NTNU
Kunnskap for en bedre verden

Institutt for datateknikk og informasjonsvitenskap

Eksamensoppgave i TDT4100 Objektorientert programmering

Faglig kontakt under eksamen: Hallvard Trætteberg

Tlf.: 91897263

Eksamensdato: 22. mai

Eksamenstid (fra-til): 9.00-13.00

Hjelpemiddelkode/Tillatte hjelpemidler: C

Kun ”Big Java”, av Cay S. Horstmann, er tillatt.

Annen informasjon:

Oppgaven er utarbeidet av faglærer Hallvard Trætteberg og kvalitetssikret av Ragnhild Kobro Runde (Ifi, UiO).

Målform/språk: Bokmål

Antall sider: 5

Antall sider vedlegg: 0

Kontrollert av:

_____

Dato Sign

Dersom du mener at opplysninger mangler i en oppgaveformulering, gjør kort rede for de antagelser og forutsetninger som du finner nødvendig. Hvis du i en del er bedt om å *implementere* klasser og metoder og du ikke klarer det (helt eller delvis), så kan du likevel *bruke* dem i senere deler.

Del 1 – Typer og funksjonelle grensesnitt (15%)

a) Typen til Java-uttrykk er basert på typen til deluttrykkene. F.eks. har uttrykket **1 + 2** typen **int**, fordi **int** + **int** gir en **int**. Bestem og forklar typen til følgende uttrykk:
   - ”Java” + ”eksamen”
   - ”Java” + ”eksamen” == ”sant”
   - 1 / 2
   - ”0” + ”123”.charAt(5)
   - ”0123”.charAt(0) – ’0’

b) Anta at en har følgende variabel-deklarasjon og initialisering:
Collection<String> strings = new ArrayList<String>().
Hva er sammenhengen mellom typene på venstre- og høyresiden av tilordningstegnet (=)?
Hvordan påvirker **String**-spesialiseringen (altså det som står mellom **< >**) bruken av **strings**-variabelen?

c) Hva er et *funksjonelt* grensesnitt?

d) Det funksjonelle grensesnittet **Predicate<T>** er definert som følger:

```
public interface Predicate<T> {
    /**
     * Evaluates this predicate on the given argument.
     * Returns true if the input argument matches the predicate,
     * otherwise false
     */
    boolean test(T t);
}
```

Anta vi har en **Person**-klasse med metodene **getGender()** (returnerer **’F’** dersom personen er en kvinne, **’M’** om personen er en mann og **’\0’** om kjønnet er ukjent) og **getAge()** (returnerer alderen). Skriv en metode **getMatchingPersons** som tar inn en **Collection** av **Person**-objekter og et **Predicate** (også for personer) og returnerer en ny **Collection** med de personene som tilfredsstiller betingelsen angitt av **Predicate**-argumentet. Vis hvordan metoden kan kalles med et predikat som sier om personen skal kalles inn til sesjon, dvs. *er 18 år og mann*.

Del 2 – Segment-, Path- og Trip-klassene (55%)

I denne oppgaven skal du implementere klasser og metoder for å representere (buss)ruter/turer og kunne estimere gjenværende tid for resten av turen. Du kan tenke deg at dette brukes av koden i en info.tavle, som informerer passasjerene om antatt ankomsttid, eller en tjeneste for å få SMS en viss tid før ankomst (for de som skal hente passasjerer på destinasjonen).

En *strekning* består av en sekvens av *segmenter* med hver sin angitte *lengde* (**s**) og *(reise)tid* (**t**). Fra lengden og tiden kan en beregne (gjennomsnitts)*farten* **v = s/t**. Tiden som knyttes til et segment kan

Segment 1	Segment 2	Segment 3
s = 20	s = 80	s = 10
t = 1200	t = 3600	t = 550

være faktisk registrert reisetid, eller antatt reisetid (basert på historiske data), avhengig av hvilke data en velger å legge inn. Et eksempel med tre segmenter er vist i Figur 1 (**s** er km, mens **t** er sekunder):

Figur 1

En antar at farten innen et segment er jevn, og passer på å dele opp en strekning i segmenter ut fra denne antakelsen. En strekning fra én by til en annen vil kanskje være delt i tre segmenter (se Figur 1) én for veien ut av den ene byen, en for delen mellom og én for veien inn til den andre byen, siden hver av disse segmentene har helt ulike kjøreforhold og dermed gjennomsnittsfart. Dersom en vei har varierende fartsgrense, så kan en dele opp i kortere segmenter. Hvis antakelsen om jevn fart innen et segment er riktig, så kan en estimere tid mellom steder underveis på en strekning.

