Modul 1-3

Konstruktion af et vektorbibliotek ved brug af objektorienteret programmering

Indhold

1 Modellering, implementering samt dokumentering og test af et vektorbibliotek til at repræsentere og operere på vektorer i 2D og senere i n-dimensioner

I denne forløbsoversigt gives en detaljeret beskrivelse af, hvorledes man der kan arbejdes i klassen med at udvikle et vektorbibliotek ved brug af objekter.

1.1 Modul 1

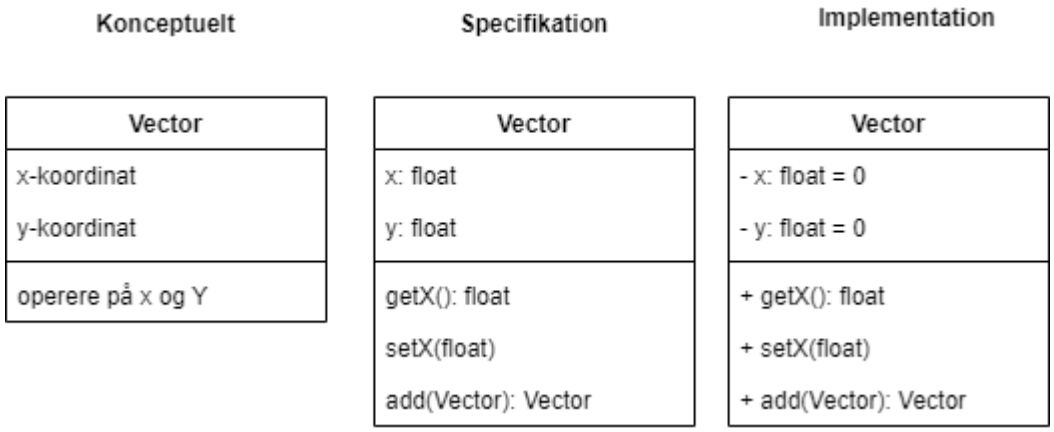
Til at starte med kan det være en god ide at tage udgangspunkt i elevernes forforståelse. I den sammenhæng kan en indledende øvelse være at få eleverne til at overveje, hvorledes de vil repræsentere og operere med vektorer uden brug af objekter. Formentlig vil de ret hurtigt indse, at det er upraktisk, hvis man f.eks. skal erklære en-to nye variabler for hver gang man skal oprette en ny vektor.

I stedet indføres begrebet klasse som en enhed i vores program, der indkapsler data og metoder og at disse klasser er repræsentanter for vektorobjekter.

Herefter indføres eleven i hvorledes en simpel klasse, der repræsenterer en vektor, kan konstrueres. I den sammenhæng anbefales det at visualisere dette med et UML-klassediagram. Det kan betones over for eleverne, at klassediagrammer kan præsenteres på forskellige niveauer:

- **Konceptuelt niveau**: Betoner koncepter inden for problemdomænet
- **Specifikationsniveau**: Betoner inter- og intrarelationelle aspekter mellem de muligvis abstrakte datatyper
- **Implementeringsniveau**: Betoner mange detaljer om attributters typer, initiale værdier metoders parametre og returtyper samt synlighed (private/offentlige)

Figur 1



Ligesom man kan arbejde iterativt og skridtvis med at forbedre sin kode, kan man indledningsvis udbygge iterativt på et klassediagram. Pointen er, at den faktiske implementering sker inkrementelt i mindre dele ud fra designprincippet ”separation af bekymringer”. Det er vigtigt at betone, at det er helt okay og ofte nødvendigt at justere på diagrammer undervejs, når ”tavs viden” erkendes.

Efter konstruktionen af et klassediagram kan man udarbejde ”et worked example” i det valgte sprog, der viser, hvorledes man implementerer klassen på kodeniveau.

Til at starte med begrænser man sig nok til attributterne x og y og konstruktøren og efterfølgende instantierer en klasse samt tester ved at tilgå attributterne. Herunder lidt pseudokode:

```
class Vector{
  constructor(x:float,y:float){
    this.x = x;
    this.y = y;
  }
}
v1 = new Vector(2,3)
print(v1.x) // printer 2
print(v1.y) // printer 3
```

Det er afgørende løbende at teste. Selv ved små ændringer bør der testes.

