Probleemoplossend denken

- Inleiding
- 1. Probleemdefinitie
- 2. Analyse
- 3. Algoritme opstellen
- 4. Programma schrijven
- 5. Testen en documenteren

Inleiding

> **Wat moet je kennen en kunnen na dit deel?**
> - Weten welke stappen er bij probleemoplossend denken gebruikt worden.
> - Weten in welke volgorde de stappen uitgevoerd worden.

Probleemoplossend denken

Algoritmisch denken wordt ook probleemoplossend denken genoemd.
Als je een probleem hebt begin je best niet willekeurig wat dingen uit te proberen om tot je doel te komen maar denk je best even na welke stappen je kan ondernemen.

Bij probleemoplossend denken gebruiken we een paar stappen om tot een goed algoritme te komen.

1. Probleemdefinitie
2. Analyse
3. Algoritme opstellen
4. Programma schrijven
5. Testen en documenteren

Elk van deze stappen zullen we nu nader bekijken in de volgende stukken.

1. Probleemdefinitie

> **Wat moet je kennen en kunnen na dit deel?**
> - Een probleem en een doel kunnen identificeren.

In deze stap probeer je eerst je probleem en je doel uit te zoeken.
Deze stap wordt soms ook de probleemstelling genoemd.

Lego voorbeeld

Probleem: Je hebt een hoop Legoblokjes maar je weet niet hoe je een kasteel kan bouwen.
Doel: Je wilt een kasteel van Legoblokjes hebben.

Oefeningen

Oefening 1

Oefening 2

Stel het is lunchpauze en je wilt je lunch eten.
Welke van deze afbeeldingen zou het probleem zijn, en welke het doel?

2. Analyse

> **Wat moet je kennen en kunnen na dit deel?**
> - Weten wat je moet doen in een analyse

