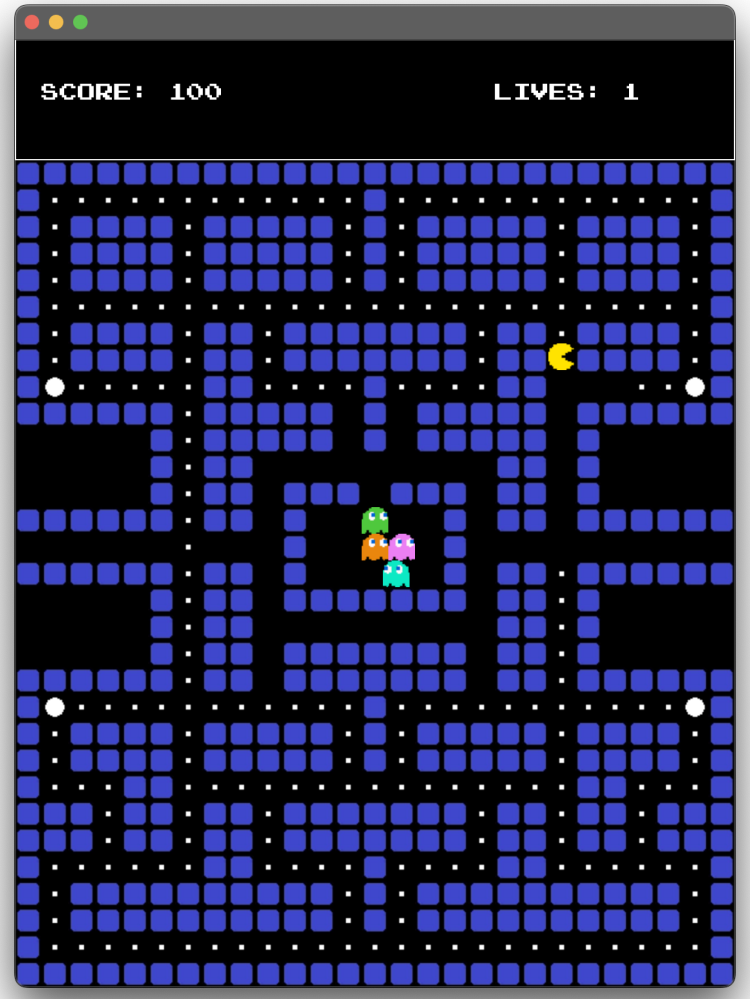
Objektorienteret Programmering Projekt PacMan

Andreas K. L. Aske W. F. Magnus R. K.

27. maj 2025



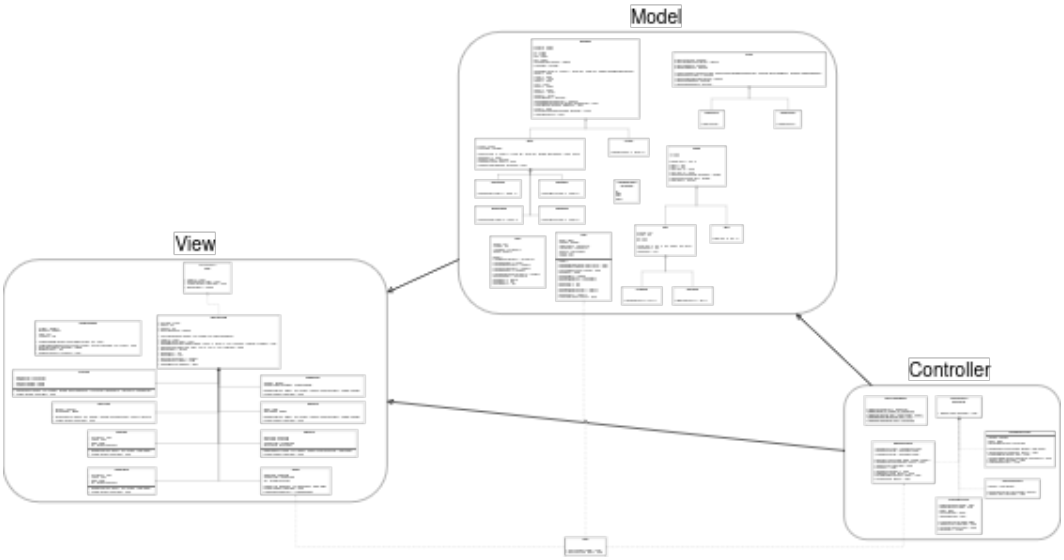
Indhold

1 Projektbeskrivelse

Til fordel for at sikre en så tro kopi til orignalen som overhovedet muligt, er dette projekts hovedformål, at udvilke spilfunktionalitet mht. kravsspecifikationen. Målet herefter, er at udvide både spillets funktionalitet samt dets brugervenlighed.

