Norges teknisk–naturvitenskapelige universitet
Institutt for datateknologi og informatikk

TDT4102 Prosedyre- og objektorientert programmering Vår 2023

Øving 7

Frist: 2023-03-10

Aktuelle temaer for denne øvingen:

- Klasser, arv, polymorfi og virtuelle funksjoner

Generelle krav:

- Bruk de eksakte navn og spesifikasjoner gitt i oppgaven.
- Teorioppgaver besvares med kommentarer i kildekoden slik at læringsassistenten enkelt finner svaret ved godkjenning.
- 70% av øvingen må godkjennes for at den skal vurderes som bestått.
- Øvingen skal godkjennes av stud.ass. på sal.
- Det anbefales å benytte en programmeringsomgivelse(IDE) slik som Visual Studio Code.

Anbefalt lesestoff:

- Kapittel 12, 13 og 14.

Merk: Siden vi bruker AnimationWindow istedenfor SimpleWindow, vil noen ting gjøres annerledes i boka. Se dokumentasjon til AnimationWindow her.

Polymorfi

Et eksempel er den abstrakte klassen`Shape`. Det finnes ingen objekter som er instanser av abstrakte klasser, og det er ikke mulig å lage et objekt av klassen. Hvis vi ser på det vi ønsker å modellere, så stemmer det overens med virkeligheten. Vi kan ikke lage eller oppfatte konseptet form, men vi kan lage og oppfatte konkrete former. En form er en tanke, sirkler og firkanter er konkrete former.

Når vi programmerer og modellerer med objekter, så er det nyttig at noe er abstrakt og noe er konkret. Alle former som kan konstrueres har fellestrekk, f.eks. posisjon, farge, osv. Det er egenskaper, men ikke nye former. En rød og en grønn trekant er fremdeles to trekanter, men med forskjellig fargeegenskap.

En konkret klasse er en spesialisering. Alle klasser som arver fra en annen er spesialiseringer, og polymorfi gjør det mulig å spesialisere atferd. Virtuelle medlemsfunksjoner kan overskrives og spesifiserer derfor ny atferd for klassen.

1 Introduksjon til arv og polymorfi (10%)

a) Teorioppgave

Hva er forskjellen på `public`, `private` og `protected`?

b) Animal, en baseklasse

Du skal nå lage klassen `Animal`, som skal inneholde følgende medlemsvariabler:

- name, av typen `string`
- age, av typen `int`

Disse skal være `protected`. `Animal` skal ha en konstruktør som skal ta inn `name` og `age` som parametere, og initialisere medlemsvariablene, i tillegg til en virtual destruktør som frigjør minne. Destruktører lærer du om senere i emnet, så det eneste du trenger å vite nå er at de skrives slik:

```
virtual ~Animal() {}
```

Klassen skal også ha en `virtual` funksjon, `toString()`. som skal returnere: `"Animal: name, age"`

c) Cat og Dog, arvede klasser

Lag klassene `Cat` og `Dog`. De skal begge arve `public` fra `Animal`.

Begge klassene skal inneholde medlemsfunksjonen `toString()` som skal være `public`, og redefinere `toString()` i `Animal`-klassen. I `Cat`-klassen skal `toString()` returnere: `"Cat: name, age"`. `toString` i `Dog`-klassen skal returnere: `"Dog: name, age"`.

Begge klassene skal ha en konstruktør som kaller konstruktøren til `Animal`.

Nyttig å vite: `unique_ptr`

En peker (pointer) er en variabel som holder på minne adressen til et objekt. En `unique_ptr` er en smart-peker i C++ som gjør det lettere å unngå vanlige feil ved minneallokering. Så `unique_ptr<Animal> dyr` lager en smart-peker, en `unique_ptr` til et objekt av class `Animal`. Pekere skal vi gå gjennom grundigere senere i kurset, så dette er det dere trenger å vite nå for å få til neste deloppgave.

d) **Test klassene** i testfunksjonen, `testAnimal()`. Opprett noen instanser av hver klasse, og kall `toString()` på instansene.