Du skal først implementere klasser for strekning og segment, som vi velger å kalle **Path** og **Segment**.

a) Skriv først **Segment**-klassen, med en konstruktør som tar inn lengde og tid for segmentet, og nødvendige felt. Det skal ikke være mulig å endre egenskapene etter opprettelsen. Skriv også én **get**-metode for hver av de tre egenskapene **distance** (lengde), **duration** (tid) og **speed** (fart).

b) Implementer **Path**-klassen, slik at den inneholder en sekvens av **Segment**-objekter. Det skal være mulig å initialisere **Path**-objekter med ingen, ett eller flere segmenter. Skriv også metoder som gjør det mulig å legge til segmenter og gå gjennom dem basert på indeks. Velg selv fornuftige metode-navn og signaturer.

c) Tegn objektdiagram for strukturen av **Path**- og **Segment**-objekter tilsvarende strekningen vist i Figur 1.

d) Skriv følgende metoder (eksemplene er alle med utgangspunkt i Figur 1):
- **double getDuration()** – returnerer (reise)tiden for hele strekningen. For tilstanden tilsvarende Figur 1 vil det være **1200** + **3600** + **550**.
- **double getDistance(Segment fromSegment, Segment uptoSegment)** – returnerer lengden fra og med **fromSegment** til, men ikke med, **uptoSegment**. Hvis **fromSegment** er **null**, så beregnes lengden fra starten av strekningen. Hvis **uptoSegment** er **null**, så beregnes lengden av resten av strekningen. Utløs et passende unntak hvis en eller begge argumentene ikke er en del av denne strekningen. Her er noen eksempler på argumenter og forventet resultat:

fromSegment	**uptoSegment**	**resultat**
Segment 1	Segment 3	20 + 80
null	Segment 2	20
Segment 2	null	80 + 10
null	null	20 + 80 + 10

- **Segment getSegmentAt(double distance, boolean next)** – returnerer segmentet som en er i etter at lengden **distance** er tilbakelagt (fra starten av strekningen). Dersom **distance** tilsvarer enden på et segment, så bestemmer **next**-argumentet om det er dette segmentet som skal returneres (**next** er **false**) eller evt. segmentet etter (**next** er **true**). Her er noen eksempler på argumenter og forventet resultat:

Segment 1	Segment 2	Segment 3
s = 20	s = 80	s = 10
t = 1400	t = 3650	t = 700

distance (s)	next	resultat
s = 0.0	false	null
s = 0.0	true	Segment 1
s = s1	false	Segment 1
s = s1	true	Segment 2
s > 20 og s < 20 + 80	false/true	Segment 2
s > 20 + 80 og s < 20 + 80 + 10	false/true	Segment 3
= 20 + 80 + 10	false	Segment 3
= 20 + 80 + 10	true	null

Klassen **Trip** representerer en faktisk (kjøre)tur og brukes for å oppdatere info.tavla på bussen med faktisk og estimert (reise)tid. Et **Trip**-objekt skal initialiseres med et **Path**-objekt som representerer *forventet* forløp, altså med segment-tider basert på historiske data. Et **Trip**-objekt skal også inneholde et **Path**-objekt med det *faktiske* forløpet, slik de registreres underveis på turen. Dette **Path**-objektet vil altså utvides med flere og flere **Segment**-objekter, etter hvert som segmentene passeres/tilbakelegges. Dette er illustrert i Figur 2:

Første segment er registrert (med **registerSegment**-metoden, se under). Lengden er forventet lengde, men det har gått litt saktere.	
Andre segment er registrert (med **registerSegment**). Her er både lengde og tid som forventet.	
Tredje segment er registrert (med **registerSegment**). Her har det gått litt saktere enn forventet.	

