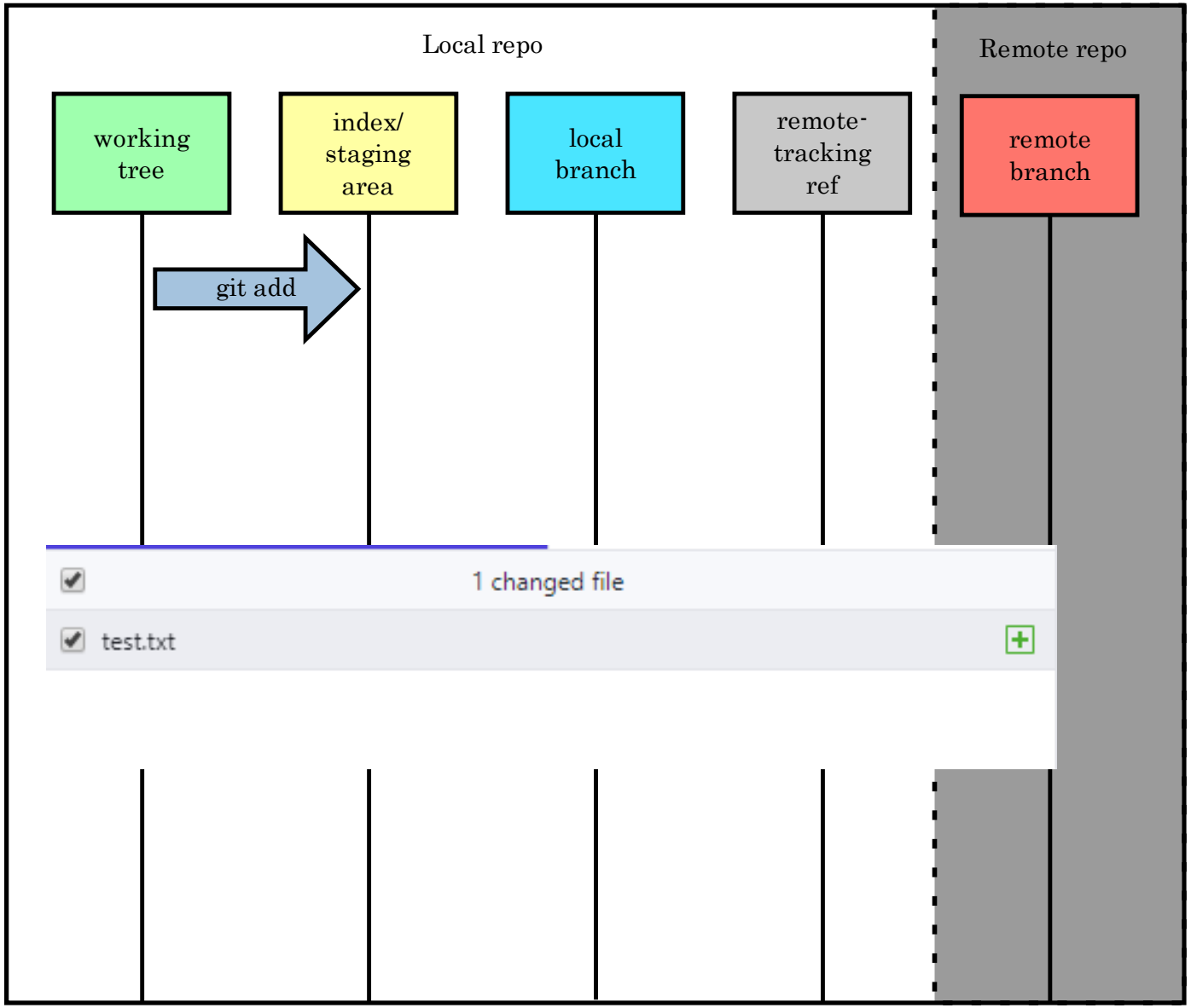
Git Basics

- Steeds afhankelijk van uit welk perspectief we kijken.
- Elk systeem dat git gebruikt heeft alle local repo stages. En kent van een remote repo enkel remote branches.
- Dit wil zeggen dat vanuit het standpunt van github je lokale computer enkel een remote branch heeft.



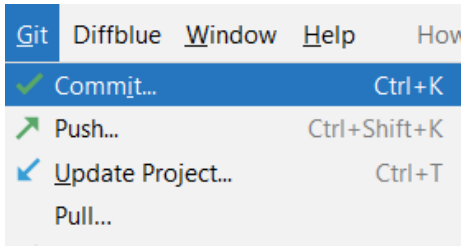
Git add

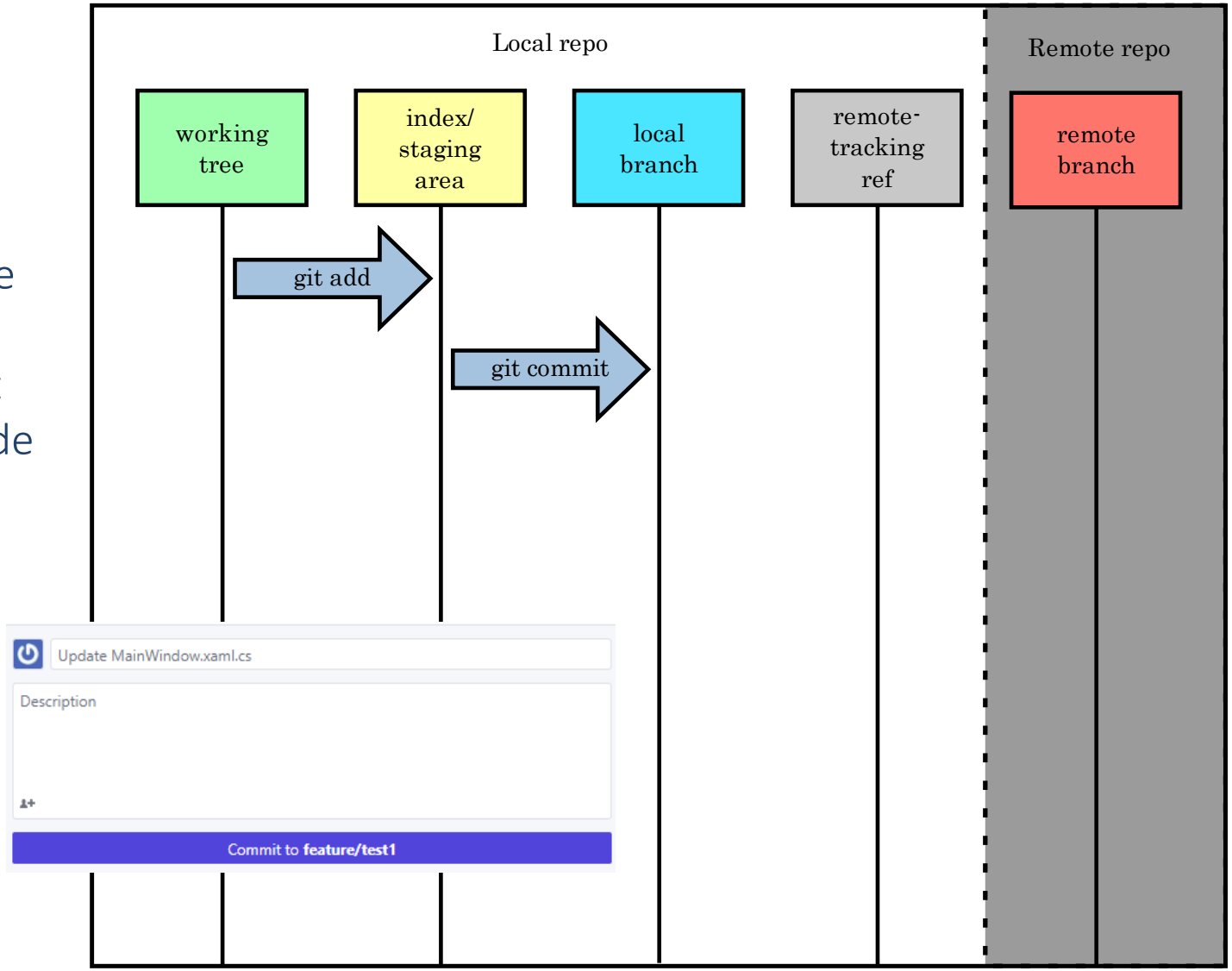
- Working tree: Wat je ziet in je folder/directory van je systeem als je aan het werken bent.
- Index/staging area: Locatie waar we aangeven welke files later naar de local branch mogen
- Met behulp van git add plaatsen we files van onze working directory naar de staging area.
  - Commando: git add <filename>
  - Github desktop: aan de hand van het vinkje
  - IntelliJ: gebeurt automatisch/ IDE bevraagt



Git commit

- Met behulp van git commit bewaren we de wijzigingen “definitief” als een tussentijdse versie in de local branch. Dit wordt enkel gedaan voor de files die op de staging area staan.
- Local branch: Een soort tak met een of meer tussentijdse versies.
  - Commando: git commit –m <message>
  - Github desktop:
  - IntelliJ:





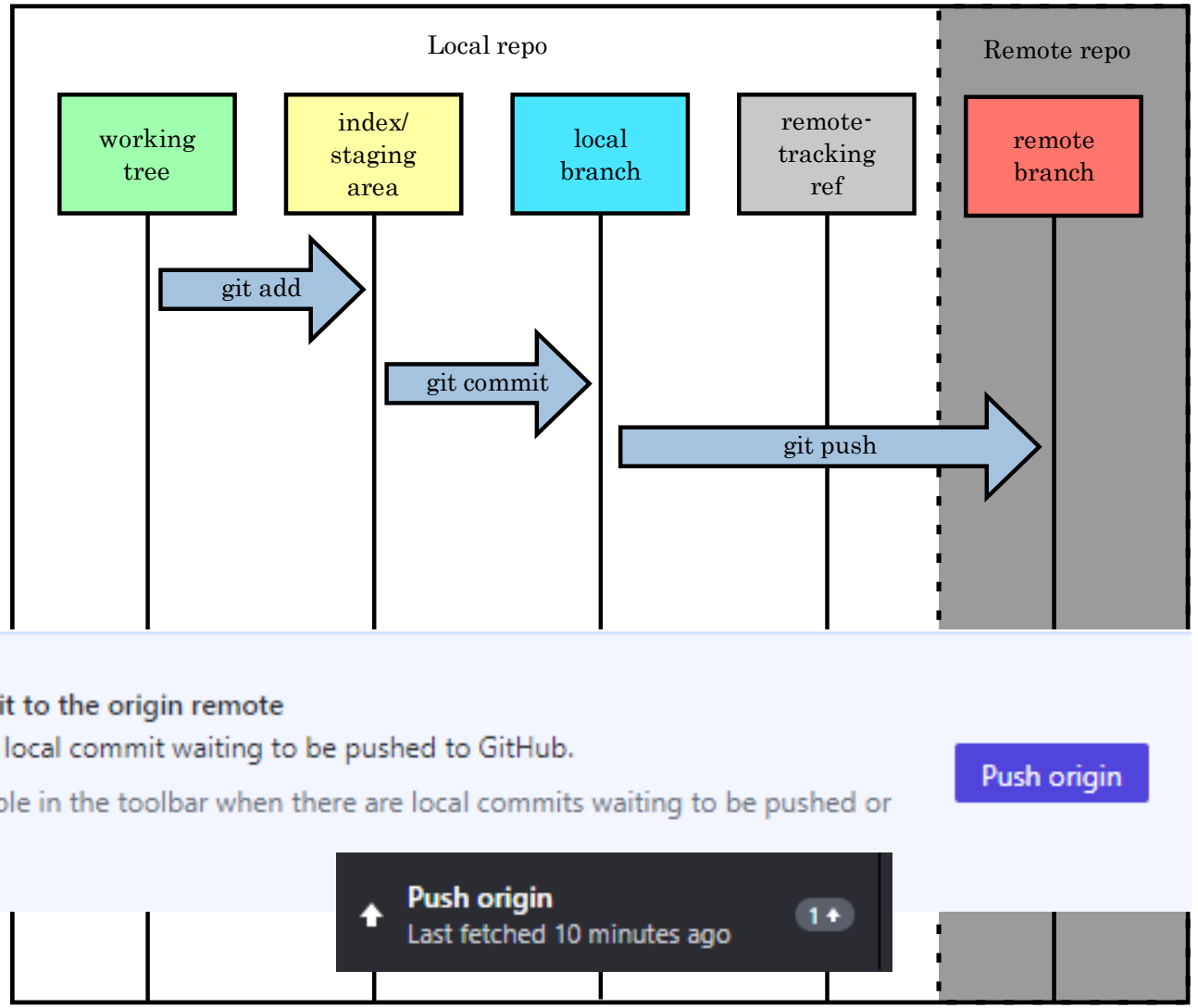
Odisee
DE CO-HOGESCHOOL

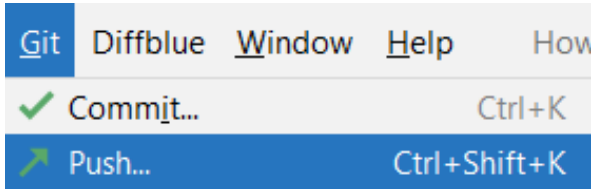
Voorbeeld local branch

- Hieronder zie je een voorbeeld van een local branch met verschillende commits. Voor elk van deze commits kan je zien wat er veranderd is en eventueel zelfs een eerdere versie terug ophalen.
  Elke versie/commit blijft bewaard in je lokaal repo zelfs al verwijder je de files waaraan je werkt van je filesystem.