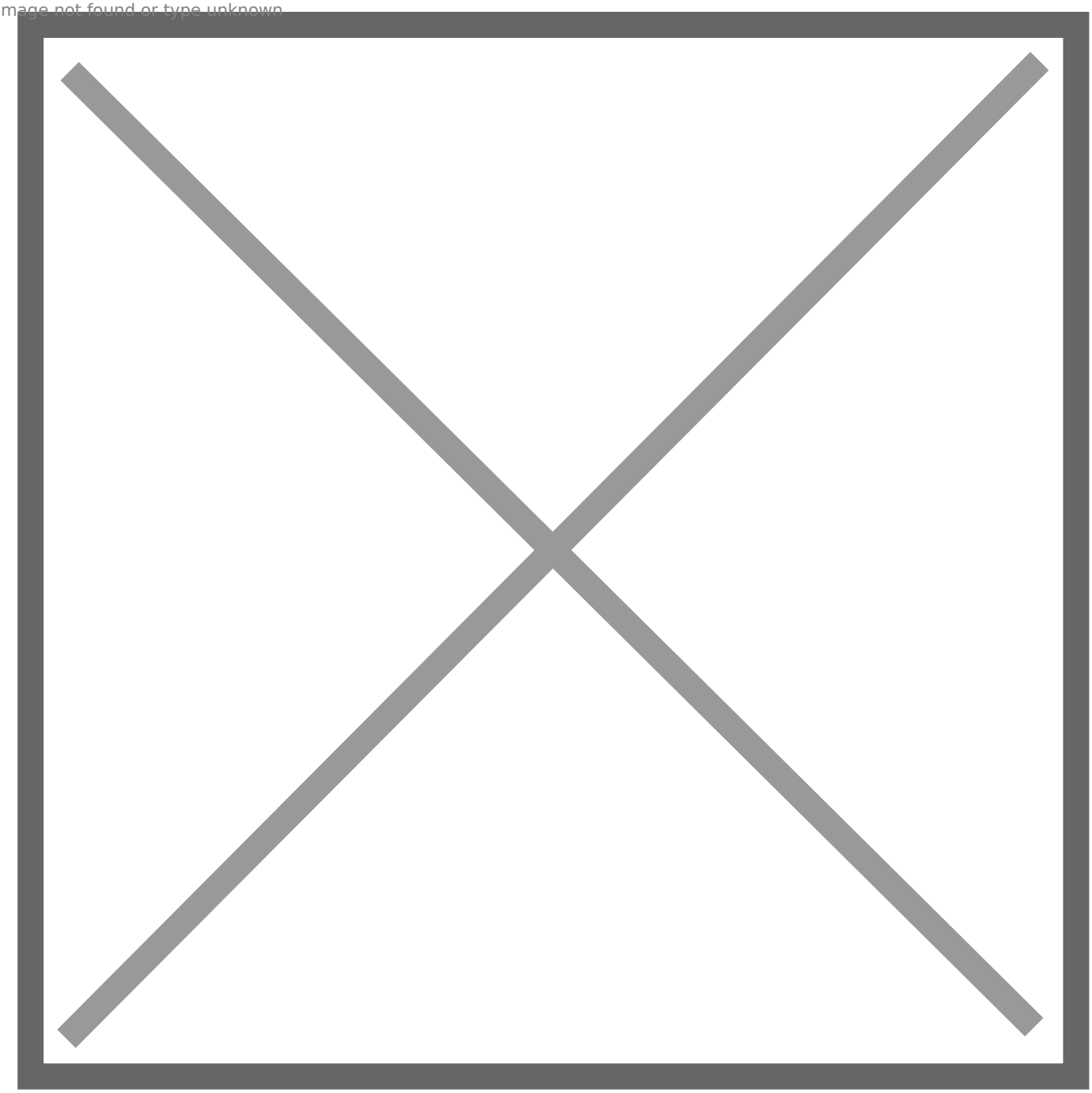
Analyse

- Wat heb je tot je beschikking?
- Welke verwerking moet er met deze beschikbare onderdelen gebeuren?

Lego voorbeeld

Ter beschikking:
We hebben witte blokken, gele blokken, grijze blokken, enz...
Mochten we zien dat we geen blauwe blokken hebben kunnen we dus later bij het opstellen van het algoritme niet zeggen “bouw een blauwe toren”, want daarvoor hebben we niet alle benodigde materiaal.

