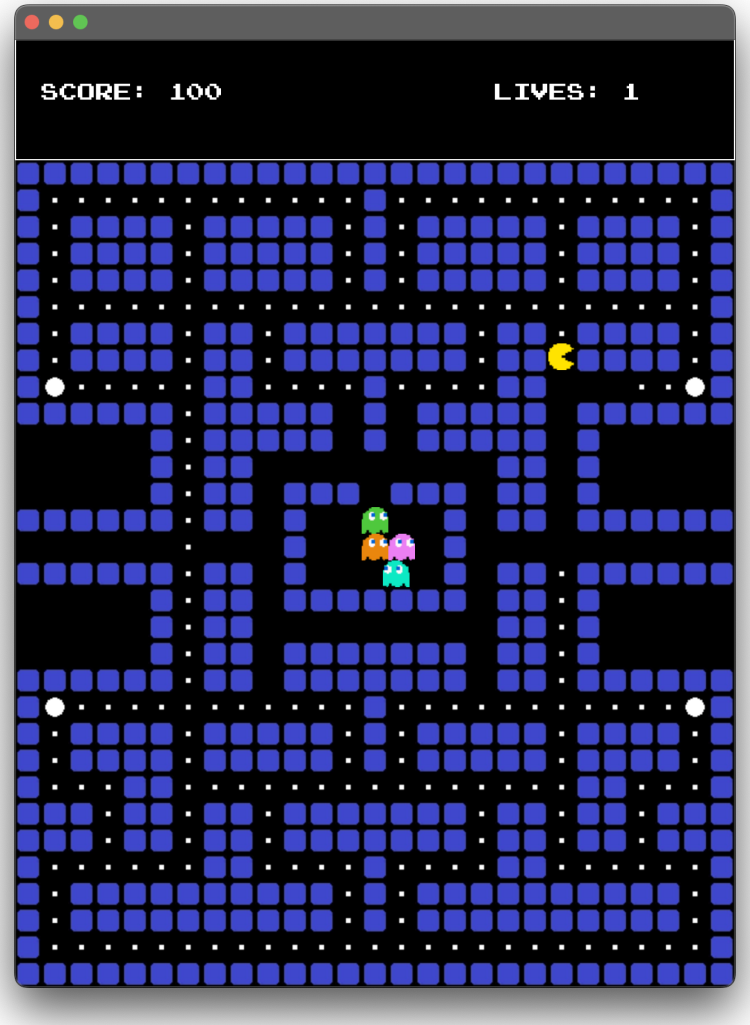
Objektorienteret Programmering Projekt PacMan

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

28. maj 2025



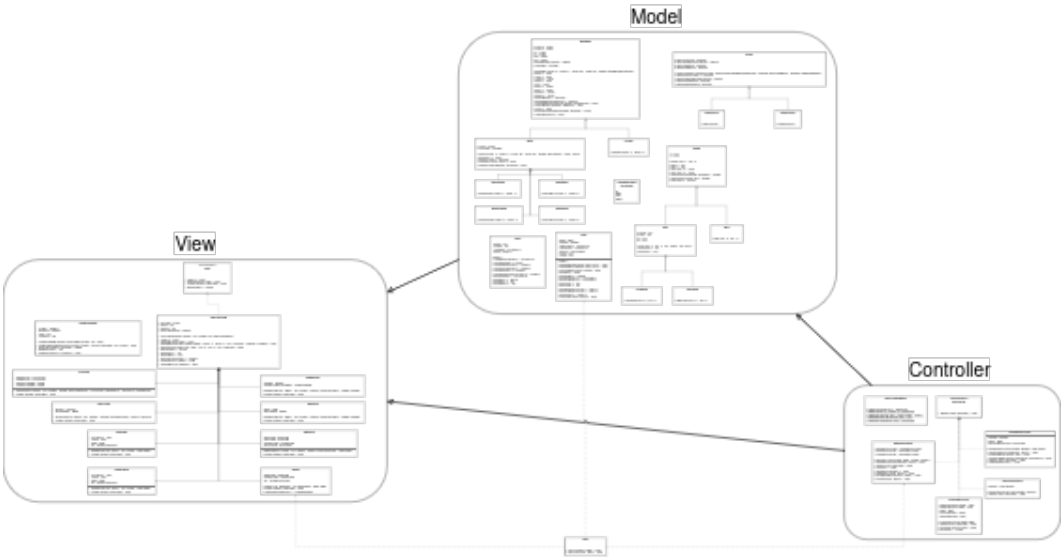
Indhold

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:

Projektets 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.

