



STORIES PROJECTWEEK 2 VAN PROJECT 1

Gamesuite: Hangman

1 Introductie

Tijdens de tweede projectweek van het opleidingsonderdeel Project 1 (MBI30X) zullen jullie in een team van 6 personen en met een opgegeven gedetailleerde “handleiding”, komen tot een mini game-suite met daarin een spelletje namelijk Hangman. Alle concepten uit de theorie van het opleidingsonderdeel OO-programmeren(MBI53X) zullen hier aan bod komen: overerving, abstracte klassen, interfaces, testklassen, ...

1.1 Agile en Pair-Programming

In 2001 werd een manifest voor agile softwareontwikkeling opgesteld. Dit manifest bevat een aantal richtlijnen ter bevordering van het ontwikkelen van “goede” software. Details met betrekking tot deze richtlijnen kan je lezen op <http://nl.wikipedia.org/wiki/Agile-softwareontwikkeling>

De hoofdkenmerken van Agile software-ontwikkeling zijn iteratie, communicatie, voortgang en werkende software.

1.1.1 Iteratie

Het **iteratie**-luik, waarbij men stelt dat goede code genereren een iteratief-proces is, zullen we realiseren door te werken met “stories”. Een story is een deelverhaal, een klein onderdeel van het gevraagde project dat tot een werkend mini-eindresultaat aanleiding geeft.

1.1.2 Communicatie

Het **communicatie**-luik (veel praten zodat documentatie zo goed als overbodig wordt) zullen we bewerkstelligen door aan pair-programming te doen. Dit houdt in dat alle softwareontwikkeling per twee gebeurt.

Eén van beide partners heeft de rol van story-owner. Telkens wanneer een story (een klein, individueel werkend stukje programma) af is, wordt de andere partner story-owner.

Eén van beide partners is bij aanvang de “doener”, hij/zij heeft controle over het toetsenbord. De andere partner is de “denker”, hij/zij denkt mee en zegt telkens wat de volgende stap in het programmeren moet zijn. Deze tweede rol-verdeling wisselt na een vaste tijd (af te spreken, en afhankelijk van het programmeer-niveau van beide partners), dit gebeurt sowieso, ook wanneer de huidige story nog niet af is.

Naast het wisselen van rol binnen de partner-paren is het ook belangrijk om af en toe gewoon van partner te wisselen. Zorg dat je iedere sessie met een andere partner in je team werkt.

- Student 1 – student 2; student 3 – student 4; student 5 – student 6
- Student 1 – student 3; student 2 – student 5; student 4 – student 6
- Student 1 – student 4; student 2 – student 6; student 3 – student 5
- ...

1.1.3 Voortgang

Voortgang wordt gemeten aan de hand van werkende software. Na elke story is er een nieuw stukje functionaliteit toegevoegd aan de applicatie. Deze functionaliteit kan dan door de klant uitgetest worden terwijl men ondertussen verder werkt aan de volgende story. Men mag dus niet aan een volgende story beginnen vooraleer de vorige story volledig werkt.

Aangezien we in groepen van 6 studenten werken, en binnen die groepen met teams van 2, zal er binnen 1 groep telkens aan 3 stories tegelijk gewerkt worden. Er werd zoveel mogelijk geprobeerd om opeenvolgende stories onafhankelijk te maken van elkaar, zodat er geen conflicten kunnen ontstaan. Hoe verder we in het proces komen, hoe minder deze manier van werken mogelijk is. Daar zal communicatie weer enorm belangrijk worden, zodat alle paren in een team weten wat ze van elkaar kunnen verwachten.

1.1.4 Werkende software

Werkende software slaat op het feit dat elke tussenliggende versie effectief werkend moeten zijn. Het is de verantwoordelijkheid van alle “doeners” om alleen werkende code te committen, zodat groepsgenoten niet vastlopen op “foutieve” code van het andere team. Code is pas juist wanneer ieder onderdeel van die code werd afgetoetst, m.b.v. unit testing.

1.2 Groepswerk

Jullie zullen in **groepen van 6 studenten** ingedeeld worden. Je blijft gedurende de hele projectweek deel uitmaken van dezelfde groep. Jullie moeten in team de taken verdelen en afwerken. Om dit mogelijk te maken zullen we gebruik maken van versiebeheer.

We starten iedere sessie (iedere dag 1 keer voormiddag 1 keer namiddag) met een “**standup meeting**” per groep. De groepsleden staan recht en praten met elkaar over de stand van zaken, wat heeft iedereen gedaan, geleerd, ... waar zaten de problemen, wie kan wie helpen, ... hoe zullen de paren tijdens deze sessie samengesteld worden?

Na het introgesprek ga je binnen je team gaan per twee studenten aan 1 PC zitten. Je beslist wie de doener zal zijn gedurende de eerste X minuten. Deze doener logt in op het systeem. Er zijn dus in totaal 3 pc's in gebruik tijdens een sessie.

1.3 Versiebeheer

Bij de start van ons verhaal is er een repository voorzien met code om van te vertrekken. Op basis van deze code zal je een Git repository aanmaken. Ieder duo binnen eenzelfde groep werkt op diezelfde repository. Tijdens het werken zal men regelmatig code op de Git repository moeten posten, alsook code van de Git repository moeten afhalen. Samen werken op dezelfde broncode kan aanleiding geven tot “conflicterende” code, ook dit zullen we leren oplossen tijdens deze sessies.

1.4 Schrijven van JUnit-testen

Je zal vanaf story 3 ook zelf testklassen moeten schrijven. Denk hierbij aan het testen van alle public functionaliteiten. Voor elke public methode moet je minstens 1 testcase voorzien voor standaardafhandeling en voor elke uitzonderingssituatie en de randvoorwaarden nog een aparte testcase. Gebruik voor elke testcase de naamgeving zoals geleerd tijdens de lessen OO-programmeren (zie hiervoor Toledo → OO-programmeren → Lesweek 1).

2 GameSuite – aanpak

De GameSuite die we willen maken zal het spelletje Hangman bevatten.

De speler moet een woord/zin raden. Dat is tekst input waar een punt mee verdiend kan worden. In het geval van “Hangman” zal er dan een tekening opgebouwd worden van een “opgehangen mannetje” telkens de gebruiker een verkeerde letter raadt.

2.1 Speler

We zullen beginnen met het concept speler.

2.2 Tekening

Daarna zullen we, in een heel aantal stappen, komen tot het maken van een tekening. Voor het tekenen zullen we het Graphics element van java.awt gebruiken. Documentatie kan je vinden op: <http://docs.oracle.com/javase/8/docs/api/java.awt/Graphics.html>. In eerste instantie willen we enkel een tekening kunnen maken, zonder het concept van een spelletje erbij. We zullen werken met punten, lijnen, vierkanten, driehoeken en cirkels. Tegen het einde van story 10 zullen jullie een programma hebben dat een huisje tekent:



Figuur 1. EINDRESULTAAT STORY 10 – TEKENING VAN EEN HUISJE

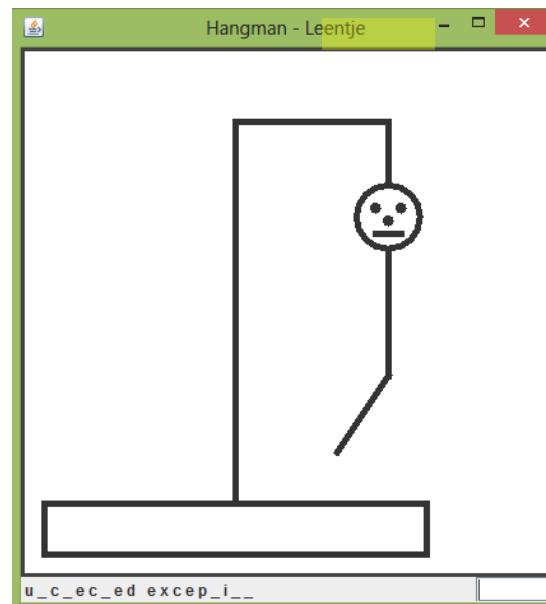
2.3 Hangman

Eens de tekening af is, kunnen we werken aan het spelletje uit onze GameSuite, namelijk Hangman. In deze HangMan versie kan je als speler je naam opgeven. Vervolgens kan je een woord raden door telkens een letter in te typen, gevolgd door “enter”. Als je letter juist was, dan wordt deze vervangen in de hint. Als je letter fout was, dan krijgt je mannetje een extra lichaamsdeel. Doel is om het woord (of het zinnetje) te raden voor het mannetje volledig is (max 14 kansen).

Het spel eindigt wanneer je het woord correct geraden hebt, of wanneer je mannetje opgehangen is. Je krijgt de mogelijkheid om opnieuw te spelen. De woorden voor het spelletje staan in een tekst-bestand: hangman.txt



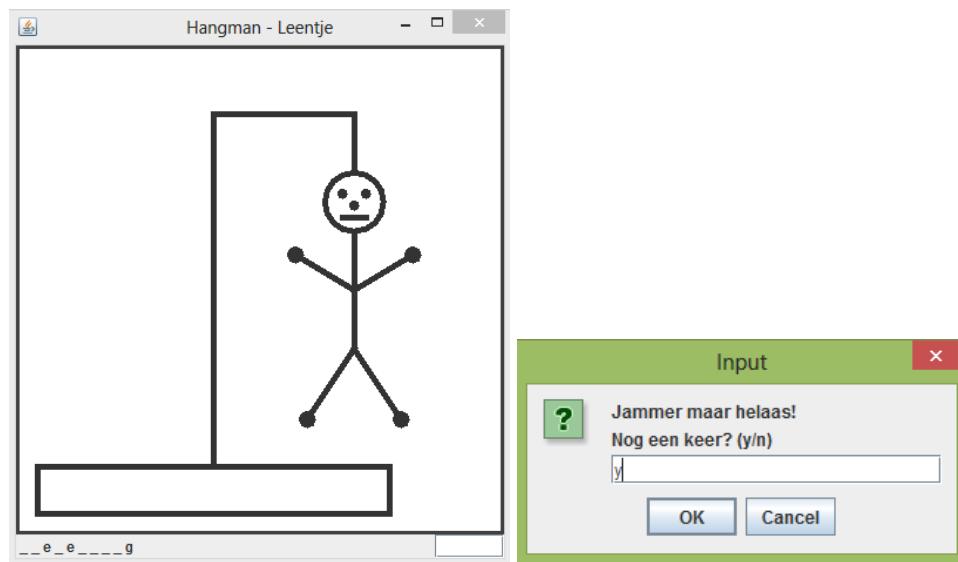
Figuur 2. EINDRESULTAAT NA STORY 15: SPELER MOET ZIJN NAAM OPGEVEN



Figuur 3. EINDRESULTAAT NA STORY 15: HET SPEL WORDT GESPEELD (LETTER+ENTER)



Figuur 4. EINDRESULTAAT NA STORY 15: GEWONNEN, NOG EEN KEER?



Figuur 5. EINDRESULTAAT NA STORY 15: MANNETJE HANGT --> VERLOREN, NOG EEN KEER?

3 Stories

We willen jullie laten proeven van alle nieuwe concepten. Daarom zullen de eerste stories op het vlak van programmeren weinig uitdagend zijn. Jullie moeten eerst vertrouwd geraken met pair-programming en de rolverdeling daarbinnen, committen en updaten Na verloop van tijd worden de stories groter en moeilijker en zal je zelf testklassen schrijven. Na de afwerking van hangman zullen jullie zelf de code kunnen uitbreiden om ook pictionary te kunnen spelen. Hiertoe zullen jullie zelfs zelf stories moeten schrijven en klassendiagramma's moeten tekenen.