Git push

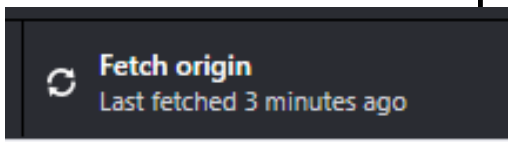
- Remote branch: Net hetzelfde als een local branch. Ten opzichte van het systeem staat deze op een andere locatie.
- Met behulp van git push kunnen we onze wijzigingen op de local branch synchroniseren met de remote branch. Elke commit die we uitgevoerd hebben op de local branch zal uitgevoerd worden op een remote branch.
  - Commando: git push
  - Github desktop:
  - IntelliJ:

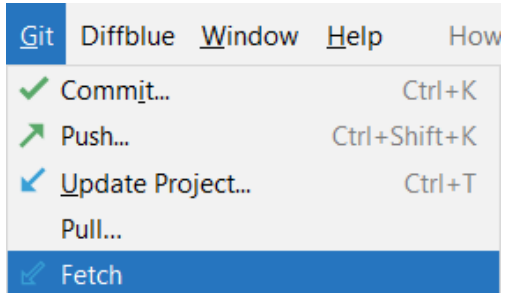


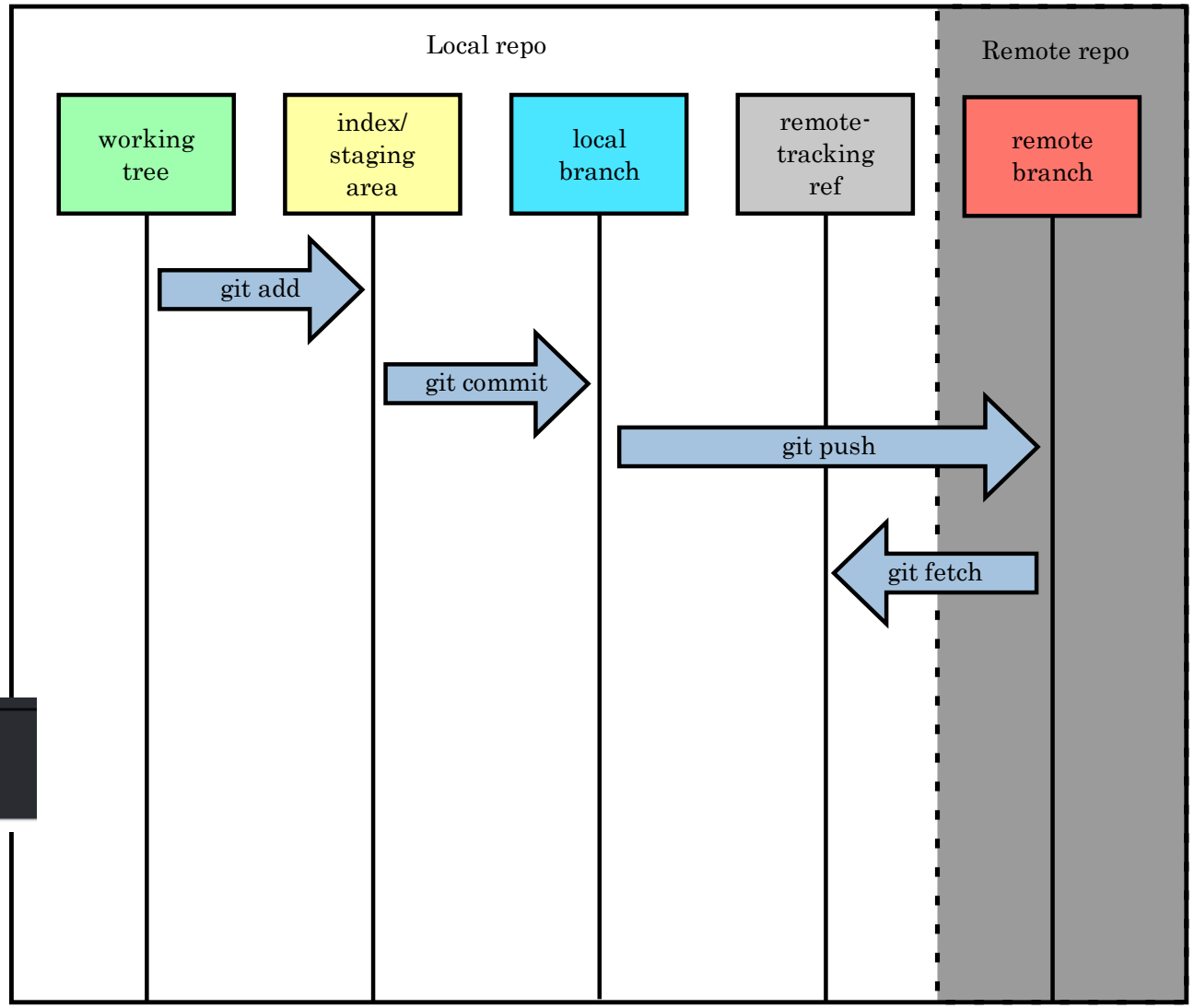


Git fetch

- Remote-tracking ref: Een referentie naar de remote branch die bijhoudt wat de huidige status van de remote branch is.
- Met behulp van git fetch kunnen we de remote tracking branch update. Deze geeft aan of er al dan niet wijzigingen zijn.
- Commando: git fetch
- Github desktop:
- IntelliJ: niet nodig

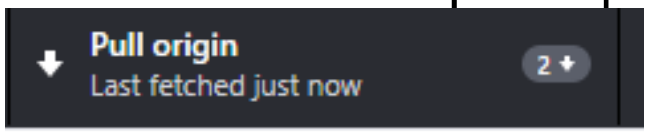


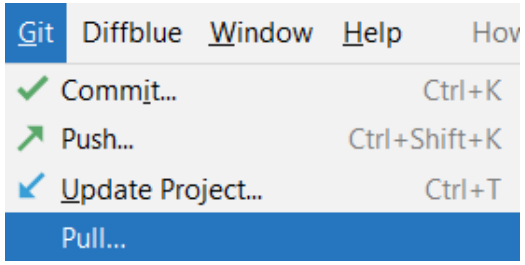


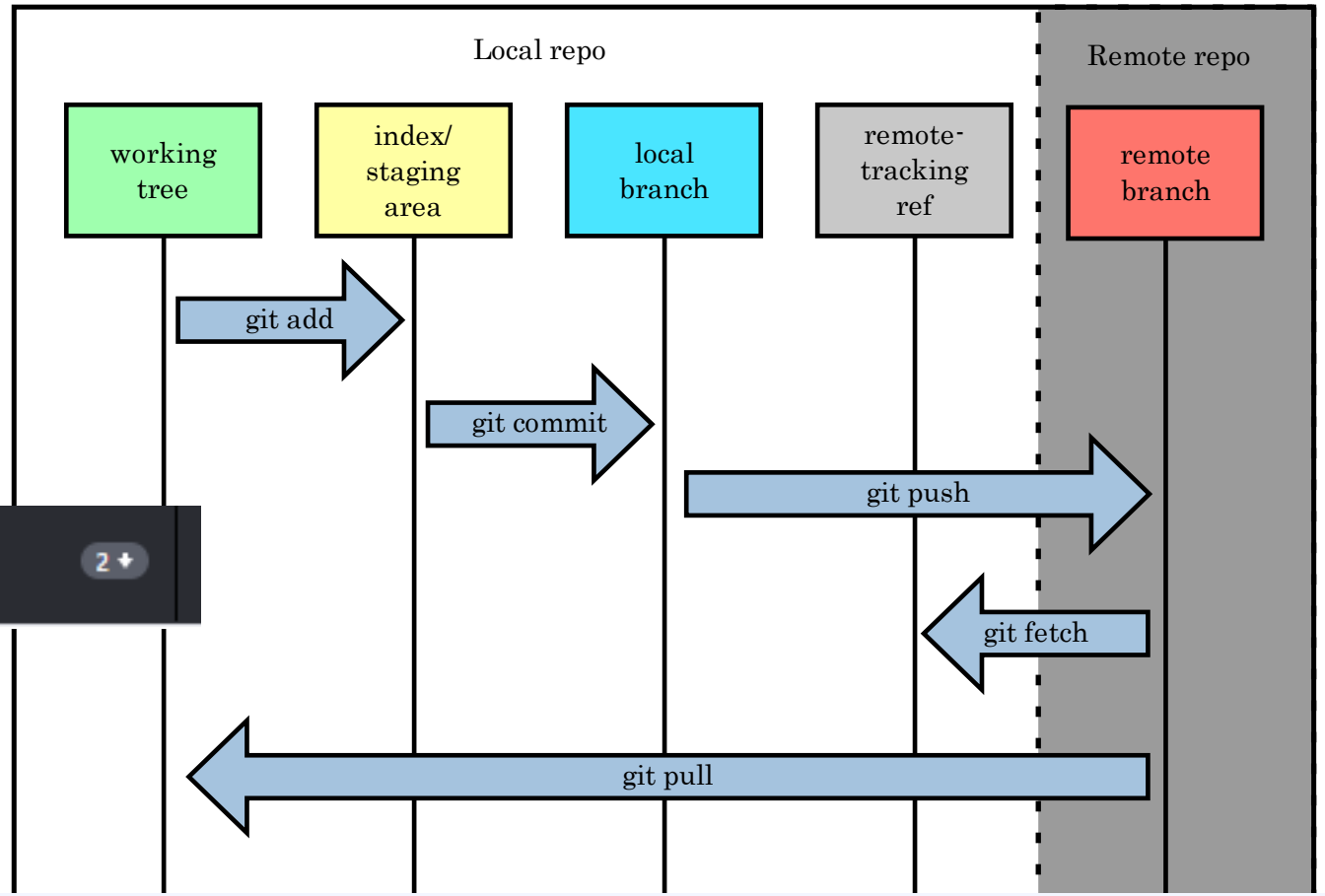


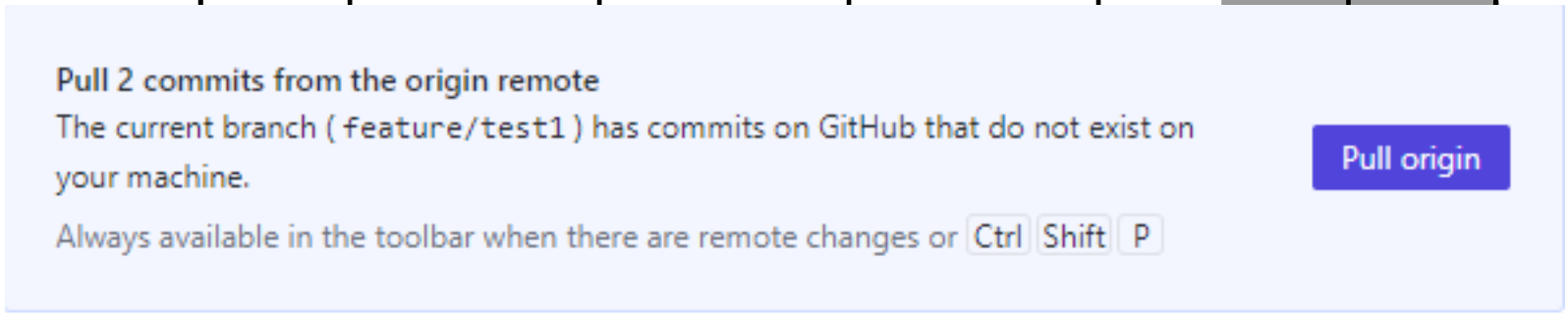
Git pull

- Met behulp van git pull kunnen we wijzigingen op een remote branch binnen hallen. Deze worden automatisch ingesteld in de working tree. Deze wijzigingen worden ook uitgevoerd op de local branch.
  - Commando: git pull
  - Github desktop:
  - IntelliJ:





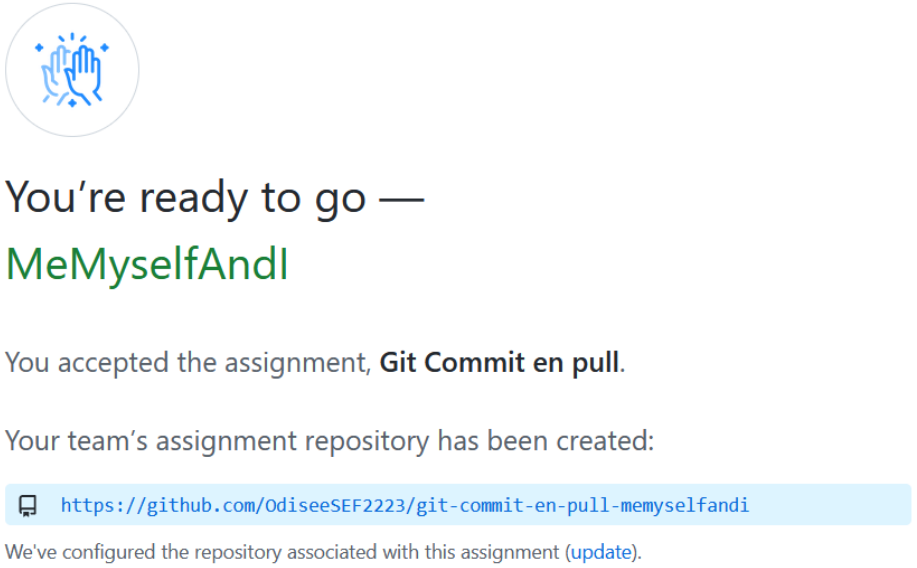




Odisee
DE CO-HOGESCHOOL

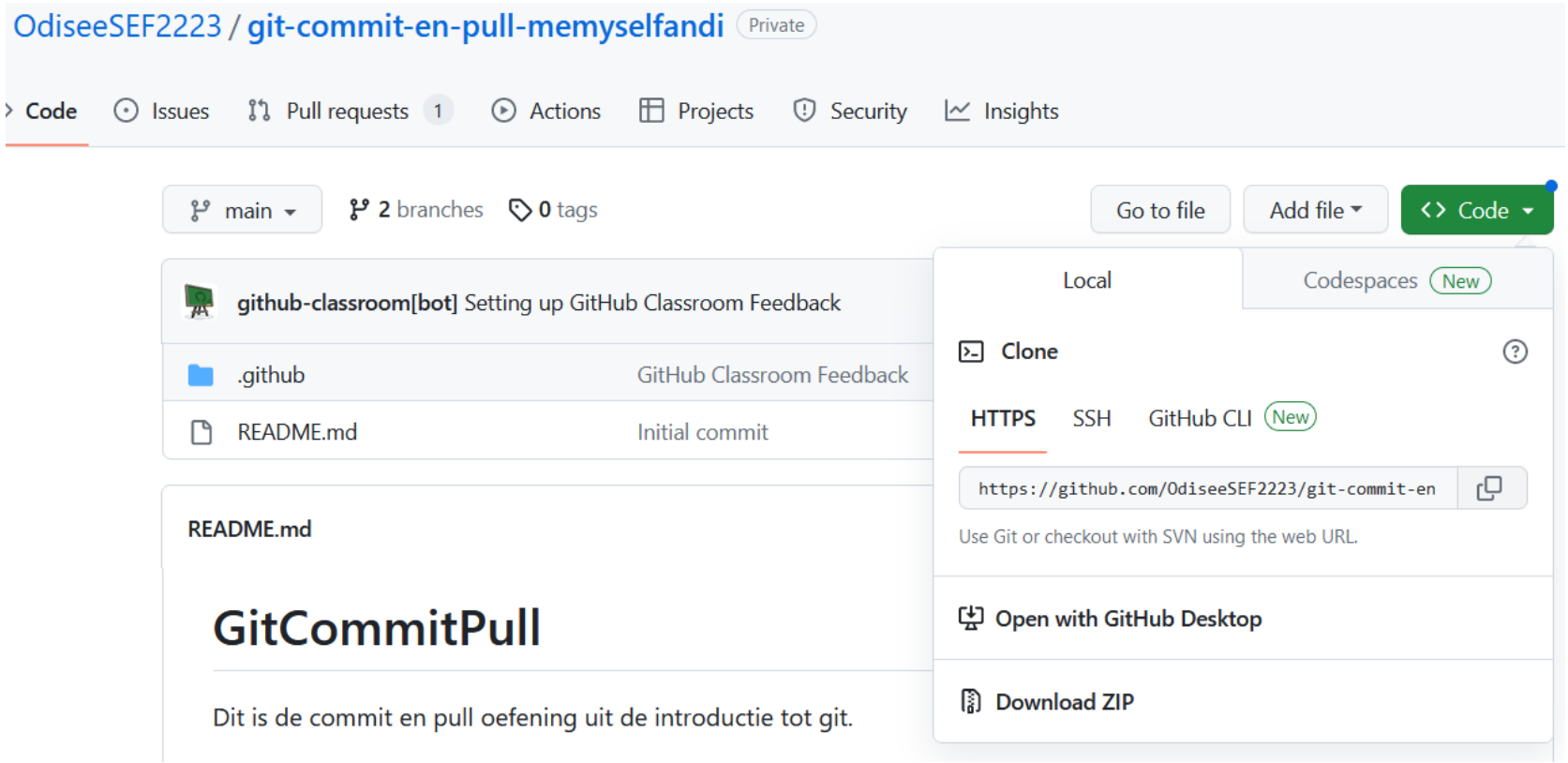
Commit en pull oefening

- Ga naar github classrooms: https://classroom.github.com/a/AY7wpRwM
- Login met je hithub account van de school.
- Werk samen met je buur: 1 van jullie maakt een team aan, de andere joint



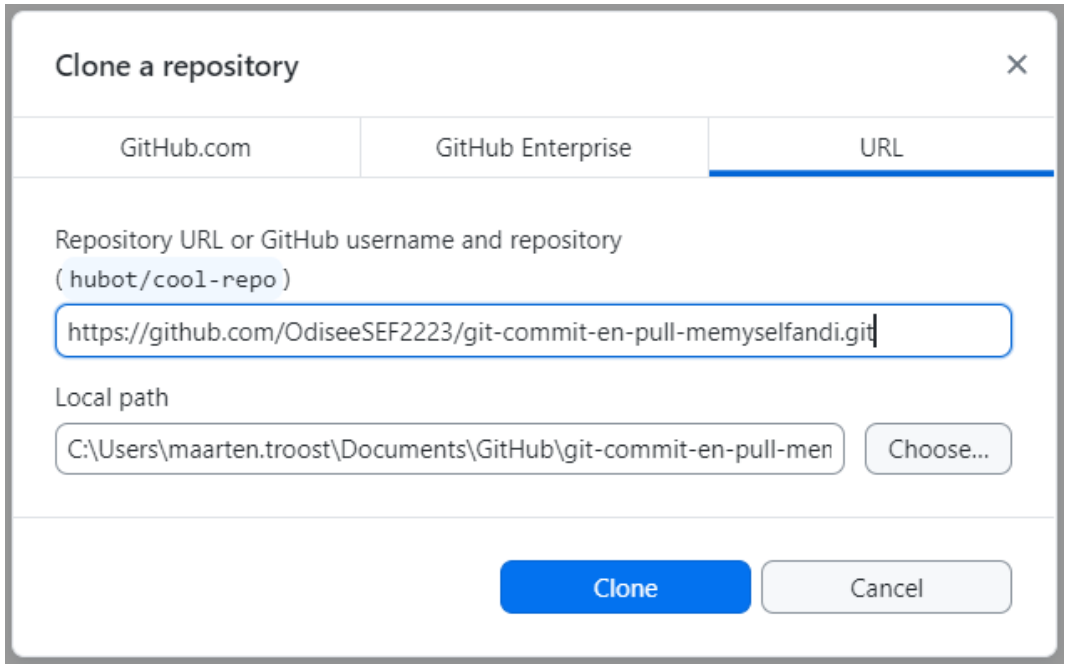
Commit en pull oefening

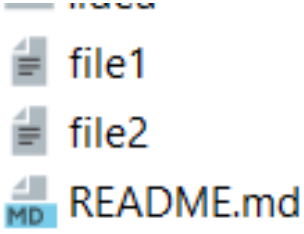
- Op de github pagina van je repo zoek je de correcte URL



Commit en pull oefening

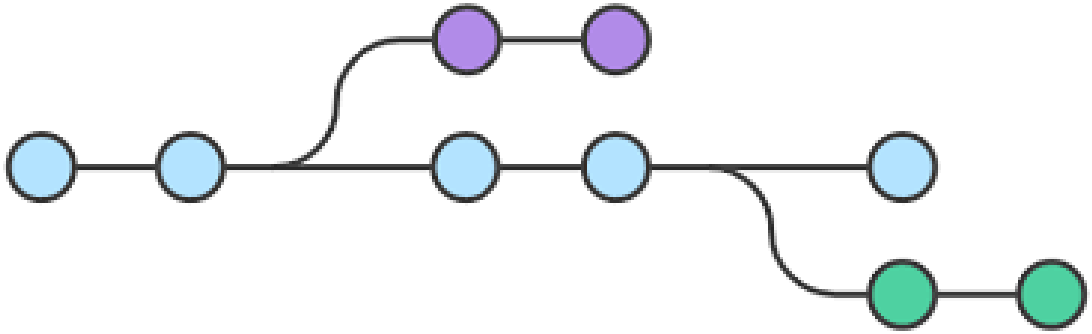
- Via Github desktop clone je je team's repository (welke zich op github bevindt). Nu heb je een eigen lokale kopij, een eigen lokaal repo.
- Ga, in je filesystem, naar de directory van je lokaal repo en voeg een extra file toe.
- Git add: voeg de nieuwe file toe aan de staging area
- Git commit: bevestig deze toevoeging in je lokaal repo
- Git push: push de file naar hithub
- Indien jij en je buur gepusht hebben:
  git fetch en pull om de extra file van je buur te ontvangen.





Git branch

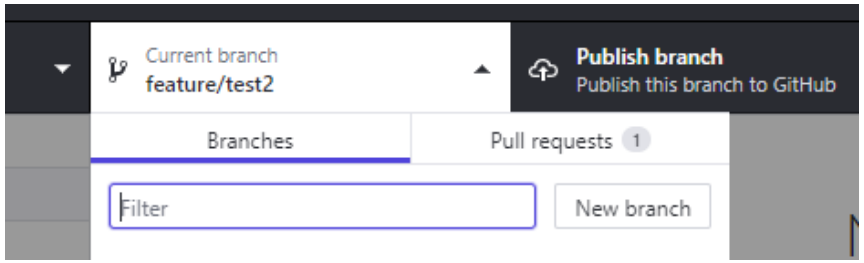
- Een branch is letterlijk vertaald een tak. Het concept van branches in git, is het creëren van aftakkingen. Elke commit gebeurt in 1 branch We kunnen zo verschillende versies maken en toch verder werken op dezelfde code zonder dat code van branch 1 gemengd wordt met code van branch 2.
  Voorbeelden:
  - Versie 1.3 branch = laatste versie voor klanten
  - Versie 1.4 branch = nieuwste versie in ontwikkeling

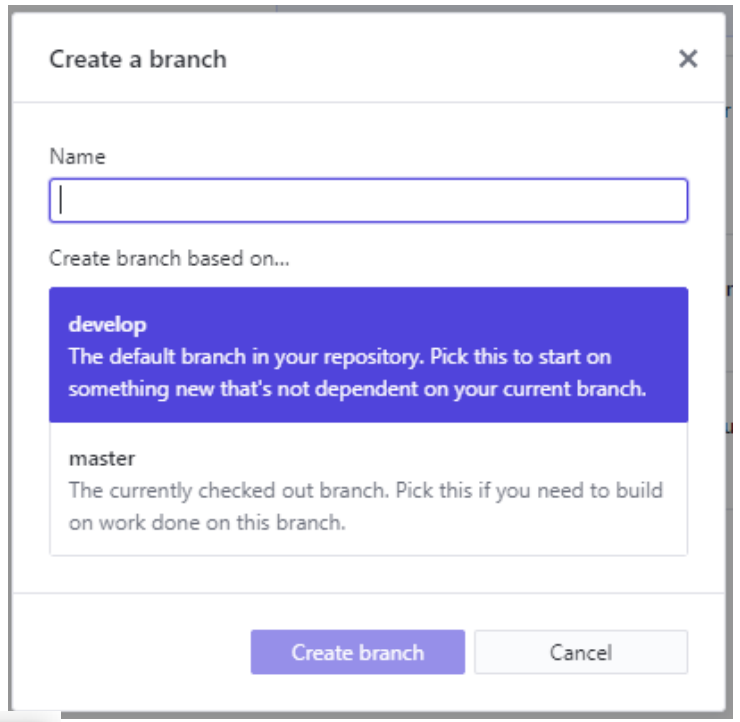


- Elke nieuwe feature wordt vaak in een aparte branch ontwikkeld tot deze voldoende stabiel is.

