Norges teknisk–naturvitenskapelige universitet
Institutt for datateknologi og informatikk

TDT4102 Prosedyre- og objektorientert programmering Vår 2020

Øving 7

Frist: 2020-02-28

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.

FLTK og Graph_lib

Før du begynner på oppgavene i denne øvingen anbefales det å gå gjennom kapittel 12 og 13 i boken for å bli kjent med hvordan vi tegner på skjermen i dette faget.

FLTK er et portabelt grafikkbibliotek som gjør det mulig å tegne grafiske grensesnitt og primitive former (nesten) uavhengig av operativsystem.

`Graph_lib` er en abstraksjon som boken har valgt å gjøre tilgjengelig. Denne abstraksjonen gjør det lettere for oss som programmerere å skrive et program som skal tegne på skjermen. Det høres kanskje litt merkelig ut å skrive enda et nivå med abstraksjon mellom programmet vårt og operativsystemet og maskinvaren, men det er faktisk en veldig vanlig problemløsningsstrategi. Ikke nok med det, det er også en god måte å sikre at programmet vi skriver ikke kobles direkte med ett grafikkbibliotek. Hvis prosjektet FLTK viser seg å bli forlatt og forsvinner, så kan vi koble `Graph_lib` oppå et annet grafikkbibliotek.

Abstraksjonen sparer oss også for en god del hodebry og kodeduplisering. La oss si at du har skrevet en prosedyre som tegner en sirkel. Du lyst til å tegne flere sirkler, men med forskjellig radius, farge, strektykkelse, osv. Det er helt i orden, men plutselig finner du en mye bedre metode å tegne sirkelen på. Å oppdatere alle stedene du har tegnet sirkelen på blir fort et lite mareritt. En annen fordel er at når en sirkel tegnes, så skal den bare tegnes, vi må ikke fortelle hvordan det gjøres. Det gjør det lettere å forklare hva programmet vil gjøre når det kjører. Algoritmen som tegner en sirkel er gitt en gang, og for å endre hvordan alle sirkelene skal tegnes må du bare oppdatere denne algroritmen. Siden vi ikke må vite hvordan hvert punkt i sirkelen tegnes, kan vi enkelt koble programmet vårt opp mot f.eks en skjerm, eller en printer. Uten en lignende abstraksjon måtte vi ha skrevet om hele programmet for hver eneste anvendelse.

Polymorfi

For å bli kjent med kjøretidspolymorfi bør kapittel 14 leses. Her forklares det hvordan `Graph_lib` er bygget opp. Hovedpoenget i kapittelet er at objektorientert programmering i mange tilfeller gjør det enklere for den som skal programmere å tenke på hvordan et problem kan løses.

`Shape` er en abstrakt klasse. 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 intisialisere medlemsvariablene.

I tillegg skal klassen 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`.

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

Opprett en `Vector_ref<Animal>`, og legg til noen instanser av hver klasse i denne. Itererer gjennom vektoren og kall `toString()` på hvert element. For at du skal kunne bruke `Vector_ref` må du inkludere `Graph.h`, og bruke navnerommet `Graph_lib`. Når du bruker `Vector_ref` vil du få warning som sier: *warning: delete called on non-final 'Animal' that has virtual functions but non-virtual destructor*. Dette blir forklart senere i øvingen, så du kan ignorere den inntil videre.

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. Du vil også her få en warning som handler om `Vector_ref` som du kan ignorere.

2 Emoji (15%)

a) Emoji, en abstrakt baseklasse

På blackboard 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**.

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 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.

`Graph_lib` tar utgangspunkt i at vinduer opprettes på skjermen og at det tegnes former inne i vinduene. For at vinduet skal få kjennskap til hvilke former som skal tegnes i det, må formene kobles til vinduet. Nesten alle Emoji har forskjellig antall og typer former: åpne og lukkede øyne, hår, strekmunn, smilemunn, øyebryn, ører, osv. Det gjør at alle de forskjellige emojiklassene selv må ta ansvar for å koble sine egne former til et vindu for at de skal tegnes korrekt. 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 koble `Shape`-ene i hver enkelt emoji til et `Window`. Medlemsfunksjonen er *pure virtual* og heter `attach_to()`.