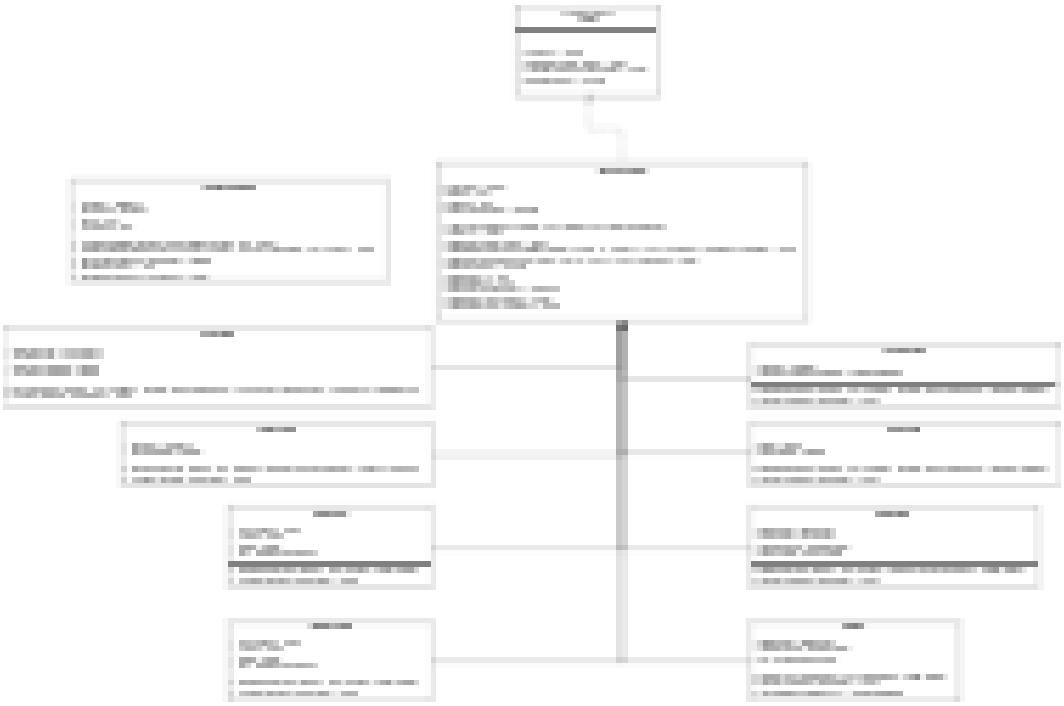
Vi har designet alt vores kode med fokus på *indkapslingsprincippet*. Alle felter i klasser er private, og har kun *getters* og *setters* der hvor det er nødvendigt. Vi har også overholdt *DRY* princippet, ved at samle ens opførsel i fælles *superklasser*.

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 klasse, `Pos2D`, til at beskrive positioner som ikke skal kunne bevæge sige. Fra denne klasse kan vi så nedarve klasser som `Pill` og `Wall`, da de bare er positioner, men vi godt vil kunne skelne imellem dem. På denne måde benytter vi *klasseafhængighedsprincippet* til at simplificerer koden, og gøre det nemmere at udvide med nye features.

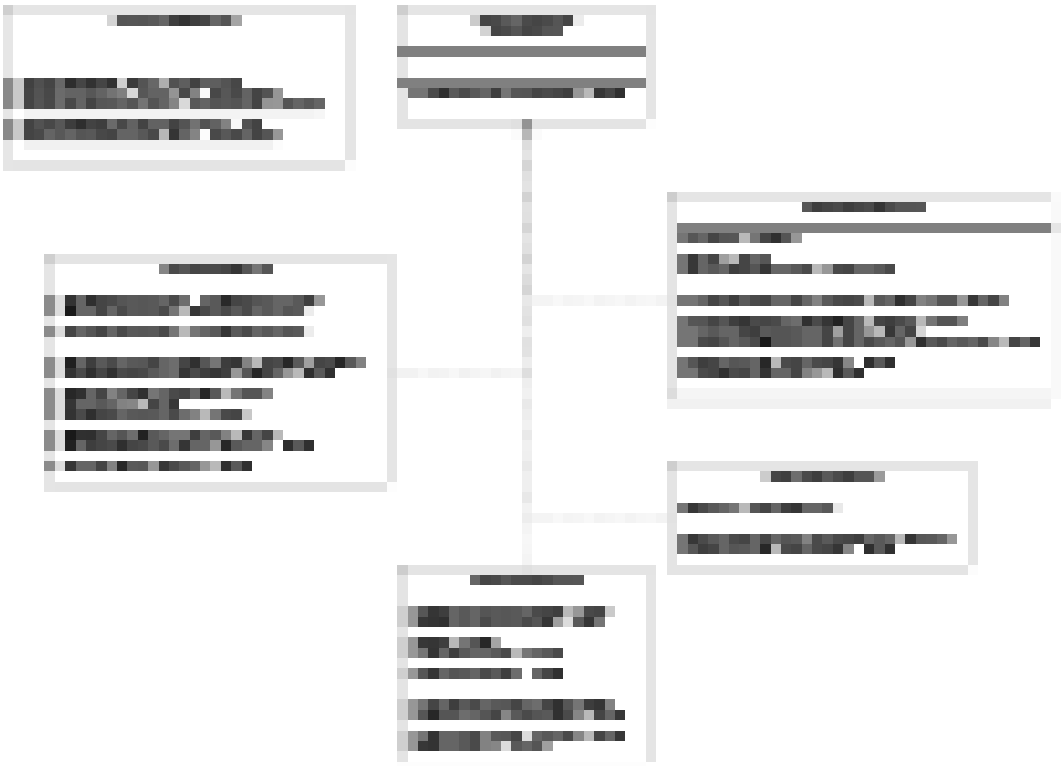
2.2 View



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, svarende til *Single Responsibility Principle* eller *SRP*.