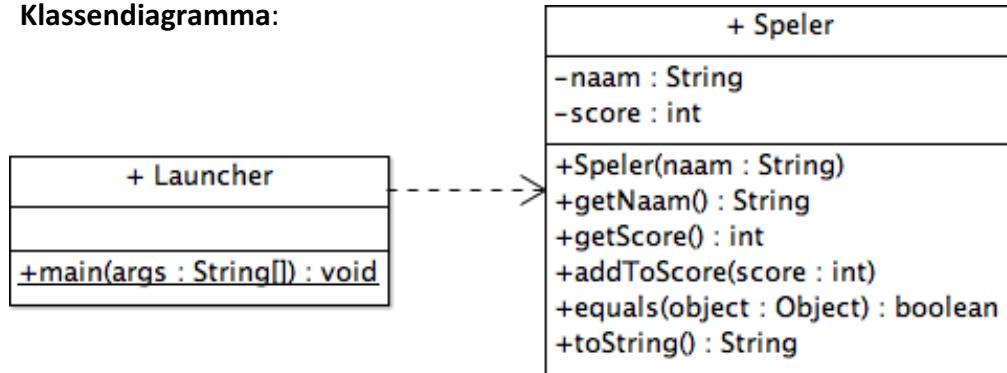
3.1 Story 01 – Speler

Als speler
kan ik mijn naam ingeven
zodat deze samen met mijn score getoond wordt.

Je kan een **speler** aanmaken met een **naam** en een default **score** 0. De naam is geldig wanneer hij niet null en niet leeg ("") is. Je kan de score van een speler veranderen door een bepaalde score toe te voegen. De resulterende score moet echter te allen tijde ≥ 0 blijven. Twee spelers zijn aan elkaar gelijk wanneer ze dezelfde naam en score hebben. Je kan de speler info omzetten naar een String-vorm.

Gegeven:

- **SpelerTest** – een volledig uitgeschreven test, die volledig moet compileren en groen kleuren aan het einde van deze story.
- **Klassendiagramma:**

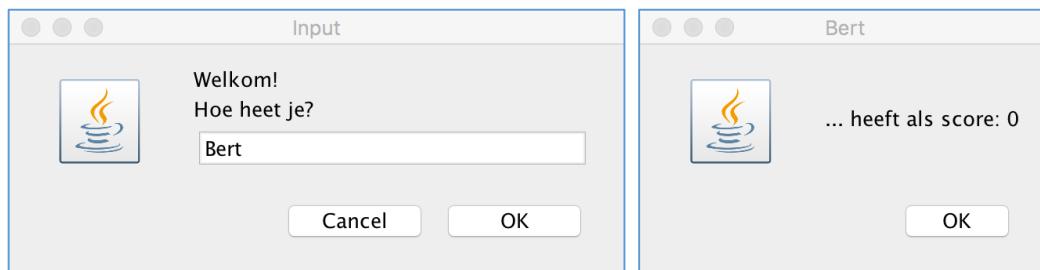


Figuur 6. KLASSEDIAGRAMMA STORY 1

Gevraagd:

- De klasse **Speler**, op zo'n manier dat **SpelerTest** slaagt.
- De klasse **Launcher** die onderstaande uitvoer geeft.

Output bij het runnen van de launcher (wanneer je het voorstel volgt):



Figuur 7. OUTPUT STORY 1

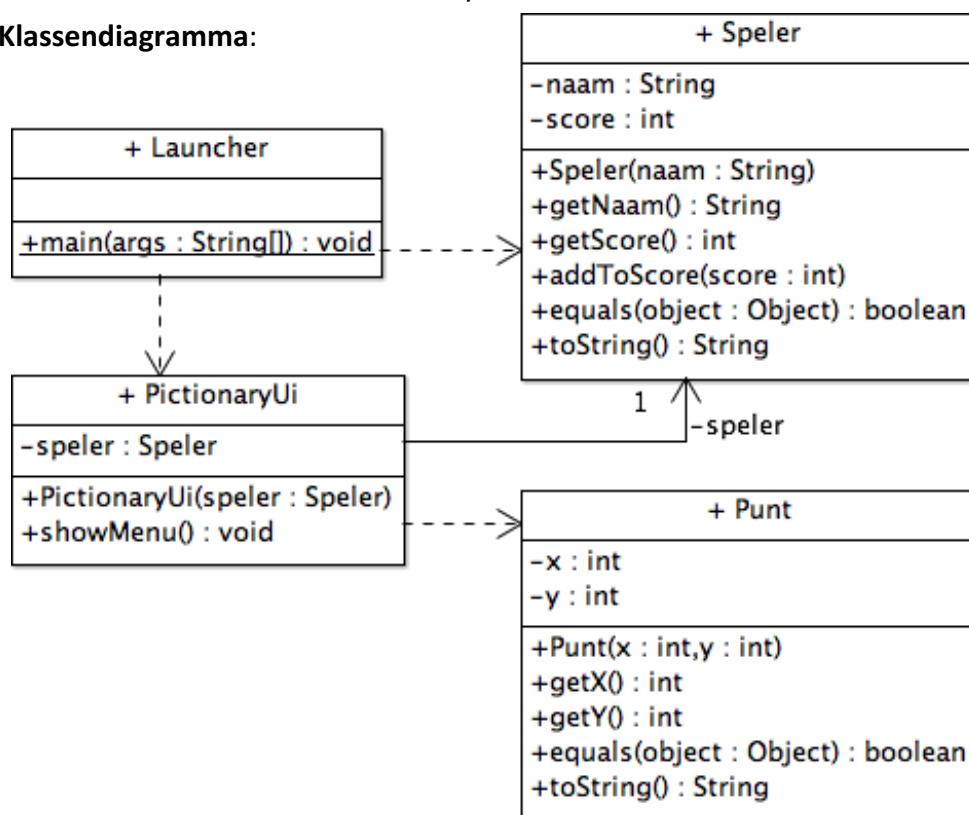
3.2 Story 02 – Punt

Als speler
kan ik een punt aanmaken
zodat dit later aan de tekening toegevoegd kan worden.

*Je kan een **punt** aanmaken met gehele getallen als coördinaten x en y. Er zijn geen randvoorwaarden voor de coördinaten. Twee punten zijn aan elkaar gelijk wanneer ze dezelfde x en y waarde hebben. Een punt wordt omgezet naar String-vorm als: (200, 200)*

Gegeven:

- **PuntTest** – een volledig uitgeschreven test, die volledig moet compileren en groen kleuren aan het einde van deze story.
- **Klassendiagramma:**



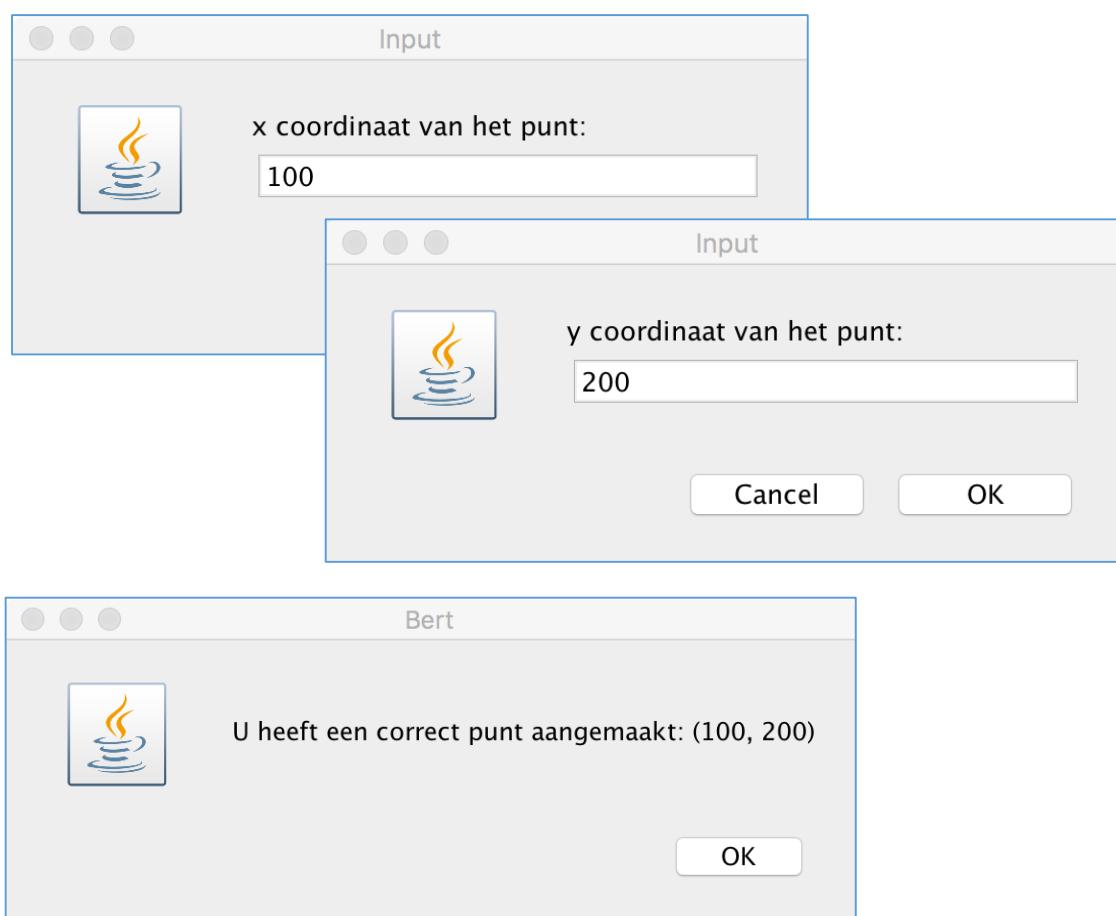
Figuur 8. KLASSENDIAGRAMMA STORY 2

Gevraagd:

- De klasse **Punt**, op zo'n manier dat PuntTest slaagt.
- De klasse **Ui**. Deze:
 - vraagt een x-coördinaat,
 - vraagt een y-coördinaat
 - maakt een punt aan en toont dit aan de gebruiker
- De aangepaste klasse **Launcher**: nadat je de Speler aangemaakt hebt toon je de Ui.

Output bij het runnen van de launcher

...



Figuur 9. OUTPUT STORY 2

3.3 Story 03 - Cirkel

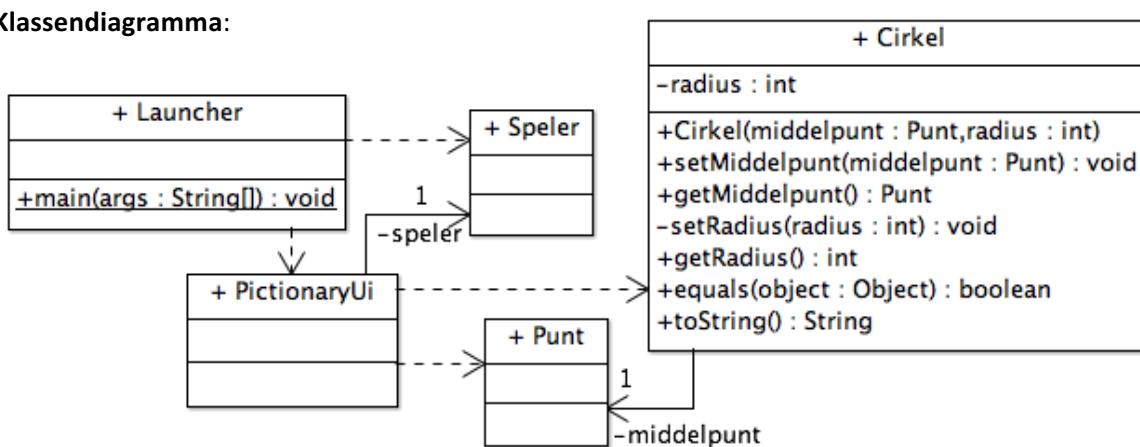
Als speler
kan ik een cirkel aanmaken
zodat dit later aan de tekening toegevoegd kan worden.