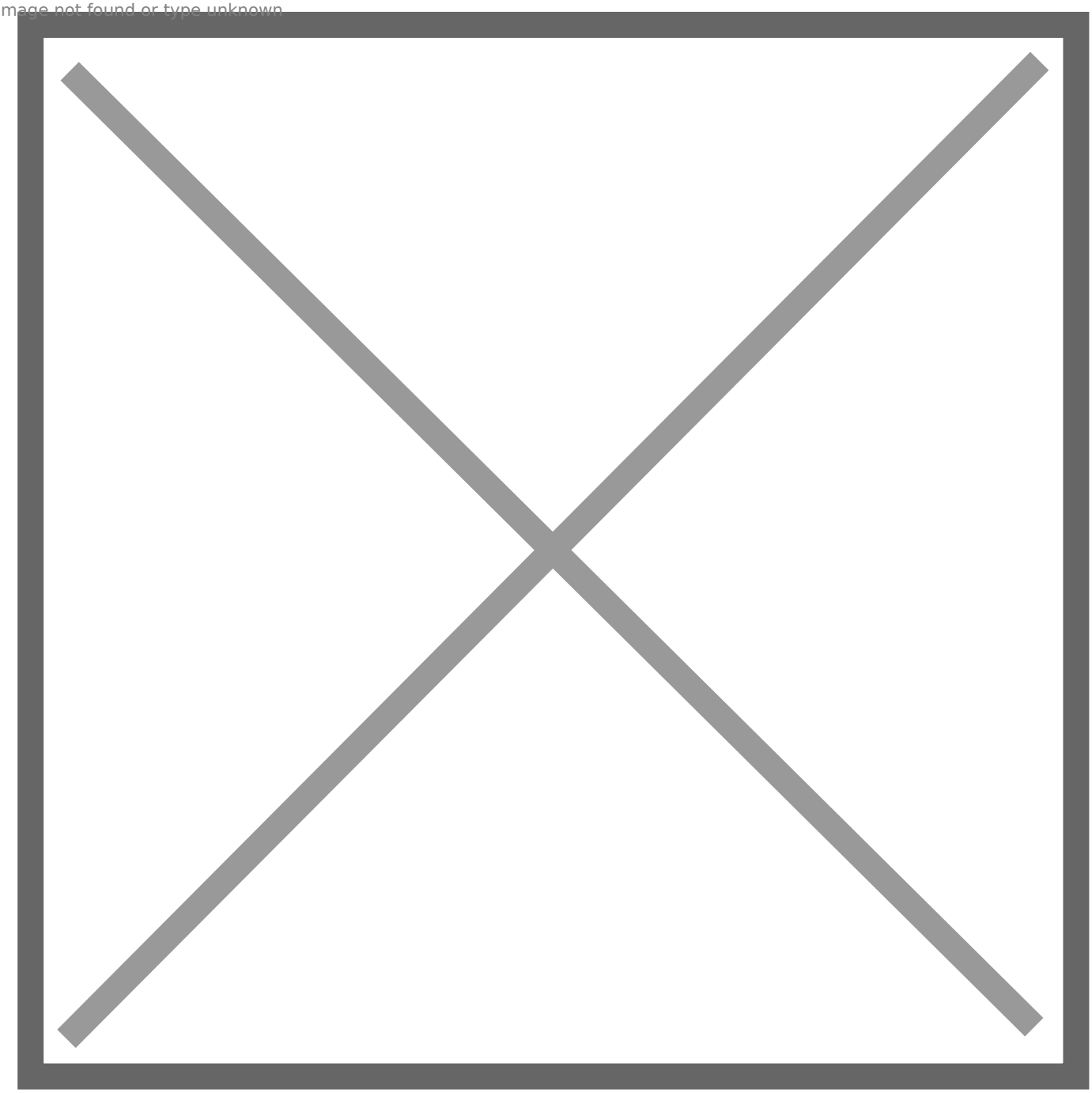
Image not found or type unknown



Welke verwerking moet er gebeuren?
De blokken moeten zo worden neergezet dat we uiteindelijk een lego kasteel hebben.

- We gaan de blokken nog niet écht neer zetten maar moeten wel ongeveer weten wat we er mee gaan doen.
  We weten dus bijvoorbeeld dat we ze niet gaan gebruiken om in het rond te gooien, maar dat we ze gaan neer zetten en op elkaar zetten.

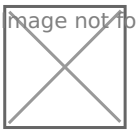
Image not found or type unknown



Kook voorbeeld

Als we bijvoorbeeld iets willen koken kunnen we in de keuken zien welke gereedschappen er zijn dat we kunnen gebruiken.

Image not found or type unknown

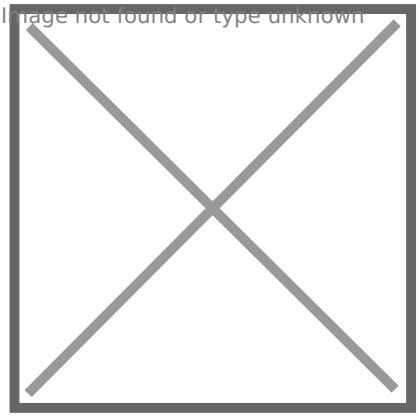


Mocht het hier blijken dat we geen deegrol hebben, dan weten we al dat het moeilijk gaat zijn om een recept op te stellen waar we een platte deeg bodem nodig hebben.

3. Algoritme opstellen

> **Wat moet je kennen en kunnen na dit deel?**
> - Weten wat een deelprobleem is
> - Zelf een simpel algoritme kunnen opstellen

Ons probleem opsplitsen in deelproblemen (decompositie)



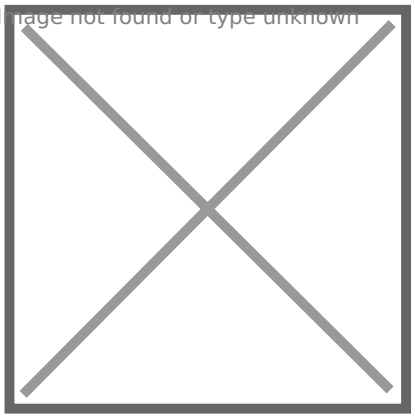
Als je probleem wat groter is dan maak je niet 1 groot algoritme maar deel je het probleem op in kleinere stukken. Deze noemen deelproblemen.
Op deze manier heb je meerder kleinere problemen die apart makkelijker op te lossen zijn dan een groot probleem in één keer.

Als je terugdenkt aan de video die in de inleiding staat (waar ik noedels maak).
Dan had ik dit algoritme ontworpen:

1. Ik zoek eten
2. Ik bereid het eten
3. Ik eet het eten op

Maar stap 2 is veel te vaag beschreven want ik wist niet hoe ik de noedels moest bereiden.
Stap 2 is dus een deelprobleem waarvoor dus een stappenplan moet worden gemaakt.

