NTNU
Institutt for datateknikk
og informasjonsvitenskap

Eksamensoppgave i

TDT4100 – Objektorientert programmering

Fredag 20. mai 2011, kl. 09:00 - 13:00

Oppgaven er utarbeidet av faglærer Hallvard Trætteberg og kvalitetssikrer Trond Aalberg.
Kontaktperson under eksamen Hallvard Trætteberg (mobil 918 97263)

Språkform: Bokmål
Tillatte hjelpemidler: C
Én valgfri lærebok i Java er tillatt.
Bestemt, enkel kalkulator tillatt.

Sensurfrist: Fredag 10. juni.

Les oppgaveteksten nøye. Finn ut hva det spørres etter i hver oppgave.

Dersom du mener at opplysninger mangler i en oppgaveformulering gjør kort rede for de antagelser og forutsetninger som du finner det nødvendig å gjøre.

Del 1 – Innkapsling (25%)

a) En kan kategorisere innkapslingsmetoder som enten lese- eller endringsmetoder. Hva er den viktigste oppgaven til endringsmetodene, bortsett fra å utføre selve endringen?

b) Gitt en **Date**-klasse med metoder **getDay()** for å lese datoens dag (1-31), **getMonth()** for måned (1-12) og **getYear()** for år (0-99). Skriv kode for en **Person**-klasse, med felt og innkapslingsmetoder for fødselsdato og kjønn. Kjønn skal kun kunne settes ved oppretting av **Person**-objektet, mens det for fødselsdatoen ikke er noen slik begrensning.

c) Skriv kode for å lagre og innkapsle personnummer i **Person**-objekter. Du kan anta at fødselsdato og kjønn allerede er satt. Et personnummer består grovt sett av fødselsdatoen, et (vilkårlig) løpenummer og to kontrollsifre. Kontrollsifrene gjør det enklere å sjekke om et personnummer er ekte. Mer spesifikt er reglene for personnummer som følger:

- Et personnummer består av 11 siffer, med følgende struktur: **D**1**D**2**M**1**M**2**Y**1**Y**2**N**1**N**2**N**3**K**1**K**2 (fargen illustrerer siffergruppene).
- De seks første sifrene, **D**1**D**2**M**1**M**2**Y**1**Y**2, tilsvarer fødselsdatoens dag (1-31), måned (1-12) og år (0-99).
- De tre neste sifrene, **N**1**N**2**N**3, kan antas å være vilkårlige, men **N**3 må være *partall* for kvinner og *oddetall* for menn.
- De to siste sifrene, **K**1**K**2, er kontrollsifre, som hver for seg beregnes ut fra de foregående sifrene. Formellen for dem begge er **11 – (VS % 11)**[1], hvor **VS** (veid sum) for **K**1 er **D**1***F**1 + **D**2***F**2 + … + **N**2***F**8 + **N**3***F**9 og **VS** for **K**2 er **D**1***G**1 + **D**2***G**2 + … + **N**3***G**9 + **K**1***G**10. **F**'ene og **G**'ene er oppgitt i tabellen under. Dersom formelen gir et ett-sifret resultat for både **K**1 *og* **K**2 så er personnummeret gyldig, mens gir formelen et to-sifret resultat for **K**1 og/eller **K**2, så er personnummeret ugyldig.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
F	3	7	6	1	8	9	4	5	2	
G	5	4	3	2	7	6	5	4	3	2

Introduser og bruk gjerne hjelpemetoder for å gjøre koden ryddigere.

d) Beskriv hvordan du kan bruke en såkalt *checked exception* for å avverge at fødselsdatoen endres etter at personnummeret er satt! Hva slags konsekvenser vil dette ha for kode som kaller endringsmetoden for fødselsdatoen?

Del 2 – Klasser (40%)

Du skal implementere klasser for å representere informasjon ifm. gjennomføring av seriespill i sporter som fotball, håndball, volleyball osv. Klassen **MatchResult** skal representere informasjon om en kamp og dens resultatet, mens klassen **LeagueTable** skal representere et sett kamper og beregne serietabellen basert på kampresultatene. I del 2 skal vi begrense oss til tabeller for *ferdigspilte fotballkamper*. Et eksempel på to kampresultater og en tabell er vist nedenfor:

```
Rosenborg - Lillestrøm: 4 - 4
Lillestrøm - Rosenborg: 1 - 5
```

[1] % er modulo-operatoren

```
Rosenborg   4
Lillestrøm  1
```

Som vi ser har Rosenborg fått totalt 4 poeng, 3 poeng for seier og 1 poeng for uavgjort, mens Lillestrøm har fått 1 poeng for den ene uavgjort-kampen. Tabellen utelater informasjon om mål for og imot og hjemme- og bortekamper.

Tenk på helheten i løsningen før du går i gang med hver del. Kanskje kan det være lurt å begynne med c) før du skrive koden for a) og b).

