Verslag Fase 2
Programmeerproject 1:
Tower Defense

Faculteit Wetenschappen en Bio-ingenieurswetenschappen
Vrije Universiteit Brussel
1BA Computerwetenschappen
Academiejaar 2022-2023

Stuker Abel
0606930
abel.stuker@vub.be

Inhoudsopgave

1 Inleiding

Dit document is het verslag van het vak Programmeerproject 1 waarin een top-down Tower Defense spel wordt geïmplementeerd. De essentie van het spel is dat de speler verschillende levels – bestaande uit meerdere rondes – overleeft. In elk level proberen per ronde een bepaald aantal monsters het einde van het pad te bereiken. Indien dit lukt verliest de speler een aantal levens, maar dat kan worden verhinderd door het plaatsen van torens die op de monsters schieten. Deze versie van het spel bevat 4 soorten monsters en 4 soorten torens. Elk type monster en toren heeft unieke eigenschappen. Wanneer monsters sterven verdient de speler geld, en soms laten ze ook power-ups achter (die kunnen worden opgeraapt). Met geld kunnen torens en power-ups worden gekocht. Er bestaan 2 power-ups: tanks en obstakels. Indien een speler alle levens heeft verloren, is het spel voorbij. Wanneer alle levels worden uitgespeeld wint de speler het spel.

1.1 Specificaties

Zoals reeds vermeld hebben de verschillende torens en monsters verschillende eigenschappen. Deze worden hieronder beschreven:

1.1.1 Monsters

- **Rood monster** is het gewone monster met 1 leven dat met een vaste snelheid over het pad beweegt.
- **Blauw monster** heeft 3 levens en beweegt sneller dan het rode monster. Echter, wanneer het nog maar 1 leven overheeft zal het trager bewegen.
- **Geel monster** beweegt trager, heeft 2 levens, en wordt beschermd door een schild. Dit schild verdwijnt kort na elke aanraking met een projectiel, en verdwijnt definitief na drie aanrakingen.
- **Paars monster** heeft 4 levens en geeft – wanneer deze sterft – een extra leven aan alle dichtstbijzijnde niet-paarse monsters.

1.1.2 Torens

- **Stenentoren** is de goedkoopste toren en schiet stenen op de voorbijkomende monsters. Wanneer een steen botst op een monster verliest dat monster een leven en verdwijnt de steen.
- **Kanonstoren** is een toren die kanonskogels werpt naar voorbijkomende monsters. Wanneer een monster geraakt wordt verliest het een leven en wordt het een korte afstand achteruitgeschoten.
- **Bommentoren** schiet op een lage frequentie bommen af op het pad. Wanneer een bom op het pad ligt ontploft deze na een kort tijdsinterval. De monsters die geraakt worden binnen de impactstraal van de ontploffing verliezen 3 levens.
- **Nettentoren** gooit een net naar het pad wanneer er monsters in de buurt zijn. De monsters die op dat moment op dat deel van het pad bewegen zullen vertraagd zijn. Na enkele seconden verdwijnt het net.

1.2 Hulp bij het spelen

Het spel kan worden opgestart door het `main.rkt`-bestand uit te voeren. Om het spel op een eenvoudige manier te kunnen testen, werden enkele keybindings toegevoegd:

- Druk op de **m** toets om 200 coins toe te voegen. Hiermee kunnen bijgevolg extra `Tower`s, `Tank`s of `Obstacle`s worden aangekocht.
- Om extra `Monster`s te spawnen op de eerste `PathCell`, kan de toets van de eerste letter van de kleur van het `Monster` worden ingedrukt (i.e., **r** voor het rode, **b** voor het blauwe, **g** voor het gele en **p** voor het paarse monster).

2 ADTs

In deze sectie bespreken we de verschillende Abstracte Data Types (ADTs) die werden geïmplementeerd, samen met hun verschillende dispatchfuncties die beschikbaar zijn.

2.1 Game ADT

Het `Game ADT` is het hart van het spel waarin leveloverschrijdende elementen worden beheerd: het geld, de levens, het huidige level, de power-ups-delegate, het screen, en de geselecteerde toren.

Naam	**Signatuur** en **Omschrijving**
`update!`	$(\emptyset \to \emptyset)$ Deze procedure stuurt een updatecall naar zowel het `Screen` als het huidige `Level`.
`mouseclick`	$(\emptyset \to \emptyset)$ Deze procedure laat het `Level` weten dat er een muisklik heeft plaatsgevonden.
`keypress`	$(\emptyset \to \emptyset)$ Deze procedure laat het `Level` weten dat er een key is ingedrukt. Indien er geen huidig level is en de spatiebalk wordt ingedrukt, dan herstart het spel.
`start!`	$(\emptyset \to \emptyset)$ Deze procedure start het spel door het `Screen` te initialiseren, een power-ups-delegate in te stellen en het eerste level te starten.
`get-power-ups-delegate`	`PowerUpsDelegate` Deze procedure geeft de `PowerUpsDelegate` terug.
`get-lives` `set-lives!`	number (number $\to \emptyset$) Deze procedures zijn verantwoordelijk zijn voor het *accesseren* en *muteren* van het aantal levens.
`get-money` `set-money!`	number (number $\to \emptyset$) Deze procedures zijn verantwoordelijk zijn voor het *accesseren* en het *muteren* van het geld.
`get-lives` `set-lives!`	number (number $\to \emptyset$) Deze procedures zijn verantwoordelijk zijn voor het *accesseren* en het *muteren* van het aantal levens.
`get-level`	`Level` Deze procedure geeft het huidige `Level` terug.
`new-level!`	$(\emptyset \to \emptyset)$ Deze procedure wordt opgeroepen wanneer een volgend level moet gestart worden. Eerst wordt de interne variabele `level-number` met 1 verhoogd, en nadien wordt een nieuwe instantie van het `Level ADT` gebonden aan de interne `level` variabele, waarna dit level wordt gestart.
`end-game!`	(boolean $\to \emptyset$) Deze procedure beëindigt het huidige spel door het `Level` te verwijderen en door aan het `Screen` te delegeren om alle spelelementen te verwijderen en om een win/game over tekstje te laten verschijnen (a.d.h.v. de enige parameter: winst?).
`get-selected-tower-type` `set-selected-tower-type!`	number (number $\to \emptyset$) Deze procedures zijn verantwoordelijk zijn voor het *accesseren* en het *muteren* van het aantal levens. Deze *mutatie* gebeurt door het `Screen ADT`, wanneer de speler de torenselectie verandert.

2.2 Screen ADT

Het `Screen ADT` staat in voor alle communicatie met de grafische bibliotheek. Alles wat op het scherm moet worden getekend zal uiteindelijk gebeuren via dit ADT. Het scherm is hoofdzakelijk opgedeeld in 2 verschillende zones: het speelveld en de User Interface. De bouw van de UI wordt bepaald door de interne variabelen `bottombar-ui-configuration` en `sidebar-ui-configuration`. Deze bevatten geneste `UIContainer ADTs` waarmee hun constructie bepaald wordt.