Je kan een **Cirkel** maken met een middelpunt (object van de klasse *Punt*) en een radius. De voorwaarde voor een geldige cirkel is dat het middelpunt niet null is en de radius **strikt** positief is. Twee cirkels zijn aan elkaar gelijk wanneer zowel hun middelpunt als hun straal gelijk zijn. Een cirkel met middelpunt (200,200) en straal 20 wordt in String-vorm gezet als:

Cirkel: middelpunt: (200, 200) – straal: 20

Gegeven:

Klassendiagramma:



Figuur 10.KLASSENDIAGRAMMA STORY 3

Gevraagd:

Cirkeltest: een volledig uitgeschreven test, die volledig moet compileren en groen kleuren aan het einde van deze story (voeg toe aan AllTests). Zorg dat het volgende getest wordt:

- *Ik kan een cirkel aanmaken met een geldig middelpunt en een geldige straal*
- *Ik krijg een exception wanneer ik een cirkel wil aanmaken met middelpunt = null*
- *Ik krijg een exception wanneer ik een cirkel wil aanmaken met straal < 0*
- *Ik krijg een exception wanneer ik een cirkel wil aanmaken met straal = 0*
- *Twee cirkels zijn gelijk wanneer ze hetzelfde middelpunt en dezelfde straal hebben*
- *Twee cirkels zijn verschillend wanneer de tweede cirkel null is*
- *Twee cirkels zijn verschillend wanneer hun middelpunt verschillend is*
- *Twee cirkels zijn verschillend wanneer hun straal verschillend is*

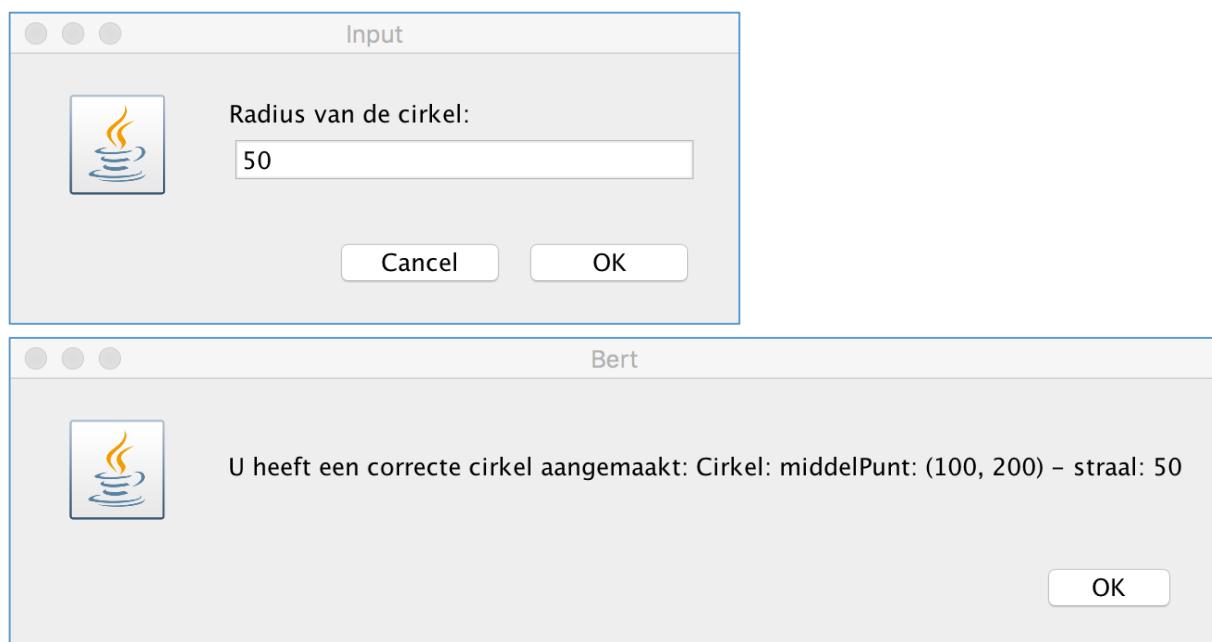
De klasse **Cirkel**, op zo'n manier dat CirkelTest slaagt.

De aangepaste klasse **Ui**. Deze:

- vraagt nadat het punt aangemaakt is ook de straal aan de gebruiker
- toont nu de cirkel (in tekstvorm) i.p.v. enkel het punt
- zorgt ervoor dat de toepassing niet crasht als er verkeerde gegevens ingegeven worden. Kies zelf hoe ver je gaat met de gebruiksvriendelijkheid, maar zorg er minstens voor dat eventuele exceptions opgevangen worden en de foutbericht getoond.

Output bij het runnen van de launcher

...



Figuur 11. OUTPUT STORY 3

3.4 Story 04 - Rechthoek

Als speler
kan ik een rechthoek aanmaken
zodat dit later aan de tekening toegevoegd kan worden.

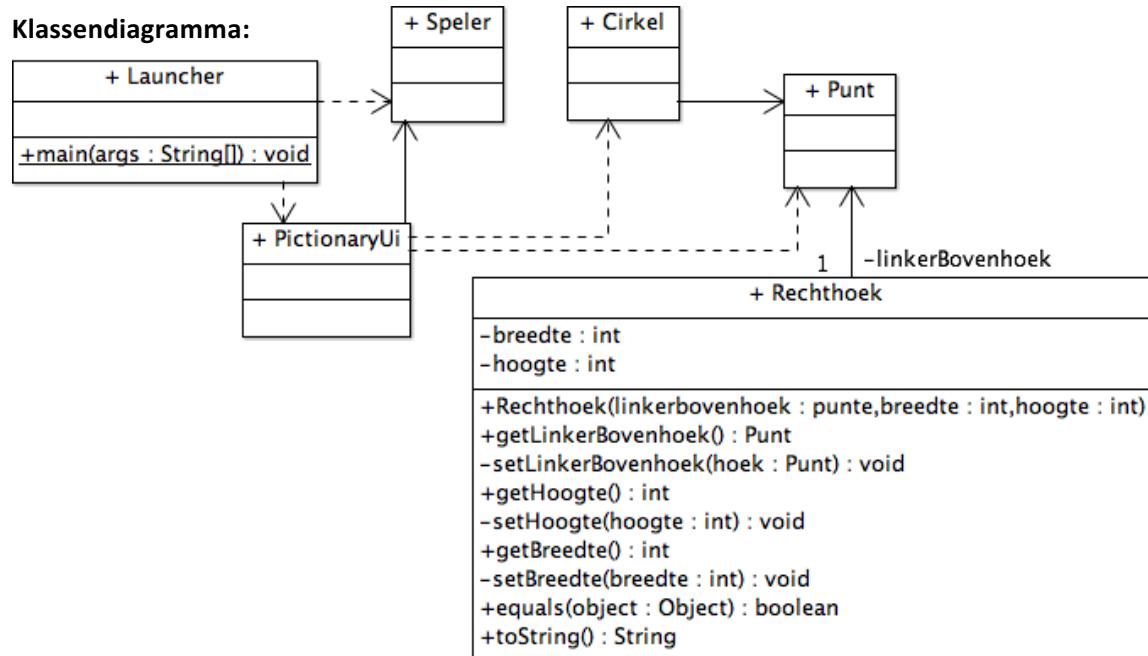
Je kan een **Rechthoek** maken met een positie (niet null), een breedte(>0) en een hoogte(>0). Twee rechthoeken zijn aan elkaar gelijk wanneer zowel hun positie als hun breedte en hoogte gelijk zijn. Een rechthoek met positie (100,200), breedte 200 en hoogte 180 wordt in String-vorm omgezet als volgt:

Rechthoek: positie: (100, 200) – breedte: 200 – hoogte: 180

Gegeven:

RechthoekTest – een volledig uitgeschreven test, die volledig moet compileren en groen kleuren aan het einde van deze story.

Klassendiagramma:



Figuur 12. KLASSENDIAGRAMMA STORY 4

Gevraagd:

De klasse **Rechthoek**, op zo'n manier dat RechthoekTest slaagt.

De aangepaste klasse **Ui**. Deze:

- vraagt nadat de speler is aangemaakt eerst wat de speler wilt aanmaken: een cirkel of een rechthoek

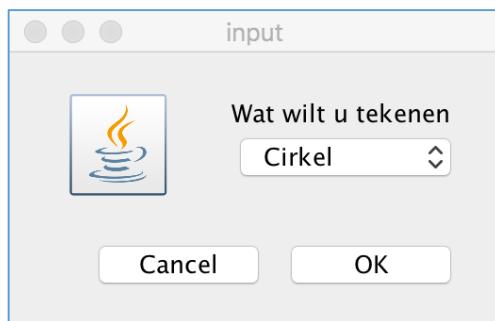
Tip: om een dropdown te krijgen in je JOptionPane-venster:

```
Object[] shapes = {"Cirkel", "Rechthoek"}  
Object keuze = JOptionPane.showInputDialog(null, "Wat wilt u tekenen",  
"input", JOptionPane.INFORMATION_MESSAGE, null, shapes, null);  
...
```

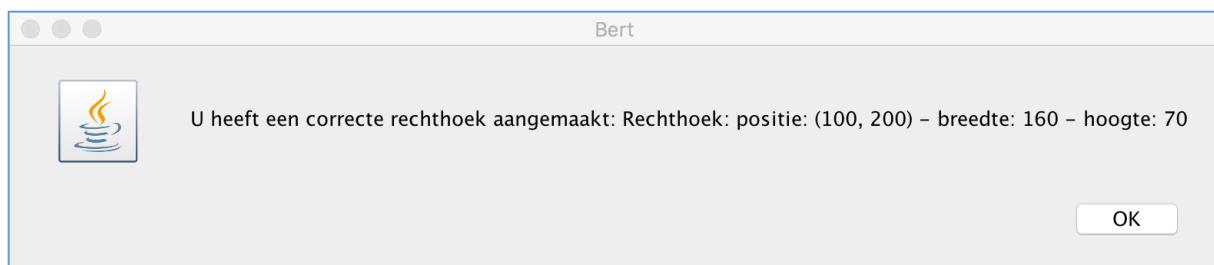
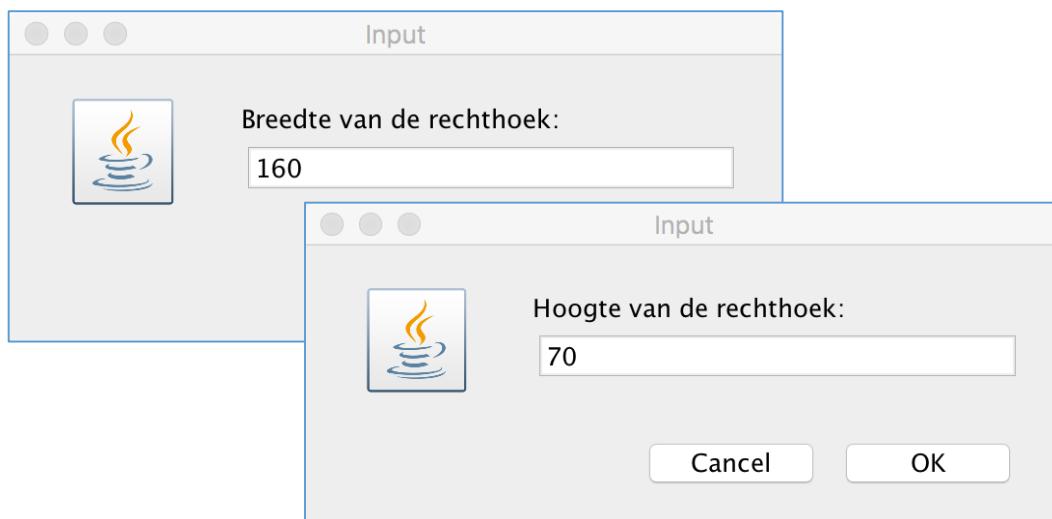
- vraagt afhankelijk van de keuze van de gebruiker de nodig input om een cirkel of een rechthoek aan te kunnen maken
- toont de aangemaakte cirkel of rechthoek (in tekstvorm)
- zorgt ervoor dat de toepassing niet crasht als er verkeerde gegevens ingegeven worden. Kies zelf hoe ver je gaat met de gebruiksvriendelijkheid, maar zorg er minstens voor dat eventuele exceptions opgevangen worden en de foutbericht getoond.