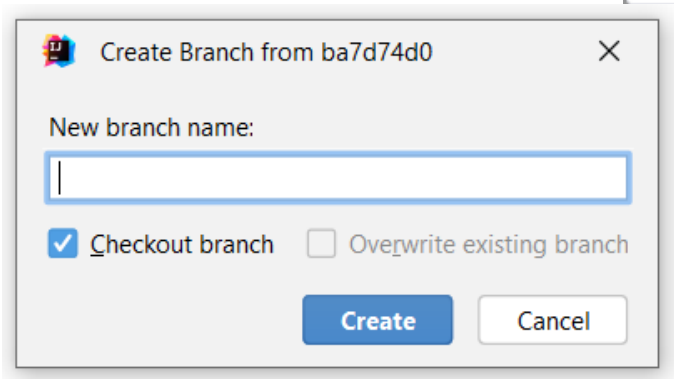
Git branch

- Commando: git branch <branchName>
- Github desktop:

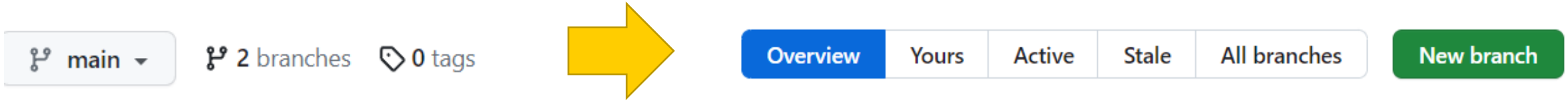




- IntelliJ: Git > New Branch…



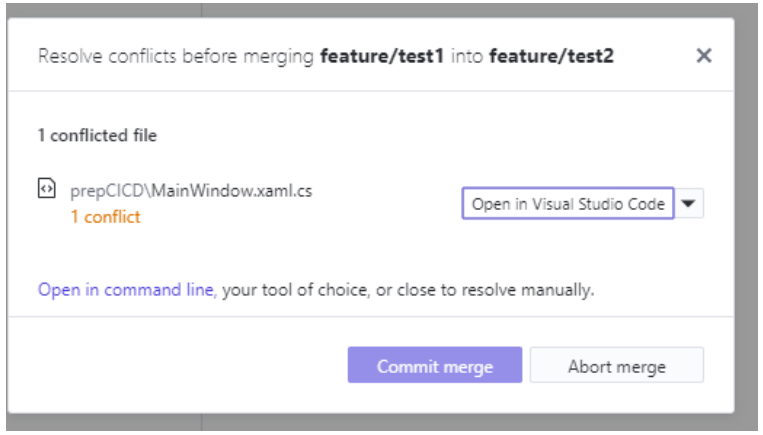
- Github:

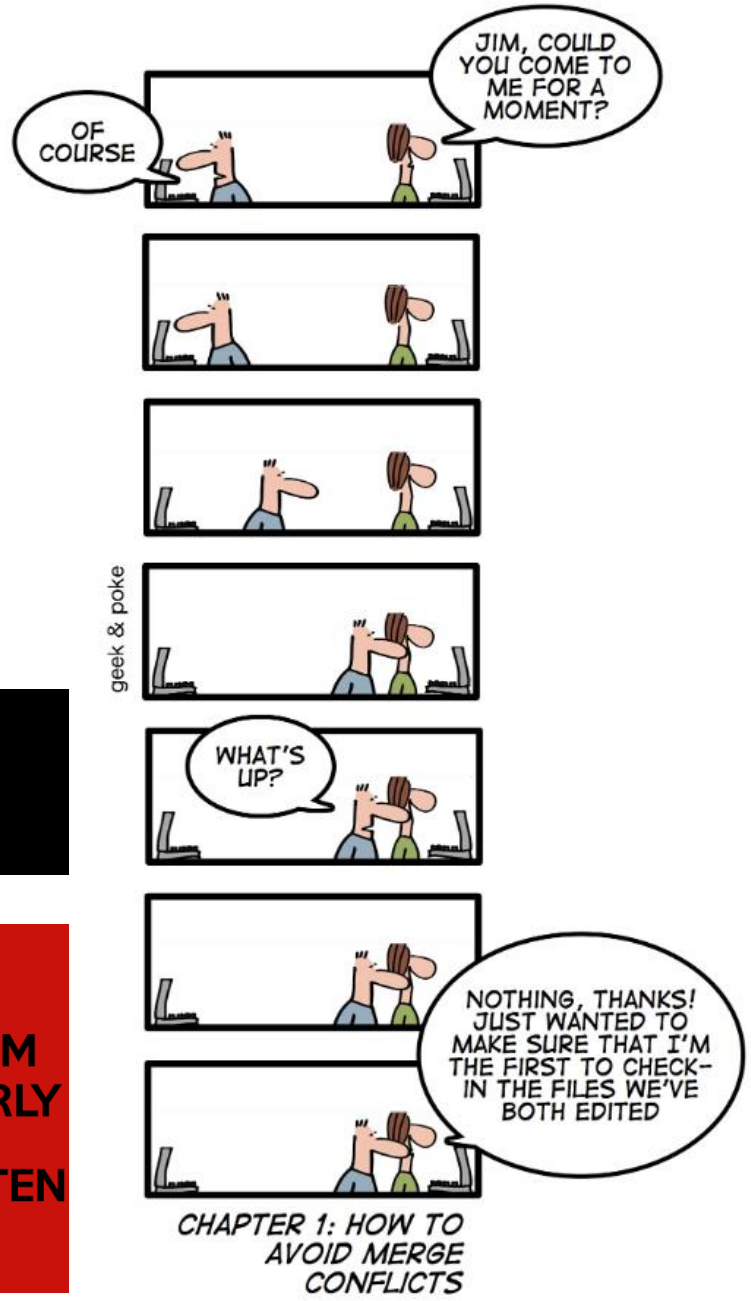


Merge conflicts

- Bij het gebruiken van branches bestaat de kans dat er op de verschillende branches dezelfde files zijn aangepast. In vele gevallen kan git dit zelf oplossen. In sommige gevallen krijgen we echter merge conflicts. Dit wil zeggen dat git niet weet welke code moet behouden worden en welke weg mag.
- In deze gevallen moeten we manueel mergen.

```
matthias.druwe@NBBRU28949 MINGW64 ~/Documents/School/ProgrammeertechniekenEnTesten/PrepCICD (feature/test2)
$ git merge feature/test1
Auto-merging prepCICD/MainWindow.xaml.cs
CONFLICT (content): Merge conflict in prepCICD/MainWindow.xaml.cs
Automatic merge failed; fix conflicts and then commit the result.
```





Manueel mergen

- Merge conflicts kunnen we herkennen aan de typische opmaak van <, = en >. Hiermee wordt aangeduid waar er conflicten zijn opgetreden.
- Hiernaast zien we bij head de code die zelf geschreven is en bij master de branch die we proberen in te mergen.
- We moeten zelf een doordachte keuze maken welke van de 2 of combinatie van de 2 die we willen behouden.

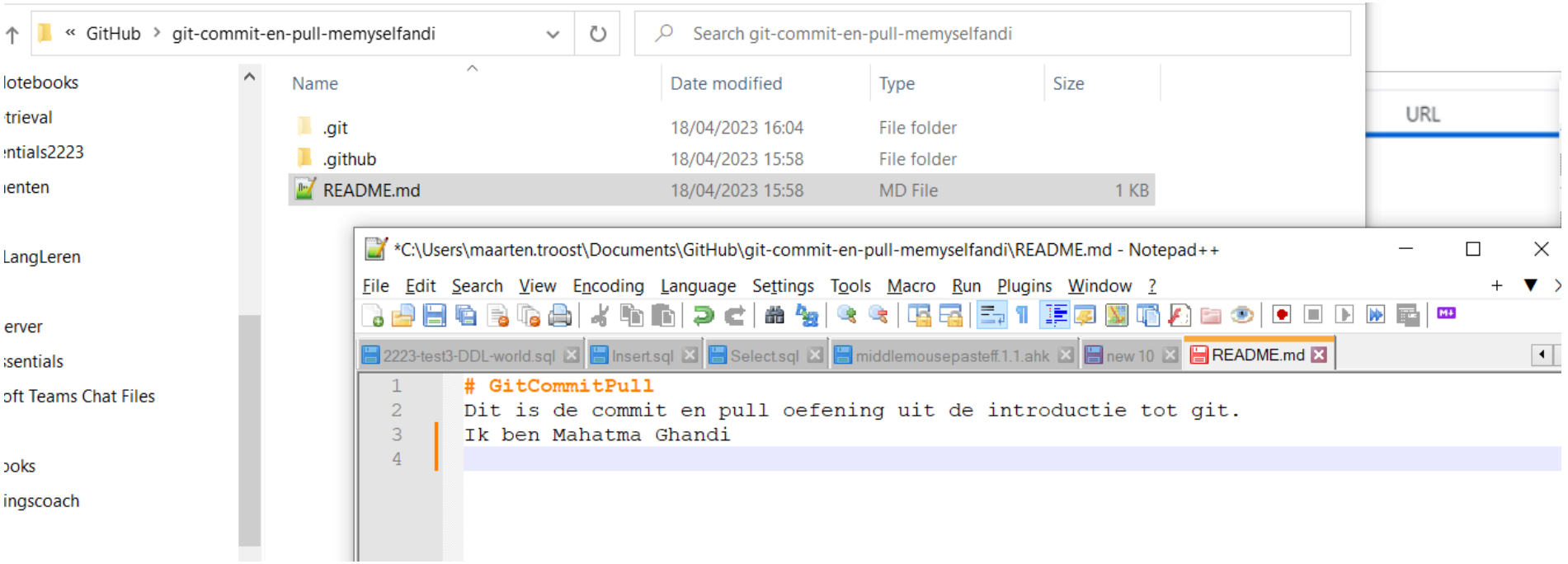
- Let op na dat merge conflicts opgelost zijn, moeten we een commit doen om deze wijzigingen definitief te maken. Deze commit wordt ook een merge commit genoemd.

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

<<<<<<< HEAD
        MessageBox.Show("test1"); // head
=======
        MessageBox.Show("test2"); // master
>>>>>>> Master
    }
}
```

Merging oefening

- Werk verder op de repo van de pull oefening.
- Ga, in je filesystem, naar de directory van je lokaal repo en wijzig de README.md file. Voeg een lijn toe: Ik ben + je naam
- Git commit

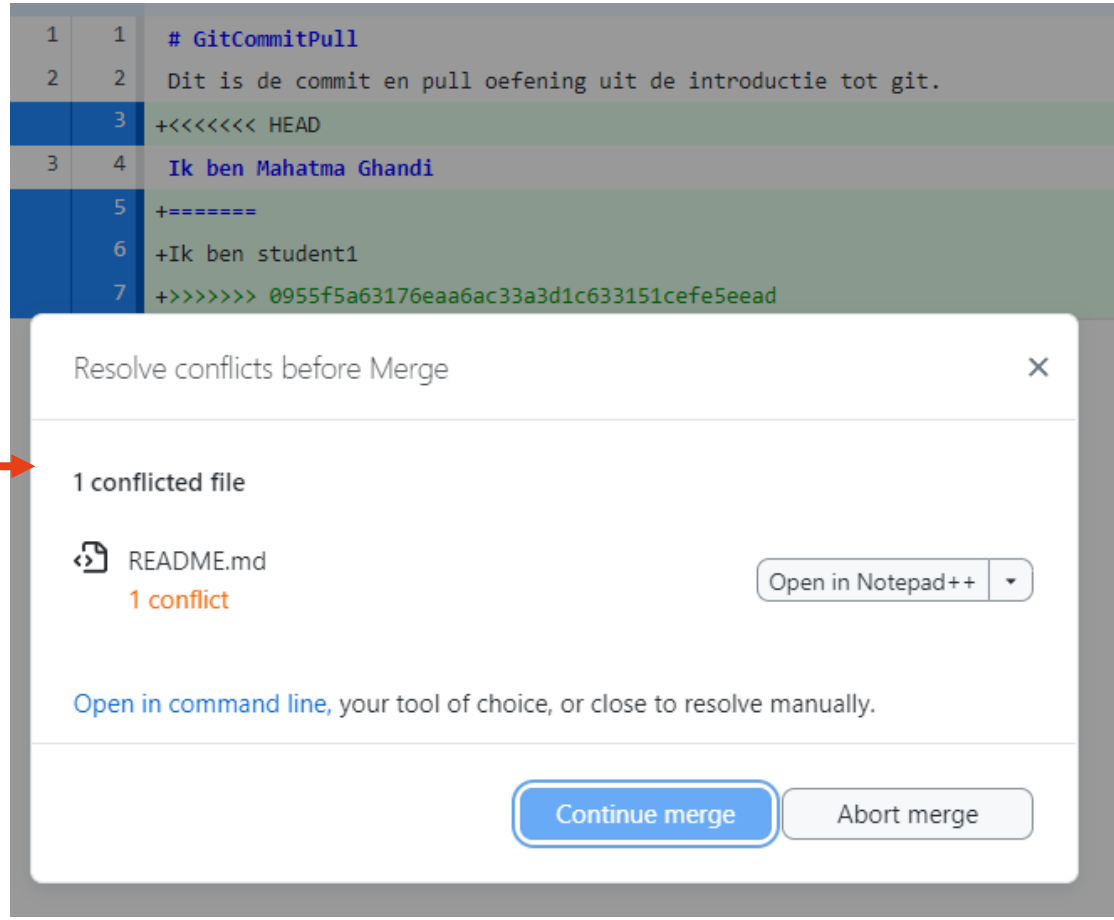


Merging oefening

- Als jij en je buur klaar zijn:
- Student 1 pusht zijn commit naar de github repo