Figur 2

e) Implementer konstruktør og nødvendige felt for **Trip**. Implementer også **registerSegment**:
- **void registerSegment(double distance, double duration)** – denne metoden kalles med en ny måling når et nytt segment er passert/tilbakelagt. Et nytt **Segment**-objekt skal legges inn med data tilsvarende målingen. Men merk at **distance**- og **duration**-argumentene er målt fra *starten* av hele turen, så de kan ikke brukes direkte i det nye **Segment**-objektet. Hvis målt avstand ikke tilsvarer det en forventer ut fra *forventet* forløp eller det allerede er registrert nok segmenter, så skal det utløses et passende unntak.

f) **Trip**-klassen skal kunne brukes til å estimere tiden som *gjenstår* fra og med et punkt underveis, basert på *forventet* forløp (som ble angitt ved opprettelsen av **Trip**-objektet). Merk at dette punktet godt kan være innen et segment. Koden under er et forsøk på implementasjon. De to argumentene er tilbakelagt lengde og tid brukt så langt.

```
public double estimateTime(double distance, double duration) {
  double remainingTime = 0.0;
  for (Segment segment : this.getActualPath()) {
        distance -= segment.getDistance();
        if (distance < 0) {
```

```
                // gjenværende tid for segmented en er kommet til
                remainingTime = distance / segment.getSpeed();
            } else if (remainingTime > 0.0) {
                // legg til gjenværende tid for etterfølgende segment
                remainingTime += segment.getDuration();
            }
        }
        return remainingTime;
    }
```

Denne implementasjonen gjør et par antakelser om **Trip**- og **Path**-klassene, hvilke? Det er også (plantet) en liten logisk feil i koden, hvor og hva er feilen?

Del 3 – Arv, grensesnitt og delegering (15%)

a) Metoden for å estimere tiden som er brukt over, tar ikke i betraktning tiden en har brukt så langt. Hvis trafikken i et segment går vesentlig saktere enn forventet, f.eks. pga. veiarbeid, så vil estimatet bli for lavt. Forklar med tekst og kode hvordan en kan bruke *arv* for å lage flere varianter av **Trip**-klassen, med ulike teknikker for estimering.

b) Gitt følgende deklarasjon av et grensesnitt for estimering av gjenværende tid:

```
public interface TimeEstimator {
    double estimateTime(Path path, Path actualPath,
                        double distance, double duration);
}
```

Metoden **estimateTime** tar inn to **Path**-argumenter, det første representerer *forventet* forløp, det andre *faktisk* forløp, slik det er registrert så langt. De to siste argumentene er total lengde kjørt og tid brukt.

Forklar med tekst og kode hvordan du kan lage en eller flere implementasjon(er) av dette grensesnittet og bruke *delegeringsteknikken* for å implementere **Trip**-klassen sin **estimateTime**-metode. Hvor *fleksibel* er denne teknikken sammenlignet med bruken av arv i a)?

Del 4 – ArrivalNotifier-klassen (15%)

Anta at du skal lage en **ArrivalNotifier**-klasse som har ansvar for å sende SMS-er en viss tid før ankomst til destinasjonen. Dette kan være nyttig for de som skal plukke opp passasjerer når de kommer frem.

a) **ArrivalNotifier**-klassen trenger å lagre mobilnumre og hvor mange minutter tid i forkant før ankomst hvert enkelt nummer skal motta SMS. Forklar med tekst og/eller kode hvordan du vil gjøre det. Du kan anta at mobilnummer kan lagres som **String**.

b) Etterhvert som tiden går og turen forløper så må SMS-ene sendes ut. Det er to ting som må trigge utsendelse av SMS-er, at tiden går og at **Trip**-objektet endres og dermed potensielt tidsestimatet. Hvilken generell teknikk kan brukes til for å informere **ArrivalNotifier**-objektet om disse endringen? Forklar med tekst og/eller kode hvordan du vil anvende denne teknikken her. Gjør de antagelser du finner nødvendig om hjelpeklasser knyttet til tid.

NTNU – Trondheim
Norwegian University of
Science and Technology

Department of computer and information science

Examination paper for TDT4100
Object-oriented programming with Java

Academic contact during examination: Hallvard Trætteberg

Phone: 91897263

Examination date: 22. May

Examination time (from-to): 9:00-13:00

Permitted examination support material: C

Only ”Big Java”, by Cay S. Horstmann, is allowed.