Opprett en `std::vector<std::unique_ptr<Animal»`, og legg til noen instanser av hver klasse i denne. Iterer gjennom vektoren og kall `toString()` på hvert element, siden vi bruker pekere gjøres dette med `v.at(i) -> toString()`. For å legge til en instans av typen `unique_ptr<Animal>` i en vektor kalt v, kaller du funksjonen `emplace_back()` slik: `v.emplace_back(new object)`, hvor "object" erstattes med et kall til konstruktøren du har opprettet for klassen din.

Hva skjer hvis du fjerner `virtual` foran `toString()` i `Animal`-klassen?

e) **Gjør `Animal`-klassen abstrakt**. Dette kan du gjøre ved å endre `toString()` til å være en `pure virtual` funksjon.

2 Emoji (15%)

a) **Emoji, en abstrakt baseklasse**

I de utdelte filene ligger det skjelettkode som dere kan bruke hvis dere vil, der er bl.a. `Emoji` allerede definert. Hvis du bruker skjelettkoden bør du forstå hvorfor `Emoji` blir abstrakt og hvilke valg som er gjort i denne oppgaven, 2 a), og klassen før du tar fatt på resten av øvingen. **Denne oppgaven forklarer den utdelte koden. Du skal altså ikke skrive noe kode i denne oppgaven, men det er viktig at du forstår hvordan den utdelte koden er bygd opp, slik at du kan bruke den i de neste oppgavene**. De utdelte filene hentes via TDT4102-extensionen som tidligere. Den utdelte filen `emoji_main.cpp` definerer en mainfunksjon. Pass på at du bare har en mainfunksjon i programmet ditt. Hvis du kompilerer og kjører skjelettkoden skal du få opp et tomt vindu.

I den utdelte koden er den abstrakte klassen `Emoji` definert. Hvis du velger å ikke bruke den utdelte koden skal `Emoji` ha egenskapene som er beskrevet i denne oppgaven.

Det finnes mange forskjellige typer emoji: ansikter, hender, biler, båter, blomster, osv. Emoji i seg selv er et abstrakt konsept, vi kan ikke tegne konseptet "emoji", men vi kan snakke om konseptet og likevel forstå hva det innebærer. I denne øvingen skal vi modellere alle typer emoji med bakgrunn i at alle emoji har en felles operasjon. Denne operasjonen er ikke helt lik for alle emoji, så formålet er å gjøre det mulig å tegne en hvilken som helst emoji gjennom det samme grensesnittet. "Tegn smileansikt" eller "tegn bil" har til felles at operasjonen er "tegn". I vårt tilfelle er "tegn" noe som byttes ut med en bestemt funksjon som alle emoji-typer selv kan skrive over for å definere hvordan den spesifikke emojien skal tegnes.

Nesten alle Emoji har forskjellig antall og typer former: åpne og lukkede øyne, hår, strek-munn, smilemunn, øyebryn, ører, osv. Det gjør at alle de forskjellige emojiklassene selv må ta ansvar for å tegne sine egne former til et vindu. Det er denne operasjonen vi bestemmer at alle emoji må ha, det felles grensesnittet.

`Emoji` har derfor en medlemsfunksjon som arvende klasser må overskrive for å bli konkrete, eller "følge kravet til grensesnittet". Medlemsfunksjonen har ansvar for å tegne de ulike øynene, munnene osv. til et `AnimationWindow`. Medlemsfunksjonen er *pure virtual* og heter `draw()`.