Naam	**Signatuur** en **Omschrijving**
`make-screen`	($\emptyset \rightarrow$ `Screen`) *Constructor.* Hiermee wordt het `Screen`object aangemaakt, en automatisch wordt gecommuniceerd met de grafische bibliotheek om een `Window` te tekenen.
`update-dynamic-counter!`	(number $\rightarrow \emptyset$) Deze updatefunctie stuurt naar de `UIDynamicText`objecten (`dynamic-text-action` en `dynamic-text-error`) een updatecall met het aantal milliseconden sinds laatste update.
`draw-game-element!`	(symbol {`Tower`, `Projectile`, `Tank`, `Obstacle`, `DroppedItem`, `PathCell`, `Position`, `Path`} $\rightarrow \emptyset$) Deze procedure is nogal vanzelfsprekend. Op basis van het type dat wordt meegegeven als eerste argument zal de visuele representatie van het object, dat als eerste element in de lijst (2de argument) wordt meegegeven, op het scherm worden getekend. (type $\in$ { 'tower, 'projectile, 'monster, 'tank, 'obstacle, 'dropped-item, 'pathcell, 'entry-exit-overlays, background})
`clear-all-game-elements!`	($\emptyset \rightarrow \emptyset$) Deze procedure haalt alle `Tower`s, `Projectiles` en `DroppedItems` weg van het scherm.
`clear-game-element!`	(symbol {`Projectile`, `Monster`, `DroppedItem`} $\rightarrow \emptyset$) Deze procedure haalt op basis van het type dat wordt meegegeven als eerste argument het corresponderende object (2de argument, behalve wanneer type = 'path of 'power-up) weg van het scherm. (type $\in$ {'projectile, 'monster, 'power-up, 'dropped-item, 'path }).
`draw-ui-element!`	(symbol list $\rightarrow \emptyset$) Deze procedure zal afhankelijk van het type dat wordt meegegeven als eerste argument een ui-element tekenen. Het 2de argument is een lijst van specificaties die kan verschillen per ui-element-type, maar de eerste twee elementen zullen steeds de x- en y-waarden zijn waarop het element moet worden getekend. (type $\in$ { 'text, 'image, 'selection-box})
`update-ui-element!`	(symbol {string number} $\rightarrow \emptyset$) Deze procedure zal afhankelijk van het elementtype (1ste argument) een bepaalde `UIDynamicText` instantie updaten waardoor diens tekst in de UI ook verandert. (symbol $\in$ {'money, 'lives, 'tanks, 'obstacles, 'power-ups-status, 'level, 'wave, 'action, 'error) De waarde waarnaar deze tekst moet worden veranderd wordt bepaald door het 2de argument.
`initialise!`	(`Game` $\rightarrow \emptyset$) Deze procedure initialiseert het spel door het meegegeven `Game`object intern op te slaan voor communicatie en door de UI te tekenen.

2.2.1 UIContainer ADT

Een `UIContainer` is de fundamentele bouwblok van de User Interface. Elke instantie hiervan zal een bepaald type hebben en zal eventueel kinderen bevatten (ook weer `UIContainer`s). Elke `UIContainer` kan van het type 'row', 'column', 'selectable-column', 'image', 'scaled-image', static-text', 'dynamic-text', 'padding', 'button', 'v-space' of 'h-space' zijn. Wanneer een `UIContainer` wordt gebouwd, worden ook diens kinderen op de juiste positie geconstrueerd. Hierdoor wordt heel de boomstructuur van `UIContainer`s getekend op het scherm.

Naam	**Signatuur** en **Omschrijving**
`make-ui-container`	(symbol . optionals → `UIContainer`) *Constructor.* Hiermee wordt op basis van het type (1ste argument) de `UIContainer`object aangemaakt. Let op: er wordt nog niets getekend op het scherm. De optionele parameter(s) kunnen info bevatten over het UI-element zelf en/of kinderen bevatten (cf. infra).
`build!`	(`Screen` number number → ∅) Deze procedure zorgt ervoor dat de `UIContainer` getekend wordt op het scherm. Hiervoor is het huidige `Screen` vereist, alsook de x- en y-waarden (in pixels) van de linkerbovenhoek waar de `UIContainer` moet worden getekend. Deze procedure tekent ook de kinderen (indien aanwezig) en bepaalt hiervan hun x- en y-coördinaten.
`click!`	(`Screen` number number → ∅) Deze procedure wordt getriggerd wanneer de gebruiker heeft geklikt binnen de dimensies van de `UIContainer`. Indien er kinderen zijn wordt deze procedure opgeroepen bij het juiste kind (afhankelijk van diens positie en dimensies binnen de huidige `UIContainer`). Indien de huidige `UIContainer` van het type 'button of 'selectable-column is worden respectievelijk de `UIClickListener` en `UISelectionListener` gebruikt om de klik uit te voeren.
`get-w!`	number
`get-h!`	number
`get-x!`	number
`get-y!`	number Deze *accessoren* zijn vanzelfsprekend; ze geven respectievelijk de breedte, hoogte, x-coördinaat en y-coördinaat van de `UIContainer` terug.

De vereiste optionele argumenten hangen af van het type `UIContainer`:

- **row/column**: (kinderen)
- **selectable-column**: (`UISelectionListener` kinderen)
- **image**: (image-path mask-path)
- **scaled-image**: (image-path mask-path scale-number)
- **static-text**: (text specifications)
- **dynamic-text**: (`UiDynamicText` specifications)
- **padding**: (padding-in-px kind)
- **button**: (`UIClickListener` kind)
- **v-space**: (v-distance)
- **h-space**: (h-distance)

2.2.2 UISelectionListener ADT

Een `UISelectionListener` wordt meegegeven aan een `UIContainer` van het type 'selectable-column. Hieraan kunnen procedures meegegeven worden (zogenaamde *updaters*), die worden opgeroepen wanneer de selected-index wijzigt. Deze nieuwe index wordt aan de updaters meegegeven als argument. Ook wordt hierin de selection-tile opgeslagen: dit is de `Tile` die de rechthoek vormt om het geselecteerde element in de kolom visueel voor te stellen.

Naam	**Signatuur** en **Omschrijving**
`make-ui-selection-listener`	(∅ → `UISelectionListener`) *Constructor.* Hiermee wordt het object aangemaakt, met 0 als initiële selected-index en zonder updaters.
`get-selected-index`	number)
`set-selected-index!`	(number → ∅) Deze procedures *accesseren* en *muteren* de selected-index. Bij de mutatie worden ook alle updaters aangeroepen met de nieuwe index als enige argument.

`get-selection-tile`	`Tile`
`set-selection-tile!`	(`Tile` → ∅) Deze procedures *accesseren* en *muteren* de selection-tile (om de selectie visueel voor te stellen).
`add-updater!`	(procedure → ∅) Deze procedure voegt de procedure die als argument wordt meegegeven toe aan de lijst van updaters die worden uitgevoerd wanneer de index wordt gewijzigd. Let op: de procedure die meegegeven wordt moet 1 parameter hebben (de nieuwe index).

2.2.3 UIClickListener ADT

Een `UIClickListener` wordt meegegeven aan een `UIContainer` van het type 'button. Hieraan kunnen procedures meegegeven worden (zogenaamde *updaters*), die worden opgeroepen wanneer een muisklik wordt uitgevoerd op de `UIContainer`.

Naam	**Signatuur** en **Omschrijving**
`make-ui-click-listener`	(∅ → `UIClickListener`) *Constructor.* Hiermee wordt het object aangemaakt, zonder initiële updaters.
`execute-click!`	(∅ → ∅) Deze procedure wordt aangeroepen door de `UIContainer` waarin dit object zit vervat, en roept vervolgens alle updaters aan zonder argumenten.
`add-updater!`	(procedure → ∅) Deze procedure voegt de procedure die als argument wordt meegegeven toe aan de lijst van updaters. Let op: de procedure die als argument wordt meegegeven mag geen parameters bevatten.

2.2.4 UIDynamicText ADT

Een `UIDynamicText` object wordt meegegeven aan een `UIContainer` van het type 'dynamic-text. Hieraan kunnen procedures meegegeven worden (zogenaamde *updaters*), die worden opgeroepen wanneer de tekst wordt gewijzigd. Optioneel kan het object ingesteld worden zodat elke keer na het instellen van een tekst, deze maar een gedurende een beperkte tijd zichtbaar blijft.