Eventuelle afvigelser fra kravsspecifikationen ses dokumenteret/diskuteret i det følgende.

2 design



Figur 1:

Projekts design følger *MVC* (Model-View-Controller) modellen. Det vil sige at vores *Model* repræsenterer hvordan PacMan spillet er bygget op med spøgelser, væge, piller osv. Så har vi vores *Controller* som står for alt logikken med hvordan ting skal kolliderer og bevæge sig, og hvornår de forskellige stadier af spillet sker. Til sidst har vi vores *View* som står for at vise spillet, med alle billederne og animationerne, samt score tekst og liv osv.

Designet kan ses i vores *UML*-diagram, hvor man kan se vi har opdelt koden i de tre dele fra *MVC* modellen, samt en *main* fil til at starte spillet, og initialiserer de andre klasser.

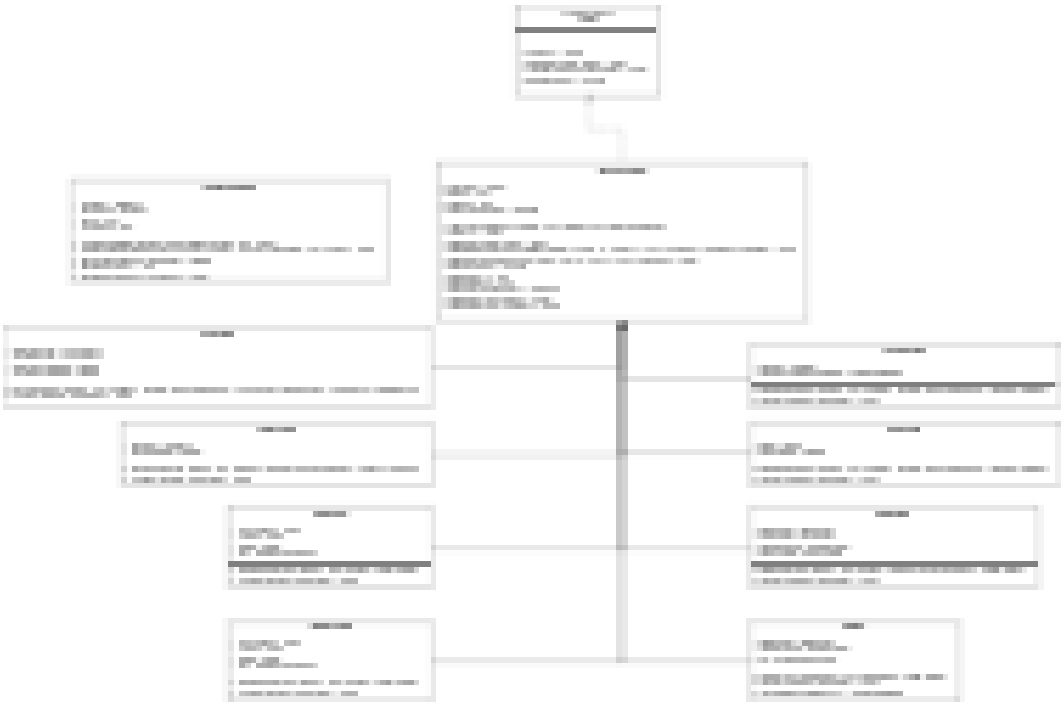
Vi har designet alt vores kode med fokus på *indkapslingsprincippet*. Alle felter i klasser er private, og har kun *getters* og *setters* der hvor det er nødvendigt. Vi har også overholdt *DRY* princippet, ved at samle ens opførsel i fælles *superklasser*.

2.1 Model

Figur 2: UML-diagram til *Model* delen af projektet.