Klassene som arver fra `Emoji` og definerer konkrete emoji inneholder flere `Shape`-variabler. `Shape` har slettet sin kopikonstruktør og tilordningsoperator. En klasse som inneholder objekter av en type som har slettet disse definisjonene mister også selv disse egenskapene. Det går med andre ord ikke an å kopiere `Emoji`-objekter, selv om vi ikke eksplisitt har slettet muligheten. For å unngå feilmeldinger som er vanskelig å tolke er det en fordel å eksplisitt slette kopikonstruktøren og tilordningsoperatoren i `Emoji`.
Som om ikke det er nok, så mister også `Emoji` sin defaultkonstruktør når andre konstruktører deklareres. Og nettopp det skjedde da kopikonstruktøren ble slettet. Derfor er defaultkonstruktøren definert eksplisitt.

Alt dette resulterer i følgende deklarasjoner.

```
virtual void Emoji::attach_to(Graph_lib::Window&) = 0;

Emoji(const Emoji&) = delete;
Emoji& operator=(const Emoji&) = delete;

Emoji() {}
// Emoji() = default; // er et annet alternativ
```

Årsaken til at `Graph_lib::` er med her, er at det i mange tilfeller finnes en type som heter `Window` i det globale scopet allerede, og da skjønner ikke linkeren hva som foregår og avslutter med en feilmelding. Vær obs på dette i resten av øvingen. Det kan gi obskure feilmeldinger å unnlate dette.

`Vector_ref` gjør en del triks bak kulissene og den abstrakte klassen `Emoji` må derfor ha noe som kalles for en virtuell destruktør. Hva dette betyr lærer vi først senere i kurset. Følgende linje er derfor public i klassedeklarasjonen til `Emoji`. Hvis det ikke gjøres kan du få advarsler fra kompilatoren og det er like greit å unngå dette først som sist.

```
virtual ~Emoji() {}
```

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.

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. `attach_to()` 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 til dette passer `Circle` godt. Formen skal lagres i klassen.

Klassen skal ha en konstruktør med to parametre, `Point c, int r`. Bruk medlemsinitialiseringsliste til å konstruere `Circle`-objektet som skal være ansiktet med posisjonen `c`, og radius `r`.

c) Fargelegg ansiktet.

Gi også ansiktet en fyllfarge, f.eks. `Color::yellow` ved å kalle medlemsfunksjonen `set_fill_color(Color)` til `Circle`, fra konstruktøren til `Face`.

d) Koble ansiktet til vinduet. Overskriv `attach_to()` slik at sirkelen som representerer ansiktet kobles til et vindu. 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 i konstruktøren.

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. 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) Koble `EmptyFace` sine former til vinduet.

Øynene må kunne kobles til et vindu. `attach_to()` må overskrives for alle `Emoji`-deriverte klasser som legger til nye `Shape`-objekter.

Former tegnes i rekkefølgen de tilkobles et vindu. For at øynene skal vises må de kobles til vinduet etter ansiktet. Ansiktet fra `Face` kobles ikke automatisk til vinduet når `EmptyFace` overskriver `attach_to()`. Derfor må det eksplisitt kalles på `Face::attach_to()` for å tegne ansiktet.
`EmptyFace::attach_to(Graph_lib& Window)` bør derfor inneholde et kall til:
`Face::attach_to(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.*

Nyttig å vite: 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 klasse for dette i `Graph_lib`, `Arc`, men den er ikke en del av læreboken. Derfor blir den forklart her.

Bruk av Arc:

Når du skal tegne buer med `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`. FLTK kan kun garantere at 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.

Komplett klassedeklarasjon vises under.

```
class Arc : public Shape {
public:
    Arc(Point center, int widt, int height, int start_degree, int end_degree);

    void draw_lines() const override;

    void set_start(int degree);
    void set_end(int degree);
    void setw(int width);
    void seth(int height);

private:
    int width;
    int height;
    int start_degree;
    int end_degree;
};
```

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 `Vector_ref<Emoji>& emojis` og et vindu emojiene skal kobles til. Det er dette som gjør polymorfi ettertraktet. Selv om alle emoji-klassene er forskjellige kan samme grensesnitt, `attach_to()`, 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 legge til en `Arc` som representerer munnen.

Hvis du ikke får `Arc` til å fungere kan det hende at du har en gammel versjon av `TDT4102`-mappen i `C:\Program Files` (windows) eller `/Library` (mac). Last ned filene for øving 0 på nytt og bytt ut `TDT4102`-mappen i `C:\Program Files` eller `/Library` med den du lastet ned. Hvis du ikke får det til spør en stud.ass eller und.ass om hjelp.

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. `Ellipse` til å representere munnen. Kan du identifisere potensielle problemer med å arve fra f.eks. `SmilingFace`?