Output bij het runnen van de launcher

...



...



Figuur 13.OUTPUT STORY 4

3.5 Story 05 – Lijnstuk

Als speler
kan ik een lijnstuk aanmaken
zodat dit later aan de tekening toegevoegd kan worden.

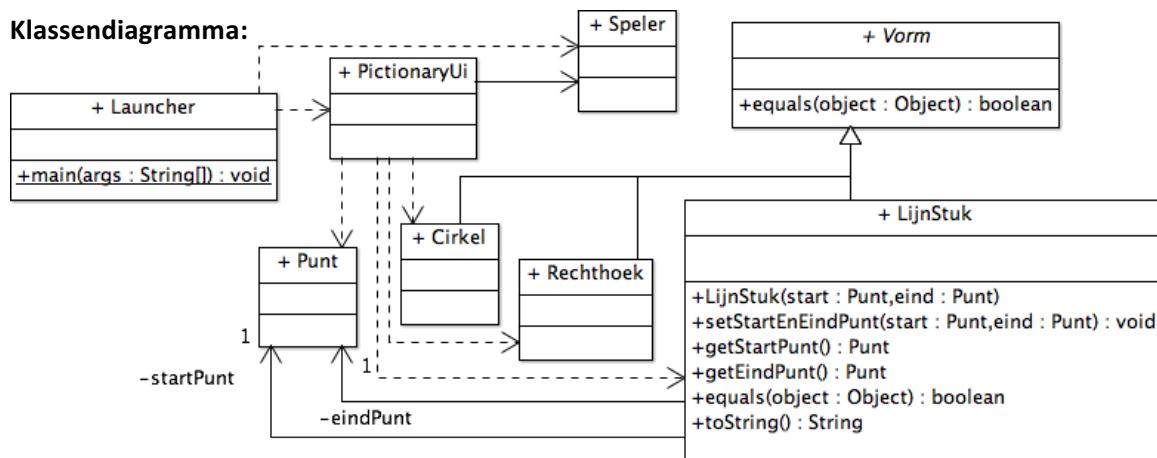
Je kan een **LijnStuk** maken met een `startPunt` (niet null) en een `eindPunt`(niet null). Een extra voorwaarde bij het aanmaken van een lijnstuk is dat `startPunt` en `eindPunt` niet aan elkaar gelijk mogen zijn. Beide punten moeten aangepast kunnen worden. **Pas op:** omdat deze voorwaarde alleen kan afgetoetst worden wanneer beide parameters beschikbaar zijn, heeft deze klasse geen klassieke setters per instantiatievariabele. Er is slechts 1 setter die de twee parameters mee krijgt. Twee lijnstukken zijn aan elkaar gelijk wanneer beide eindpunten aan elkaar gelijk zijn. **Pas op:** de volgorde van de eindpunten is in deze niet relevant. Een lijnstuk met startpunt (100,150) en eindpunt (200,250) wordt omgezet in String-vorm als:

Lijn: startpunt: (100, 150) – eindpunt: (200, 250)

Gegeven:

LijnStukTest: een deels uitgeschreven test

Klassendiagramma:



Figuur 14.KLASSENDIAGRAMMA STORY 5

Gevraagd:

De aangepaste **LijnStukTest** – denk na over ontbrekende testen en voeg ze toe. De test moet volledig compileren en groen kleuren aan het einde van deze story.

De klasse **LijnStuk** en de klasse **Vorm**: met het oog op het later toevoegen van lijnstukken, rechthoeken, ... aan een Tekening maken we nu reeds de abstracte superklasse **Vorm**. Deze bevat voorlopig zo goed als niets (voorlopig zijn er geen generalisaties mogelijk, maar dat zal snel volgen). Pas ook Cirkel (story 3) en Rechthoek (story 4) aan (controleer eerst of ze correct gecommit werden!) zodat ze overerven van je nieuwe klasse Vorm. De aangepaste klasse **Ui**. Deze geeft nu ook de keuze om een lijnstuk te tekenen.

3.6 Story 06 – Driehoek (is een Vorm)

Als speler
kan ik een lijnstuk aanmaken
zodat dit later aan de tekening toegevoegd kan worden.

*Je kan een **Driehoek** maken met drie hoekpunten (niet null): hoekPunt1, hoekPunt2 en hoekPunt3. Een driekoek is een vorm. De hoekpunten moeten aangepast kunnen worden.*

Extra voorwaarden bij het aanmaken van een driehoek zijn:

- *de hoekpunten mogen niet samenvallen (twee aan twee)*
- *de hoekpunten mogen niet op 1 lijn liggen.*
Algoritmisch: drie hoekpunten: $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ liggen op 1 lijn wanneer de twee lijnen gevormd door hoekpunt 1 en 2, en door hoekpunt 1 en 3 dezelfde richting/helling hebben. Dit is het geval wanneer: $(x_2-x_1)/(y_2-y_1) = (x_3-x_1)/(y_3-y_1)$. Om ervoor te zorgen dat we geen deling door nul krijgen wanneer twee punten dezelfde y coördinaat hebben, herschrijven we deze formule: $(x_2-x_1)(y_3-y_1) = (x_3-x_1)*(y_2-y_1)$*

Pas op: omdat deze extra voorwaarden alleen kunnen afgetoest worden wanneer alle parameters beschikbaar zijn, heeft deze klasse geen klassieke setters per instantievariabele. Er is slechts 1 setter die de drie parameters meekrijgt.

Twee driehoeken zijn aan elkaar gelijk wanneer alle eindpunten aan elkaar gelijk zijn.
Pas op: de volgorde van de eindpunten is in deze niet relevant.

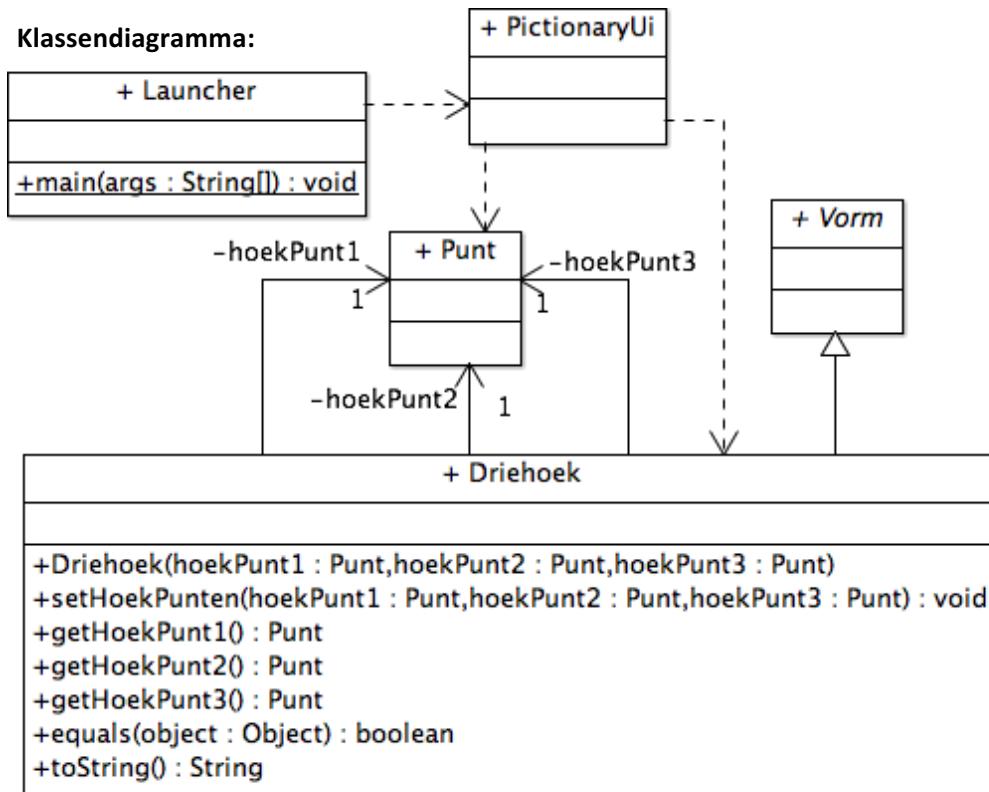
Een driehoek met hoekpunten (100,200), (300,200) en (200,100) wordt omgezet in String-vorm als:

Driehoek: hoekpunt1: (100, 200) – hoekpunt2: (300, 200) – hoekpunt3: (200, 100)

Gegeven:

DriehoekTest: een deels uitgeschreven test

Klassendiagramma:



Figuur 15.KLASSENDIAGRAMMA STORY 6

Gevraagd:

De aangepaste **DriehoekTest** – denk na over ontbrekende testen en voeg ze toe. De test moet volledig compileren en groen kleuren aan het einde van deze story (voeg toe aan AllTests).

De klasse **Driehoek** die van **Vorm** moet overerven. Controleer dat **Vorm** correct gecommit werd.

De aangepaste klasse **Ui**. Deze geeft nu ook de keuze om een driehoek te tekenen. Vergeet niet dat eventuele exceptions opgevangen moeten worden en de foutbericht getoond.

OP HET EINDE VAN DEZE STORY MOETEN CIRKEL, RECHTHOEK, LIJNSTUK EN DRIEHOEK OVERERVEN VAN DE ABSTRACTE KLASSE VORM. PAS WANNEER AAN DIE VOORWAARDE VOLDAAN IS MAG JE DEZE STORY ALS “AF” BESCHOUWEN.

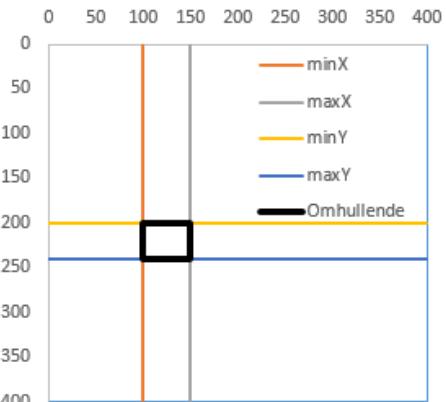
3.7 Story 07 – Omhullende van een rechthoek

Als speler

kan ik de omhullende vragen van een rechthoek
zodat ik later kan controleren of hij in een tekening past.

*Met het oog op het toekomstige tekenen van verschillende vormen in een TekenVenster hebben we nood aan een klasse **Omhullende**, die de minimale ruimte zal beschrijven die nodig is voor het tekenen van een welbepaalde vorm. Een Omhullende heeft een Punt positieLinksBoven (mag niet null zijn), een breedte(≥ 0) en een hoogte(≥ 0). Aan een omhullende moet je kunnen opvragen wat zijn minimale X, maximale X, minimale Y en maximale Y waarde is. Twee omhullenden zijn gelijk aan elkaar wanneer hun positie, breedte en hoogte gelijk zijn aan elkaar. Wanneer je een omhullende omzet in String-vorm met positieLinksBoven (100,200), breedte 50 en hoogte 40 krijg je:*

Tekening met omhullende



Omhullende: (100, 200) - 50 - 40

Voor iedere vorm kan er een omhullende berekend worden. In deze story gaan we dit enkel voor de rechthoek doen. Schrijf voor de Rechthoek de methode `getOmhullende()`, die de omhullende voor de rechthoek terug geeft:

Rechthoek: positie: (100, 200) - breedte: 200 - hoogte: 180
 Omhullende: (100, 200) - 200 - 180

Gegeven: **OmhullendeTest** – een deels uitgeschreven test.

