UiT
NORGES
ARKTISKE
UNIVERSITET

SENSORVEILEDNING

For eksamen i: **INF-1400 Objektorientert Programmering**
Dato: **Torsdag 29. september 2016**

Sensorveiledningen er på 7 sider inklusiv forside

Fagperson/intern sensor: John Markus Bjørndalen.
Telefon/mobil: 90148307

Les gjennom hele oppgavesettet før du begynner å løse oppgavene.

Oppgavene kan besvares på Norsk, Engelsk, Svensk eller Dansk.

I besvarelsen kan du bruke Python, pseudokode eller en kombinasjon.

Noen oppgaver har kun en “a)”. Dette er for å tydeliggjøre hva selve spørsmålet som skal besvares er, det har ikke falt ut noen delspørsmål.

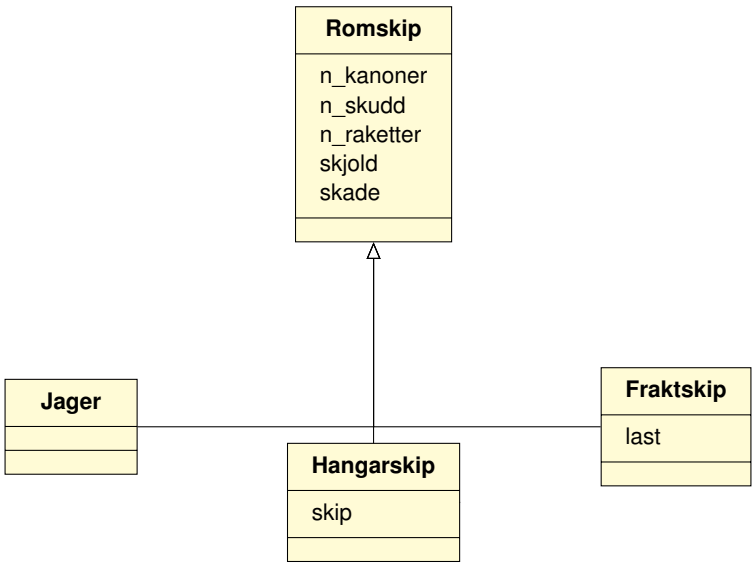
Noen norske termer som er brukt i stedet for de engelske fra boka:

- Klasse - class
- Arv - inheritance
- Atributt - Attribute
- Metode - Method

Del A

En veldig nyttig måte å lære programmering på er å delta i større prosjekter. Etter å ha meldt deg frivillig på et ambisiøst prosjekt som ønsker å lage et *mye* bedre spill enn No Man’s Sky oppdager du at man kanskje også kan lære noe om hvordan man *ikke* skal gjøre ting. Det er mange gode idéer her, men ting må ryddes opp i før noen kan begynne å leke seg med et enkelt tidlig spill.

Oppgave 1 (20%)



a) Implementer klassehierarkiet som er angitt i UML-diagrammet over. Få tydelig fram arv og attributter.

b) Skriv kode for å lage et objekt av typen Fraktskip. Hvilke attributter kan du nå fra objektet?

a) Det viktigste her er at studentene får med seg hvor hver attributt skal initialiseres og hvor hver attributt hører hjemme. Overføring av attributter er mindre viktig siden det ikke er angitt at de skal gjøre det.

```
class Romskip:
    def __init__(self, n_kanoner, n_skudd, n_raketter, skjold, skade):
        self.n_kanoner = n_kanoner
        self.n_skudd = n_skudd
        self.n_raketter = n_raketter
        self.skjold = skjold
        self.skade = skade

class Jager(Romskip):
    def __init__(self, n_kanoner, n_skudd, n_raketter, skjold, skade):
        super().__init__(n_kanoner, n_skudd, n_raketter, skjold, skade)

class Hangarskip(Romskip):
    def __init__(self, n_kanoner, n_skudd, n_raketter, skjold, skade, skip):
        super().__init__(n_kanoner, n_skudd, n_raketter, skjold, skade)
        self.skip = skip

class Fraktskip(Romskip):
    def __init__(self, n_kanoner, n_skudd, n_raketter, skjold, skade, last):
        super().__init__(n_kanoner, n_skudd, n_raketter, skjold, skade)
        self.last = last
```

b) Eksempel under. De kan nå attributtene listet i uml-boksene for Romskip+Fraktskip. Tester grunnleggende forståelse av arv.

```
skip = Fraktskip(2, 2000, 30, 40, 5, ['foo', 'bar'])
```