Efterfølgende kan klassen udvides med relevante metoder og de kan prøves af. Herunder vektoraddition. Man kan overveje om metoden fremfor at ændre på instansen af værdierne i stedet bare bør returnere sumvektoren. Ligesom man også kan overveje at indføre statiske metoder, der eksempelvis kan summe vektorer og hvor man ikke behøver at oprette en instans af klassen for at kalde dem.

```
 add(v: Vector){
  this.x += v.x
  this.y += v.y
 }
 static Vector add(v1: Vector, v2: Vector){
  return new Vector(v1.x + v2.x,v2.y+v2.y)
 }
//…
v1 = new Vector(2,3)
v2 = new Vector(1,4)
v1.add(v2)
v3 = Vector.add(v1,v2)
print(v1.x) // printer 3
print(v1.y) // printer 7
print(v3.x) // printer 4
```

Man kan med fordel implementere en metode, der tegner en (sted)vektor på skærmen, så det visuelle element tidligt inddrages. Samtidigt trænes eleverne i at bruge det eksterne grafikbibliotek.

1.2 Modul 2

Når den indledende klasse er konstrueret, kan man indføre abstraktionsprincippet, som bl.a. betoner, at vi bør skjule den faktiske implementation, og at vi skal forsøge ekstrahere fælles egenskaber fra specifikke eksempler. Dvs. at se bort fra/fjerne ting som er unødvendige. Det handler med andre ord om at gøre sig klart, hvad der er essentielt for en vektor, hvad der er mindre relevant, og få disse ting ekspliciteret i en pæn grænseflade.

I forbindelse med indføringen af indkapsling og abstraktion kan det være en god ide at indføre begreber såsom ”privat” og ”offentlig” ud fra klassediagrammet og hvorledes vi kan implementere accessor- og mutator-metoder, der kan tilgå de private attributter. Det er en god øvelse for eleverne at skrive nogle simple getters og setters og teste dem af i praksis:

```
class Vector{
 constructor(x,y){
  this.#x = x; //private attribut ved brug af #
  this.#y = y;
 }
 setX(x){
  this.x = x
 }
 setY(y){
  this.y = y
 }
 float getX(){
```

```
  return this.x
 }
 float getY(){
  return this.y
 }

}
```

Belys gerne de mange fordele ved brug af getters og setters, som nogle elever måske kan have svært ved at se nødvendigheden af. Herunder kan nævnes:

- Skalerbar: Det er meget nemmere at refaktorisere en enkelt getters/setters metode end at skulle søge hele koden igennem for tildelinger eller lignende.
- Debug: Man kan sætte breakpoints i getters og setters.
- Generelt pænere og mere elegant kode. På den korte bane kan getters og setters virke omkostningstungt og ressourcekrævende, men på den længere bane får man som regel pænere kode.

Nedenfor er illustreret en anden getter metode i pseudokode, der beregner længden af den konkrete vektor:

```
float getLength(){
 return sqrt(this.x^2 + this.y^2)
}
```

I forhold til abstraktionsprincippet er det vigtigt at bemærke, at der ofte deltager mange udviklere i et større software projekt. Ofte er udviklerne inddelt i mindre grupper, der arbejder med hver deres del af koden. Ofte har hver gruppe brug for at bruge de andre gruppers dele uden nødvendigvis at kende den konkrete implementering. Man kan med andre ord overveje, om man bør formulere vektorklassen som en abstrakt klasse til at starte med. Endelig bør det også betones, at selvom der er ligheder mellem abstraktion og indkapsling, såsom at skjule unødig data og unødig kompleksitet, så er abstraktion noget, der typisk sker i designfasen ved brug af abstrakte klasser/interfaces, mens indkapsling sker i implementationsfasen.

Efterfølgende kan man begynde at udvide klassen til et egentligt vektorbibliotek. Her fordres igen, at man udvider klassen inkrementelt ud fra et klassediagram. Nogle af attributterne kunne være vektorens farve, startpunkt, koordinater m.m.

Oplagte metoder at implementere kunne være:

- **Skalering [`scalar(k)`]**: En vektor kan skaleres i længde med værdien k. Det sker ved at multiplicere k på både x og y. Metoden skal returnere den skalerede vektor.