Naam	**Signatuur** en **Omschrijving**
`make-ui-dynamic-text`	(string . (number) → `UIDynamicText`) *Constructor.* Aan de hand van de initiële tekst en optioneel de duurtijd (na hoelang de tekst verdwijnt) wordt het `UIDynamicText` object gemaakt.
`update!`	(number → ∅) Deze procedure, die wordt aangeroepen met als enige argument het aantal verstreken milliseconden sinds laatste update, zorgt voor het intern aftellen van de timer waarna –indien nodig– de tekst wordt leeggemaakt (lege string:).
`get-text`	string
`update-text!`	(string → ∅) De *accessor* geeft simpelweg de dynamische tekst terug. De *mutator* update de dynamische tekst waarna de updaters worden aangeroepen (die verantwoordelijk zijn voor het effectief veranderen van de tekst op het scherm).
`add-updater!`	(procedure `Screen` list `Tile` → ∅) Deze procedure voegt de procedure die als eerste argument wordt meegegeven toe aan de lijst van updaters, samen met het `Screen`, een lijst van specificaties en de tekst`Tile`. Deze wordt uitsluitend gebruikt door de `UIContainer`, die een interne procedure meegeeft waardoor de text effectief wordt aangepast op het scherm wanneer aangeroepen.

2.3 Level ADT

Het `Level ADT` is de kern van elk level. Elke keer wanneer een nieuw level wordt opgestart wordt er een nieuwe instantie van dit ADT aangemaakt en opgeslagen in het `Game ADT`. Hierin zitten ook alle objecten vervat die nodig zijn in een level, zoals de `Monster`s, `Projectile`s, `Tower`s, `DroppedItems` en het `Path`.

Naam	**Signatuur** en **Omschrijving**
`make-level`	(`Game Screen` number → `Level`) *Constructor.* Deze procedure maakt een nieuwe instantie van het ADT aan, met de huidige `Game`, het `Screen` en het levelnummer als parameters.
`start!`	($\emptyset \to \emptyset$) Deze procedure start het level door o.a. het pad te initialiseren.
`mouseclick`	(symbol `Position` → $\emptyset$) Deze procedure wordt aangeroepen vanuit het `Screen ADT` om aan te geven dat er een muisklik heeft plaatsgevonden in het speelveld op de meegegeven `Position`, en zorgt ervoor dat ofwel de power-up wordt opgeraapt, ofwel een toren wordt geplaatst in het speelveld.
`keypress`	(symbol → $\emptyset$) Deze procedure wordt aangeroepen vanuit het `Screen ADT` om aan te geven dat er een toets werd ingedrukt, en zal de gepaste actie uitvoeren: tank plaatsen (indien mogelijk) met `t`, obstakels plaatsen (indien mogelijk) met `o`, en `space` om de volgende ronde te starten of het level te beëindigen.
`update!`	(number → $\emptyset$) Deze procedure roept de updatefunctie van alle `Monster`s, `Tower`s, `Projectile`s, `DroppedItems` en de `PowerUpsDelegate` aan, en controleert ook steeds de voortgang in de huidige ronde. Het enigste argument (dat ook wordt doorgegeven aan de andere ADTs) is het aantal verstreken milliseconden sinds laatste update.
`get-game`	`Game` Deze procedure geeft simpelweg de `Game` terug waarin het `Level` werd aangemaakt. Dit maakt het mogelijk voor ADTs die wel toegang hebben tot het `Level ADT` maar niet rechtstreeks tot het `Game ADT`, om toch bijvoorbeeld geld en levens aan te passen in dat `Game ADT`.
`can-place-tower?`	(`Position` boolean → boolean) Deze procedure controleert of het plaatsen van een `Tower` op de meegegeven `Position` mogelijk is. Dit wordt niet alleen intern gebruikt door de `mouseclick` procedure, maar ook extern door het `Screen ADT` dat instaat voor het hoveren (meebewegen met de muisaanwijzer) van torens over vrije plaatsen in het speelveld. Het tweede argument bepaald of er op het scherm een error moet worden getekend wanneer het plaatsen van de `Tower` niet mogelijk is (`#f` bij het hoveren, `#t` bij het plaatsen).
`add-projectile!`	(`Projectile` → $\emptyset$) Deze procedure wordt door de `Tower`s opgeroepen om een `Projectile` toe te voegen.
`add-dropped-item!`	(`DroppedItem` → $\emptyset$) Deze procedure wordt door de `Monster`s (met een bepaalde kans) opgeroepen om een `DroppedItem` achter te laten.
`remove-monster!`	(`Monster` → $\emptyset$)
`remove-projectile!`	(`Projectile` → $\emptyset$)
`remove-tank!`	(`Tank` → $\emptyset$)
`remove-power-up!`	($\emptyset \to \emptyset$)
`remove-dropped-item!`	(`DroppedItem` → $\emptyset$) Deze procedure verwijdert het meegegeven object uit het `Level` en van het `Screen`. Bij `remove-power-up!` wordt niets meegegeven aangezien maar 1 power-up op hetzelfde moment kan actief zijn, en deze dus wordt verwijderd.
`get-path`	`Path` Deze procedure geeft het `Path` mee dat wordt gebruikt in het huidige `Level` (indien dat reeds werd geïnitialiseerd door `start!` op te roepen).

2.4 Position ADT

Het `Position ADT` bevat informatie over een bepaalde plaats op het speelveld. Het speelveld wordt conceptueel opgedeeld in een n aantal rijen en m aantal kolommen (n en m zijn constanten die kunnen aangepast worden in `constants-and-auxfunctions.rtk`). Hierdoor kan het speelveld worden voorgesteld als een rooster (of 'grid') met in het totaal $i \times j$ cellen.
Zowel achtergrond-tiles, `Tower`-tiles als `PathCell`-tiles bedekken één volledige cel wanneer aanwezig op het speelveld. Hun `Position` kan bijgevolg eenvoudig worden voorgesteld door hun overeenkomstige (x_g, y_g)-coördinaat in het rooster, met $x_g \in 0, 1, 2, ..., m$ en $y_g \in 0, 1, 2, ..., n$, en waarbij $(0, 0)$ de linkerbovenhoek van zo'n cel is.
Echter, `Monster`-tiles en `Projectile`-tiles dienen zicht op een flexibele manier voort te bewegen, zonder gebonden te worden aan het gelimiteerd mogelijk aantal posities in een rooster. Hiervoor werden relatieve waarden geïntroduceerd, namelijk x_r- en y_r met $x_r, y_r \in [0, 1[$ waardoor nu ook elke mogelijke positie binnen een cel kan omschreven worden ($x_g \leq x_g + x_r < x_g + 1$ en $y_g \leq y_g + y_r < y_g + 1$). Deze twee relatieve waarden zijn standaard gelijk aan 0, tenzij anders gespecificeerd.
Een `Position` bestaat dus uit vier getallen: een x_g en y_g die de positie van een cel in het conceptuele rooster aangeven, en een x_r en y_r die binnen deze cel de relatieve positie aangeven. Hierdoor blijft het rooster-systeem behouden, maar is er ook flexibelere positionering mogelijk. Bijgevolg kan elke positie op het speelveld worden omschreven a.d.h.v. een `Position`.
Dit brengt uiteraard wat nieuwe complexiteit met zich mee. Wanneer een `Position` wordt gemuteerd (door bijvoorbeeld `move!`), zou het kunnen dat de constraints van x_r en y_r worden doorbroken, i.e. dat hun waarde onder 0 zakt of groter (of gelijk aan) 1 wordt. Dit wordt opgelost door hierop adequaat te controleren en indien nodig de `Position` zodanig te veranderen dat x_r en y_r weer aan hun constraints voldoen (cf. `check-update-position!`).
Merk op dat in de code x_g en y_g worden aangeduid als `x-grid` en `y-grid`, x_r en y_r als `x-rel` en `y-rel`, en $x_g + x_r$ en $y_g + y_r$ als `x` en `y`.