Gevraagd:

Een **klassendiagramma** dat toont wat er aan de bestaande klassen toegevoegd/aangepast worden.

De aangepaste **OmhullendeTest**: voeg testen toe voor de methodes *getMinimumX()*, *getMinimumY()*, *getMaximumX()* en *getMaximumY()*.

De klasse **Omhullende**.

De aangepaste **RechthoekTest**: voeg een test toe die controleert of de berekende omhullende (*getOmhullende()*) van een rechthoek gelijk is aan de omhullende die je zou verwachten. Voor een rechthoek is dit eenvoudig: de omhullende heeft dezelfde positie, hoogte en breedte als de rechthoek zelf.

De aangepaste **Rechthoek**-klasse: deze krijgt een methode *getOmhullende()*. Bij het omzetten naar een string-vorm van de rechthoek wordt ook de omhullende getoond:

```
Rechthoek: positie: (100, 200) - breedte: 200 - hoogte: 180
Omhullende: (100, 200) - 200 - 180
```

3.8 Story 08 – Tekening

Als speler
 kan ik een tekening maken met vormen
 zodat mijn tegenspelers kunnen raden wat ik getekend heb.

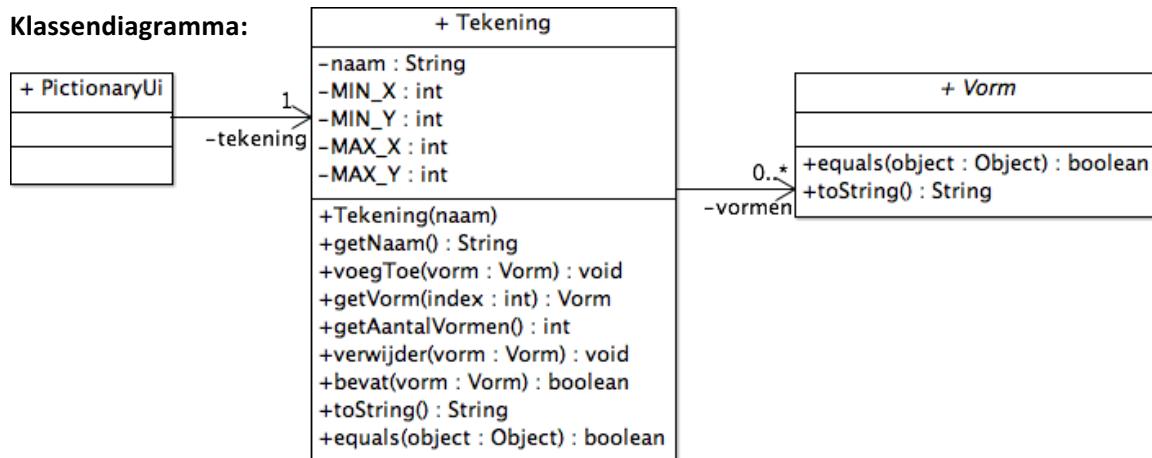
Je kan een **Tekening** maken met een naam (niet null, niet leeg ""). Bij het aanmaken van de tekening zijn er 0 vormen aanwezig. Je kan nu vormen toevoegen aan de tekening. De voorwaarden voor een vorm zijn dat die niet null mag zijn en dat die nog niet mag voorkomen in de tekening. Voorlopig zijn er geen andere voorwaarden. . De klasse **tekening** heeft constanten die de minimale en maximale afmetingen van de **Tekening** voorstellen:

```
private static final int MIN_X = 0;      private static final int MIN_Y = 0;
private static final int MAX_X = 399;     private static final int MAX_Y = 399;
```

Gegeven:

TekeningTest – een deels uitgeschreven test

Klassendiagramma:



Figuur 16.KLASSENDIAGRAMMA STORY 8

Gevraagd:

De aangepaste **TekeningTest** – voeg testen toe voor de methodes **voegToe()**, **getVorm()** en **verwijder()**.

De klasse **Tekening**.

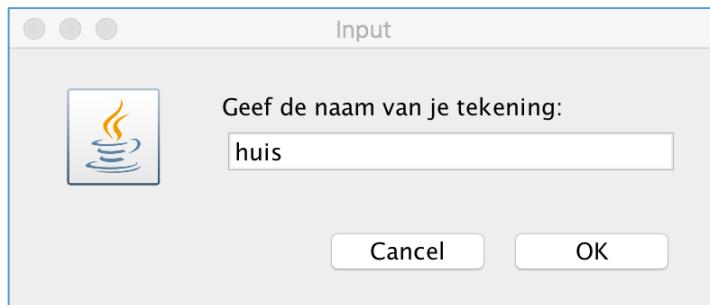
De aangepaste klasse **Ui**:

- de klasse heeft een **Tekening**
- vraag in het begin de naam van de tekening zodat je deze met de juiste naam kan aanmaken
- zorg dat alle vormen die je maakt toegevoegd worden aan deze tekening.

De aangepaste **AllTests** klasse: voeg **TekeningTest** toe.

Output bij het runnen van de launcher

...



...



Figuur 17. OUTPUT STORY 8

3.9 Story 09 – Omhullende van andere vormen

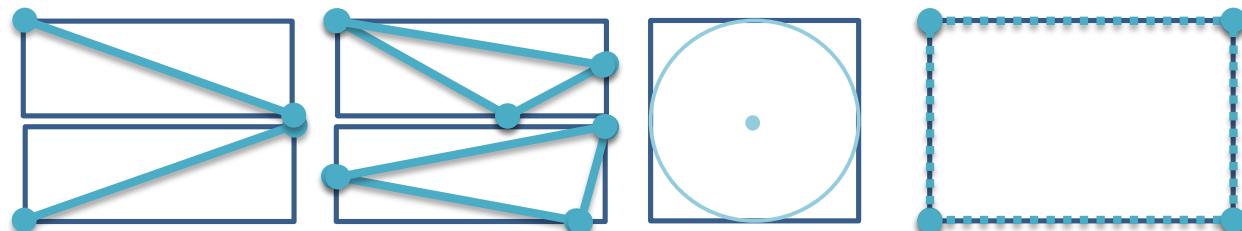
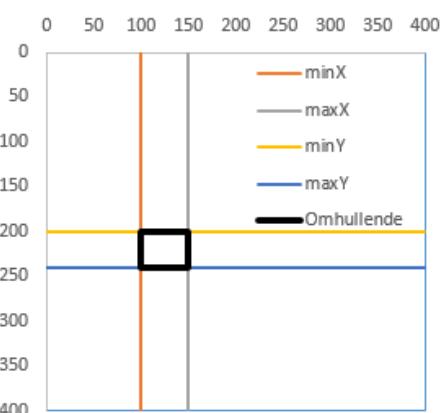
Als speler

kan ik de omhullende vragen van alle vormen
zodat ik later kan controleren of ze in een
tekening passen.

Ter herinnering (zie story 7): een Omhullende heeft een Punt positieLinksBoven (mag niet null zijn), een breedte(≥ 0) en een hoogte(≥ 0). Aan een omhullende moet je kunnen opvragen wat zijn minimale X, maximale X, minimale Y en maximale Y waarde is

Voorzie een methode getOmhullende() voor alle vormen. Kan je de omhullende op dit super niveau berekenen? Indien niet, hoe los je dat dan op?

Tekening met omhullende



Figuur 18.STORY 9 - VOORBEELDEN VAN OMHULLENDEN VOOR EEN LIJNSTUK, DRIEHOEK CIRKEL EN EEN RECHTHOEK

Gevraagd:

De aangepaste **CirkelTest**, **LijnStukTest**, **DriehoekTest**: voeg een test toe die controleert of de berekende omhullende (`getOmhullende()`) van de vorm gelijk is aan de omhullende die je zou verwachten.

Controleer of deze testklassen toegevoegd zijn aan **AllTests**.

De aangepaste klassen **Vorm**, **Cirkel**, **LijnStuk**, **Driehoek**. Voeg in de klasse **vorm** de methode `getOmhullende()` toe. Bij het omzetten naar String-vorm van de vorm wordt ook de omhullende getoond:

```

Cirkel:      middelpunt: (200, 200) - straal: 20 -
            Omhullende: (180, 180) - 40 - 40
Lijn:        startpunt: (100, 150) - eindpunt: (200, 250) -
            Omhullende: (100, 150) - 100 - 100
Driehoek:    hoekpunt1: (100, 200) - hoekpunt2: (300, 200) - hoekpunt3: (200, 100) -
            Omhullende: (100, 100) - 200 - 100
  
```

Zorg ervoor dat je het gemeenschappelijk van deze printout niet dubbel codeert. Waar moet het terecht komen om dit te realiseren?

3.10 Story 10 – Tekening met passende vormen

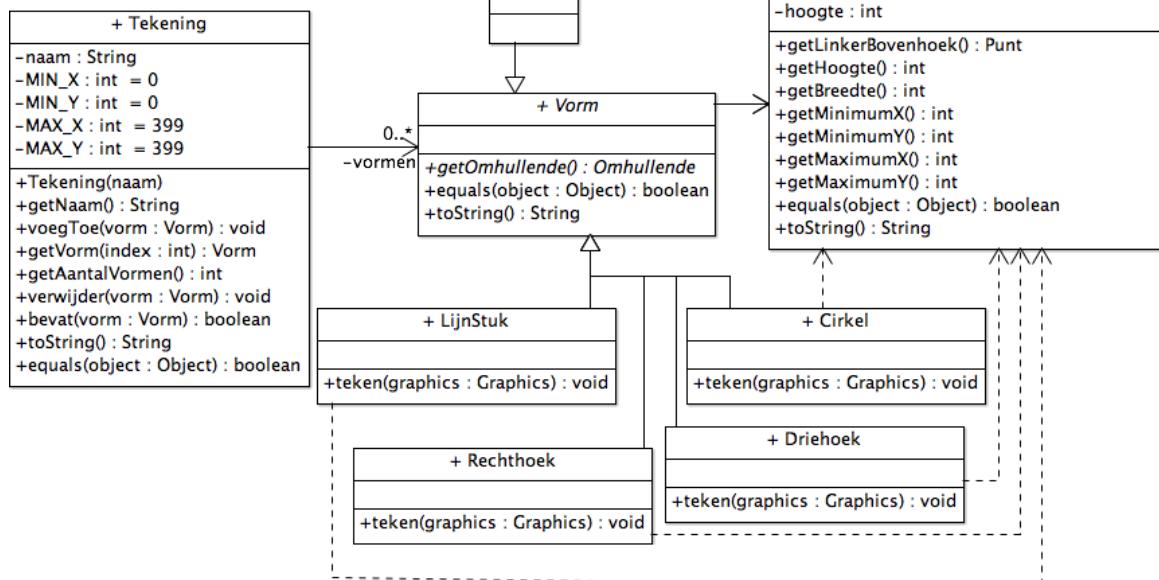
Als speler

kan ik enkel vormen toevoegen die op de tekening passen
zodat ik te zien krijg wat ik getekend heb.

We willen ervoor zorgen dat vormen die buiten de grenzen van de tekening vallen niet kunnen toegevoegd worden. Om dit te controleren, gaan wij de Omhullenden vergelijken met de (grenzen van de) Tekening. Eens de omhullenden correct berekend worden moet je ervoor zorgen dat Vormen waarvan de omhullende buiten de tekening valt (`min_x, max_x, min_y, max_y`) niet toegevoegd kunnen worden. Implementeer hiertoe een extra check in `voegToe(Vorm vorm)` methode van `Tekening`.

Gegeven:

Het **klassendiagramma**, met hierin alleen de klassen en methodes die toegevoegd of gewijzigd moeten worden.



Figuur 19. KLASSENDIAGRAMMA STORY 10

Gevraagd:

De aangepaste **TekeningTest**: voeg extra testen toe om te controleren of:

- de methode `voegToe()` een exception gooit wanneer de `minimumX` van de omhullende van de vorm kleiner is dan de `minimumX` van de tekening,
- ...