Nyttig å vite: arv og spesifisering av rettigheter

I lærebokens kapittel 14.3.2 er det representert flere måter å beskrive samme type arv. 14.3.5 lister flere måter for hvordan `public`- og `protected`-medlemmer arves fra en baseklasse til klassen som arver egenskapene. I denne øvingen er det tilstrekkelig å bevare `public`- og `protected`-egenskapene til medlemmene. De to følgende måtene er ekvivalente når vi ønsker oppførselen som er beskrevet over:

```
class Face : public Emoji {};
// eller
struct Face : Emoji {};
```

3 Ansikt (20%)

I denne oppgaven jobber vi med abstrakte klasser, og det blir derfor vanskelig å gjøre tester på dette stadiet. I oppgave 4 får vi derimot testet at alt fungerer som det skal, da vi skal lage konkrete klasser som arver fra disse abstrakte klassene. Dokumentasjon til AnimationWindow kan være nyttig i oppgavene videre.

a) Et utgangspunkt for ansikts-emoji.

Definer klassen `Face`, den skal arve fra `Emoji`.

Emojipedia holder en oversikt over hvilke smilefjes som finnes. Her finnes ingen smilefjes helt uten egenskaper. Derfor er det ikke ønskelig at smilefjes av typen `Face` skal kunne konstrueres.

Gjør derfor `Face` abstrakt. Det kan gjøres på samme måte som tidligere. `draw()` skal spesifiseres til å være pure virtual også for denne klassen.

b) `Face` sine egenskaper.

Selv om det ikke skal kunne instantieres objekter av typen `Face` har det en attraktiv egenskap som alle ansikter trenger. Nemlig et ansikt.

Klassen skal representere et sirkelformet ansikt og vi vil derfor at klassen skal inneholde medlemsvariablene `centre` og `radius` som er henholdsvis sirkelens posisjon og radius.

Klassen skal ha en konstruktør med to parametre, `Point c, int r`. Bruk medlemsinitialiseringsliste til å initialisere disse verdiene til medlemsvariablene `centre` og `radius`.

c) Tegn ansiktet

Overskriv `draw()` slik at det i vinduet tegnes en sirkel. Dette gjøres vha.

```
win.draw_circle(centre, radius, color);
```

`color` er fargen på sirkelen og er av typen `Color`, for eksempel `Color::yellow`.

Her bør du skrive `override` til slutt i deklarasjonen, slik at du får beskjed fra kompilatoren hvis du har skrivefeil e.l. i funksjonsnavn og parameterliste.

Selv om en funksjon er pure virtual kan den ha en definisjon. I dette tilfellet betyr pure virtual bare at klassen ikke kan brukes til å lage objekter.

4 Konkret emoji-klasse (20%)

a) Ansikt med øyne, endelig en ekte konkret klasse.

Det er flere ansikter som har to åpne øyne som fellestrekk. Derfor skal du opprette en ansiktsklasse som har to øyne og arver fra den abstrakte klassen `Face`. Dette er en konkret emoji som kalles «empty face» eller «face without mouth». For å gjøre typenavnet litt ryddig skal denne klassen hete `EmptyFace`. Den arver fra, og konstrueres på samme måte som `Face`. Klassen må inneholde to øyne, gi også øynene fyllfarge når de tegnes.

Bruk medlemsinitialiseringslisten til å gjenbruke konstruktøren fra klassen det arves fra og initialisere begge øynene. For `EmptyFace` betyr det at ansiktet som allerede initialiseres i `Face` sin konstruktør gjenbrukes. Syntaksen for å oppnå det ligner på normal initialisering av et medlem. Der det normalt ville vært navnet på et medlem, er navnet til en klasse brukt. Dette kalles *delegerende konstruktør*. Det vil se slik ut for `EmptyFace` som bruker `Face` sin initialisering:

```
EmptyFace(<parameterliste>) : Face{<argumenter>} /*, andre medlemmer */
```

Se også kapittel 13.15 i boken for andre eksempler.

Størrelse på øyne og plassering av øyne i ansiktet bestemmer du selv. Plassering skjer ut fra ansiktets sentrum. Det er ikke et krav at øynenes plassering og størrelse skaleres i forhold til endringer i størrelsen til emojien. Det gjelder også resten av øvingen.

b) Tegn `EmptyFace` i vinduet.