Naam	**Signatuur** en **Omschrijving**
`make-position`	(number number → `Position`) *Constructor.* Op basis van een initiële x- en y-waarde (in het spelgrid) wordt een nieuw `Position`object aangemaakt.
`get-x` `get-y`	number number Deze *accessoren* geven respectievelijk de x- en y-waarden van de gridcel van de `Position` terug, (i.e., de teruggegeven waarde zal een integer zijn).
`get-grid-x` `get-grid-y`	number number Deze *accessoren* geven respectievelijk de x- en y-waarden van de `Position` terug, waarbij ook de relatieve positionering is inbegrepen (i.e., de teruggegeven waarde zal decimalen bevatten).
`move!`	(`Movement` number → number) Deze *mutator* verwacht als argumenten een `Movement` en het aantal verstreken milliseconden sinds de laatste update. Op basis hiervan wordt de nodige relatieve $x_{r_{move}}$ en $y_{r_{move}}$ verandering berekent, waarna deze variabelen worden gemuteerd (m.b.v. `set-x-rel!` en `set-y-rel!`). De totaal afgelegde afstand $\sqrt{{x_{r_{move}}}^2 + {y_{r_{move}}}^2}$ wordt uiteindelijk teruggegeven.
`compare?`	(`Position` → boolean) Deze procedure gaat na of de huidige `Position` en de meegegeven `Position` zich in dezelfde gridcel bevinden.
`object-at-position-with-dimensions?`	(list<V> → number)

	Op basis van een lijst met objecten (V) die de dispatchmessage 'get-position en 'get-dimensions verstaan zal deze procedure het eerste object in deze lijst teruggeven waarbij de huidige `Position` binnen de `Position` met `Dimension`s van het object (V) geloceerd is.
`object-in-position?`	({list<V> vector<V>} → number) Deze procedure geeft het eerste procedureobject (V) in de gegeven lijst of vector waarvan de `Position` zich in dezelfde gridcel bevindt als de huidige `Position`. Merk op dat elk object V de dispatchmessage 'get-position moet verstaan. Wanneer een vector wordt meegegeven als 1ste argument kan ook optioneel nog een 2de argument worden meegegeven, namelijk de index vanaf waar moet worden gezocht (voor performantieverbeteringen).
`distance-to-other-position`	(`Position` → number) Deze procedure geeft de afstand tussen de huidige `Position` en de meegegeven `Position`.
`make-center-position`	(∅ → `Position`) Aangezien posities op het speelveld de linkerbovenhoek van de spelelementen aangeven, zijn we soms ook geïnteresseerd in het middelpunt van deze elementen. Dit is nuttig voor bijvoorbeeld een `Tower`, die haar `Projectile`s uiteraard vanuit het centrum van de toren wilt laten schieten, en niet vanuit de linkerbovenhoek. Deze procedure geeft dat middelpunt terug.
`make-anchored-position`	(`Position` → number) Deze procedure de linkerbovenhoekspositie van de huidige `Position` terug. Dit is nodig bij het plaatsen van een `Tower`, aangezien de muisklik kan hebben plaatsgevonden op eender welke `Position` binnen de gridcel (maar de toren moet zelf de hele cel vullen en dus in de linkerbovenhoek worden geplaatst).

2.4.1 Movement ADT

Het `Movement ADT` bevat informatie over de beweging van het ADT waarin het geëncapsuleerd is. Het wordt aangemaakt met een bepaalde snelheid (`speed`), en de totaal af te leggen (relatieve) x- en y-afstand (`total-x-distance` en `total-y-distance`). Uiteraard is het onmogelijk om met deze waarden gestandaardiseerde bewegingen te laten plaatsvinden. Daarom worden intern de waarden `normalised`, `x-distance` en `y-distance` bijgehouden. Dit zijn de genormaliseerde afstanden die worden berekend op basis van de `total-x-distance` en `total-y-distance`, en laten toe om alle bewegingen gestandaardiseerd te laten verlopen. Deze genormaliseerde waarden moeten berekend worden bij aanmaak van het ADT, maar ook elke keer dat de totaalafstanden worden geüpdatet (door `set-total-distances!`). Het ADT wordt gebruikt door de `Position`s van het `Projectile` en het `Monster ADT` om zich voort te bewegen.

Naam	**Signatuur** en **Omschrijving**
`make-movement`	(number number number → `Movement`) *Constructor.* Bij aanmaak van het ADT worden 3 getallen verwacht als argumenten: de `speed`, de `total-x-distance` en de `total-y-distance`. Op basis hiervan wordt ook meteen `update-movement!` opgeroepen om de genormaliseerde afstanden te kunnen berekenen.
`get-speed`	number Deze *accessor* geeft de snelheid terug zoals meegegeven bij constructie.
`get-x-movement` `get-y-movement`	number number Deze *accessoren* worden gebruikt door een `Position` om te bewegen, en geven de relatieve x- en y-waarden terug waarmee moet worden bewogen. Hiervoor vermenigvuldigen ze de genormaliseerde x/y-afstand met de `speed`.
`set-total-distances!`	(number number → ∅)

	Deze procedure *muteert* de totale te bewegen afstanden (i.e., `total-x-distance`, `total-y-distance`). Na deze mutatie worden uiteraard de genormaliseerde waarden herberekend.
`mirror-movement`	($\emptyset \rightarrow$ `Movement`) Deze procedure maakt een nieuw `Movement` object door de huidige beweging te spiegelen naar de tegenovergestelde richting, en wordt gebruikt wanneer `Monsters` achteruit worden gevuurd door een kanonskogel-`Projectile`.
`slowed-down-movement`	($\emptyset \rightarrow$ `Movement`) Deze procedure maakt een nieuw `Movement` object door de huidige beweging te vertragen. Dit wordt gebruikt wanneer `Monsters` geraakt worden door een net-`Projectile`.
`sped-up-movement`	($\emptyset \rightarrow$ `Movement`) Deze procedure maakt een nieuw `Movement` object door de huidige beweging te versnellen. Dit wordt gebruikt wanneer `Monsters` achter worden gevuurd door een kanonskogel-`Projectile`, wat natuurlijk met een hoge snelheid gebeurd.

2.5 Path ADT

In het begin van elk level moet het unieke en vooraf gedefinieerde `Path` worden aangemaakt. Een padvector bestaat uit een bepaald aantal `PathCells`, die elk een deel zijn van het pad op een bepaalde `Position` van het speelveld. Bij aanmaak wordt het pad gegenereerd m.b.v. een instructielijst. Het eerste element van de instructielijst is de rijhoogte van de `Position` van de eerste `PathCell` van het pad. De volgende elementen zijn `symbols` die de uitgangsrichting van opeenvolgende cellen aangeven. 3 mogelijke `symbols` zijn toegelaten, namelijk `'R` (right), `'U` (up), en `'D`. Merk op dat een pad steeds links moet beginnen en rechts moet eindigen.