Other information:

This examination paper is written by teacher Hallvard Trætteberg, with quality assurance by Ragnhild Kobro Runde (Ifi, UiO).

Language: English

Number of pages: 5

Number of pages enclosed: 0

Checked by:

Date Signature

If you feel necessary information is missing, state the assumptions you find it necessary to make. If you are not able to *implement* classes and method that a part asks for, you may still *use* these classes and methods later.

Del 1 – Types and functional interfaces (15%)

a) The type of a Java expressions is based on the type of the sub-expressions. E.g. **1 + 2** has the type **int**, because **int** + **int** gives an **int**. Determine and explain the type of the following expressions:
   - ”Java” + ”eksamen”
   - ”Java” + ”eksamen” == ”sant”
   - 1 / 2
   - ”0” + ”123”.charAt(5)
   - ”0123”.charAt(0) – ’0’

b) Assume the following variable declaration and initialisation:
Collection<String> strings = new ArrayList<String>().
What is the relation between the types on the left and right hand sides of the assignment (=)? How does the **String** specialisation (i.e. the part between < >) affect the use of the **strings** variable?

c) What is a *functional* interface?

d) The functional interface **Predicate<T>** is defined as follows:

```
public interface Predicate<T> {
    /**
     * Evaluates this predicate on the given argument.
     * Returns true if the input argument matches the predicate,
     * otherwise false
     */
    boolean test(T t);
}
```

Assume we have a **Person** class with the methods **getGender()** (returns **’F’** if the person is a woman, **’M’** if the person is a man and **’\0’** if the gender is undetermined) and **getAge()** (returns the age). Write a method **getMatchingPersons** that takes as arguments a **Collection** of **Person** objects and a **Predicate** (also for persons) and returns a new **Collection** with the persons that satisfy the **Predicate** argument. Show how the method can be called with a predicate that decides whether a person should be considered for military service, i.e. *is 18 years old and male*.

Del 2 – Segment-, Path- og Trip-klassene (55%)

In this part you will implement classes and methods for representing (bus) trips and estimating remaining time to the destination. The code could possibly be useful in an information board, that tells passengers about estimated time of arrival, or a service for receiving an SMS some time before arrival (for those picking up the passengers at the destination).

An *path* consists of a sequence of *segments*, each with a given *length* (**s**) and *(travel)time* (**t**). From the length and time we can compute the (average) *speed* **v = s/t**. The time for a segment can be actual

Segment 1	Segment 2	Segment 3
s = 20 t = 1200	s = 80 t = 3600	s = 10 t = 550

registered travel time, or assumed travel time (based on historical data), depending on what data are provided. An example with three segments are shown in Figure 1 (**s** is km, while **t** is seconds):

Figure 1

We assume the speed within a segment is invariant, and make sure to split a path in segments based on this. A path from one city to another will perhaps be divided into three segments (see Figure 1) one for the part out of the first city, one for the part between the cities and one for the part into the second city, since each of these parts have different driving conditions and thus average speed. If the road has varying speed limits, it can be split into even shorter segments. If the assumption of invariant speed within a segment is correct, you can estimate the time between places within a segment.

You'll first implement classes for path and segment.

a) First write the **Segment** class, with a constructor that takes as argument the length and time for the segment, and necessary fields. It should not be possible to change these properties after creation. Also write a **get** method for each of the three properties **distance** (length), **duration** (time) and **speed**.

b) Implement the **Path** class, so it contains a sequence of **Segment** objects. It should be possible to initialise a **Path** object with zero, one or more segments. Also write methods that make it possible to add segments and iterate over them based on index. Choose sensible method names and signatures.

c) Draw an object diagram for the structure of **Path** and **Segment** objects corresponding to the path shown in Figure 1.

d) Write the following methods (all the examples are based on Figure 1):
   - **double getDuration()** – returns the (travel)time for the whole path. For the state of Figure 1 it will be **1200 + 3600 + 550**.
   - **double getDistance(Segment fromSegment, Segment uptoSegment)** – returns the length from and including **fromSegment** to, but not including, **uptoSegment**. If **fromSegment** is **null**, the distance is computed from the beginning of the path. If **uptoSegment** is **null**, the distance is computed for the rest of the path. Throw a suitable exception if one or both of the arguments are not part of this path. Here are some examples of arguments and expected results:

fromSegment	**uptoSegment**	**resultat**
Segment 1	Segment 3	20 + 80
null	Segment 2	20
Segment 2	null	80 + 10
null	null	20 + 80 + 10