Een stappenplan (algoritme) opstellen voor elk deelprobleem



Je gaat nu voor elk deelprobleem stap voor stap uitleggen hoe je dit deelprobleem kan oplossen.
Voor stap 2 in het noedelprobleem waren dit de stappen:

1. Doe 200ml water in een pannetje
2. Doe de kruiden erbij
3. Wacht tot het water kookt
4. Voeg dan de noedels toe
5. Wacht tot het water is weggekookt

Nu weten we exact hoe we de noedels moeten bereiden.
Het volledige algoritme voor de video is dus dit:

1. Ik zoek eten
2. **Ik bereid het eten**:
    1. Doe 200ml water in een pannetje
    2. Doe de kruiden erbij
    3. Wacht tot het water kookt
    4. Voeg dan de noedels toe
    5. Wacht tot het water is weggekookt
3. Ik eet het eten op

Deelproblemen in deelproblemen

Het kan zijn dat in een deelprobleem nog andere deelproblemen zitten (En daarin nog eens enz...). Hiervoor moet dan ook een algoritme worden opgesteld.

Stel dat ik niet wist hoe ik 200 ml water in het pannetje moest doen, dan had dit het volledige algoritme kunnen zijn:

1. Ik zoek eten
2. Ik bereid het eten:
    1. **Doe 200ml water in een pannetje**:
        1. Zoek een pannetje en een maatbeker
        2. Doe water in de maatbeker tot het water even hoog staat als de 200 ml lijn.
        3. Doe dit water in het pannetje
    2. Doe de kruiden erbij
    3. Wacht tot het water kookt
    4. Voeg dan de noedels toe
    5. Wacht tot het water is weggekookt
3. Ik eet het eten op

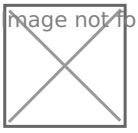
Lego voorbeeld

Deelproblemen: We gaan ons lego kasteel niet in 1 keer helemaal bouwen, maar we bouwen *onderdeel per onderdeel*.
Een deelprobleem in ons kasteel zou kunnen zijn:

- Hoe bouwen we een torentje?
- Of hoe bouwen we de poort?

Dit zijn kleinere problemen binnen ons hoofdprobleem.

Stappenplan:
Om bijvoorbeeld een torentje te bouwen zou dit ons algoritme kunnen zijn.



4. Programma schrijven

> **Wat moet je kennen en kunnen na dit deel?**
> - Weten dat een programma een vertaling is van een algoritme voor een computer.
> - Weten dat deze vertaling gebeurt aan de hand van een programmeertaal of computertaal.

Programma schrijven

Nadat je hebt bepaald welke stappen er moeten worden uitgevoerd om van je probleem tot je doel te komen kan je dit automatiseren door een computer het algoritme te laten uitvoeren.

Maar zoals gezegd in het deel “Wat is een programma?” kunnen we niet gewoon ons algoritme aan de computer geven.
Het algoritme moet eerst worden omgezet in een taal die de computer verstaat. Een programmeertaal, of *code*.

Met deze code beschrijven we elke stap in het algoritme.

Een algoritme is dus zo neergeschreven dat mensen het kunnen verstaan, en een programma is datzelfde algoritme vertaald naar een computertaal zodat een computer dat kan verstaan.

Lego voorbeeld

Voor deze robot is een speciaal programma geschreven gebaseerd op een algoritme om een lego torentje te bouwen.

Voorbeeld algoritme & code

Een voorbeeld van het algoritme en de code zou dit kunnen zijn:

Image not found or type unknown

> **Opgelet!** Dit is geen echt algoritme en geen echte code, het is maar een voorbeeld om aan te tonen hoe verschillend deze 2 er kunnen uitzien.
> Het programma is geschreven in de programmeertaal Python.

5. Testen en documenteren

> **Wat moet je kennen en kunnen na dit deel?**
> - Weten waarom je test
> - Weten waarom je documenteert