- **Tegne vektor [`drawVector(xstart=0,ystart=0)`]**: Metoden skal tegne vektoren fra startpunktet (xstart,ystart). Hvis ingen argumenter angives, tegnes stedvektoren.
- **Til streng [`toString()`]**: Metoden skal returnere en streng, der består af koordinaterne og vektorens længde.
- **Prikproduktet [`dotProduct(v)`]**: Metoden beregner prikproduktet med en anden vektor `v`. Det er summen af koordinaterne multipliceret koordinatvis.
- **Normalisering [`normalize()`]**: Metoden konstruerer en såkaldt enhedsvektor med samme retning som den oprindelige vektor men med længden 1. Det sker ved at skalere med skalaren 1/"Længden af vektoren".
- **Ens retning [`isParallel(v)`]**: Normaliseres vektorerne, beregnes prikproduktet og det giver 1, så betyder det, at vektorerne er ensrettede. Metoden skal returnere sand eller falsk.
- **Modsat retning [`isOpposite(v)`]**: Normaliseres vektorerne, beregnes prikproduktet og det giver -1, så betyder det, at vektorerne er modsatrettede. Metoden skal returnere sand eller falsk.
- **Vinkelret [`isPerpendicular(v)`]**: Normaliseres vektorerne, beregnes prikproduktet og det giver 0, så betyder det, at vektorerne står vinkelret på hinanden. Metoden skal returnere sand eller falsk.
- **Negering af vektor [`negate()`]**: Ønskes konstruktion af en vektor i modsat retning kan den oprindelige vektor negeres ved at skalere med -1. Metoden skal returnere den negeret vektor.

Ud over listen ovenfor findes en lang række metoder, som kunne være relevante at udvide klassen med. Det kan være en god diskussion at tage i klassen omkring metoder og attributter. Ved at have en fælles grænseflade, kan man evt. vælge at lade forskellige grupper arbejde med forskellige metoder, men vær opmærksom på, at nogle afhænger af andre. I den sammenhæng kan man vælge at indføre skeletmetoder, som kompileres fejlfrit, men ikke gør noget. Ydermere bør man overveje at differentiere udviklingsarbejdet, så de mere øvede grupper får de sværere udviklingsopgaver.

I øvelserne er en lang række ekstra metoder, der kunne være interessante at implementere. Igen bør det overvejes, om nogle af metoderne bør være statiske.

Et væsentlig krav til implementeringen af et API/bibliotek er, at metoderne testes og dokumenteres. I den sammenhæng anbefales det at indføre sprogets konventionelle måde at dokumentere kode (eksempelvis javadoc til Java, Jsdoc til JS/TS eller docstrings/pep8 til Python). Det kan være en opgave for eleverne at gøre det for deres udvalgte metoder. Det giver den fordel, at man i den faktiske brug af biblioteket får forslag til, hvorledes biblioteket kan benyttes i sin IDE. Det er muligt, at man bliver nød til at tage dele af 3.modul i brug, hvis grupperne skal nå at dokumentere deres metoder.

Herunder et eksempel fra JsDoc med Javascript:

```
/** Class representing a Vector. */
class Vector {
```

```
  /**
   * Create a Vector.
   * @param {number} x – The x value.
   * @param {number} y – The y value.
   */
  constructor(x, y) {
    // ...
  }

  /**
   * Get the x value.
   * @return {number} The x value.
   */
  getX() {
    // ...
  }

  /**
   * Get the y value.
   * @return {number} The y value.
   */
  getY() {
    // ...
  }

  /**
   * Scales a vector
   * @param {float} flo–t - The scalar a real number.
   * @return {Vector} A scaled vector.
   */
  scale(flt) {
    // ...
  }
  //...
}
```

1.3 Modul 3

Softwareudvikling er i høj grad en ikke-lineær proces, hvor man arbejder iterativt med at udvide, forfine og refaktorisere sin kode. Det er vigtigt for eleverne at erkende, så de ikke mister lysten for hurtigt.

I den sammenhæng kan det være en spændende opgave at modificere vektorbiblioteket, så det kan håndtere ikke bare vektorer i planen, men vektorer i 3D eller n-dimensioner. I førstnævnte tilfælde kan

man nøjes med at udvide med et z-koordinat, mens det med n-dimensioner kræver indføring af arrays eller lister.

I øvelserne foreslås bl.a. at der udvikles en højere ordens map-funktion på ens vektorer. Det giver mulighed for mange spændende udvidelser af biblioteket for de særligt interesserede og dygtige elever. Eksempelvis at udvikle vektorfunktioner, der senere kan bruges til at modellere særlige "stier", som fiskene følger.

2 Øvelser

Herunder følger en lang række øvelsesskitser til de enkelte dele. Det er på ingen måde et krav, at man når igennem dem alle. De kan suppleres med relevante kodeeksempler i det valgte sprog.

Til at starte med arbejdes i to dimensioner. Senere i øvelserne skal alle resultaterne generaliseres til n-dimensioner. Det er dog en god ide at starte i to dimensioner. Generelt opfordres til, at man i alle øvelserne starter med at justere klassediagrammet, udvide med en skeletmetode og herefter test den. Endelig bør man teste og dokumentere alle udvidelser.