   - **Segment getSegmentAt(double distance, boolean next)** – returns the segment you are in after traveling the provided **distance** (from the beginning of the path). If **distance** corresponds to the end of a segment, then the **next** argument determines if that segment should be returned

Segment 1	Segment 2	Segment 3
s = 20	s = 80	s = 10
t = 1400	t = 3650	t = 700

(**next** is **false**) or the segment after (**next** is **true**). Here are some examples of arguments and expected results:

distance (s)	**next**	**resultat**
s = 0.0	false	null
s = 0.0	true	Segment 1
s = s1	false	Segment 1
s = s1	true	Segment 2
s > 20 og s < 20 + 80	false/true	Segment 2
s > 20 + 80 og s < 20 + 80 + 10	false/true	Segment 3
= 20 + 80 + 10	false	Segment 3
= 20 + 80 + 10	true	null

The **Trip** class represents an actual (driving) trip and is used for updating an information board on the bus with actual and estimated (travel) time. A **Trip** object must be initialised with a **Path** object representing *expected* travel, i.e. with segment times based on historical data. A **Trip** should also contain a **Path** object representing the *actual* travel, as registered during the trip. This **Path** object will be extended with **Segment** objects, as they are covered/passed. This is illustrated in Figure 2:

The first segment is registered (with the **registerSegment** method, see below). The length is as expected, but the speed is lower.	
The second segment is registered (with **registerSegment**). Both length and time is as expected.	
The third segment is registered (with **registerSegment**). The speed is a bit lower than expected.	

Figure 2

e) Implement constructor and necessary fields for **Trip**. Also implement **registerSegment**:
- **void registerSegment(double distance, double duration)** – this method is called with a new measurement when another segment is covered/passed. A new **Segment** object must be added with data for the measurement. Note that the **distance** and **duration** arguments are measured from the beginning of the trip, so they cannot be used as is in the new **Segment** object. If the measured distance is not what is *expected* or there are already enough segments, a suitable exception must be thrown.

f) It should be possible to use the **Trip** class for estimating the *remaining* travel time from a certain point, based on expected travel (provided when the **Trip** object was created). Note that this point can be within a segment. The code below is a suggested implementation. The two arguments are travel distance and duration so far.

```
public double estimateTime(double distance, double duration) {
  double remainingTime = 0.0;
  for (Segment segment : this.getActualPath()) {
        distance -= segment.getDistance();
```

```
        if (distance < 0) {
            // gjenværende tid for segmented en er kommet til
            remainingTime = distance / segment.getSpeed();
        } else if (remainingTime > 0.0) {
            // legg til gjenværende tid for etterfølgende segment
            remainingTime += segment.getDuration();
        }
    }
    return remainingTime;
}
```

This implementation makes some assumptions about the **Trip** and **Path** classes, which ones? There is also a (deliberate) small logical error in the code, where and what is it?

Del 3 – Inheritance, interface and delegation (15%)

a) The method used for estimating time used above, does not consider the time used so far. If the traffic in a segment is a lot slower than expected, e.g. due to road work, the estimate will be too low. Explain with text and code how you can utilise inheritance to make several variants of the **Trip** class, with different estimation techniques.

b) Given the following declaration of an interface for estimating remaining time:

```
public interface TimeEstimator {
    double estimateTime(Path path, Path actualPath,
                        double distance, double duration);
}
```

The **estimateTime** method takes two **Path** arguments, the first representing *expected* travel, and the other *actual* travel, as registered so far. The two last arguments are total length and time travelled.

Explain with text and code how you would make one or more implementations of this interface and use *delegation* to implement the **Trip** class' **estimateTime** method. How *flexible* is this technique compared to the use of inheritance in a)?

Del 4 – ArrivalNotifier-klassen (15%)

Assume you must make an **ArrivalNotifier** class with the task of sending SMS-es a certain time before a bus arrives at its destination. This can useful for those picking up arriving passengers.

a) The **ArrivalNotifier** class needs to store mobile phone numbers and how many minutes in advance before arrival each number should receive an SMS. Explain with text and/or code how you will do it. You can assume mobile numbers can be represented as a **String**.

b) As time passes and the trip proceeds, the SMS-es must be sent. There are two things that must trigger sending SMS-es, the passing of time and that the **Trip** object is changed and possibly the time estimate. What generic technique can be used for notifying the **ArrivalNotifier** object about these events? Explain with text and/or code how you will use this technique here. Make assumptions about supporting time related classes that you find necessary.