```
1   # GitCommitPull
2   Dit is de commit en pull oefening uit de introductie tot git.
3   Ik ben student1
4
```

- Student 2 fetcht en pullt
  met een mergeconflict tot gevolg



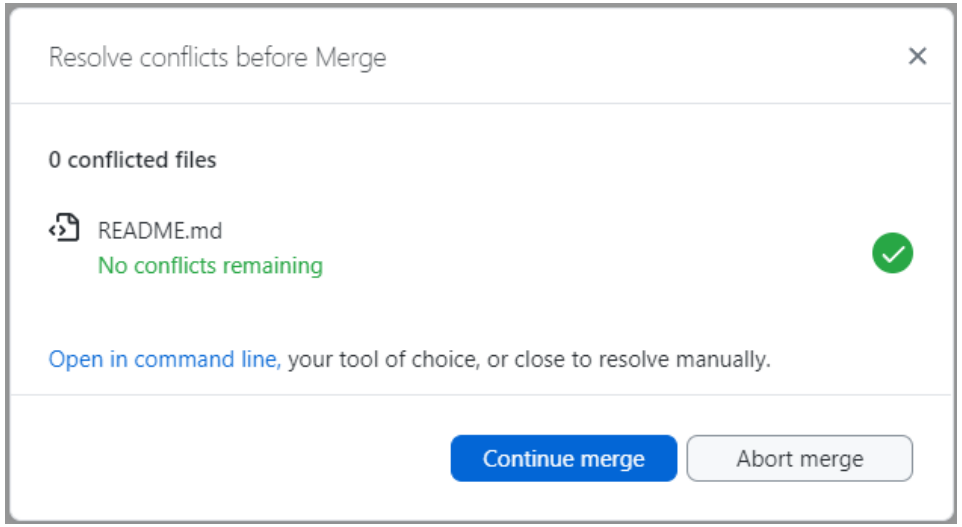
Odisee
DE CO-HOGESCHOOL

Merging oefening

- Student 2 heeft nu een merge van beide aanpassingen en past de README aan naar het gewenste eindresultaat. Vb.

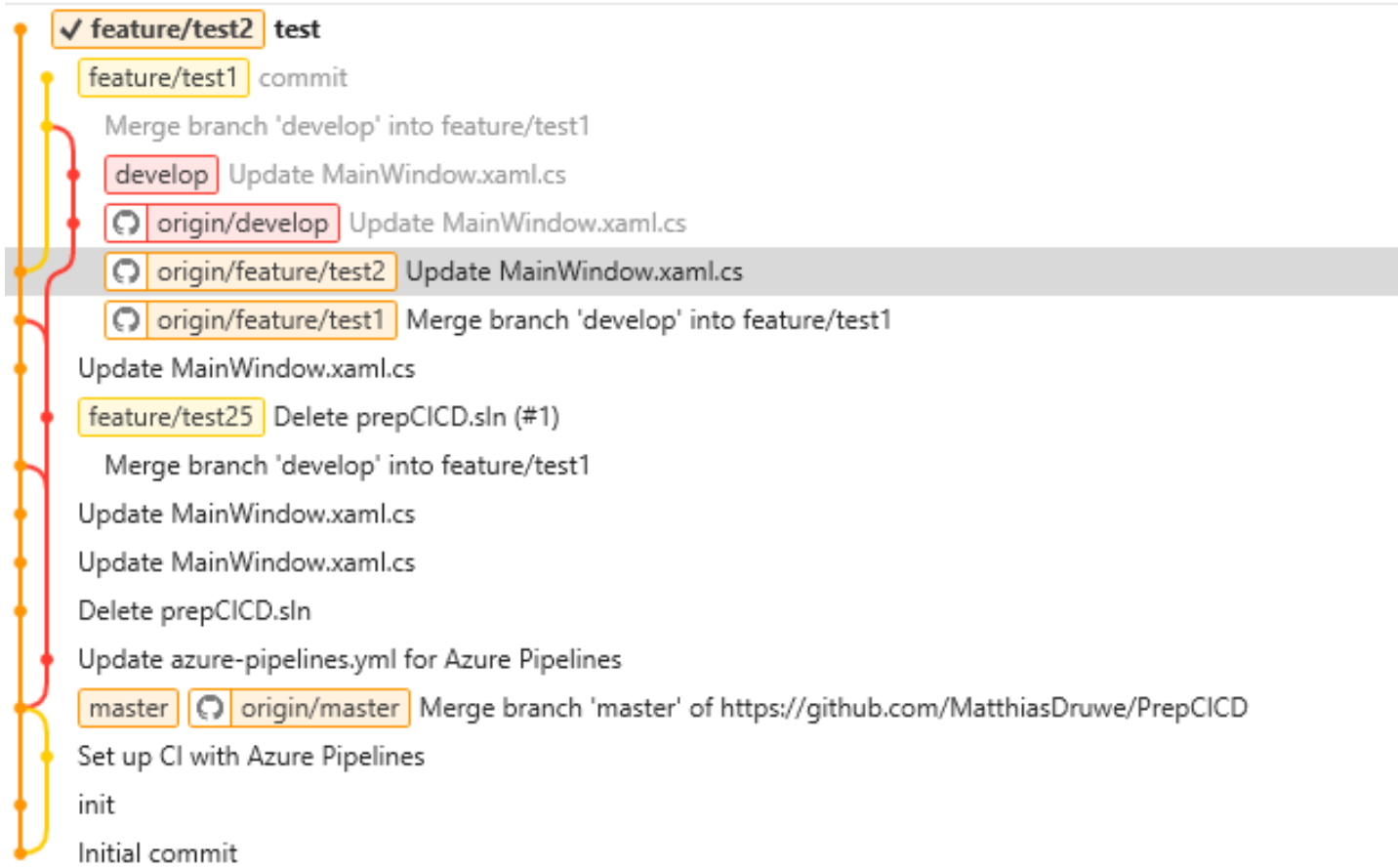
```
# GitCommitPull
Dit is de commit en pull oefening uit de introductie tot git.
Wij zijn student1 en Mahatma Ghandi
```

- Continue merge.
  Dit maakt een nieuwe commit aan.
- Student 2 pusht



Git history

- Sommige tools helpen door een git tree te tonen. Github desktop doet dit helaas niet.
- Tools waarmee dit wel gaat zijn:
- Git gui
- Fork
- Sourcetree



Odisee
DE CO-HOGESCHOOL

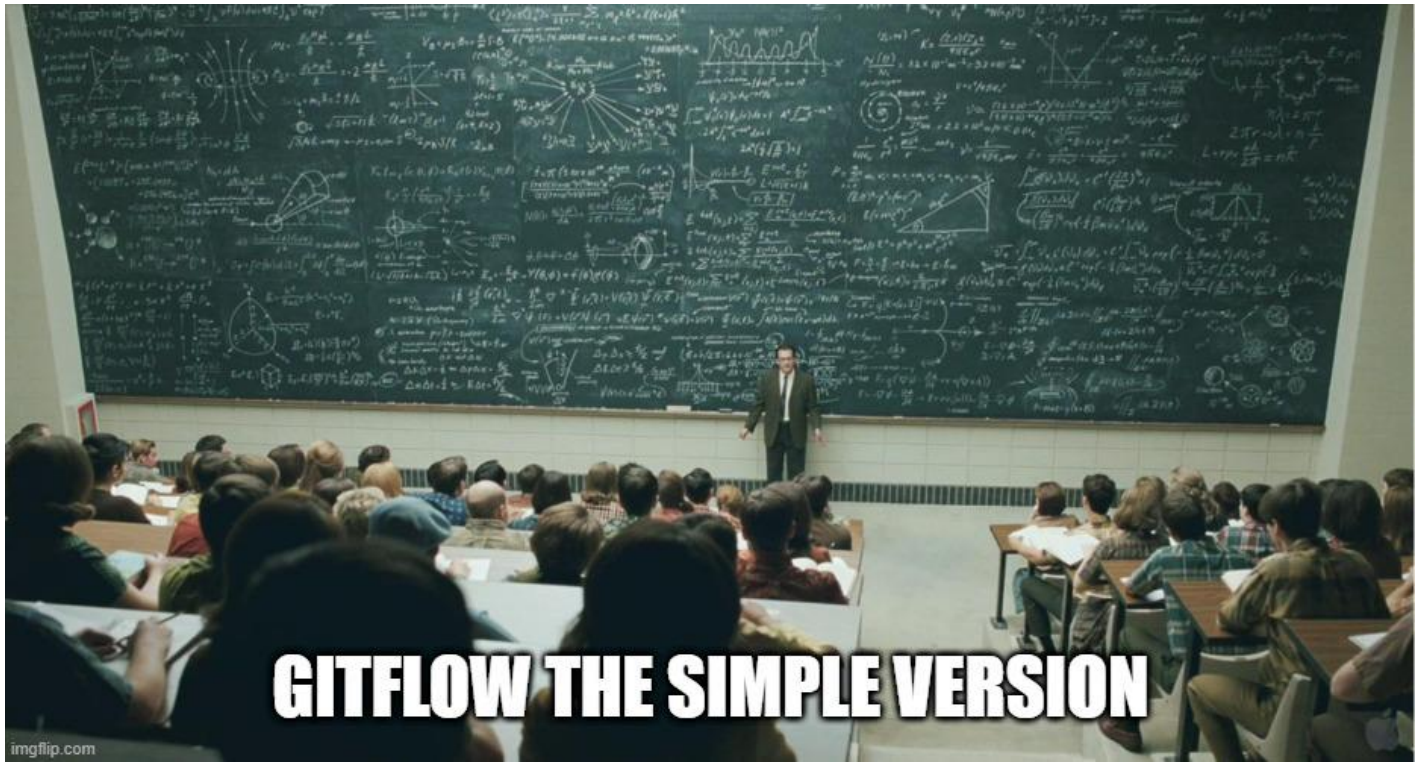
GithubFLOW

GithubFLOW

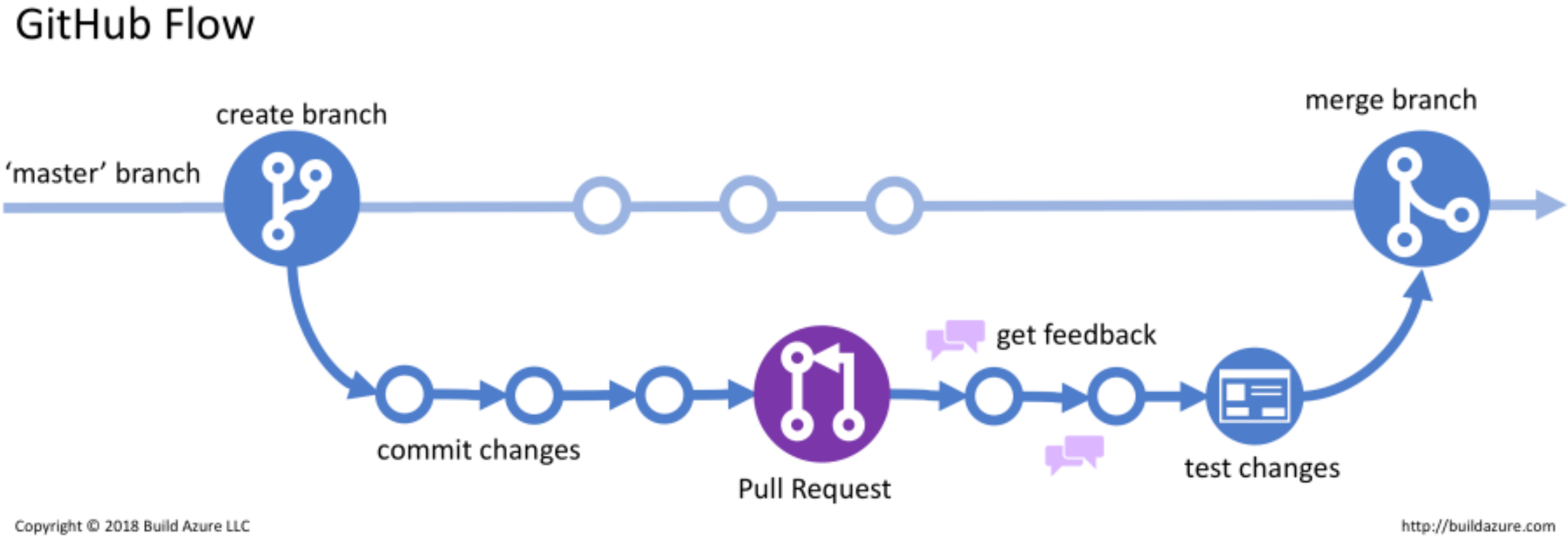
- Githubflow is een workflow om efficiënt en gestructureerd met git en github te werken
- Een manier om branches op te zetten en te mergen
- Er bestaan ook nog ander workflows zoals:
  - Trunk based development
  - Three-Flow
  - Git flow

Waarom?

- Workflow?
  - Structuur
  - Standaarden
  - Efficiëntie
  - Veiligheid



Overzicht Github Flow

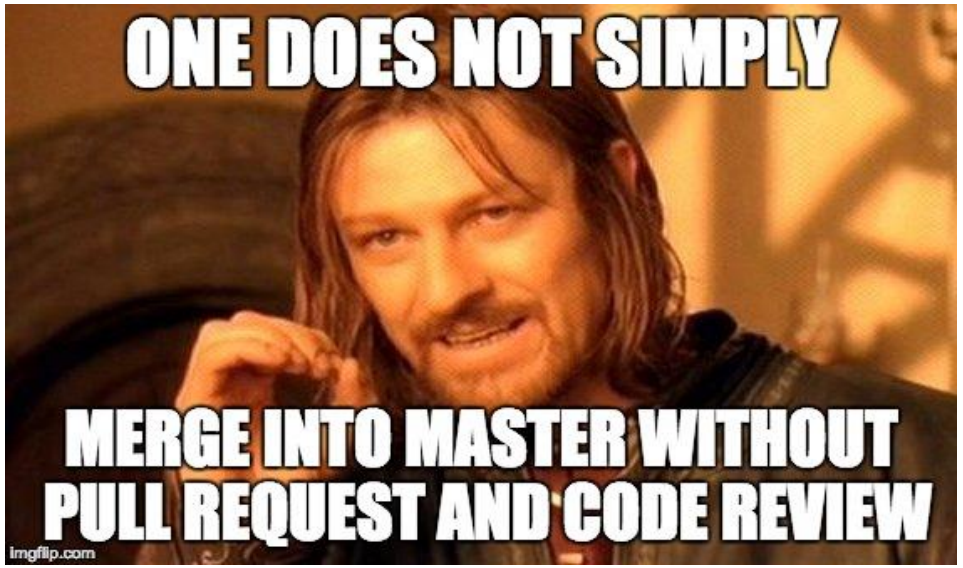


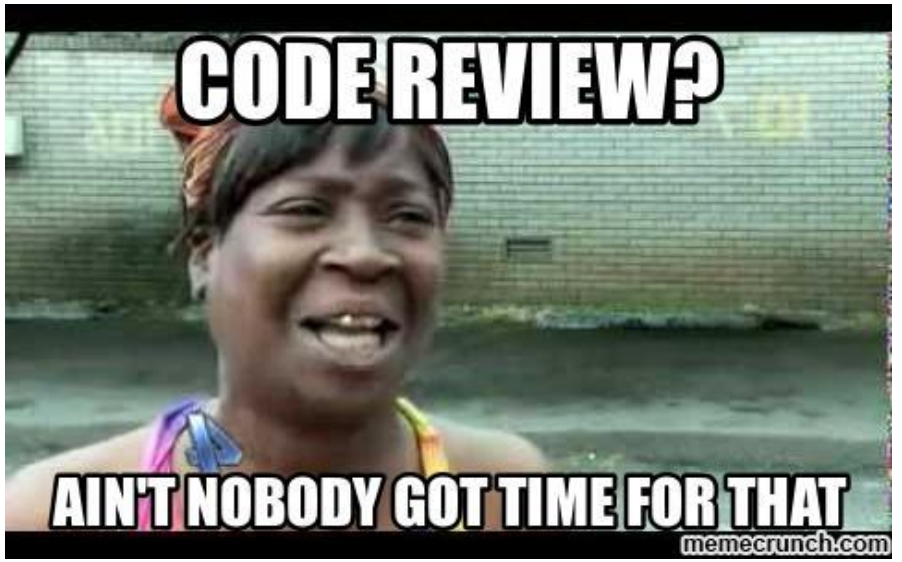
Overzicht Github Flow

- Voor elk onderdeel / requirement / feature:
  - Maak een nieuwe branch
  - Implementeer en test
  - Maak een pull request van je branch om deze in main in te voegen
  - Iemand anders bekijkt je code, voegt de code (lokaal) samen en test
  - Indien goed wordt de branch gemerged en gepusht

GitFlow mergen

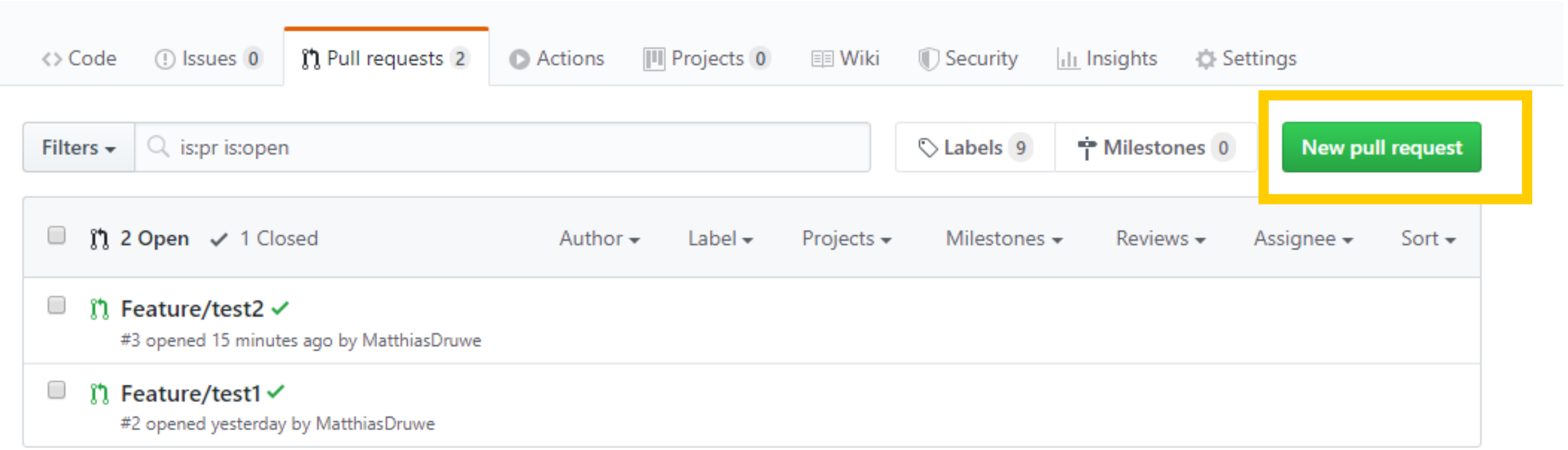
- We gaan niet zomaar mergen in een team. We gaan dit doen aan de hand van pull requests.