De aangepaste klasse **Tekening**: voeg de controle toe.

Als we nu vormen toevoegen die buiten de tekening vallen moeten we een foutbericht te zien krijgen.

3.11 Story 11 – De tekening getekend!

Als speler
kan ik de tekening tonen
zodat mijn tegenspelers kunnen raden wat ik getekend heb.

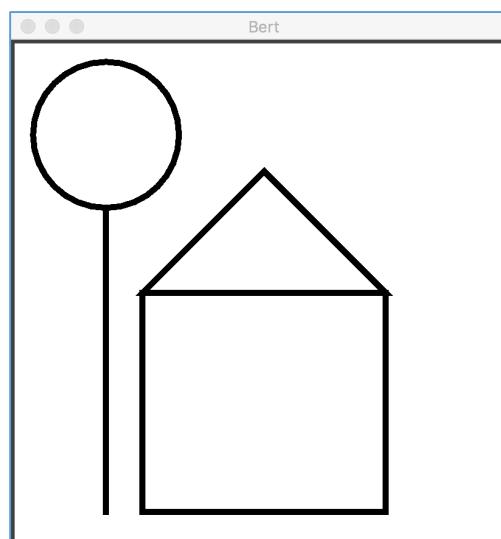
Aan het eind van deze story kan je een **zelf gemaakte tekening** tekenen in een tekenvenster. Deze story vormt een uitzondering op alle vorige en de meeste volgende, in die zin dat we hier geen testklassen zullen schrijven. We zullen hier tekenen op een canvas, iets wat je visueel kan controleren, maar wat je niet kan unit-testen, of toch niet met de conventionele testen die jullie tot nu toe besproken/gebruikt hebben.

Alles wat je wilt kunnen tekenen (een Tekening, een Vorm, ...) moet de interface **Drawable** implementeren. Deze interface bevat één methode **teken(Graphics)**. De klasse **Graphics** is een bestaande Java-klasse die rendering informatie bijhoudt (kleur, font, ...).

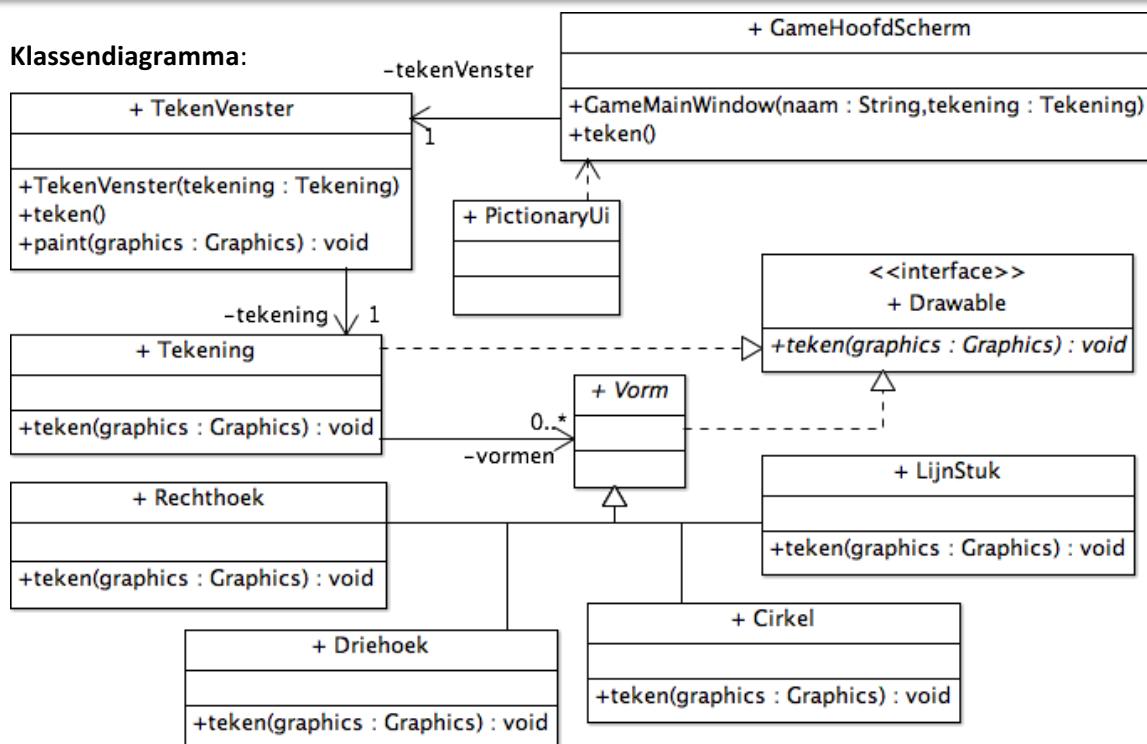
Gegeven:

De klasse **GameMainWindow**. Deze erft over van de Java-klasse **JFrame**. Een **JFrame** is het hoofdvenster van de grafische interface. Alle andere elementen (zoals de titel, de tekening, ...) worden hieraan toegevoegd.

De klasse **TekenVenster**. Deze erft over van de Java-klasse **Canvas**. Op een Canvas kan getekend worden. De klasse heeft een methode **teken()**, waarin de methode **repaint()** wordt opgeroepen. Je kan niet zien wat deze methode van de klasse **Canvas** doet, maar hij roept de methode **paint(Graphics)** op. Op dit moment is de tekening die hier wordt getekend hardgedecodeerd, zonder gebruik te maken van bestaande Tekening-klasse. Dit moet je uiteraard aanpassen.



Figuur 20. HARDGECODEERDE TEKENING – TE VERVANGEN



Figuur 21.KLASSENDIAGRAMMA STORY 11

Gevraagd:

De klasse **UiException** voor het geval er een exception moet gegooid worden in de user-interface klassen.

De aangepaste klasse **Ui**. Vervang de code om de tekening te tonen door het volgende (probeer te begrijpen wat deze code doet):

```

GameMainWindow view = new GameMainWindow(speler.getNaam(), tekening);
view.setVisible(true);
view.teken();
    
```

De interface **Drawable**, met daarin een publieke methode teken(Graphics g).

Aanpassingen in de klassen **Lijn**, **Rechthoek**, **Driehoek** en **Cirkel**. Deze klassen moeten de interface **Drawable** implementeren. Ga voor de inhoud van de teken() methode eens spieken in de klasse **TekenVenster** om te zien hoe je cirkels, lijnen, rechthoeken en driehoeken kan tekenen.

De aangepaste klasse **Tekening**. Deze klasse moet ook de interface **Drawable** implementeren. Om een **Tekening** te tekenen moeten alle vormen getekend worden.

De aangepaste klasse **TekenVenster**: zorg er nu voor dat deze niet meer hard gecodeerde tekeningen maakt, maar dat ze een tekening meekrijgt en deze tekent!

3.12 Story 12 – Hint (het raden van een verborgen woord)

Als speler
kan ik letters raden
zodat ik te weten kan komen over welk woord het gaat.

Vanaf nu gaan we ons niet meer op HangMan focussen. In deze story laten we de speler toe om woorden te raden. Om de speler te helpen wordt steeds een hint getoond.

Stel: woord = "test" → hint = " _ _ _ "

Iedere letter van woord werd vervangen door " _ " (spatie, underscore). Een woord kan ook spaties bevatten (zin), een spatie wordt in de hint vervangen door " " (2 spaties).

Eens je een hint hebt, kan je raden. Om te raden geef je 1 letter (als char) mee. Als je juist geraden hebt, dan worden in de hint alle " _ " die overeenkomen met je letter vervangen door die letter.

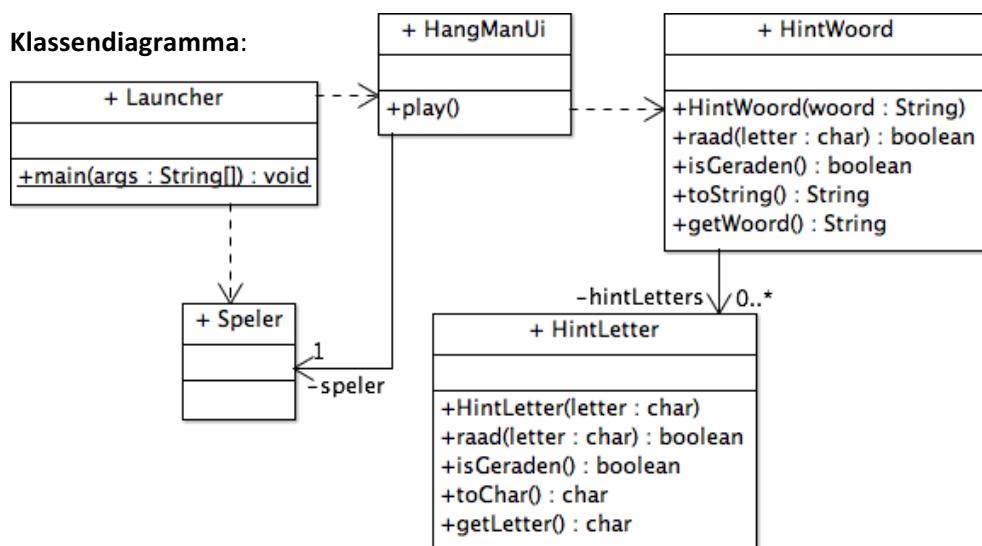
Voorbeeld: raad('t') → hint = " t _ _ t"

Voor een hint voorzien we de klasse HintWoord. Deze bestaat uit een aantal objecten van de klasse HintLetter.

Gegeven:

De testklassen **HintLetterTest** en **HintWoordTest**: twee volledig uitgeschreven testen, die volledig moet compileren en groen kleuren aan het einde van deze story.

Klassendiagramma:



Figuur 22. Klassendiagramma Story 12

Gevraagd:

De klasse **HintLetter**, op zo'n manier dat **HintLetterTest** slaagt.

De klasse **HintWoord**, op zo'n manier dat **HintWoordTest** slaagt.

De aangepaste **AllTests** klasse: voeg de gegeven testklassen toe.

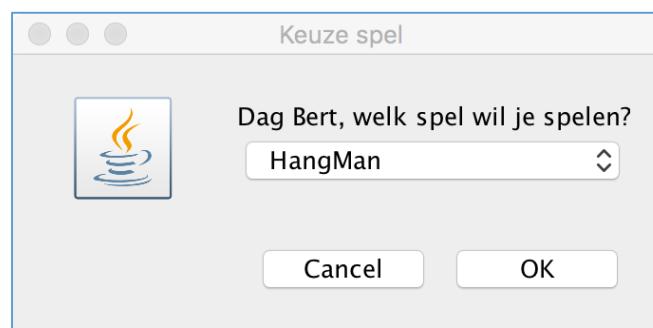
De aangepaste klasse **Launcher**: deze moet nu vragen welk spel je wilt spelen: Pictionary of HangMan (zie output hieronder)

De klasse **HangManUi** (zie output hieronder):