Det kan være lurt å ha i bakhodet at i del 3 skal koden generaliseres til å håndtere sporter med andre regler for å beregne tabeller og i del 4 skal koden håndtere at kamper kan legges inn i tabellen før de er ferdigspilt. Men la ikke dette gjøre koden i del 2 unødvendig komplisert.

a) Implementer **MatchResult**:
- Representasjon og innkapsling av informasjon om navnene på hjemme- og bortelagene. Navnene skal ikke kunne endres etter at et **MatchResult**-objekt er laget.
- Representasjon og innkapsling av informasjon om kampresultatet, dvs. antall hjemme- og bortemål.
- Metodene **isParticipant(String participant)** for å spørre om (laget) **participant** spiller i denne kampen, **isDraw()** for å spørre om resultatet ble uavgjort og **isWinner(String participant)** for å spørre om (laget) **participant** vant denne kampen.

b) Implementer **LeagueTable**:
- Representasjon av (navnene til) alle lagene. Navnene skal ikke kunne endres etter at **LeagueTable**-objektet er laget.
- Representasjon av informasjon om kampresultater, dvs. **MatchResult**-objekter. Det skal kun være lov å legge inn nye kamper, ikke fjerne dem. Det skal ikke være lov å legge inn kamper for lag som ikke er med i tabellen.
- Metoden **getParticipantPoints(MatchResult matchResult, String participant)** som returnerer antall poeng som (laget) **participant** fikk for resultatet **matchResult**. Ta høyde for at **participant** faktisk ikke er med i **matchResult**.

Tabellpoeng summeres basert på kampresultatene, og tabellen sorteres etter antall poeng, med laget med flest poeng øverst. Når et kampresultat legges inn, så skal tabellen oppdateres.

Merk at du antageligvis vil trenge en klasse for å representere hver rad i tabellen, f.eks. kalt **LeagueTableRow**, som lagrer data om et lag og summen av poengene laget har fått for seier og uavgjort i sine kamper. Denne klassen kan implementere relevante grensesnitt ifm. sortering.

c) Tegn et objekt/instansdiagram (figur som viser instansene i en objektstruktur og deres innhold) som tilsvarer tabellen over, for strukturen av **MatchResult**-, **LeagueTable**- og evt. **LeagueTableRow**-objekter.

d) I en ordentlig fotballtabell så spiller også totalt antall mål for og imot og resultat i innbyrdes kamper inn på tabellen. Beskriv kort med tekst og kode hvordan du vil støtte dette.

Del 3 – Generalisering og arv (20%)

Ulike sporter har ulike regler for poenggiving i både kamper og serietabeller og sortering av tabellene. I tennis er kampen slutt når én deltaker har vunnet 2 eller 3 sett for hhv. kvinner og menn, så sluttresultatet 1-0 finnes ikke i tennis. Fotball gir 3 tabellpoeng for seier, mens håndball gir 2. I noen sporter teller innbyrdes oppgjør før målforskjell ved sortering, mens det for andre er omvendt.

a) Forklar med tekst og kode hvordan du kan lage sport-spesifikke resultat-klasser, f.eks. **TennisResult**, som arver fra **MatchResult** og implementerer logikk for å håndheve sportens regler for poenggiving, bl.a. for å unngå registrering av ugyldige resultater. Beskriv også nødvendige endringer av **MatchResult**.

b) Forklar med tekst og kode hvordan du kan lage sport-spesifikke tabell-klasser, f.eks. **FootballLeagueTable** og **TennisTournamentTable**, som arver fra **LeagueTable** og implementerer logikk iht. sportens regler for å beregne tabellpoeng. Beskriv også nødvendige endringer av **LeagueTable**.

Del 4 – Diverse (15%)

a) Hva slags generell teknikk kan anvendes for å sikre at en serietabell automatisk oppdateres hvis et **MatchResult**-objekt endres etter at det er lagt inn i tabellen? Forklar med tekst og kode hvordan du vil implementere støtte for dette. Fokusér på å forklare hvordan den generelle teknikken anvendes/tilpasses, ikke på å få logikken for poengberegning helt riktig (kan være litt intrikat i dette tilfellet).

b) Forklar hvordan du vil teste at:
- metoden **isWinner(…)** i 2 a) pkt. 3 er riktig implementert
- innbyrdes sammenligning av tabellrader i 2 b) er korrekt
- løsningen i 1 d) er korrekt

Du trenger ikke skrive kode, men forklaringen må være detaljert nok til å illustrere testemetoden.

NTNU
Institutt for datateknikk
og informasjonsvitenskap

Exam for

TDT4100 – Object-oriented programming

Friday 20. may 2010, 09:00 - 13:00

The exam is made by responsible teacher Hallvard Trætteberg and quality assurer Trond Aalberg.
Contact person during the exam is Hallvard Trætteberg (mobile 918 97263)

Language: English
Supporting material: C
One printed Java textbook is allowed. Other printed books are not allowed.