I *Model* (se Figur 2) har vi lavet et abstrakt klasse der hedder `Moveable`, som er en abstrakt enhed der kan bevæge sig. Dette tillader os at *nedarve* fra den når vi skal lave ting der skal bevæge sig som `Ghost` og `PacMan`. Ud over dette er `Ghost` også en abstrakt klasse, så vi kan udvide de enkelte spøgelser fra den. Udover `Moveable` klasse, har vi også en abstrakt klass, `Pos2D`, til at beskrive positioner som ikke skal kunne bevæge sige. Fra denne klasse kan vi så nedarve klasser som `Pill` og `Wall`. På denne måde benytter vi *klasseafhængighedsprincippet* til at simplificerer koden, og gøre det nemmere at udvide med nye features.

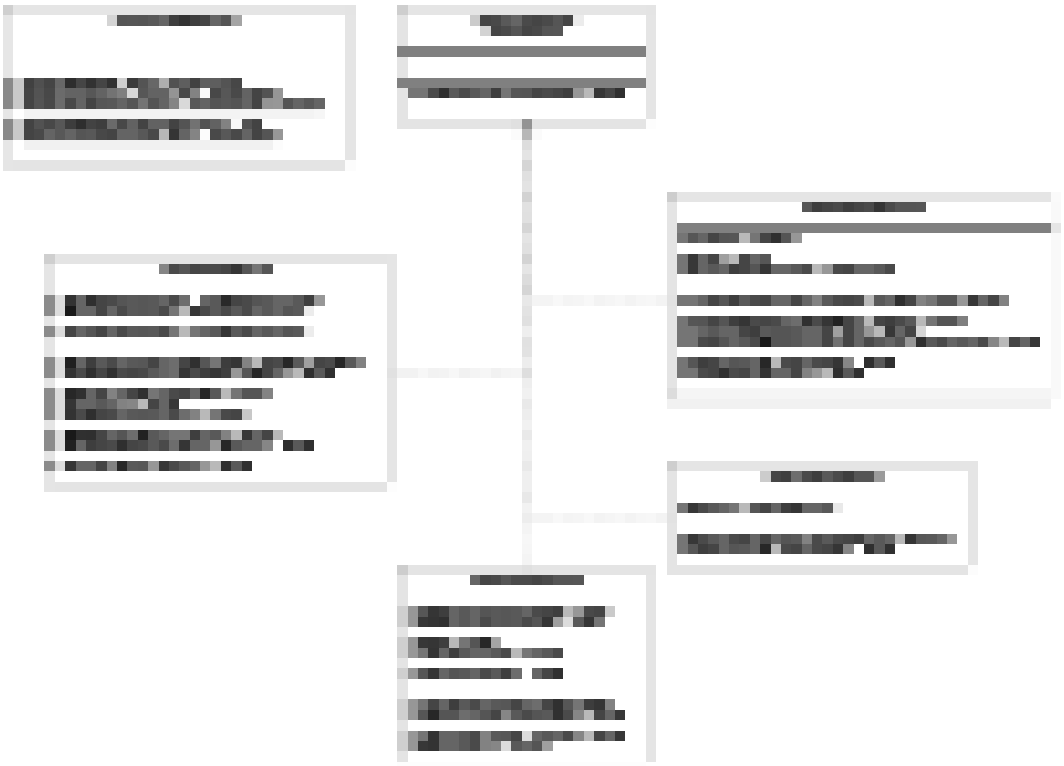
2.2 View



Figur 3: UML-diagram til *View* delen af projektet.

I *View* (se Figur 3) har vi benyttet en *grænseflade* ved navn `View` som specificere hvilke metoder et *View* skal have. Så har vi lavet en abstrakt klasse `AbstractView`, som implementerer dette interface. Vi nedarver på denne måde fra det abstrakte *View* hver gang vi laver et nyt *View* som står for at vise noget andet. Så kan vi lave specifikke views som står for at tegne kun én slags ting, f.eks. `PacManView`. På denne måde benytter vi princippet om et enkelt ansvar.

2.3 Controller



Figur 4: UML-diagram til *Controller* delen af projektet.

Til *Controller* (set Figur 4) har vi også lavet en grænseflade, for at specificerer hvad en "Controller"gør. Så har vi lavet controllers til at styre henholdsvis PacMan, spøgelser og de forskellige stadier af spillet. Med dette har vi lavet en `MainController` som bestemmer hvornår alle de andre controllers skal kaldes, og hvornår vores *View* bliver kaldt.