NTNU
Kunnskap for ei betre verd

Institutt for datateknikk og informasjonsvitenskap

Eksamensoppgåve i TDT4100
Objektorientert programmering

Fagleg kontakt under eksamen: Hallvard Trætteberg

Tlf.: 91897263

Eksamensdato: 22. mai

Eksamenstid (frå-til): 9.00-13.00

Hjelpemiddelkode/Tillatne hjelpemiddel: C

Berre ”Big Java”, av Cay S. Horstmann, er tillaten.

Annan informasjon:

Oppgåva er utarbeidd av faglærer Hallvard Trætteberg og kvalitetssikra av Ragnhild Kobro Runde (Ifi, UiO).

Målform/språk: Nynorsk

Sidetal: 5

Sidetal vedlegg: 0

Kontrollert av:

_____

Dato Sign

Om du meiner at opplysningar manglar i ein oppgåveformulering, gjer kort greie for dei føresetnader som du finn naudsynt. Om du i ein del er beden om å implementere klasser og metodar og du ikkje klarar det (heilt eller delvis), så kan du likevel nytte dei i seinare delar.

Del 1 – Typar og funksjonelle grensesnitt (15%)

a) Typen til Java-uttrykk er basert på typen til deluttrykka. T.d. har uttrykket **1 + 2** typen **int**, fordi **int** + **int** gjev ein **int**. Bestem og forklar typen til fylgjande uttrykk:
   - ”Java” + ”eksamen”
   - ”Java” + ”eksamen” == ”sant”
   - 1 / 2
   - ”0” + ”123”.charAt(5)
   - ”0123”.charAt(0) – ’0’

b) Anta ein har fylgjande variabel-deklarasjon og initialisering:
Collection<String> strings = new ArrayList<String>().
Korleis heng typane på venstre- og høgresida av tilordningsteknet (=) saman? Korleis verker **String**-spesialiseringa (altså det som står mellom < >) inn på bruken av **strings**-variabelen?

c) Kva er eit *funksjonelt* grensesnitt?

d) Det funksjonelle grensesnittet **Predicate<T>** er definert som fylgjer:

```
public interface Predicate<T> {
    /**
     * Evaluates this predicate on the given argument.
     * Returns true if the input argument matches the predicate,
     * otherwise false
     */
    boolean test(T t);
}
```

Anta vi har ein **Person**-klasse med metodane **getGender()** (returnerer **’F’** om personen er ei kvinne, **’M’** om personen er ein mann og **’\0’** om kjønnet er ukjend) og **getAge()** (returnerer alderen). Skriv ein metode **getMatchingPersons** som tek inn ein **Collection** av **Person**-objekt og eit **Predicate** (og for personar) og returnerer ein ny **Collection** med dei personane som tilfredsstiller krava angitt av **Predicate**-argumentet. Vis korleis metoden kan kallast med eit predikat som seier om personen skal kallast inn til sesjon, dvs. *er 18 år og mann*.

Del 2 – Segment-, Path- og Trip-klassene (55%)

I denne oppgåva skal du implementere klasser og metodar for å representere (buss)ruter/turar og kunne estimere tida som er att for resten av turen. Du kan tenkje deg at dette nyttast av koden i en informasjonstavle, som informerer passasjerane om antatt tid for ankomst, eller ei teneste for å få SMS ein viss tid før ankomst (for dei som skal hente passasjerar på destinasjonen).