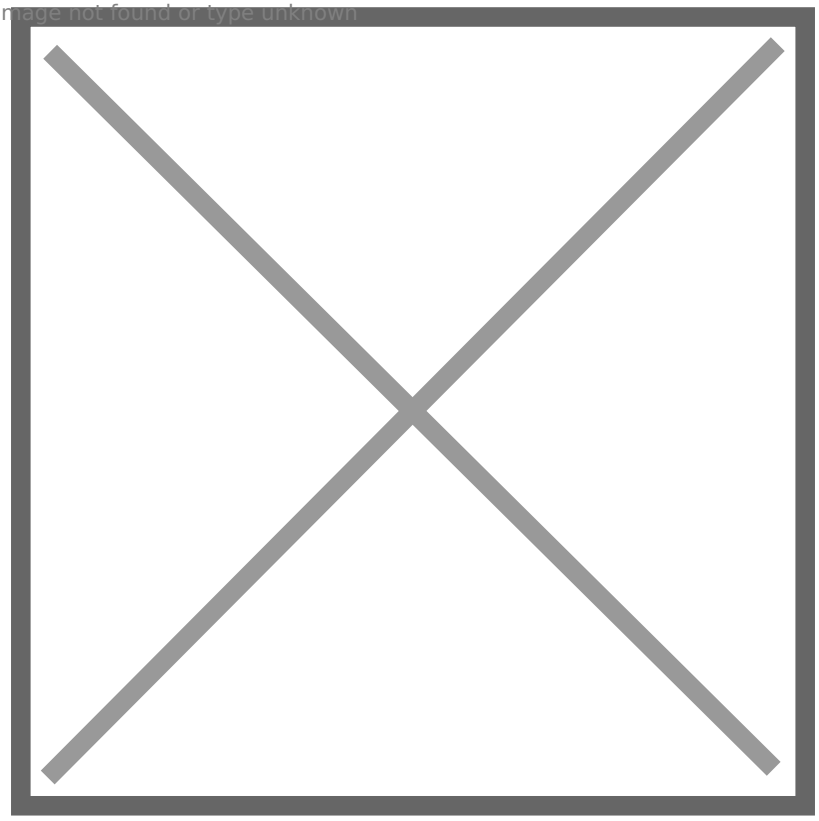
Testen

Nadat we ons programma hebben geschreven moeten we testen of het werkt.

Lego voorbeeld

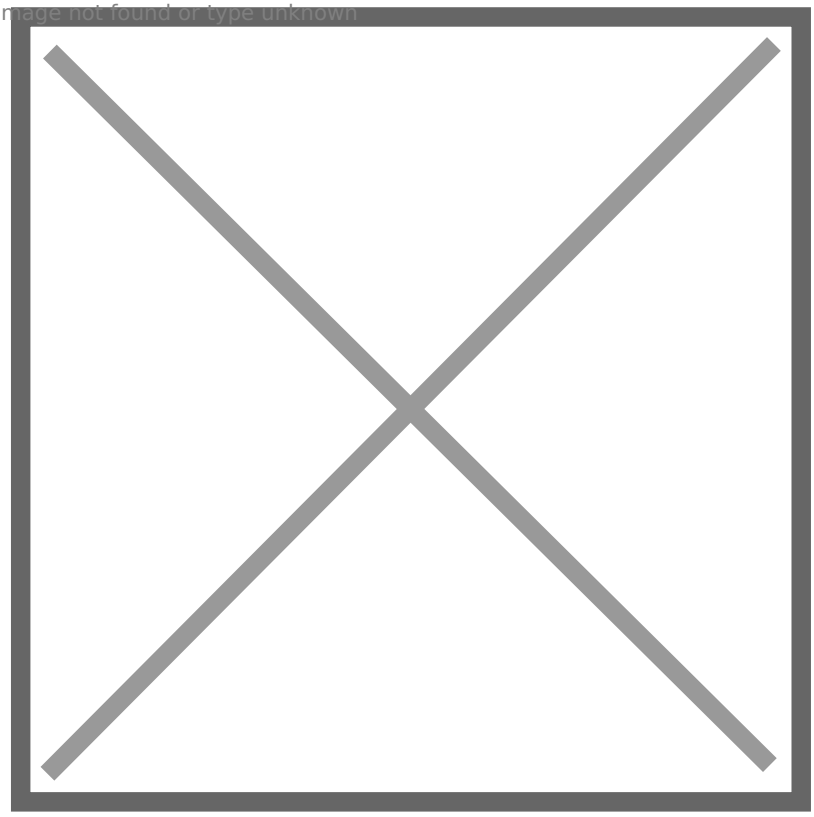
Werkt ons programma ook als we blauwe blokken gebruiken in plaats van rode? En met gele?