Naam	**Signatuur** en **Omschrijving**
`make-path`	(pair $\rightarrow$ `Path`) *Constructor.* Deze aanmaakprocedure verwacht een lijst met instructies die nodig is bij het genereren van het pad (cf. supra).
`get-path-vector`	vector<`PathCell`> Deze *accessor* geeft het `Path` terug als een vector bestaande uit alle `PathCells`.
`for-each-pathcell`	(procedure $\rightarrow \emptyset$) Deze procedure zal de meegegeven procedure uitvoeren op elke `PathCell` in de padvector.
`get-nearest-pathcell?`	(`Position` $\rightarrow$ (boolean $\cup$ `PathCell`)) Deze procedure geeft de `PathCell` terug die het dichtst bij de gegeven `Position` ligt, indien deze bestaat.
`get-pathcell-from-index`	(number $\rightarrow$ `PathCell`) Deze procedure geeft de `PathCell` terug die op een meegegeven index staat in de padvector.
`get-path-length`	($\emptyset \rightarrow$ number) Deze procedure geeft de lengte van de padvector terug.

2.5.1 PathCell ADT

Een `PathCell` is één stuk van een `Path`. Het heeft een unieke `Positie` op het speelveld, zonder relatieve waarden, wat dus betekent dat het gebonden is aan de conceptuele 'grid'. Elke `PathCell` bestaat uit een `entry`-symbool en een `exit`-symbool, die de richting van respectievelijk de ingang en de uitgang aangeven. De mogelijke `entry`-symbolen zijn `'left`, `'down` en `'up`, en de mogelijke `exit`-symbolen zijn `'right`, `'down` en `'up`. Deze informatie is essentieel voor de `Monsters` en `Tanks` die over deze `PathCell` bewegen, om te weten 'hoe' zij zich moeten voortbewegen.

Naam	**Signatuur** en **Omschrijving**
`make-pathcell`	(symbol symbol `Position` boolean boolean → `PathCell`) *Constructor.* Deze procedure maakt een `PathCell` aan op basis van een entry- en exit-symbool, de `Position` van de cel en twee booleans die aangeven of het de eerste en of het de laatste cel van het pad is.

`get-entry`
`get-exit`
`get-position`
	symbol
	symbol
	`Position`)
	Deze procedures geven respectievelijk het entry- en exit-symbool terug, en de `Position` van de `PathCell`.

`is-first?` `is-last?`	boolean boolean Deze procedures geven terug of de `PathCell` de eerste / de laatste van het pad is, respectievelijk. Deze booleans werden reeds als argumenten meegegeven bij constructie.
`is-slowed?` `set-slowed!`	boolean (boolean → ∅) Deze procedures zullen de vertragingsstaat van de `PathCell` respectievelijk *accesseren* en *muteren*. Een `PathCell` komt in vertraagde staat wanneer er een net-`Projectile` op komt te liggen, waardoor alle `Monster`s die er overheen gaan trager zullen bewegen.
`has-active-obstacle?` `set-active-obstacle!`	boolean (boolean → ∅) Deze procedures zullen de obstakelstaat van de `PathCell` respectievelijk *accesseren* en *muteren*. Een `PathCell` kan geblokkeerd zijn wanneer de `Obstacle` power-up is geactiveerd, waardoor `Monster`s hier niet voorbij kunnen.

2.6 Monster ADT

Een `Monster` is een spelelement dat tijdens een ronde (`wave`) door `start-wave!` (in het `Level ADT`) wordt aangemaakt en getekend in het begin van het `Path`. `Monster`s kiezen bij het betreden van elke `PathCell` een willekeurige uitgangs`Position` aan de uitgangskant van de `PathCell`, en proberen zo tot het einde van het `Path` te geraken zonder te sterven. Elk `Monster` heeft een vast aantal levens en elk type `Monster` heeft bepaalde eigenschappen.

Naam	**Signatuur** en **Omschrijving**
`make-monster!`	(`PathCell Level` number . boolean → `Monster`) *Constructor.* Op basis van de initiële `PathCell`, het `Level` en het type wordt een `Monster` object aangemaakt. Het 4de argument is optioneel een boolean die aangeeft of het `Monster` direct moet tevoorschijn komen (true) of op een willekeurige afstand voor de eerste `PathCell` buiten het speelveld moet beginnen (false – default).
`update!`	(number `Path Screen` list → ∅) Deze procedure update (indien nodig) de `Position` van het `Monster`, en zal ook het `Shield` updaten (indien nodig).

`get-position`
`get-movement`
`get-underlying-pathcell`
`get-lives`
`get-exit-direction`
`get-type`
	`Position`
	`Movement`
	`PathCell`
	number
	symbol
	number
	Deze procedures *accesseren* de `Position`, `Movement`, onderliggende `PathCell`, het aantal levens, de uitgangspositie en het type, respectievelijk, van het `Monster`.

`hurt!`	(number list<`Monster`> boolean boolean → symbol) Deze procedure wordt aangeroepen door een `Projectile`, om het `Monster` levels te laten verliezen. Het 1ste argument bepaald het aantal te verliezen levens, het 2de argument is een lijst van `Monster`s (want een paars `Monster` moet bij dood de omliggende `Monster`s een leven kunnen geven), het 3de argument bepaald of het `Monster` ook achteruit moet worden geschoten en het 4de argument bepaald of het `Monster` geld geeft wanneer het sterft.
`hurt-with-tank`	(`Tank` list<`Monster`> → boolean) Wanneer een `Monster` wordt geraakt door een `Tank` wordt deze procedure aangeroepen, die ook levens zal afnemen indien dat niet reeds eerder is gebeurd door diezelfde `Tank`.
`get-dimensions` `set-dimensions!`	`Dimensions` (`Dimensions` → ∅) Deze procedures zullen de dimensies respectievelijk *accesseren* en *muteren*. Deze zijn nodig om bijvoorbeeld collisions met een projectiel te berekenen.
`get-shield`	{`Shield` ∪ boolean} Deze procedure geeft het `Shield` terug indien het `Monster` deze eigenschap heeft.
`add-single-life!`	(∅ → ∅) Deze procedure voegt een enkel leven toe aan het `Monster` en wordt gebruikt door het paarse `Monster` om aan andere `Monster`s in de buurt te geven wanneer het sterft.

2.6.1 Shield ADT

In het spel hebben de gele `Monster`s een `Shield`, waardoor `Projectile`s het `Monster` niet raken. Bij elke aanraking met het `Shield` wordt het wel enkele seconden inactief, en na 3 aanrakingen verdwijnt het.

Naam	**Signatuur** en **Omschrijving**
`make-shield`	(∅ → `Shield`) *Constructor.* Deze procedure maakt een `Shield` object aan.
`update!`	(number → ∅) Deze updateprocedure wordt opgeroepen door het `Monster`. Deze zorgt ervoor dat de interne `Timer` wordt geüpdatet en het schild indien nodig terug wordt geactiveerd.
`hit-shield!`	(∅ → ∅) Deze procedure wordt opgeroepen wanneer een `Monster` wordt geraakt, waardoor het `Shield` (indien actief) wordt gedeactiveerd (al dan niet tijdelijk).
`is-active?`	boolean Deze procedure geeft terug of het `Shield` al dan niet actief is.

2.7 Tower ADT

Een `Tower` is een spelelement dat door een muisklik van de speler in het speelveld kan geplaatst worden. Elke `Tower` is een vierkant dat in precies één cel van het conceptuele rooster past, en kan niet op een `PathCell` of op een andere `Tower` worden geplaatst. Er bestaan 4 soorten `Tower`s, elk met een eigen soort `Projectile`s.

Naam	**Signatuur** en **Omschrijving**
`make-tower`	(`Position` `Level` number → `Tower`) *Constructor.* Op basis van een grid`Position`, het `Level` en het typenummer wordt de `Tower` aangemaakt.
`update!`	(number list<`Monster`> `Path` → ∅) Deze updateprocedure is verantwoordelijk voor het updaten van de interne counter, en moet – wanneer de counter is afgelopen – deze resetten en een `Projectile` lanceren.