Oppgave 2 - 20 %

Vi antar at det kan være flere lag med i spillet som alle har mange romskip. For å regne ut styrken til hvert lag trenger vi en funksjon som regner ut styrken til et gitt lag basert på en liste over romskipene til laget.

```
POENG_JAGER = 10
POENG_HANGARSKIP = 100
POENG_FRAKTSKIP  =  70

def team_score(romskip):
    """Regner ut en styrkefaktor for et lag basert på en liste (romskip) med
    romskipene til laget"""
    pass
```

a) Lag funksjonen `team_score` som regner ut poeng basert på hvilken type hvert romskip er. Konstantene øverst (`POENG_JAGER`, `POENG_HANGARSKIP`, `POENG_FRAKTSKIP`) angir hvor mange poeng hver type romskip gir.

Ulempen med dette er at vi må endre `team_score` og legge til konstanter hvis vi ønsker å legge til flere klasser. Vi kan gjøre dette mer fleksibelt ved å erstatte konstantene over med en **klasseattributt** (`STYRKEPOENG`) i hver av romskipklassene.

b) Vis hvordan du gjør dette ved å bruke `Jager`-klassen som eksempel. Vis hvordan du implementerer `team_score` når du kan bruke klasseattributten STYRKEPOENG.

Et lite hint: funksjonen i b) burde bli merkbart enklere enn den i a).

a) Oppgaven tester at de kan finne typen til et objekt. De trenger ikke huske eksakt Python-syntax hvis de istedet forklarer hva de gjør og angir pseudokode.

```
def team_score_a(romskip):
    score = 0
    for r in romskip:
        if isinstance(r, Jager):
            score += POENG_JAGER
        elif isinstance(r, Hangarskip):
            score += POENG_HANGARSKIP
        elif isinstance(r, Fraktskip):
            score += POENG_FRAKTSKIP
    return score
```

b) Her tester vi en vanlig bruk av klasseattributter for å erstatte globale konstanter. Vi tester også at de skiller mellom objekt- og klasseattributter, men kan likevel gi noen poeng hvis de bruker objekt-attributter.

```
class Jager(Romskip):
    STYRKEPOENG = 10
    def __init__(self):
        super().__init__()

def team_score_b(romskip):
    return sum([r.STYRKEPOENG for r in romskip])
```

Oppgave 3 - 25 %

En romstasjon i spillet trenger forsvarsverk rundt seg. Foreløpig har vi bare lagt til raketter og kanoner. Weapon-klassen under kan vi riktignok bruke, men det vil være mer hensiktsmessig å *spesialisere*[1] den slik at vi enkelt kan legge til flere våpentyper i framtiden.

```
class Weapon:
    def __init__(self, weapon_type, cannon_shots=0, rockets=0, pos=None):
        self.weapon_type = weapon_type
        self.cannon_shots = cannon_shots
        self.rockets = rockets
        self.pos = pos

    def aim(self, target):
        """Returns a vector towards the target"""
         # placeholder, you don't need to change this method
        return []

    def fire(self, target):
        target_vector = self.aim(target)
        if self.weapon_type == "cannon":
            if self.cannon_shots > 0:
                print("Booom")
                objects.append(CannonBall(self.pos, target_vector))
                self.cannon_shots -= 1
        elif self.weapon_type == "rocket_launcher":
            if self.rockets > 0:
                print("Wooosh")
```

[1] Tenk i forhold til *generaliseringen* vi gjorde i oppgave 1

a) Bruk spesialisering av Weapon-klassen for å forbedre koden. Beskriv hvordan du ønsker å gjøre det og hvordan du bruker arv.

b) Implementer koden. Du trenger ikke å skrive noe av støttekoden rundt Weapon-klassen. Du trenger heller ikke å implementere `aim` (bare angi hvor du vil ha den).