2.4 Ændringer

I løbet af projektet har vores design ændret sig en del. I starten var vores UML-diagram meget mere simpelt, som kan ses på Figur 5. Enkeltheden er både grundet de manglende felter og metoder, men også at kompleksiteten af programmet ikke var blevet realiseret endnu. `View` var bare én enkelt klasse som stod for at tegne alt, og `Controller` stod kun for at tage imod input fra brugeren, og bevæge pacman i `Maze`. Alt andet logik til spillet var i `Maze`, som gjorde at det fyldte rigtig meget. Da vi så begynde at implementerer mere, lagde vi mærke til hvor god en idé det ville være at splitte det endnu mere op, og specialisere klasserne vi laver, som er idéen med *Single Responsibility Principle*.

Figur 5: Det første UML-diagram til projektet.

I et forsøg på at simplificerer `Maze` delte vi den op, og lavede en `Game` klasse, hvis formål var at repræsenterer spillet. På denne måde kunne `Controller` stå for at håndterer alt logikken, som også er mere typisk af en *Controller* at gøre i *MVC*-modellen. Med denne ændring ville maze bare stå for at repræsenterer labyrinten. UML-diagrammet for denne forbedring kan ses på Figur 6.

Figur 6: Det andet UML-diagram til projektet.

3 Implementation

- Formålet med denne rapportsektion er at give den interesserede læser et overblik over de interessante implementationsdetaljer, som er værd at kigge nærmere på i jeres kodebase, samt nødvendige detaljer for at køre jeres kode.
- Angiv hvilken version af Java I har brugt til at teste og kompilere jeres kode, og inkludérkorte instruktioner til hvordan man kompilerer og kører koden.
- Giv en beskrivelse på højniveau af interessante implementationsaspekter. F.eks., aspekter, I har brugt særligt meget tid eller energi på.
- Det kunne f.eks. være mere avancerede aspekter såsom hvordan I håndterer AI, hvordan I håndterer spilhandlinger, animation, eller andet.
- Hold beskrivelsen overordnet. Vi kan læse jeres kode for detaljerne.

4 Kvalitetssikring

Til fordel for at sikre, at vores kode/program/spil lever op til kravsspecifikationen (se sektion 2), har vi valgt at anvende unit-tests. Unit-testing defineres i denne kontekst som test af enkelte komponenter af programmet, som til sammen skaber det ønskede overblik.

I overenstemmelse med vores valg af unit-tests, er det også underforstået, at vi tester i en form for white-box testing. Med dette betyder det, at dem der skriver testsne (os som udviklere af spillet i dette tilfælde), kender til alt logikken bag implementeringen. Med dette, har man som "*tester*", altså mulighed for at tjekke, om en given invariant for en given metode overholdes under kørsel.

Mere specifikt; har vi valg at benytte os af *Java*s framework `JUnit`, som b.la. understøtter behjælpelige sammenligningsmetoder (f.eks. *assertions* via `assert`).

Med dette, anvender vi altså også unit-testing som middel til at krydstjekke med vores kravsspecifikation, om disse (krav) samt eventuelle invarianter er overholdt.

Med anvendelse af ovennævnte, kan fejfindingprocessen under udviklingen, i nogle tilfælde, forkortes markant. Man kan med andre ord, nogle gange, såfremt man skriver tilstrækkelige unit-tests, forsimple samt forbedre design og udviklingsprocessen.

INDKLUDER ANDEN TILGANG TIL TEST?

5 Proces

- Arbejdede I i faser i løbet af projektet?
- Hvordan gik samarbejdet, og hvordan sikrede I lige deltagelse?
- Brugte I tekniske værktøjer til at få samarbejdet til at glide nemmere på tværs af maskiner?
- Har I brugt AI som støtte under udviklingen af jeres projekt? I så fald, hvordan?

6 Diskussion

- Ville I gøre noget anderledes hvis I skulle implementere projektet forfra?
- Var der dele af projektbeskrivelsen I ikke nåede? I så fald, hvordan er disse dele kompatible med jeres design? Ville I foretage ændringer for at imødekomme ændringer?