2.3 Controller



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

Til *Controller* (se Figur 4) har vi også lavet en grænseflade, for at specificerer hvad en "Controller"gør. Derefter 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 opdateres, og hvornår vores *View* bliver opdateret.

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 labyrinten. Alt andet logik til spillet var i `Maze`, som gjorde projektet uoverskueligt, samtidig med at det bryder *SRP*. 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 *SRP*.

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.

Siden da har vi udvidet endnu mere på projektet, og det ser nu ud som det gør på Figur 1.

3 Implementation

States

3.1 PacMan kontrol

Siden vi har gjort at en *Moveable*'s position er beskrevet med decimal tal, så skal der noget smart logik til for at PacMan kan bevæge sig "smooth"omkring hjørner i labyrinten, da den består af en masse væge på heltals-positioner, og PacMan er samme størrelse som væggene, så han passer kun lige akurat imellem dem.

Idéen er at man kigger frem foran PacMan for at se om der er en passage til den side man gerne vil hen, indikeret med et tastetryk. Hvis der er en passage venter controlleren med at dreje PacMan indtil han er lige ved siden af passagen. Dette har vi gjort ved at gemme retningn i et felt kaldet for `waitingDirection`, og så hver gang `PacManController`'s `update()` metode bliver kaldt, så tjekker den om PacMan's position er en heltalsværdi. Det må nemlig betyde at han nu står foran den passage som der tidligere blev set. I så fald drejer PacMan nu til retningen af `waitingDirection`.

Der er også casen hvor PacMan skifter retning til det modsatte af hvad han bevæger sig. Så kan vi ikke kigge efter en passage der, for det vil der altid være, og han vil dermed ikke skifte retning med det samme. Så i dette tilfælde skifter vi bare retning med det samme. Hvis det er sådan at PacMan står stille, så er vi ikke interreseret i at kigge efter en passage, og han skal bare bevæge sig i den retning som man trykker med det samme.

For at lave alt dette med retninger, lavede vi en `enum` klasse til at holde styr på de fire retninger vi tillader: op, ned, højre og venstre. For at finde ud af hvilken en af disse retninger vores tastetryk svarer til, lavede vi også en `DirectionAdapter` klasse, som står for at konverterer imellem `Direction` og andre representationer af en retning.

3.2 Ghost AI

3.3 Animationer

Da vi gerne ville have animationer i spillet, lavede vi en klasse kaldet `AnimatedImage`, som står for at repræsenterer de forskellige frames af en animation, indlæse frames fra en sti, og ændre farver i dem hvis der er brug for det. Det med at ændre farver har vi gjort så vi ikke selv skal tegne et nyt set frames for hvert spøgelse vi vil lave i en anden farve, men i stedet for kan lave ét set med en meget specifik farve (hvis hex værdi er #000001), og så udskifte den farve med farven af spøgelset.

Når man så laver et nyt `AnimatedImage`, så specificerer man hvor lang tid én frame varer i nanosekunder. På denne måde kan vi i `getFrame()` metode regne ud hvilken frame vi giver, ud fra den nuværende tid som metoden for. Det gør den ved først at regne ud hvor lang tid hele animationen tager. Derefter finder vi hvor langt inde i den animation cyklus vi er ved at tage den nuværende tid modulus længden af animationen. Med dette kan vi så dividerer med længden af én frame og få det frame indeks som vi er nået til.

For at farve spøgelsernes animationerne de rigtige farver benytter vi vores metode `replaceColorsInFrames()`, som udskifter alle farver som er lig med `fromColor` i en animations frames, og sætter dem til farven `toColor`. Dette gør den ved at gå igennem hver frame pixelvis og tjekke om de matcher `fromColor`, for så at udskifte det med `toColor` på en kopi af framet, så det til sidst kan blive overskrevet.

- 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.

Ydermere, har vi til håndtering af versionsstyring anvendt `git` i form af nye branches under udvikling af nye programfunktionaliteter. Dette konstruerer en form for automatiseret testing, da man ikke kan publicere ny funktionalitet, før det passer med den resterende kodebase (f.eks. håndtering af mergeconflicts mm.). Denne form for *automatiseret* testing er også kendt som *Continuous integration testing*.

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?