Inden vi startede med at lave diagrammerne og koden, var tanken at projektet skulle være så objektorienteret som muligt. Klasser skulle kunne holde referencer til objekter fra andre klasser.

Målet var også at have en klar opdeling af strukturen. Main skulle udelukkende bruges til at starte programmet. Al data, logik og lister skulle initialiseres af Engine. Menu skulle være grænsefladen og stå for al kommunikation med brugeren. Engine og de andre klasser må ikke indeholde spor af at det er en konsolapplikation. Det vil sige System.out.print i Engine er ”fy fy”, og klasserne må heller ikke have metoder til at formatere pænt til konsolen. Ved at holde det helt adskilt ville man kunne skifte menuklassen ud med f.eks. en JavaFX GUI, uden at skulle rydde op i hvad der kunne have blevet spaghetti.

Efter at have designet klassediagrammet, lavede vi alle klasserne med constructors, getters & setters og toString metoderne.

Næste skridt var at lave en loadData() metode i Engine, som kunne lave super og sub klasse objekter ud fra tekstfiler.

For at kunne have mellemrum i vores String fields, og for at kunne skelne mellem super og sub klasser bruger vi delimiters i vores tekst, der splitter hver linje op i et String array.

’:’ indikerer at det er et nyt token og ’#’ at det er tokens, der er unikke til subklasser.

For at gøre det lettere at arbejde med super og subklasser, har vi et boolean field, der fortæller om objektet er super eller sub klasse.

Derfor kan vi f.eks. lave et ”if statement”, der checker om et medlem er af subklassen CompetetitiveMember, mens vi læser en linje i filen ved at parse en bestemt plads i arrayet til boolean og dermed lave en instans af den rigtige klasse.

Nogle af disse objekter skulle også have fields med referencer til andre objekter. Et eksempel på dette er Record, som har et Member field, der indikerer hvem der har sat rekorden.

Når det kommer til at objekter skal indkapsle andre objekter, er det essentielt at de fremmede objekter er blevet initialiseret først.

Det betyder at vi har høj binding, når vi loader dataene fra tekstfilerne. Vi bliver nødt til at loade alt og i den rigtige rækkefølge.

En af de første idéer vi fik var, at i stedet for at havde et ID for hvert medlem, kunne vi bruge et CPR-nummer. Vi kan så bruge CPR-nummeret til at regne ud, hvor gammel en person er – og finde det passende kontingent ved programstart.

Vi har også en liste med konkurrencehold, et for junior og senior, for hvert køn. Ved at tage det sidste ciffer i CPR-nummeret, kan vi se hvilket køn konkurrencemedlemmet er, og sammen med alderen placere dem på det rigtige hold.

Der blev diskuteret indbyrdes, om vi skulle lave et login modul, og om hvordan man skulle fortolke opgavebeskrivelsen med hensyn til, at forskeligt personale skal bruge forskellige funktioner i programmet.

Vi kom til den konklusion, at vi ville springe dette trin over, bla. fordi hele tiden at skulle tjekke brugerens privilegier, ville blive snørklet, og at fordi vi på første semester kun har lært at gemme oplysninger i plaintext lokalt. Derfor skulle passwords enten hardcodes i kildekoden eller også ville alle med adgang til systemet kunne bruge operativsystemets stifinder til at åbne filerne og aflæse koderne.

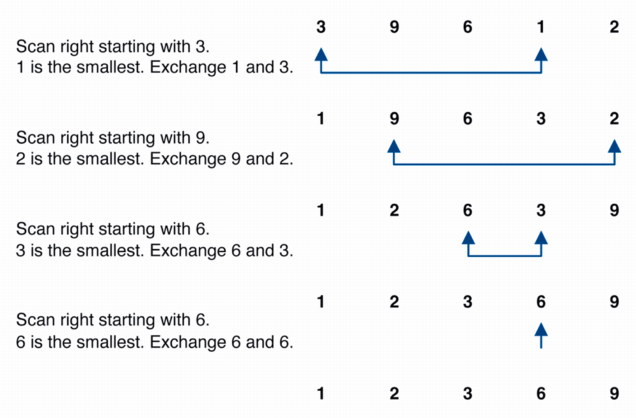
Efter at fundamentet var på plads, uddelegerede vi forskellige undermenuer, så alle havde ansvar for at lave en del af programmet. Vi blev enige om at bruge git og github som versionsstyring, så man kunne se de ændringer og opdateringer de andre havde lavet, og kunne splejse vores kode sammen.

Vi aftalte på forhånd, at objekter med persistens, dvs. rekorder og medlemmer skulle have CRUD funktioner.

Discipliner, kontingenttyper og hold står beskrevet statisk i opgavebeskrivelsen, så i vores program har vi ikke lavet menu funktioner til at ændre dem.

Rekord funktioner er presset ind i undermenuen ”Competitive Menu” og hvor der er endnu 3 undermenuer, Personal records, Competition records og Teams.

Som beskrevet i opgavebeskrivelsen skulle en træner kunne trække sine bedste 5 svømmere inden for hver disciplin. Det blev en udfordring, fordi vores liste med rekorder ikke er sorteret efter tid, men efter tilføjelse. En normal Collections.sort() kunne ikke bruges, fordi vores liste ikke består af simple datatyper, men af Record objekter. Vi skulle bruge en algoritme til at sortere vores rekorder efter tid. En hurtigt tur på Wikipedia [[1]](#footnote-1), [[2]](#footnote-2) afslørede, at ”Selection sort” vil være en simpel algoritme at kode og implementere. Som vist på billedet, går man listen igennem, og bytter den nuværende værdi rundt med den mindste værdi i listen, der kommer efter sig selv.

[[3]](#footnote-3)

Efter at listen er sorteret, looper vi den igennem og returnerer de 5 første rekorder, hvor disciplinen er lig med den disciplin brugeren har spurgt efter, og hvor rekorden tilhører et medlem på holdet.

Da vi ikke overskriver eller sletter et medlems rekorder, når medlemmet laver en ny, tjekker vi også om medlemmet allerede er blevet trukket som en af de 5 bedste.

Rekorderne og de ting man skal kunne med dem ifølge opgavebeskrivelsen, virker ret beskedne. Vi diskuterede om vi skulle udbygge systemet, så der også er distancer og bedre integration med turneringer m.m. Men det endte med, at vi besluttede os for at holde os ret tæt på de krav der blev stillet.

1. <https://en.wikipedia.org/wiki/Sorting_algorithm> [↑](#footnote-ref-1)
2. <https://en.wikipedia.org/wiki/Selection_sort> [↑](#footnote-ref-2)
3. <http://dopey.cs.vt.edu/courses/cs1706/slides/sorting.html> [↑](#footnote-ref-3)