Det finnes bedre løsninger enn den under (`fire` kunne vært generalisert bedre f.eks), men den vil være god nok. Det viktige her er at de får med seg hva som må gjøres i hver klasse.

```
class Weapon:
    """Nytt navn slik at jeg kan ha koden samlet i samme fil"""
    def __init__(self, pos=None):
        self.pos = pos

    def aim(self, target):
        """Returns a vector towards the target"""
         # placeholder, you don't need to change this method
        return []

    def fire(self, target):
        raise Exception("Base class should not fire")

class Cannon(Weapon):
    def __init__(self, cannon_shots=0, pos=None):
        super().__init__(pos=pos)
        self.cannon_shots = cannon_shots

    def fire(self, target):
        target_vector = self.aim(target)
        if self.cannon_shots >= 0:
            print("Booom")
            objects.append(CannonBall(self.pos, target_vector))
            self.cannon_shots -= 1

class RocketLauncher(Weapon):
    def __init__(self, rockets=0, pos=None):
        super().__init__(pos=pos)
        self.rockets = rockets

    def fire(self, target):
        target_vector = self.aim(target)
        if self.rockets >= 0:
            print("Woosh")
            objects.append(Rocket(self.pos, target_vector))
            self.rockets -= 1
```

Del B

Oppgave 4 (20%)

Noen sjekket inn kildekode i prosjektet, men ingen av de andre forstår helt hva den gjør og hvorfor. Den kjører stort sett uten problemer, men av og til krasjer spillet spektakulært. Klassenavnene kan umulig gi mening, men klassene brukes enkelte steder i koden, så vi må finne ut hva den gjør og hvor den feiler.

```
def significant_match(template_list, snarf):
    return True # TODO: can't be bothered today - fix later

class Platypus(object):
    def __init__(self, templates):
        self.templates = templates

    def oinker(self, snarf):
        return significant_match(self.templates, snarf)

class Zebra(Platypus):
    def __init__(self, templates):
        Platypus.__init__(self, templates)

    def oinker(self, snarf):
        return False # Cannot be

    def hoot(self, msg):
        print("cannot hoot - no owls present")


class Bat(Platypus):
    def __init__(self, templates):
        Platypus.__init__(self, templates)

    def hoot(self, msg):
        print("Almost owl...ish")


z = Zebra([1,2,3])
b = Bat([4,5,6])
p = Platypus([7,8,9])
```

For hvert av uttrykkene under, angi hva som skrives ut. Hvis noe feiler, angi hvorfor. Vi antar at `obs` er et objekt av typen `Observasjon`.

a) `print z.oinker(obs)`

b) `print b.oinker(obs)`

c) `print p.oinker(obs)`

d) `z.hoot("detected")`

e) `b.hoot("detected")`

f) `p.hoot("detected")`

De trenger ikke å få med at det blir en exception, men de bør få med at siste deloppgave kommer til å trigge en feil på grunn av en manglende metode.
Ellers tester oppgaven forståelse av hvordan arv og polymorfi faktisk virker.

a) `False`

b) `True`

c) `True`

d) `cannot hoot - no owls present`

e) `Almost owl...ish`

f)
```
Traceback (most recent call last):
  File "platypus_test.py", line 12, in <module>
    p.hoot("detected")
AttributeError: 'Platypus' object has no attribute 'hoot'}
```

Oppgave 6 -15%

a) Gi en kort beskrivelse av 2 av de følgende uttrykk/konsepter:

- Klasse vs. objekt
- Polymorfi (det holder å angi 1 variant av polymorfi)
- Multiple inheritance
- Mutable/immutable
- getter/setter

Det holder med en kort beskrivelse av hvert av de to punktene de velger som tar med hovedtrekkene/konseptet.
For polymorfi er det sannsynlig at de beskriver valg av metodeimplementasjon basert på objektets klasse, men de kan ha vært borti en av de andre variantene. Hvilken de velger er ikke viktig.