En *strekning* består av en sekvens av *segmenter* med kvar si gjevne *lengde* (**s**) og *(reise)tid* (**t**). Frå lengda og tida kan ein berekne (gjennomsnitts)*farten* **v = s/t**. Tida som knytast til eit segment kan være faktisk registrert reisetid, eller antatt reisetid (basert på historiske data), avhengig av kva for data en vel å leggje inn. Eit døme med tre segment er vist i Figur 1 (**s** er km, mens **t** er sekunder):

Segment 1	Segment 2	Segment 3
s = 20 t = 1200	s = 80 t = 3600	s = 10 t = 550

Figur 1

Ein antar at farta innan eit segment er jamn, og passer på å dele opp ei strekning i segmenter ut frå det. En strekning frå ein by til ein annan vil kanskje vere delt i tre segment (sjå Figur 1) ein for vegen ut av den eine byen, ein for delen mellom og ein for vegen inn til den andre byen, sidan kvar av desse segmenta har heilt ulike kjøreforhold og dermed gjennomsnittsfart. Dersom ein vei har varierande fartsgrense, så kan en dele opp i kortare segment. Om antakinga om jamn fart innan eit segment er riktig, så kan ein estimere tid mellom steder underveis på ein strekning.

Du skal fyrst implementere klasser for strekning og segment, som vi vel å kalle **Path** og **Segment**.

a) Skriv fyrst **Segment**-klassen, med ein konstruktør som tar inn lengde og tid for segmentet, og naudsynte felt. Det skal ikkje vere mogleg å endre eigenskapa etter opprettinga. Skriv og ein **get**-metode for kvar av dei tre eigenskapa **distance** (lengde), **duration** (tid) og **speed** (fart).

b) Implementer **Path**-klassen, slik at den inneheld ein sekvens av **Segment**-objekt. Det skal vere mogleg å initialisere **Path**-objekt med inga, eitt eller fleire segment. Skriv og metodar som gjer det mogleg å leggje til segment og gå gjennom dei basert på indeks. Vel sjølv fornuftige metode-namn og signaturar.

c) Teikn objektdiagram for strukturen av **Path**- og **Segment**-objektar som svarar til strekninga vist i Figur 1.

d) Skriv fylgjande metodar (døma er alle med utgangspunkt i Figur 1):
- **double getDuration()** – returnerer (reise)tida til heile strekninga. For tilstanden som svarar til Figur 1 vil det være **1200 + 3600 + 550**.
- **double getDistance(Segment fromSegment, Segment uptoSegment)** – returnerer lengda frå og med **fromSegment** til, men ikkje med, **uptoSegment**. Om **fromSegment** er **null**, så bereknast lengda frå starten av strekninga. Om **uptoSegment** er **null**, så bereknast lengda av resten av strekninga. Utløys et passande unntak om ein eller båe argumenta ikkje er ein del av denne strekninga. Her er nokre døme på argument og forventa resultat:

fromSegment	**uptoSegment**	**resultat**
Segment 1	Segment 3	20 + 80
null	Segment 2	20
Segment 2	null	80 + 10
null	null	20 + 80 + 10

- **Segment getSegmentAt(double distance, boolean next)** – returnerer segmentet som ein er i etter at lengda **distance** er tilbakelagt (frå starten av strekninga). Dersom **distance** svarar til enden på eit segment, så bestemmer **next**-argumentet om det er dette segmentet som skal returnerast (**next** er **false**) eller evt. segmentet etter (**next** er **true**). Her er nokre døme på argument og forventa resultat:

distance (s)	**next**	**resultat**

Segment 1	Segment 2	Segment 3
s = 20 t = 1400	s = 80 t = 3650	s = 10 t = 700

s = 0.0	false	null
s = 0.0	true	Segment 1
s = s1	false	Segment 1
s = s1	true	Segment 2
s > 20 og s < 20 + 80	false/true	Segment 2
s > 20 + 80 og s < 20 + 80 + 10	false/true	Segment 3
= 20 + 80 + 10	false	Segment 3
= 20 + 80 + 10	true	null

Klassen **Trip** representerer en faktisk (kjøre)tur og nyttast for å oppdatere info.tavla på bussen med faktisk og estimert (reise)tid. Et **Trip**-objekt skal initialiserast med et **Path**-objekt som representerer *forventa* forløp, altså med segment-tider basert på historiske data. Et **Trip**-objekt skal og innehalde eit **Path**-objekt med den *faktiske* forløpet, slik de registrerast undervegs på turen. Dette **Path**-objektet vil altså utvidast med fleire og fleire **Segment**-objekt, etter kvart som segmenta passerast/tilbakeleggjast. Dette er illustrert i Figur 2:

Første segment er registrert (med **registerSegment**-metoden, se under). Lengden er forventet lengde, men det har gått litt saktere.	
Andre segment er registrert (med **registerSegment**). Her er både lengde og tid som forventet.	
Tredje segment er registrert (med **registerSegment**). Her har det gått litt saktere enn forventet.	