- Een pull request is een vraag aan de team leden om te mogen mergen
- Een pull request mag pas gemerged worden wanneer het team een “go” geeft.
- Een pull request review is een “statische” vorm van testing, waarbij code gelezen en nagekeken wordt om zo fouten te ontdekken.





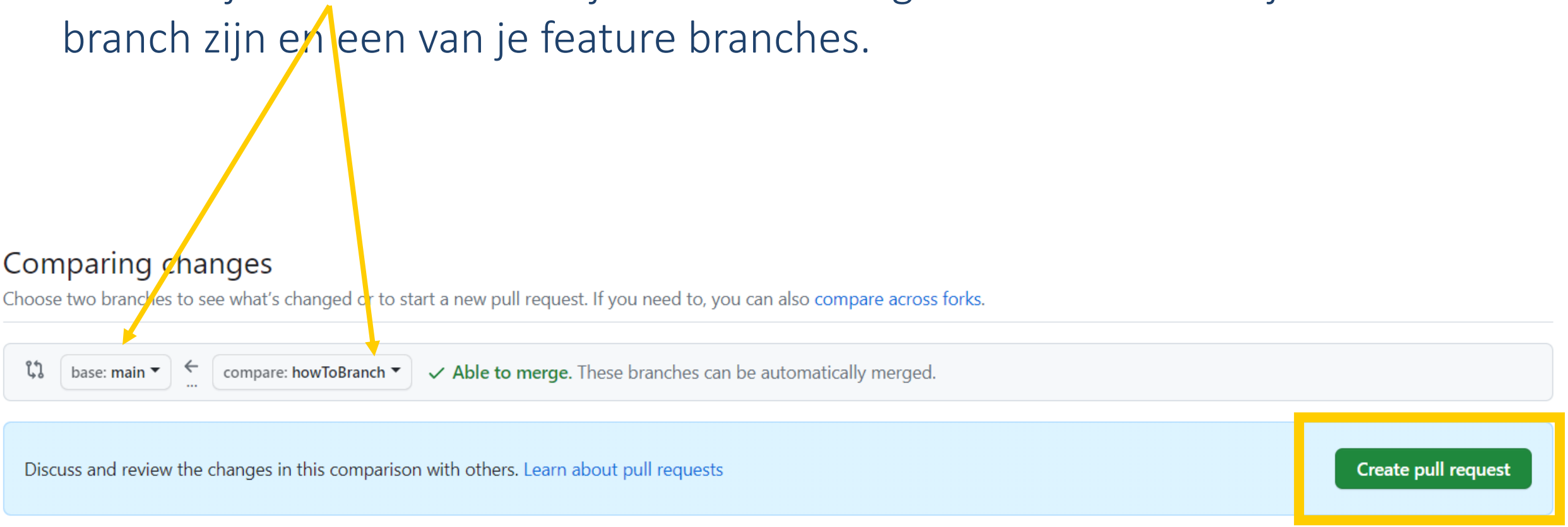
Pull request maken

- Op Github kunnen we pull requests maken aan de hand van de knop new pull request onder de tab Pull requests

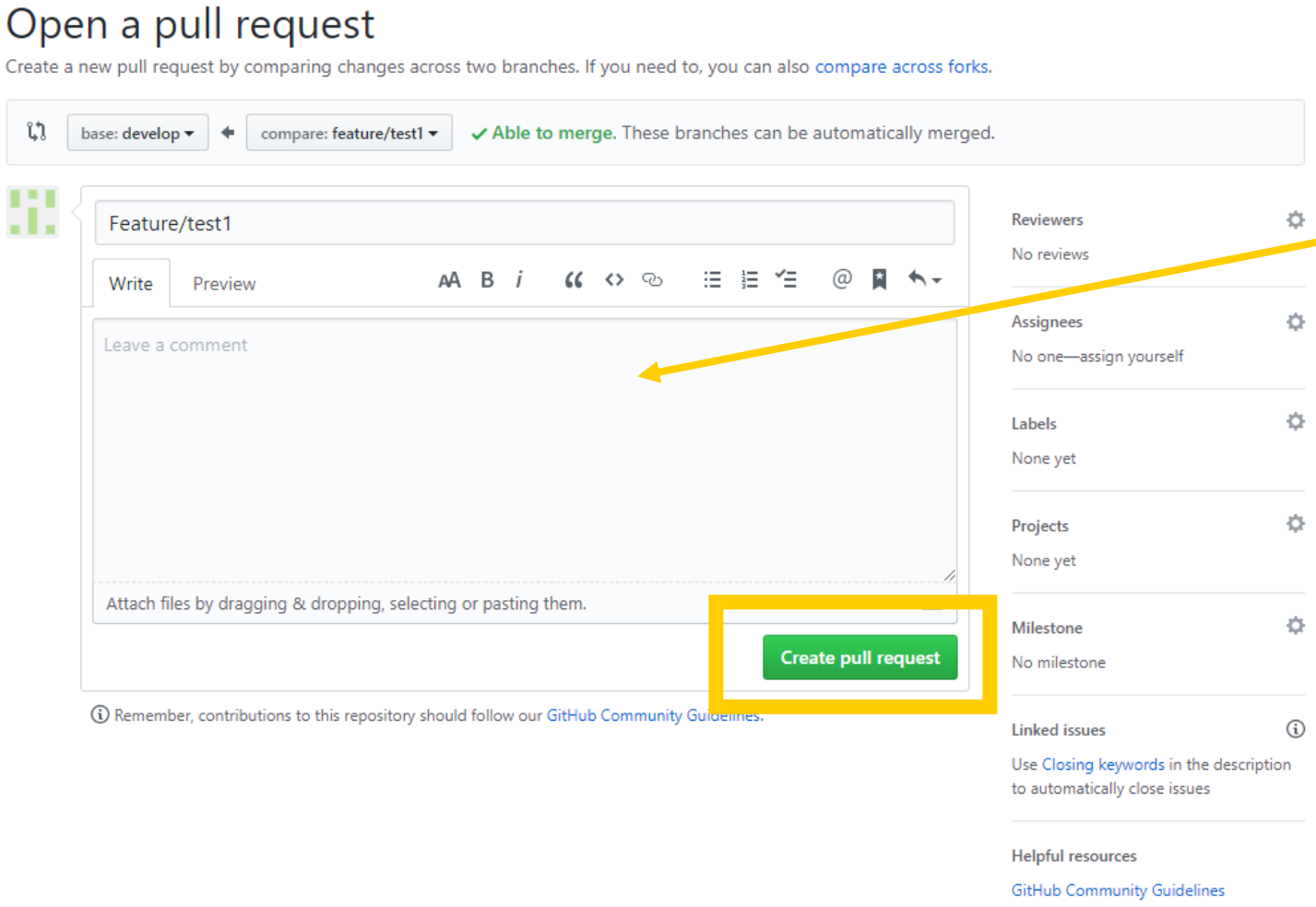


Pull request maken

- Kies de juiste branches die je wil samenvoegen. Meestal zal dit je main branch zijn en een van je feature branches.



Pull request maken

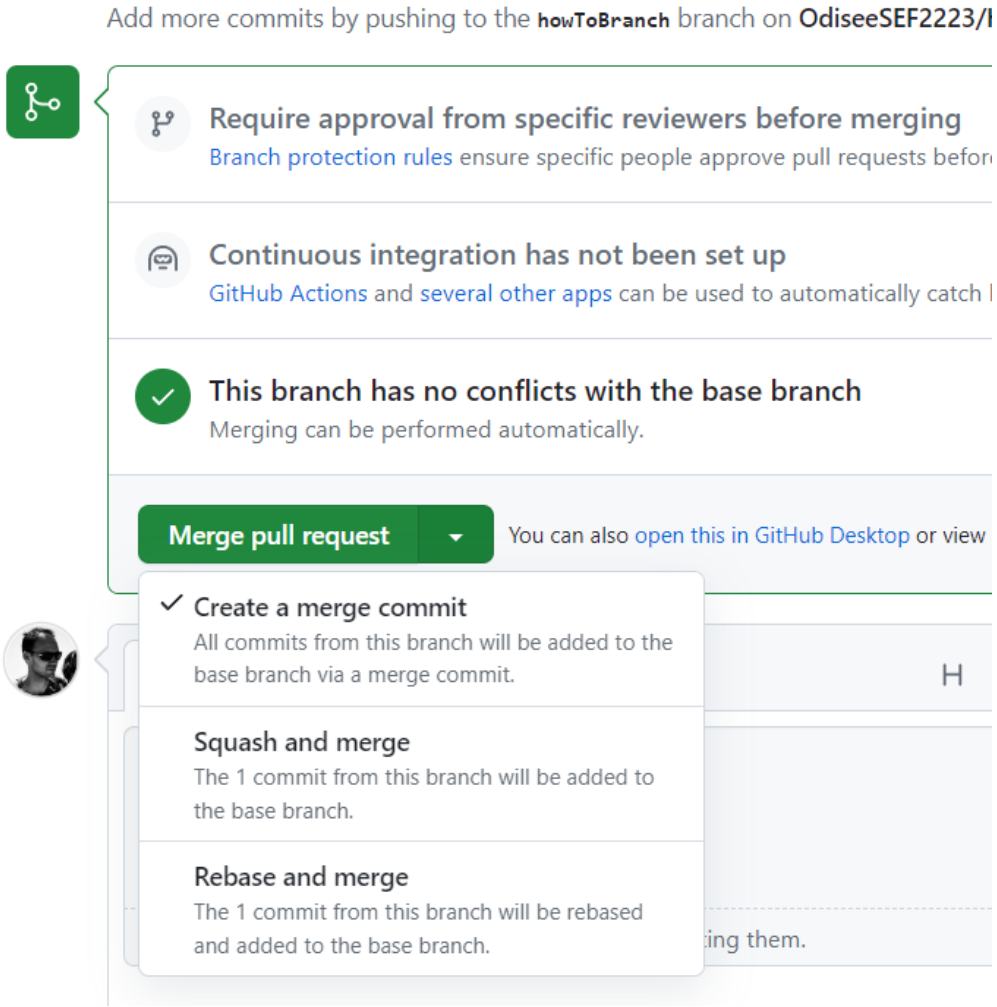


Voorzie je pull request van een woordje uitleg en kies de juiste reviewers

Pull request feedback

- De reviewers krijgen een uitnodiging om de pull request te na te kijken.
- Reviewers kunnen de pull request:
  - Approven => Merge mag gebeuren
  - Declinen => Merge mag niet gebeuren wegens enkele te grote fouten
  - Commenten => Feedback die eerst moet aangepast worden alvorens de reviewer beslist
- Wanneer je wijzigingen pushed op de branch waarvoor een PR (pull request) bestaat wordt deze automatisch upgedate.

Pull request feedback



.gitignore in je project

- Vaak zorgt het builden, of zelfs het editen van code voor files in onze repo die we niet altijd online willen plaatsen op een git server. Dit maakt het vaak moeilijk om de code te reviewen aangezien er veel wijzigingen lijken te zijn terwijl dit vooral automatisch gegenereerde files zijn.
- Met behulp van een .gitignore file kunnen we files aanduiden die we liever niet syncen met remote repositories.
- In IDE's wordt dit vaak automatisch toegevoegd.