`draw()` må overskrives for alle `Emoji`-deriverte klasser for å bli tegnet i vinduet.
Former tegnes i rekkefølgen de kalles. For at øynene skal vises må de tegnes til vinduet etter ansiktet. Ansiktet fra `Face` tegnes ikke automatisk til vinduet når `EmptyFace` overskriver `draw()`. Derfor må det eksplisitt kalles på `Face::draw()` for å tegne ansiktet.
`EmptyFace::draw(AnimationWindow& win)` bør derfor inneholde et kall til:
`Face::draw(win);`

c) Tegn et tomt ansikt på skjermen. Opprett et vindu og tegn ansiktet i vinduet. Juster programmet til du er fornøyd med plassering av øynene i ansiktet. *Dette er første gangen du kan teste koden din.* I den utdelte filen `emoji_main.cpp` er det allerede opprettet et vindu som kan brukes.

Nyttig å vite: `draw_arc()`

I resten av øvingen skal du lage flere emoji. Mange emoji har nytte av at det kan tegnes buer og ikke komplette sirkler. Det finnes en funksjon for dette som for circle, ved å skrive `draw_arc()` istedet.

Bruk av `draw_arc()`

`draw_arc()` tar inn sentrum av buen, bredden på buen, høyden på buen, startvinkel, sluttvinkel og farge slik som dette:

```
AnimationWindow::draw_arc(Point center, int width, int height, int start_degree,
int end_degree, Color color);
```

Når du skal tegne buer med `draw_arc` kan du se for deg enhetssirkelen eller en klokke med utgangspunkt i hhv. 0 grader og klokken 3. Buene tegnes «mot klokken» fra start til og med slutt. Buen tegnes fra og med 0 grader pluss `start_degree` til og med 0 grader pluss `end_degree`. Buen tegnes korrekt om `start_degree <= end_degree`.

Bredden og høyden til sirkelen som buen spenner over kan også justeres. Resultatet er buer som strekker seg i retning av den aksen som er størst. Her er det bare å prøve seg fram til former som kan passe inn i dine emoji.

5 Flere emoji (35%)

Lag minst 5 forskjellige emojis. Du står fritt til å lage hvilke emoji du måtte ønske. a) til e) inneholder forslag til emojis du kan lage hvis du står helt fast.

Til slutt skal alle emojiene tegnes på skjermen. Du kan f.eks. lage en funksjon som tar inn `std::vector<std::unique_ptr<Emoji>>& emojis` og et vindu emojiene skal tegnes til. Det er dette som gjør polymorfi ettertraktet. Selv om alle emoji-klassene er forskjellige kan samme grensesnitt, `draw()`, brukes for å utrette samme operasjon på de forskjellige instansene av alle klassene som arver fra `Emoji`.

Kunst til inspirasjon.

a) Smilefjes

Lag en smilefjesklasse, `SmilingFace`. Klassen skal arve fra `EmptyFace`, da er alt du trenger å gjøre å tegne en `arc` som representerer munnen.

b) Lei seg-fjes

Lag klassen `SadFace`. Du står fritt til å velge om du ønsker å arve fra smilefjeset i forrige deloppgave og justere på munnen fra det ansiktet, eller arve direkte fra `EmptyFace` og legge til en ny munn.

Hvorfor har du valgt det ene alternativet framfor det andre?

c) Sint ansikt

Lag klassen `AngryFace`. Se emojipedias angry face for inspirasjon.

d) Blunkeansikt

Lag klassen `WinkingFace`. Ansiktet skal ha ett åpent øye og ett øye som blinker. Det blinkende øyet kan f.eks. være to linjer som former en <-lignende form eller en halvsirkel. Se emojipedias winking face for inspirasjon.

e) Overrasket ansikt

Lag klassen `SurprisedFace`. Ansiktet du får når du skriver `:o` eller `:O`.

Du bestemmer selv om dette ansiktet skal arve fra et smilende ansikt med arc-munn som endres til å tegne 360 grader, eller om du benytter f.eks. `draw_circle()` til å representere munnen. Kan du identifisere potensielle problemer med å arve fra f.eks. `SmilingFace`?