`get-position`	`Position`
`get-type`	number Deze procedures *accesseren* respectievelijk de `Position` en het type van de `Tower`.
`get-nearest-pathcell`	(number → ∅) Deze procedure zal de dichtstbijzijnde `PathCell` van de `Tower` teruggeven, waarop netten en bommen (`Projectiles`) kunnen worden gelanceerd.

2.7.1 Projectile ADT

Een `Projectile` is een spelelement dat wordt afgevuurd door een `Tower` in de richting van een `Monster` met een bepaalde `Movement`. Wanneer deze een `Monster` raakt, verliest het `Monster` een leven en verdwijnt het `Projectile`. Wanneer het `Projectile` echter haar (`range`) verlaat moet het ook verdwijnen.

Naam	**Signatuur** en **Omschrijving**
`make-projectile`	(`Level Tower Position Movement` number → `Projectile`) *Constructor.* Deze procedure maakt op basis van het `Level`, de `Tower`, de `Position`, een begin`Movement` en het type een `Projectile` object aan.
`update!`	(number list<`Monster`> `Screen` → ∅) Deze updateprocedure staat in voor het veranderen van het gedrag van het `Projectile` zoals gewenst en afhankelijk van het type. Ook moet na verplaatsing worden gecontroleerd op een eventuele botsing met een `Monster`.
`get-position`	`Position`
`get-type`	number Deze procedures *accesseren* respectievelijk de `Position` en het type van het `Projectile`.
`get-scale`	number Deze procedure geeft de schaal terug waarop een `Projectile` moet worden getekend. Dit wordt gebruikt door het `Screen ADT` om bijvoorbeeld netten vloeiend uit de `Tower` te laten verschijnen.
`set-dimensions!`	(`Dimensions` → ∅) Deze *mutator* stelt de `Dimensions` van het `Projectile` in, wat nodig is voor het detecteren van botsingen met `Monsters`.
`collide!`	(`Monster` → (`Monster` ∪ `#f`)) Deze procedure staat nog steeds in voor het uitvoeren van de botsing, maar verwacht geen `Level` meer aangezien deze al in het ADT werd meegegeven bij constructie.

2.8 Power Ups Delegate ADT

Verschillende Power Ups kunnen worden gebruikt door de speler als extra hulpmiddel om de `Monsters` te verhinderen tot het einde te geraken. Deze Power Ups kunnen op 2 manieren worden verkregen:

1. Door te klikken op Power Ups die door gestorven `Monsters` willekeurige worden achtergelaten.
2. Door deze aan te kopen met geld dat verdiend kan worden door `Monsters` te doden.

Er kan maximaal 1 Power Up tegelijkertijd actief zijn, en na gebruik ervan is er ook een cool-down periode van een aantal seconden. Het overkoepelende ADT dat al dit beheert is het `Power Ups Delegate ADT`.

Naam	**Signatuur** en **Omschrijving**
`make-power-ups-delegate`	(`Game Screen` → `PowerUpsDelegate`) *Constructor.* Op basis van de huidige `Game` en het `Screen` wordt een `PowerUpsDelegate` object aangemaakt. Initieel zijn er nog geen Power Ups ingezameld of actief.
`update!`	(number `Screen Level` list<`Monster`> → ∅) Deze procedure update de actieve Power Up of – indien er geen Power Up actief is – werkt de cool-down`Timer` bij.

`use-power-up!`	(symbol `Path` → {boolean ∪ `Tank` ∪ `ObstacleDelegate`}) Deze procedure wordt opgeroepen wanneer een Power Up wordt ingezet. Het type wordt gespecificeerd door het 1ste argument ('tank of 'obstacle-delegate) en ook het `Path` wordt meegegeven aangezien deze Power Ups op het pad moeten worden geplaatst.
`collect!`	(symbol boolean → ∅) Deze procedure voegt een nieuwe Power Up toe aan de verzameling van inzetbare Power Ups. Het type wordt bepaald door het eerste argument ('tank of 'obstacles), en of de Power Ups eerst moet worden aangekocht wordt bepaald door het 2de argument.
`get-active-tank` `get-active-obstacle-delegate`	{`Tank` ∪ boolean} {`ObstacleDelegate` ∪ boolean} Deze *accessoren* geven respectievelijk de actieve `Tank` en actieve `ObstacleDelegate` terug, indien deze actief zijn.
`get-collected-tanks` `get-collected-obstacles`	number number Deze *accessoren* geven respectievelijk het aantal verzamelde tanks en obstakels terug.
`deactivate-power-up!`	(boolean boolean → ∅) Deze *mutator* schakelt de huidige Power Up uit, en past de status aan op basis van de 2 argumenten (1ste argument: nieuw level?, 2de argument: einde game?). Als een nieuw level begint wordt de status 'available, als het spel eindigt 'unavailable en anders 'cool-down.

2.8.1 Tank ADT

Het `Tank ADT` is een Power Up waarbij – na activatie – en tank verschijnt aan het begin van het `Path`, waarna het de `Monster`s op dat `Path` overrijdt en hun eenmalig 1 leven afneemt. Als een `Monster` hierdoor sterft brengt het geen extra geld op.

Naam	**Signatuur** en **Omschrijving**
`make-tank`	(`Path` → `Tank`) *Constructor.* Deze procedure staat in voor de aanmaak van een `Tank`, a.d.h.v. een gegeven `Path` om de af te leggen we te kunnen bepalen.
`update!`	(number `Screen Level` list¡`Monster`¿ → ∅) Deze updateprocedure is verantwoordelijk voor het verplaatsen van de `Tank` bij elke update. Hierbij moeten ook de geraakte `Monster`s eenmalig een leven verliezen.
`get-position` `get-exit`	`Position` symbol Deze *accessoren* geven respectievelijk de `Position` en de uitgangspositie (richting) van de `Tank`. Dat laatste is nodig om de oriëntering van de `Tank` te kunnen bepalen.

2.8.2 ObstacleDelegate ADT

Het `ObstacleDelegate ADT` is verantwoordelijk voor het willekeurig plaatsen van `Obstacle`s op het `Path`. Ook wordt er een interne `Timer` bijgehouden aangezien de actieve tijd van elke Power Up beperkt is.

Naam	**Signatuur** en **Omschrijving**
`make-obstacle-delegate`	(`Path` → `ObstacleDelegate`) *Constructor.* Deze procedure staat in voor de aanmaak van een `ObstacleDelegate`, a.d.h.v. een gegeven `Path` om hierop de `Obstacle`s willekeurig te kunnen positioneren.
`update!`	(number `Screen Level` list¡`Monster`¿ → ∅) Deze procedure is verantwoordelijk voor het updaten van de `Timer`. Indien deze is afgelopen moeten de `Obstacle`s worden verwijderd.

`get-obstacles`	vector<`Obstacle`>
`get-time-left`	number
	Deze *accessoren* geven respectievelijk een vector met alle `Obstacle`s en de de resterende actieve tijd terug. Dat laatste is nodig om aan de speler te kunnen tonen hoelang de `Obstacle`s nog actief zijn.

2.8.3 Obstacle ADT

Het `Obstacle ADT` is een enkel obstakel dat door de `ObstacleDelegate` wordt geplaatst op een willekeurige `PathCell`. `Monster`s die hiertegen botsen kunnen niet verder bewegen.