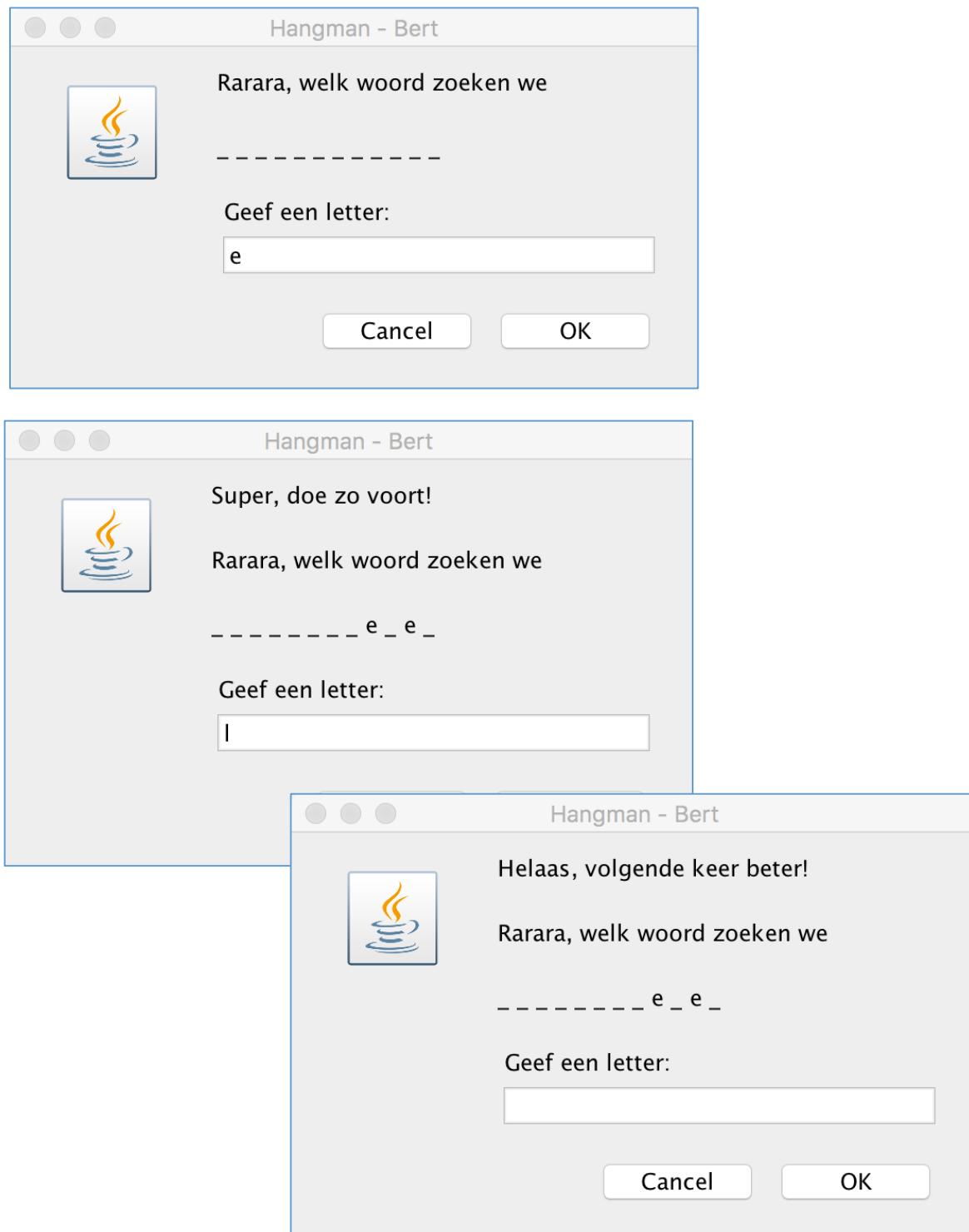
- gebruik voorlopig een hard-coded woord
- toon de tip en laat de speler raden door gebruik te maken van `JOptionPane.showInputDialog()`
- toon de naam van het spel + de naam van de speler in de titelbalk
- herhaal dit tot alle letters geraden zijn
- toon telkens een boodschap of de speler al dan niet juist heeft geraden

Output:

...



Figuur 23. LAUNCHER NA HET VRAGEN VAN DE SPELERAAM IN STORY 12



Figuur 24. SPELVERLOOP STORY 12

3.13 Story 13 – Woordenlijst

Als administrator
kan ik woorden toevoegen aan het spel
zodat deze door de spelers geraden kunnen worden.

In deze story lezen we een woordenlijst in, zodat de speler een random woord krijgt om te raden. Woorden in een woordenlijst mogen niet null en niet "" (=leeg) zijn. Ze mogen ook geen twee keer worden toegevoegd. De belangrijkste functionaliteit is dat je uit de lijst van woorden een random woord moet kunnen opvragen. Zolang je lijst leeg is zal je hier null terug krijgen. Als de lijst 1 woord bevat moet je dat woord terug krijgen. In alle andere gevallen krijg je gewoon een willekeurig woord terug uit de lijst met woorden. Daarnaast moet je aan een WoordenLijst kunnen opvragen hoeveel woorden er in zitten.

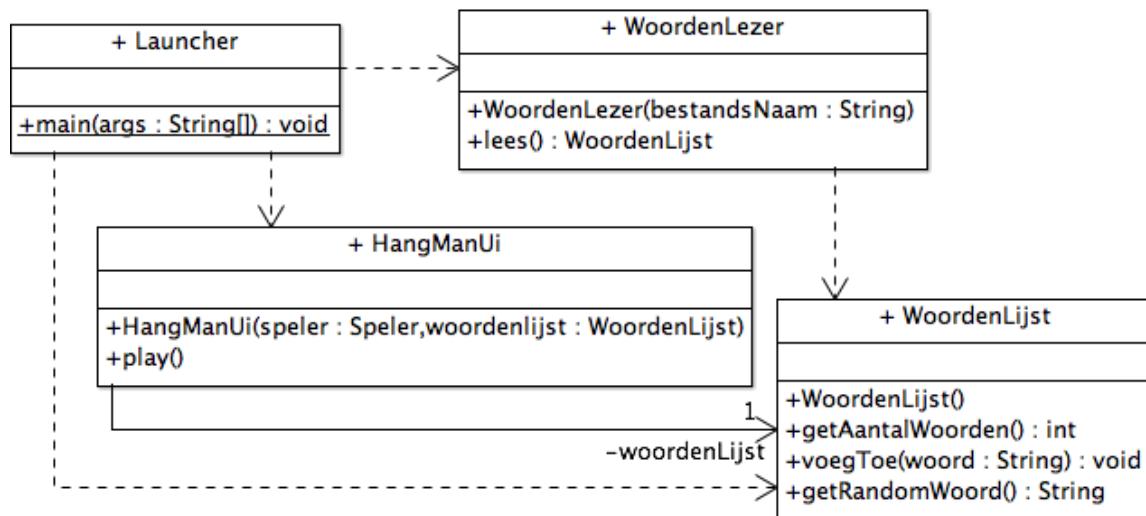
Om een woordenlijst te maken, zullen we vertrekken van een tekstbestand dat een aantal woorden/zinnen bevat: hangman.txt. Er is een voorbeeldbestand gegeven!

Gegeven:

WoordenLijstTest: een deels uitgeschreven test

hangman.txt: een voorbeeldbestand met een aantal woorden/zinnen

Klassendiagramma:



Figuur 25.KLASSENDIAGRAMMA STORY 13

Gevraagd:

De aangepaste **WoordenLijstTest** – voeg testen toe voor de methode `getRandomWoord()`. De test moet volledig compileren compileren en groen kleuren aan het einde van deze story.

De aangepaste **AllTests** klasse: voeg **WoordenLijstTest** toe.

De klasse **WoordenLijst**, zodat de testklasse compileert.

Een package **db**, waar klassen inkomen die verantwoordelijk zijn voor het lezen uit/schrijven naar externe opslag.

De klasse **DbException** voor het geval er een exception moet gegooid worden in de db klassen.

De klasse **WoordenLezer**. Deze komt in de package *db*. De klasse zorgt voor het uitlezen van de woorden uit een exten bestand. Gebruik hiervoor `java.util.Scanner`.

De aangepaste klasse **Launcher**: bij het opstarten moeten alle woorden uit het tekstbestand uitgelezen worden en toegevoegd aan de woordenlijst.

De aangepaste klasse **HangManUi**: deze krijgt een woordenlijst mee en haalt daaruit het woord dat geraden moet worden.

3.14 Story 14 – TekeningHangMan – tekenen en zichtbaar maken.

Als speler
kan een tekening van een galg zien
zodat ik daarop het verloop van het spel zal kunnen volgen.

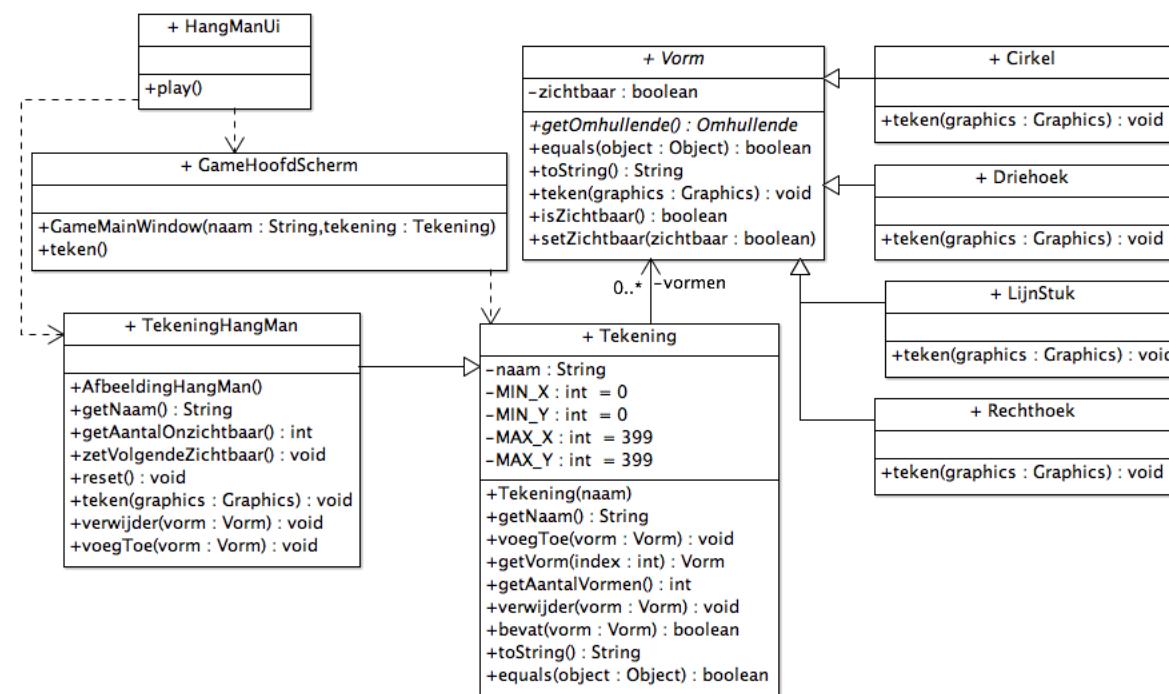
In deze story voegen we de tekening toe die gebruikt zal worden om Hangman te tekenen in stapjes. Het tekenen in stapjes betekent niet dat er in ieder stapje 1 vorm wordt toegevoegd aan de tekening. Het betekent dat vormen in een tekening zichtbaar en onzichtbaar kunnen zijn. Alleen de zichtbare vormen mogen getekend worden. Standaard is iedere vorm die aangemaakt wordt zichtbaar.

Het HangManMannetje is een zeer specifieke tekening, met 4 altijd-zichtbare onderdelen: de galg, en met een initieel onzichtbaar mannetje dat bestaat uit 14 onderdelen. Om dit mannetje gemakkelijk te kunnen aanmaken, voorzien we een TekeningHangMan, een subklasse van Tekening. In de constructor van deze klasse voegen we hard gecodeerd de onderdelen van ons mannetje toe. Het is belangrijk dat het toevoegen en verwijderen van vormen voor deze subklasse geen effect mag hebben.

Je moet van een Hangmannetje de onderdelen van het mannetje één voor één zichtbaar kunnen maken. Je moet ook de mogelijkheid hebben om het mannetje terug te herstellen in zijn originele zichtbaarheidstoestand.

Gegeven:

Klassendiagramma:



Figuur 26.KLASSENDIAGRAMMA STORY 14

Gevraagd:

De aangepaste klasse **Vorm**, om zichtbaarheid van een vorm te regelen. Een vorm is default wél zichtbaar.

De aangepaste **subklassen** van vorm, zodat onzichtbare vormen niet getekend worden.