```
## Ignore Visual Studio temporary files, build results, and
## files generated by popular Visual Studio add-ons.
##
## Get latest from https://github.com/github/gitignore/blob/master/VisualStudio.gitignore


# User-specific files
*.rsuser
*.suo
*.user
*.userosscache
*.sln.docstates

# User-specific files (MonoDevelop/Xamarin Studio)
*.userprefs

# Build results
[Dd]ebug/
[Dd]ebugPublic/
[Rr]elease/
[Rr]eleases/
x64/
x86/
[Aa][Rr][Mm]/
[Aa][Rr][Mm]64/
bld/
[Bb]in/
[Oo]bj/
[Ll]og/

# Visual Studio 2015/2017 cache/options directory
.vs/
# Uncomment if you have tasks that create the project's static files in wwwroot
#wwwroot/

# Visual Studio 2017 auto generated files
Generated\ Files/

# MSTest test Results
[Tt]est[Rr]esult*/
[Bb]uild[Ll]og.*

# NUNIT
*.VisualState.xml
TestResult.xml
```

Odisee
DE CO-HOGESCHOOL

Git branch en pull request oefening

- Student 1 maakt een nieuwe branch aan
- In die nieuwe branch voeg je een nieuwe file toe en commit deze
- Student 1 maakt een pull request aan om de branch in main in te voegen
- Student 2 doet een review, merget de code en pushed deze
- Student 1 pullt en verifieert dat de wijzigingen in main zijn aangebracht .

Single responsibility principe

SOLID – principes van software architectuur

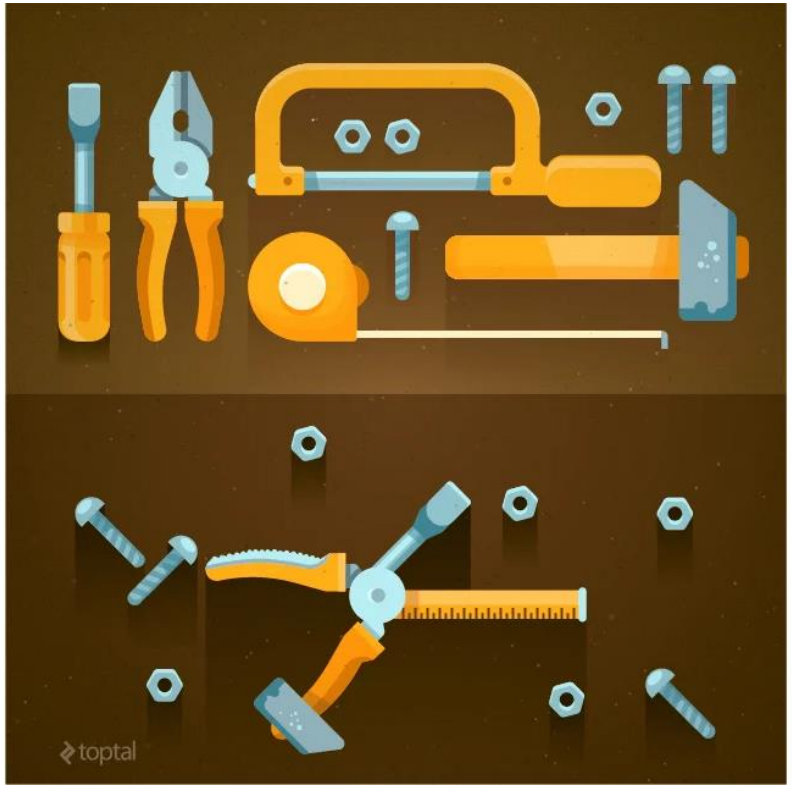
- S – Single Responsibility Principle
- O – Open-closed principle
- L – Liskov Substitution principle
- I – Interface Segregation Principle
- D – Dependency inversion principle

Robert C. Martin (aka Uncle Bob)

https://www.youtube.com/watch?v=7EmboKQH8lM&list=PLUxszVpqZTNShoypLQW9a4dEcffsoZT4k

SOLID – principes van software architectuur

- S – Single Responsibility Principle
  - A class should have only one job
  - Bvb niet het berekenen van waarden en output genereren in 1 klasse
- O – Open-closed principle
- L – Liskov Substitution principle
- I – Interface Segregation Principle
- D – Dependency inversion principle



Wat is er slecht aan volgende code?

```
public class Main {

    public static void main(String[] args) {
        ArrayList<String[]> docenten = new ArrayList<>();
        ArrayList<String[]> studenten = new ArrayList<>();
        ArrayList<String[]> scores = new ArrayList<>();

        while(true) {
            // Print menu window
            System.out.println("Welkom bij Odisee administratie");
            System.out.println("Welke taak wil je uitvoeren (geef het nummer in of x om af te sluiten)");
            System.out.println("1) Voeg Docent toe");
            System.out.println("2) Voeg Student toe");
            System.out.println("3) Voeg score toe");
            System.out.println("4) Print overzicht van alle docenten");
            System.out.println("5) Print alle studenten uit en het aantal vakken waarvoor ze geslaagd zijn");
            System.out.println("x) Sluit de applicatie");
        }
    }
}
```

- Alle data staat in de main als pure String
  - Hierdoor is het moeilijk om handige functies te schrijven om bepaalde features te implementeren

Is dit beter? Zijn er nog opmerkingen?

```java
public static void main(String[] args) {

    ArrayList<Docent> docenten = Docent.docenten;
    ArrayList<Student> studenten = Student.studenten;
    ArrayList<Score> scores = Score.scores;

    while(true) {
        // Print menu window
        System.out.println("Welkom bij Odisee administratie
        System.out.println("Welke taak wil je uitvoeren (ge
        System.out.println("1) Voeg Docent toe");
```

```java
public class Docent {
    2 usages
    public static ArrayList<Docent> docenten = new ArrayList<>();

    1 usage
    private String voornaam;
    1 usage
    private String achternaam;

    no usages
    public Docent(String voornaam, String achternaam){
        this.voornaam = voornaam;
```

De docent klasse conflicteert met het single responsibility principe

- De klasse is verantwoordelijk voor
  - Bijhouden van alle informatie van 1 docent
  - Bijhouden van alle docenten
  - Heeft dus ook alle functies nodig voor beiden te manipuleren / bewaren / in te laden

```java
public class Main {

    public static void main(String[] args) {

        ArrayList<Docent> docenten = School.docenten;
        ArrayList<Student> studenten = School.studenten;
        ArrayList<Score> scores = School.scores;
```

```java
3 usages
public class School {
    1 usage
    public static ArrayList<Docent> docenten = new ArrayList<>();
    1 usage
    public static ArrayList<Score> scores = new ArrayList<>();
    1 usage
    public static ArrayList<Student> studenten = new ArrayList<>();
}
```

```java
public class Docent {

    1 usage
    private String voornaam;
    1 usage
    private String achternaam;

    no usages
    public Docent(String voornaam, String achternaam){
        this.voornaam = voornaam;
        this.achternaam = achternaam;
    }
}
```

Veelgemaakte fouten

- Combineren business logica en in/output

```
public class OrderProcessor {
    public void processOrder(Order order) {
        // perform business logic
        // save order to database
        // send confirmation email
    }
}
```

- Plaats in/output in aparte klassen

```java
public class OrderProcessor {
    private final OrderRepository repository;
    private final ConfirmationSender sender;

    public OrderProcessor(OrderRepository repository, ConfirmationSender sender) {
        this.repository = repository;
        this.sender = sender;
    }

    public void processOrder(Order order) {
        // perform business logic
        repository.save(order);
        sender.send(order);
    }
}
```

Veelgemaakte fouten

- Utility klasses met niet-gerelateerde functies

```java
public class StringUtil {
    public static boolean isNullOrEmpty(String str) {
        return str == null || str.isEmpty();
    }

    public static String trimString(String str) {
        return str.trim();
    }

    public static int countOccurrences(String str, char c) {
        int count = 0;
        for (int i = 0; i < str.length(); i++) {
            if (str.charAt(i) == c) {
                count++;
            }
        }
        return count;
    }
}
```

- Maak kleinere utility klasses