Naam	**Signatuur** en **Omschrijving**
`make-obstacle`	(`PathCell` symbol → `Obstacle`)
	Constructor. Deze procedure staat in voor de aanmaak van een `Obstacle`, op basis van de `PathCell` waarop deze geplaatst is (1ste argument) en de oriëntatie van het obstakel (2de argument ∈{'horizontal, 'verical})
`get-position`	`Position`
`get-orientation`	symbol
	Deze *accessoren* geven respectievelijk de `Position` en oriëntatie van het `Obstacle` terug.
`get-dimensions`	`Dimensions`
`set-dimensions!`	(`Dimensions` → ∅)
	Deze procedures zijn verantwoordelijk voor het *accesseren* en *muteren* van de `Dimensions`. Dit is nodig om te kunnen detecteren wanneer `Monster`s botsen met dit `Obstacle`.

2.8.4 DroppedItem ADT

Wanneer een `Monster` sterft wordt met een bepaalde kans een `DroppedItem` achtergelaten. Dit is een Power Up (tank of obstakels) die kan worden opgeraapt door erop te klikken. Na een bepaalde tijd verdwijnt dit element echter, en kan het niet meer worden opgeraapt.

Naam	**Signatuur** en **Omschrijving**
`make-dropped-item`	(`Position` → `DroppedItem`)
	Constructor. Een `DroppedItem` object wordt aangemaakt op een bepaalde `Position`. Hierbij wordt ook meteen een `Timer` geïnitialiseerd zodat het element op tijd van het `Screen` kan verdwijnen.
`update!`	symbol
	Deze procedure staat in voor het updaten van de interne `Timer` en het deactiveren van zichzelf wanneer deze `Timer` is afgelopen.
`get-position`	`Position`
`get-type`	symbol
	Deze *accessoren* geven respectievelijk de `Position` en het type van het `Obstacle` terug.
`get-dimensions`	`Dimensions`
`set-dimensions!`	(`Dimensions` → ∅)
	Deze procedures zijn verantwoordelijk voor het *accesseren* en *muteren* van de `Dimensions`. Dit is nodig om te kunnen detecteren wanneer er op de visuele representatie van dit object wordt geklikt op het speelveld.

2.9 Hulpobjecten

De onderstaande ADTs dienen als extra hulpabstracties voor veel voorkomende toepassingen.

2.9.1 ObjectList ADT

Een `ObjectList` wordt gebruikt door het `Level ADT` om spelobjecten V met hun aantal bij te houden.

Naam	**Signatuur** en **Omschrijving**
`make-object-list`	$(\emptyset \rightarrow \emptyset)$ *Constructor.* Deze procedure maakte een lege `ObjectList` met lengte 0 aan.
`get-count` `get-objects`	number list<V> Deze procedures geven respectievelijk het aantal objecten en de objecten als een lijst terug.
`add-object!` `remove-object!`	(V → boolean) $(V \rightarrow \emptyset)$ Deze procedures zijn verantwoordelijk voor het toevoegen en verwijderen van een object, respectievelijk. Ze passen hierbij ook de interne 'count' aan.
`contains-object?!`	(V → boolean) Deze procedure gaat na of een gegeven object V aanwezig is in de `ObjectList`.

2.9.2 Timer ADT

Het `Timer ADT` wordt gebruikt wanneer een object een interne counter nodig heeft. Deze bevat een updateprocedure waarmee kan worden afgeteld. Dit wordt o.a. gebruikt door de `Tower`s (om met een bepaalde frequentie `Projectile`s af te vuren) en de `ObstacleDelegate` (om slechts een bepaalde tijd `Obstacle`s actief te houden).

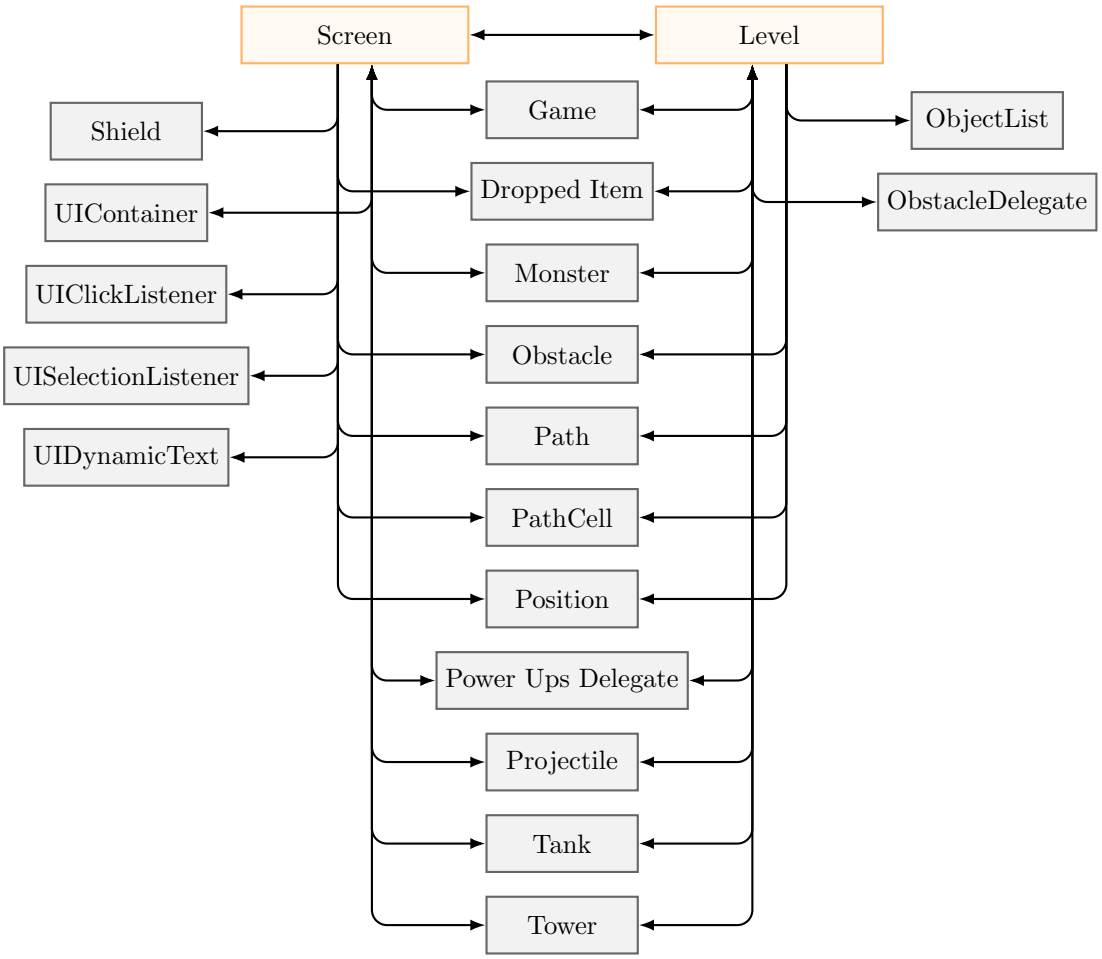
Naam	**Signatuur** en **Omschrijving**
`make-timer`	(number → $\emptyset$) *Constructor.* Aan de hand van een starttijd wordt het `Timer` object aangemaakt.
`update!`	(number → $\emptyset$) Deze procedure wordt opgeroepen met het aantal milliseconden dat moet worden afgetrokken van de timer, waardoor deze dus wordt geüpdatet.
`get-time` `set-time!`	number (number → $\emptyset$) Deze procedures *accesseren* en *muteren* de timer.
`ended?`	($\emptyset$ → boolean) Deze procedure gaat na of de tijd is afgelopen.