Figur 2

e) Implementer konstruktør og naudsynte felt for **Trip**. Implementer og **registerSegment**:
- **void registerSegment(double distance, double duration)** – denne metoden kallast med ein ny måling når eit nytt segment er passert/tilbakelagt. Eit nytt **Segment**-objekt skal leggjast inn med data som svarar til målinga. Men merk at **distance**- og **duration**-argumenta er målt frå *starten* av hele turen, så dei kan ikkje nyttast direkte i det nye **Segment**-objektet. Om målt avstand ikkje svarar det ein forventar ut frå *forventa* forløp eller det allereie er registrert nok segmenter, så skal det utløysast et passande unntak.

f) **Trip**-klassen skal kunne nyttast til å estimere tida som *står att* frå og med eit punkt undervegs, basert på *forventa* forløp (som ble gjeve inn ved opprettinga av **Trip**-objektet). Merk at dette punktet godt kan vere innan eit segment. Koden under er eit forsøk på implementasjon. Dei to argumenta er tilbakelagt lengde og tid brukt så langt.

```
public double estimateTime(double distance, double duration) {
  double remainingTime = 0.0;
  for (Segment segment : this.getActualPath()) {
        distance -= segment.getDistance();
        if (distance < 0) {
              // gjenværende tid for segmented en er kommet til
```

```
                remainingTime = distance / segment.getSpeed();
            } else if (remainingTime > 0.0) {
                // legg til gjenværende tid for etterfølgende segment
                remainingTime += segment.getDuration();
            }
        }
        return remainingTime;
    }
```

Denne implementasjonen gjer eit par føresetnader om **Trip**- og **Path**-klassene, kva for føresetnader? Det er og (planta) ein liten logisk feil i koden, kor og kva er feilen?

Del 3 – Arv, grensesnitt og delegering (15%)

a) Metoden for å estimere tida som er nytta over, tar ikkje i betraktning tida ein har brukt så langt. Om trafikken i eit segment går vesentleg saktare enn forventa, t.d. pga. vegarbeid, så vil estimatet bli for lågt. Forklar med tekst og kode korleis ein kan nytta *arv* for å lage fleire variantar av **Trip**-klassen, med ulike teknikkar for estimering.

b) Gitt fylgjande deklarasjon av et grensesnitt for estimering av tida som står att:

```
public interface TimeEstimator {
    double estimateTime(Path path, Path actualPath,
                        double distance, double duration);
}
```

Metoden **estimateTime** tar inn to **Path**-argument, det fyrste representerer *forventa* forløp, det andre *faktisk* forløp, slik det er registrert så langt. De to siste argumenta er total lengde kjørt og tid nytta.

Forklar med tekst og kode korleis du kan lage ein eller fleire implementasjon(ar) av dette grensesnittet og nytte *delegeringsteknikken* for å implementere **Trip**-klassen sin **estimateTime**-metode. Kor *fleksibel* er denne teknikken samanlikna med bruken av arv i a)?

Del 4 – ArrivalNotifier-klassen (15%)

Anta at du skal lage en **ArrivalNotifier**-klasse som har ansvar for å sende SMS-er en viss tid før ankomst til destinasjonen. Dette kan være nyttig for dei som skal plukke opp passasjerar når dei kjem fram.

a) **ArrivalNotifier**-klassen treng å lagre mobilnumre og kor mange minutt tid i forkant før ankomst kvart enkelt nummer skal teke imot SMS. Forklar med tekst og/eller kode korleis du vil gjere det. Du kan anta at mobilnummer kan lagrast som **String**.

b) Etterkvart som tida går og turen held fram så må SMS-ene sendast ut. Det er to ting som må trigge utsending av SMS-er, at tida går og at **Trip**-objektet endrast og dermed potensielt tidsestimatet. Kva for generell teknikk kan nyttast til for å informere **ArrivalNotifier**-objektet om desse endringane? Forklar med tekst og/eller kode korleis du vil nytte denne teknikken her. Gjør dei føresetnadene du finner naudsynt om hjelpeklasser knytt til tid.