TekeningHangManTest, een JUnit testklasse die het volgende moet testen:

- AfbeeldingHangman maakt een hangman afbeelding met de naam hangman en met 18 vormen waarvan 14 onzichtbaar
- zetVolgendeZichtbaar moet eerstvolgende onzichtbare vorm zichtbaar zetten
- zetVolgendeZichtbaar moet exception gooien als alle vormen zichtbaar
- reset moet alle vormen behalve de galg weer onzichtbaar zetten
- voegToe moet een exception gooien
- verwijder moet een exception gooien

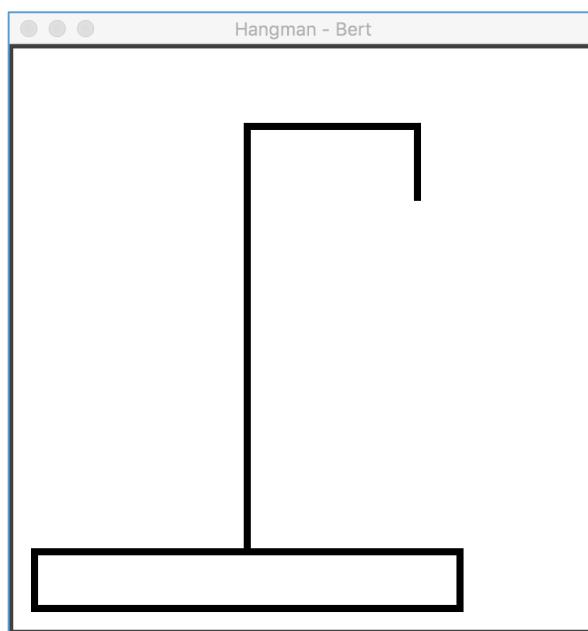
De klasse **TekeningHangMan**, zodat de test slaagt. TekeningHangMan is een Tekening met de naam "HangMan" en heeft volgende vormen:

```
Vorm galgBodem = new Rechthoek(new Punt(10, 350), 300, 40); // altijd zichtbaar
Vorm galgStaaf = new LijnStuk(new Punt(160, 350), new Punt(160, 50)); // altijd zichtbaar
Vorm hangbar = new LijnStuk(new Punt(160, 50), new Punt(280, 50)); // altijd zichtbaar
Vorm koord = new LijnStuk(new Punt(280, 50), new Punt(280, 100)); // altijd zichtbaar
Vorm hoofd = new Cirkel(new Punt(280, 125), 25); // zichtbaar na 1 fout
Vorm oogLinks = new Cirkel(new Punt(270, 118), 2); // zichtbaar na 2 fouten
Vorm oogRechts = new Cirkel(new Punt(290, 118), 2); // ...
Vorm neus = new Cirkel(new Punt(280, 128), 2);
Vorm mond = new LijnStuk(new Punt(270, 138), new Punt(290, 138));
Vorm lijf = new LijnStuk(new Punt(280, 150), new Punt(280, 250));
Vorm beenLinks = new LijnStuk(new Punt(280, 250), new Punt(240, 310));
Vorm beenRechts = new LijnStuk(new Punt(280, 250), new Punt(320, 310));
Vorm voetLinks = new Cirkel(new Punt(240, 310), 5);
Vorm voetRechts = new Cirkel(new Punt(320, 310), 5);
Vorm armLinks = new LijnStuk(new Punt(280, 200), new Punt(230, 170));
Vorm armRechts = new LijnStuk(new Punt(280, 200), new Punt(330, 170));
Vorm handLinks = new Cirkel(new Punt(230, 170), 5);
Vorm handRechts = new Cirkel(new Punt(330, 170), 5);
```

Figuur 27. DE VORMEN NODIG VOOR EEN HANGMANMANNETJE

De aangepaste klasse **HangManUi**. Wanneer de speler gekozen heeft voor HangMan, dan wordt het initiële scherm getoond (zie Output), vooraleer het raden begint.

Output:



Figuur 28. DE TEKENING BIJ HET BEGIN VAN HET SPEL

3.15 Story 15 – Hangman

Als speler

kan ik het mannetje zichtbaar zien worden terwijl ik speel
zodat ik daarop het verloop van het spel kan volgen.

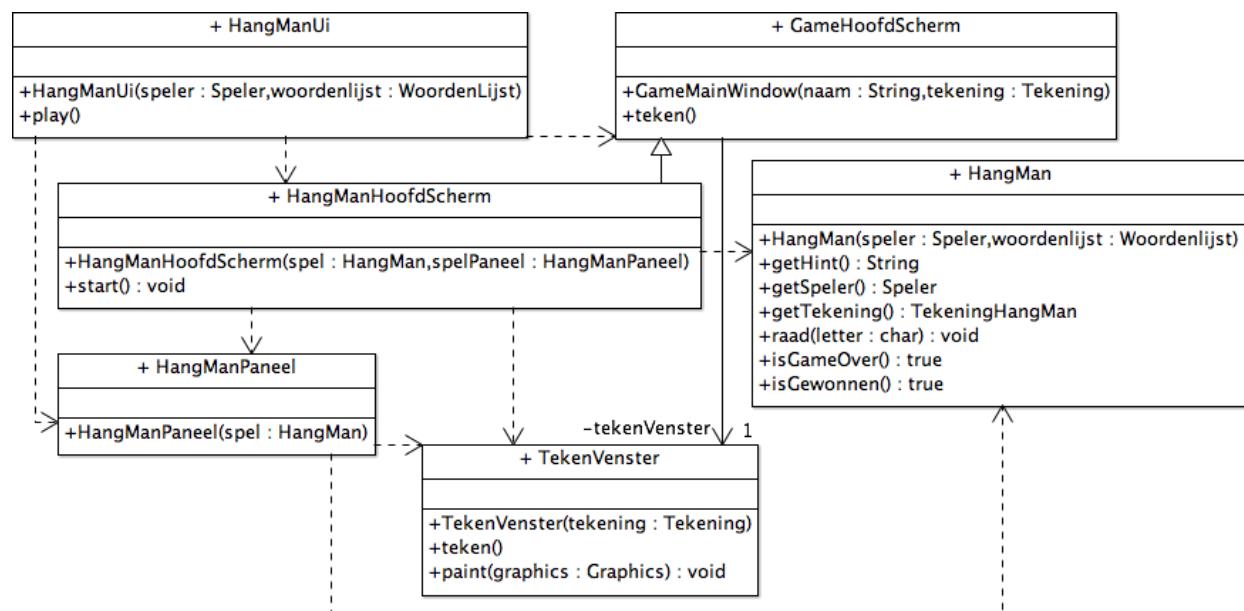
In deze story maken we eindelijk de eerste echte **HangMan** versie. Na het succesvol voltooien van alle vorige stories is dit niet eens zo'n moeilijke taak. We moeten alle vorige elementen samenbrengen in een nieuwe klasse. **HangMan** heeft een speler, een **TekeningHangMan**, een **WoordenLijst** (die via een **WoordenLezer** moet worden gevuld moet woorden uit een bestand **hangman.txt**) en een **HintWoord** met **HintLetters** voor het huidige te raden woord. Daarnaast wordt er bijgehouden of je gewonnen hebt dan wel of je game over bent. Het spel moet alleen correct geinitialiseerd worden, en dan kan het raden beginnen...

Gegeven:

HangManTest – een volledig uitgeschreven test, die volledig moet compileren en groen kleuren aan het einde van deze story

De user interface klassen **HangManHoofdScherm** en **HangManPaneel**. Deze laatste is onvolledig: je zal ze zelf moeten afwerken.

Klassendiagramma:



Figuur 29.KLASSENDIAGRAMMA STORY 15

Gevraagd:

De aangepaste **AllTests** klasse: voeg **HangManTest** toe.

De klasse **Hangman**, zodat de testklasse slaagt en zodat het spel gespeeld kan worden.

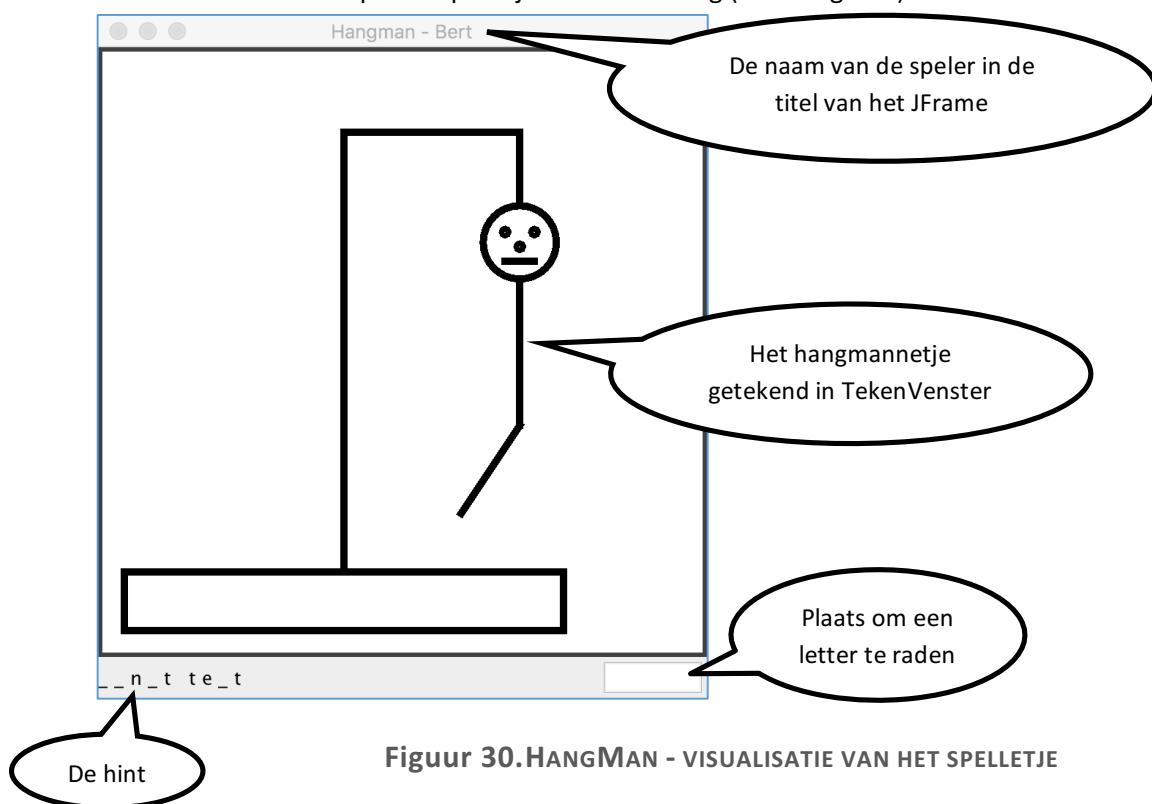
De aangepaste klasse **HangManPaneel**: vervang de todo's in commentaar door Java code om het spel te spelen.

De aangepaste klasse **HangManUi**: vervang de inhoud van de play methode door volgende stappen:

- aanmaken van een HangMan spel
- aanmaken van een HangManPaneel
- aanmaken van een HangManHoofdScherm
- starten van het HangManHoofdScherm

Output:

Voor de flow van het spel: zie prentjes in de inleiding (2.3 Hangman)



Figuur 30. HANGMAN - VISUALISATIE VAN HET SPELLETJE

OP HET EINDE VAN DEZE SESSIE MOET DE TEST DIE ALLE TESTEN COMBINEERT (ALLTESTS**) VOLLEDIG GROEN KLEUREN. JE MOET OOK HET SPELLETJE HANGMAN KUNNEN SPELEN DOOR LAUNCHERSTORY14 TE LAUNCHEN. PAS ALS DAT HET GEVAL IS MAG JE DEZE SESSIE ALS "VOLTOOID" BESCHOUWEN.**