3 Afhankelijkheidsdiagram

In Figuur 1, 2 en 3 wordt de afhankelijkheid tussen de verschillende ADTs visueel voorgesteld. Wanneer ADT a afhankelijk is van ADT b, wordt dit voorgesteld met een pijl van a naar b. In Sectie 2 worden alle ADTs beschreven, waaruit hun afhankelijkheid van andere ADTs kan blijken. Het afhankelijkheidsdiagram werd opgesplitst in 3 diagrammen om het overzichtelijk te houden:

- **Figuur 1** bevat alle ADTs die afhankelijk zijn van het `Level` en/of `Screen` ADT, alsook de ADTs waarvan het `Level` en/of `Screen` zelf afhankelijk zijn.
- **Figuur 2** bevat alle ADTs die te maken hebben met de Power Ups, samen met hun afhankelijkheden.
- **Figuur 3** bevat alle andere ADTs met hun afhankelijkheden.

Het afhankelijkheidsdiagram wijkt zeer veel af van de voorspelde afhankelijkheden zoals beschreven in de voorstudie van fase 2. Dit komt voornamelijk omdat veel van de ADTs die daar werden beschreven uiteindelijk significant aangepast zijn. Ook zijn er verschillende ADTs toegevoegd die daar niet beschreven stonden.

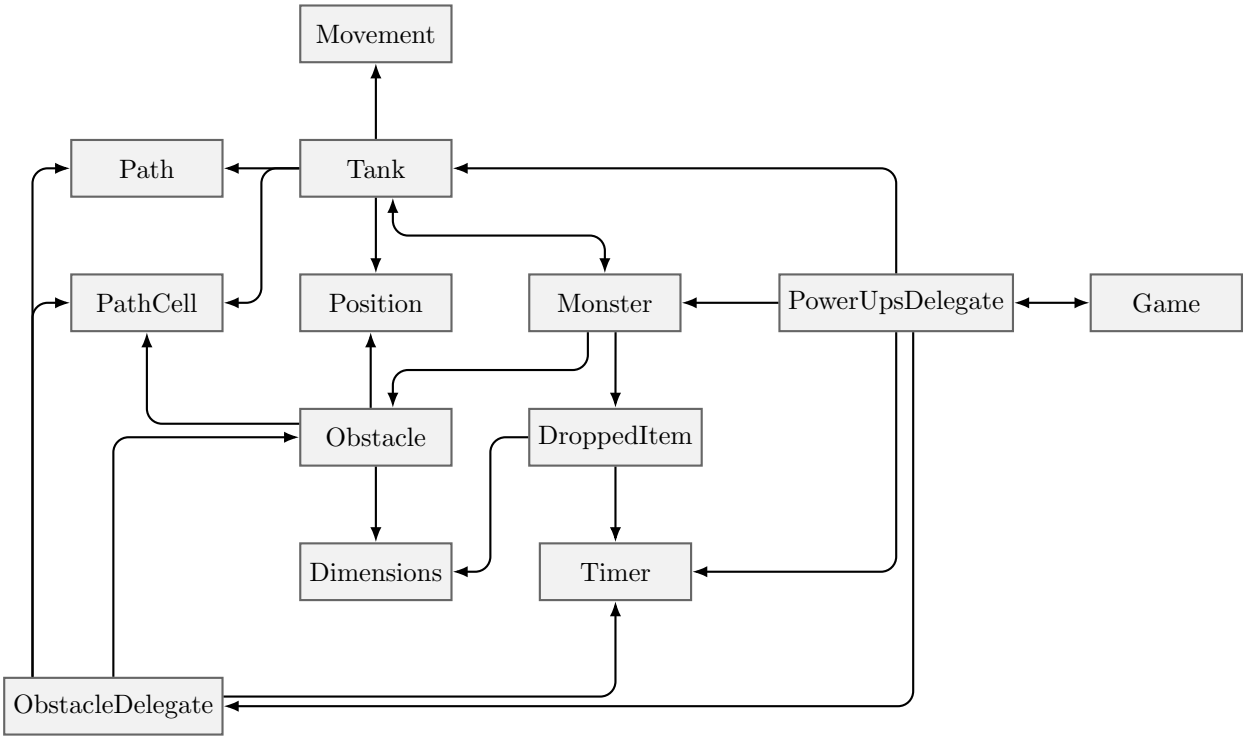


Figuur 1: Level en Screen – afhankelijkheden

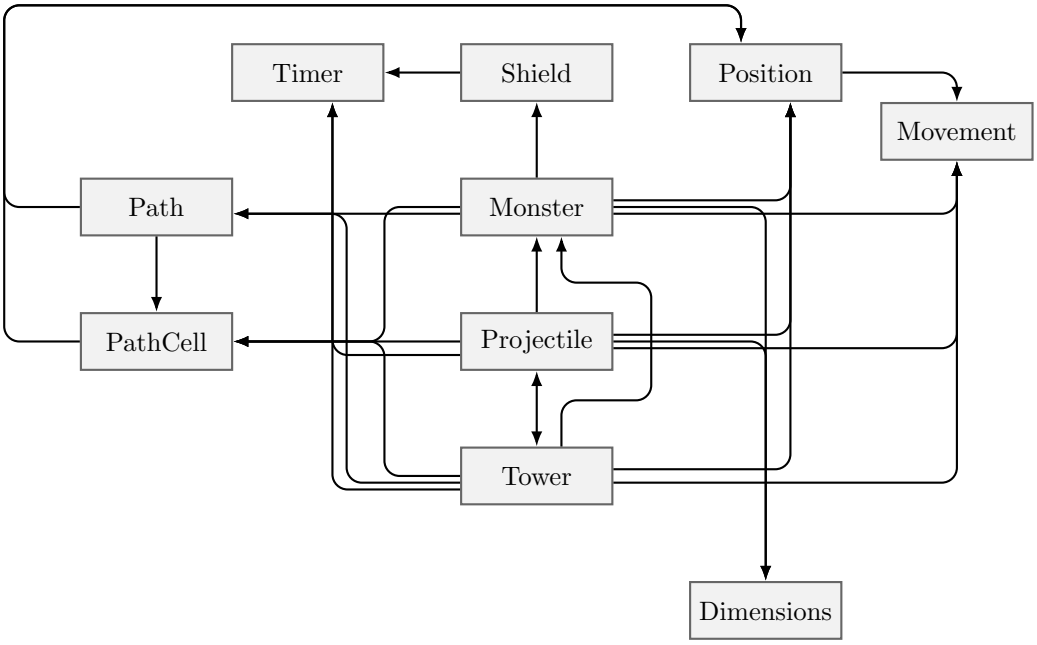
4 Planning

Week	Omschrijving
26	Voorstudie gemaakt.
27	Hulpobjecten geïmplementeerd (`ObjectList` en `Timer`) en andere verbeteringen aangebracht op basis van de feedback tijdens de mondelinge verdediging.
28	Geld en levens geïmplementeerd. Nieuwe drawables gemaakt voor de monsters, torens, projectielen, powerups en user interface.
29-31	Verschillende soorten `Tower`s met hun specifieke `Projectile`s geïmplementeerd.
31-32	Verschillende soorten `Monster`s geïmplementeerd.
33-34	Power Ups geïmplementeerd (`PowerUpsDelegate`, `Tank`, `ObstacleDelegate`, `Obstacle`, `DroppedItem`).
35	Meerdere levels en meerdere waves geïmplementeerd.
35-37	User Interface constructie vereenvoudigd (`UIContainer`, `UIClickListener`, `UISelectionListener` en `UIDynamicText`). Verslag geschreven.

De uiteindelijke implementatie van het spel volgde min of meer de opgestelde planning (cf. voorstudie fase 2). Echter nam het implementeren van de User Interface en het schrijven van het verslag veel meer tijd in beslag dan ingeschat.



Figuur 2: Power Up objecten – afhankelijkheden



Figuur 3: Overige afhankelijkheden