Eksempler

1. Modeller vektorklassen ved at konstruere et klassediagram, hvor du tilføjer relevante attributter til at starte med. Blandt de relevante attributter x- og y-koordinater, vektorens farve, tykkelse og navn. Indiker typerne i diagrammet.

2. Udvid med en klasse, der initialiserer de indførte attributter.

3. Overvej hvilke attributter som bør være private. Juster klassediagrammet.

4. Indfør relevante mutator-metoder (getters og setters) for alle dine attributter og tilføj metoderne til klassediagrammet. Prøv metoderne af og se om de virker.

5. Udvid klassediagrammet med metoder `scalar(k)`, `drawVector(xstart=0,ystart=0)`, `toString()`, `dotProduct(v)`,`normalize()`, `isParallel(v)`, `isOpposite(v)`, `isPerpendicular(v)` og `negate()`.

6. Udarbej skeletmetoder for hver af metoderne, hvor du tilføjer kommentaren og prøv herefter at implementere de relevante metoder.

7. Prøv alle metoderne af og se, om de giver det ønskede resultat.

8. Dokumenter alle metoder ved brug af sprogets standard for dokumentering.

9. Udvid med en metode, der undersøger om en vektor er lig med eller forskellig fra en anden vektor. Dvs. hvorvidt koordinaterne er ens.

10. Tilføj en metode der giver mulighed for at lave elementvis multiplikation og division mellem to vektorer. Husk at teste og dokumentere dem.

11. Udvid vektorklasen med nedenstående statiske sammenligningsmetoder (ved at skrive `static` i erklæringen af metoderne). Vær opmærksom på, at ved at erklære metoderne for `static` kan de kaldes uden, at man behøver at instantiere dem. Husk at teste og dokumentere dem.

- `lessThan(a,b)`: Undersøger om hvert koordinat i vektor a er skarpt mindre end tilsvarende koordinat i vektor b. Returner sand hvis det er tilfældet ellers falsk.
- `lessThanOrEqualTo(a,b)`: Undersøger om hvert koordinat i vektor a er mindre end eller lig tilsvarende koordinat i vektor b. Returner sand hvis det er tilfældet ellers falsk.
- `greaterThan(a,b)`: Undersøger om hvert koordinat i vektor a er skarpt større end tilsvarende koordinat i vektor b. Returner sand hvis det er tilfældet ellers falsk.
- `greaterThanOrEqualTo(a,b)`: Undersøger om hvert koordinat i vektor a er større end eller lig tilsvarende koordinat i vektor b. Returner sand hvis det er tilfældet ellers falsk.

12. Udvid nu med selvvalgte statiske metoder `max(a,b)` og `min(a,b)`, der finder det hhv. største og mindste koordinat i to vektorer, a og b.

13. Udvid biblioteket med følgende statiske metoder:

- `abs(v)`: Returnerer en ny vektor med koordinater som vektor v, men hvor den absolutte/numeriske værdi er taget på hver af koordinaterne.
- `floor(v)`: Returnerer det største hele tal mindre end eller lig med hvert koordinat i vektoren.
- `ceil(v)`: Returnerer det mindste hele tal større end eller lig med hvert koordinat i vektoren.

14. Udvid biblioteket med følgende statiske metoder:

- `power(vec,n)`: Metoden opløfter hvert element i heltallet `n`.

- `power(a,b)`: Metoden opløfter hvert element i vektor a med tilhørende element i vektor b.

15. Generaliser vektorbiblioteket så det kan håndtere og operere på vektorer i n-dimensioner.

16. Generaliser alle metoderne, som du har implementeret i 2d til n-dimensioner. F.eks. bør getters og setters tage parametre, der indikerer hvilke koordinat i form af et heltal fra 0 til n-1, der indikerer hvilket koordinat man ønsker at hente eller sætte. Overvej desuden hvilken datastrukturer, der skal bruges til at håndtere data.

17. I det følgende skal anvendes en såkaldt højere ordens funktion ved navn `map`. Funktionen `map` tager en funktion og lader den virke på hvert element i et array. Herunder et eksempel fra det aktuelle sprog, som skal prøves af.

Udvid vektorbiblioteket, så det inkluderer en `map(f)` funktion, der lader funktionen f virke på alle koordinater i vektoren. Prøv map af i praksis med forskellige funktioner. Bemærk at `f` skal være en funktion, der tager et reelt tal og returnerer et reelt tal.