Image not found or type unknown



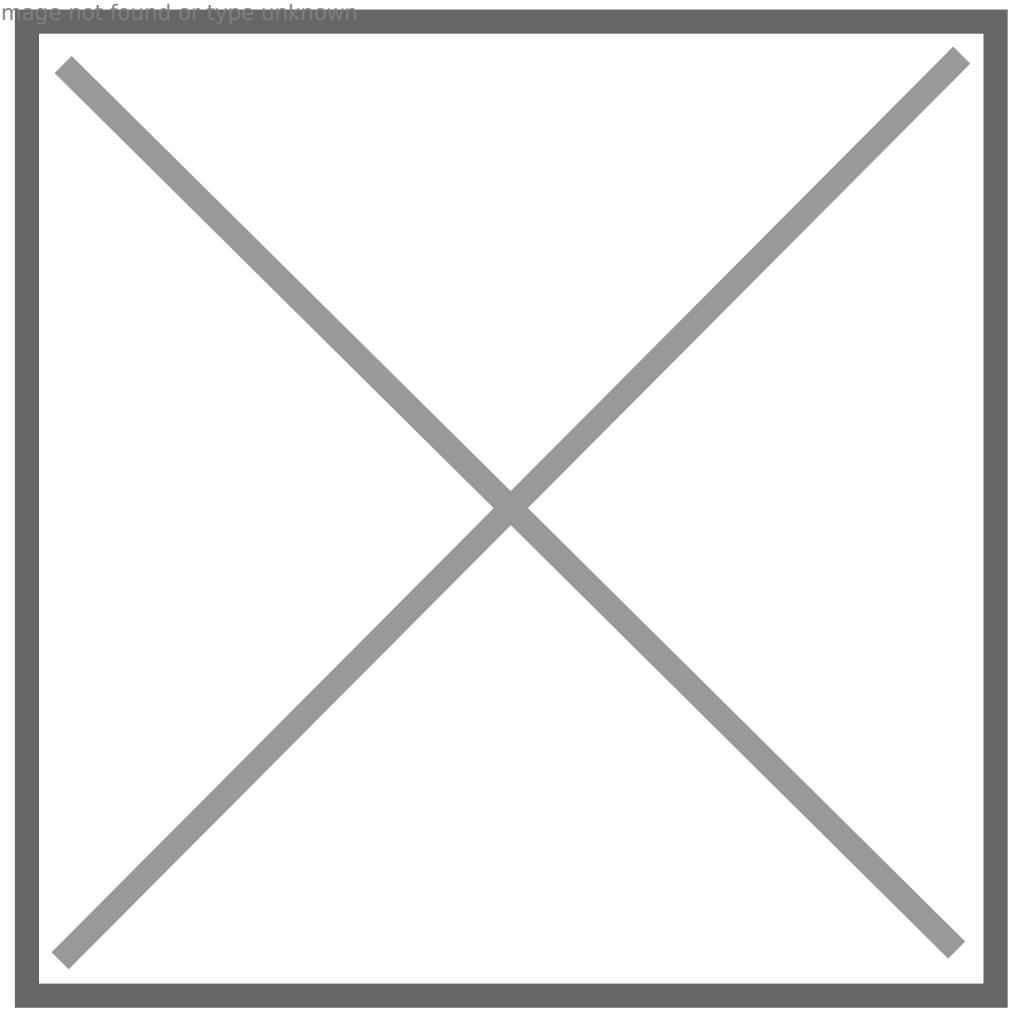
Bouwt ons programma wel een toren? Of bouwt het iets anders?

Image not found or type unknown



Kan onze robot de blokjes nog vinden als ze op deze tafel liggen?

Image not found or type unknown



Documenteren

Na het schrijven van een programma is het belanglijk dat andere mensen je code kunnen begrijpen. Daarom kan het handig zijn om documentatie schrijven die verklaart wat je code doet. Één manier om dit te doen is om commentaar te schrijven in je code.

In de code hieronder is de tekst die achter de 2 schuine strepen staat commentaar. (De tekst die in het grijs staat)
Zelfs zonder te kunnen programmeren weet je nu ongeveer wat de code doet.

```
class MainClass {
  public static void Main (string[] args) {
    var blokken = new List<LegoBlokje>();

    // Hier maak ik 10 nieuwe lego blokjes aan en steek ze in een lijst die
    "blokken" noemt.
    foreach (int i in Enumerable.Range(1, 10)) {
      var nieuwBlokje = new LegoBlokje();
      blokken.Add(nieuwBlokje);
    }
  }
}
```

Iedereen is vergeetachtig

Documenteren is ook handig voor jezelf!
Het gebeurt vaak dat als je een programma hebt geschreven, en je moet er een paar maanden later iets aan veranderen dat je helemaal geen idee meer hebt hoe je code ineen steekt!