```java
public class StringUtil {
    public static boolean isNullOrEmpty(String str) {
        return str == null || str.isEmpty();
    }
}

public class StringTrimmer {
    public static String trimString(String str) {
        return str.trim();
    }
}

public class StringCounter {
    public static int countOccurrences(String str, char c) {
        int count = 0;
        for (int i = 0; i < str.length(); i++) {
            if (str.charAt(i) == c) {
                count++;
            }
        }
        return count;
    }
}
```

Veelgemaakte fouten

- Database operaties en object mapping

```
public class UserRepository {
    public User getUserById(int id) {
        // execute SQL query
        // map result set to User object
        // return User object
    }

    public void saveUser(User user) {
        // execute SQL query to save User object
    }
}
```

- Splits functionaliteiten in verschillende klassen

```
public class UserMapper {
    public static User mapFrom(UserRecord record) {
        // map UserRecord to User object
        // return User object
    }

    public static UserRecord mapTo(User user) {
        // map User object to UserRecord
        // return UserRecord
    }
}
```

```
public class UserRepository {
    private final UserDatabase db;

    public UserRepository(UserDatabase db) {
        this.db = db;
    }

    public User getUserById(int id) {
        // execute SQL query
        UserRecord record = db.getUserRecordById(id);
        // map result set to User object
        User user = UserMapper.mapFrom(record);
        // return User object
        return user;
    }

    public void saveUser(User user) {
        // execute SQL query to save User object
        UserRecord record = UserMapper.mapTo(user);
        db.saveUserRecord(record);
    }
}
```

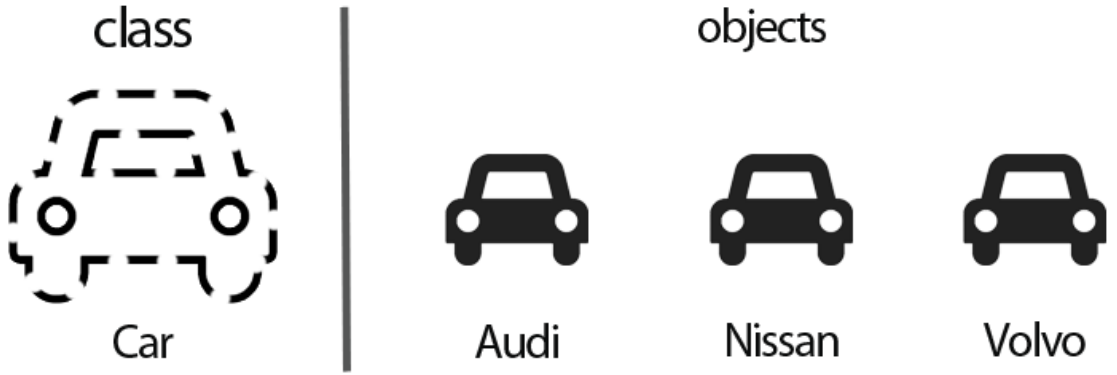
Oefening Single Responsibility principe

- Bestudeer het project in de SRP opgave
  - Wat doet dit project?
  - Welke data wordt er bijgehouden? Wat stelt deze voor?
  - Wat kan er beter?

- Pas de code in het project aan om je voorgestelde verbetering toe te voegen

Refresher OOP

Object-georiënteerde concepten



- Programmeertaal-onafhankelijk
  - Klassen, objecten, methoden, properties, …
  - Inkapseling
  - Overerving, Interfaces, Packages
  - Software architectuur

Work phases in building planning (HOAI[4])	**Phases in software engineering**
1 Basic evaluation	Analyzes
2 Preliminary planning	Definition of requirements
3 Blueprint planning	Software architecture design
4 Approval planning	
5 Implementation planning	Design of components & classes
6 Preparation in award of contract	
7 Participation in award of contract	
8 Supervision/controlling of the construction	Implementation
	Integration and tests
9 Handling of warrantee claims and documentation	Documentation (introduc-tion/training)
	Maintenance

Programma's en realiteit

- Doel van ontwikkelen software is
  - Het maken van een bruikbaar model van de realiteit dat net genoeg is om het probleem op te lossen
- Voorbeeld loonberekening

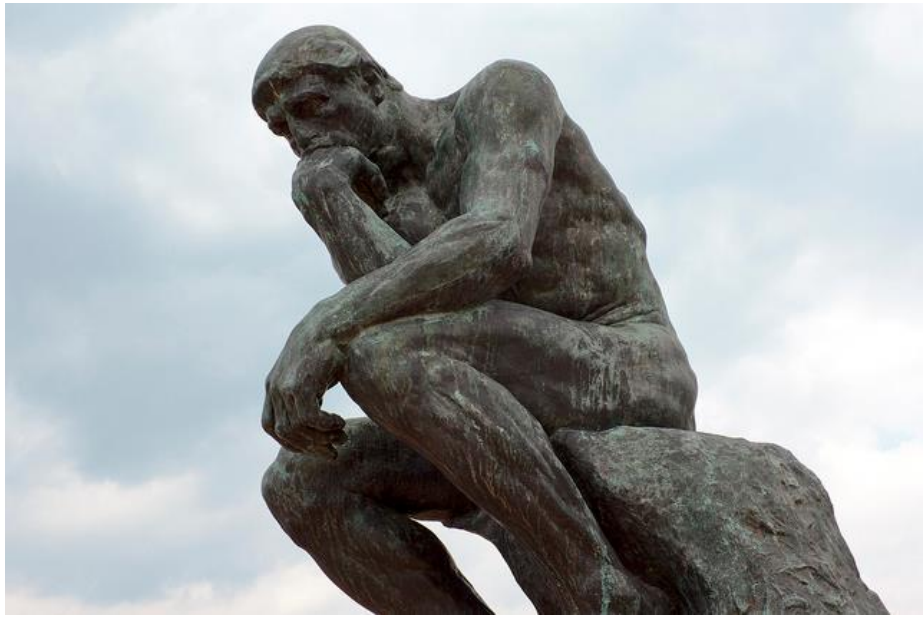
Relevant	Niet-relevant
Naam	Geloofsovertuiging
Voornaam	Haarkleur
Adres	Schoenmaat
Functie	Geaardheid
Aantal dienstjaren	Kleur van ogen
Gehuwd	Lievelingseten
…	…

Programma's en realiteit

- Voorbeeld 2: Administratie huisarts
  - Welke kenmerken zijn relevant?

Kenmerken	
Naam	Aantal dienstjaren
Voornaam	Gehuwd
Adres	Lichaamsgewicht
Geaardheid	Schoenmaat
Kleur van ogen	Gevaccineerd tegen
Beoefende sporten	Lievelingseten
Ziektehistoriek	…

Programma's en realiteit



- In beide voorbeelden gaat het over een Persoon
  - Relevante informatie verschilt van toepassing
  - Soms relevant soms irrelevant
- Vertrek vanuit de werkelijkheid
  - Is een verzameling van objecten die met elkaar samenwerken
  - Objecten verzamelen een reeks data over “iets”
  - Bvb: Een object is een driehoek dat informatie bevat over de drie punten waaruit het bestaat

Oefening: Observeer een kruispunt en merk het volgende op

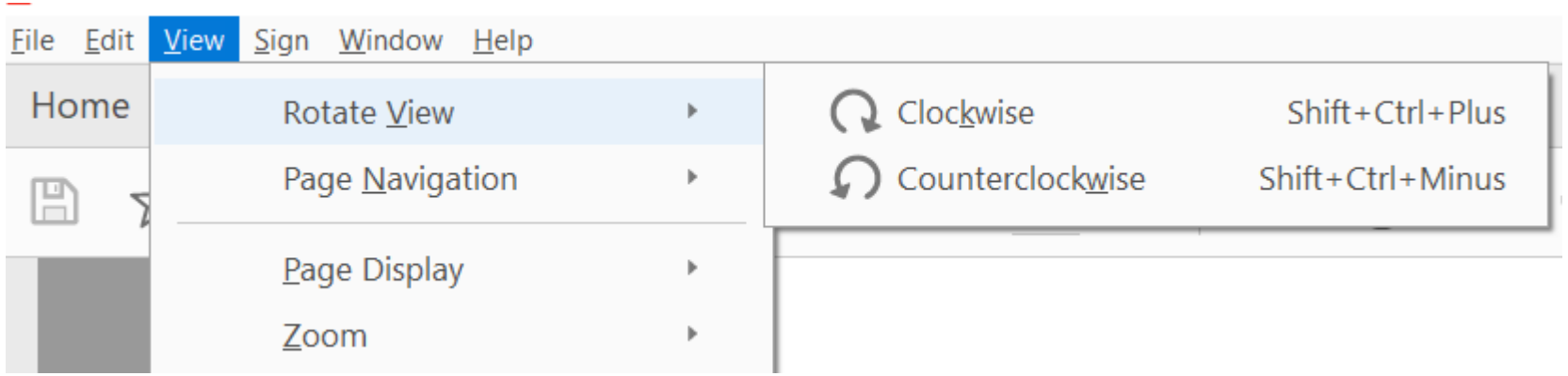
Entiteit	Data	Gedrag
Auto	kleur = groen, merk = Volvo	vertraagt
Auto	kleur = blauw, merk = Audi	rijdt aan 30 km/u
Auto	kleur = rood, merk = BMW	staat stil
Voetganger	geslacht = vrouw, haarkleur = blond, kleding = jas	steekt straat over
Voetganger	type = kind, leeftijd = 10 jaar	wacht voor rood licht
Fietser	geslacht = man, voertuig = fiets (kleur = rood)	wacht voor rood licht
Fietser	geslacht = man, voertuig = fiets (kleur = blauw)	wacht voor rood licht
Fietser	type = meisje, voertuig = fiets (kleur = geel)	rijdt door het groene licht
Verkeerslicht	hoogte = 4m, aantal lichten = 3	rood licht brandt
Verkeerslicht	hoogte = 4m, aantal lichten = 3	groen licht brandt

- Welke klassen zijn er? Hoeveel instanties zijn er van elke klasse? Welke eigenschappen hebben de verschillende klassen? Welke operaties zijn er?
- Welke interacties tussen objecten zijn er?

Oplossing

- Klassen:
  - Auto met 3 instanties
    - Eigenschappen: kleur, merk, …
    - Operaties: rijden aan een snelheid, vertragen, stilstaan
  - Voetganger met 2 instanties
  - Fietser met 3 instanties
    - Eigenschappen: type, geslacht, voertuig
    - Operaties: wachten, rijden
  - Verkeerslicht met 2 instanties
- Interacties
  - Voetganger wacht bij een verkeerslicht
  - auto vertraagt omdat het een verkeerslicht nadert

Oefening: Applicatie menu



Entiteit	**Data**	**Gedrag**
Menu?	Naam=View, bevat=<andere menu's>, plaats=boven	Klapt open en dicht, Toont geselecteerd
Menu?	Naam=Rotate View, bevat=<andere menu's>, plaats=niveau 1 onder	Klapt open en dicht, Toont geselecteerd
Menu?	Naam=Clockwise, bevat=NULL, plaats=niveau 2 onder, shortcut=shift+ctrl+plus	Start actie, Toont geselecteerd

Oplossing

- Klassen:
  - MenuBranch
    - Eigenschappen: bevat andere MenuBranches en MenuLeaves, naam, kleur
    - Operaties: selecteer, klapOpen, klapDicht
  - MenuLeaf
    - Eigenschappen: naam, kleur, icoon, shortcut
    - Operaties: selecteer, startActie

- Waarom niet 1 klasse Menu waar we alles inzetten?
  - Single responsibility principle

Belangrijke terminologie

- Een programma is de realisatie van een model van een stukje werkelijkheid.
- OOP is een methode van programmeren die het toelaat zo dicht mogelijk bij concepten van de te modelleren realiteit te blijven.
- Een klasse is de beschrijving van een reeks gelijksoortige objecten.
- Een dergelijk object heet dan een instantie van de klasse.

Nieuwe termen en begrippen

Klasse
Object
Instantie

Wat gaan we zien in de loop van dit vak over OOP?

Principle	What it is	Why we use it	How to do it
Data encapsulation	Technique that hides information from the users of that class	To reduce the level of software development complexity	Use access modifiers such as `public`, `private`, and `protected`
Inheritance	Technique to allow a derived or child class to use parts of a base or parent class	To promote the re-use of the software	Use the `extends` keyword
Polymorphism	Technique which supports different behavior of methods that is dependent on the object the method is executing against	To make an application more maintainable	Inherent to the Java language

Java – Naming conventions

Name	Convention
class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

Opdrachten

Leerobjecten bekijken

- Beschikbaar op Toledo
- Herhaling van de les

Opdrachten

- De opdracht staat op github classroom en kan bereikt worden via de link
  - TI 1-2: https://classroom.github.com/a/FqyseyW6
  - TI 3-4: https://classroom.github.com/a/cjkvnn4j
  - TI 5-6: https://classroom.github.com/a/TLNOfoDh
  - TI 7-8: https://classroom.github.com/a/QBQpxQAU

- Dien de code in door het te pushen naar github
  - eventueel via IntelliJ git integratie
  - of externe git applicatie zoals github desktop