Deadline for results: Friday 10. June.

Read the text carefully. Make sure you understand what you are supposed to do.

If information is missing you must clarify what assumptions you find necessary.

Note the percentages for each part, so you use your time wisely.

Part 1 – Encapsulation (25%)

a) You can categorise encapsulation methods as either read or modify methods. What is the main task of the modify methods, besides performing the modification itself?

b) Given a **Date** class with methods **getDay()** for reading the date's day (1-31), **getMonth()** for the month (1-12) and **getYear()** for the year (0-99). Write code for a **Person** class, with fields and encapsulation methods for date of birth and gender. You should only be able to set the gender when the **Person** object is created, while there for the date of birth is no such limitation.

c) Write code for storing and encapsulating person identification numbers (PIDs) in **Person** objects. You may assume that the date of birth and gender is set alread. A PID consists of the date of birth, a (arbitrary) counter and two control digits. The control digits makes it easier to detect illegal/faulty PIDs. The full rules for PIDs are as follows:

- A PID consists of 11 digits, with the following structure: **D**1**D**2**M**1**M**2**Y**1**Y**2**N**1**N**2**N**3**K**1**K**2 (the shading illustrates the grouping of the digits).
- The first six digits, **D**1**D**2**M**1**M**2**Y**1**Y**2, are the date of birth's day (1-31), month (1-12) and year (0-99).
- The next three digits, **N**1**N**2**N**3, are (almost) arbitrary, but **N**3 must be *even* for women/females and *odd* for men/males.
- The last two digits, **K**1**K**2, are control digits, both of which are computed from the preceding digits. The formula for both is **11 – (WS % 11)**[2], where **WS** (weighted sum) for **K**1 is **D**1***F**1 + **D**2***F**2 + … + **N**2***F**8 + **N**3***F**9 and **WS** for **K**2 is **D**1***G**1 + **D**2***G**2 + … + **N**3***G**9 + **K**1***G**10. The **F**'s and **G**'s are given in the table below. If the formula results in a one-digit value for both **K**1 *and* **K**2 the PID is considered valid, whereas if the formula results in a two-digit value for **K**1 and/or **K**2 the PID is considered invalid.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
F	3	7	6	1	8	9	4	5	2	
G	5	4	3	2	7	6	5	4	3	2

Introduce and use helper methods, to make the code more tidy.

d) Describe how you can use a so-called *checked exception* to prevent that the date of birth is changed after the PID is set! What consequences will this have for the code that calls the modify method for the date of birth?

Part 2 – Classes (40%)

You will implement classes for representing information about leagues/series for sports like football, handball, volleyball etc. The class **MatchResult** will represent information about a match and its result, while the class **LeagueTable** will represent a set of matches and compute the table based on the match results. In part 2 we will constrain the solution to tables for *finished football matches*. An example of two match results and a table is shown below:

```
Rosenborg - Lillestrøm: 4 - 4
Lillestrøm - Rosenborg: 1 - 5
```

[2] % is the modulo operator

```
Rosenborg   4
Lillestrøm  1
```

We see that Rosenborg has a total of 4 points, 3 points for a victory and 1 for a draw, while Lillestrøm has 1 points for a draw. The table omits information about goals for and against and results home and away.

Consider the whole solution, before doing each part. It may be best to start with c) before writing the actual code for a) and b).

You should also keep in mind that we in part 3 will generalise to other sports and other rules for computing tables and we in part 4 will handle unfinished matches. But don't let this make the kode in part 2 more complicated than necessary.

a) Implement **MatchResult**:
- Represent and encapsulate information about the names of the two teams (home and away). It should not be possible to change the names after a **MatchResult** object has been created.
- Represent and encapsulate information about the match result, i.e. the number of goals for each team.
- The metods **isParticipant(String participant)** for asking if (the team) **participant** plays in this match, **isDraw()** to ask if the result was a draw and **isWinner(String participant)** to ask if (the team) **participant** won this match.

b) Implement **LeagueTable**:
- Represent (the names of) all the teams. It should not be possible to change the names after a **LeagueTable** object has been created.
- Represent information about match results, i.e. **MatchResult** objects. It should only be possible to add new matches, not remove them. It should not be possible to add matches for teams that are not part of the table.
- The method **getParticipantPoints(MatchResult matchResult, String participant)** that returns the number of points that (the team) **participant** received for the result **matchResult**. Take into consideration that **participant** may not actually play in **matchResult**.

Table points are aggregated based on the match results, and the table is sorted on the number of points, with the team with the highest number of points at the top. The table must be updated when a new match result is added.

Note that you will probably need a class for representing each row in the table, e.g. named **LeagueTableRow**, that stores data about a team and the sum of the points that this team has received for victories and draws is its matches. This class can implement relevant interfaces concerned with sorting.

c) Draw an object/instance diagram (figure that shows the instances in an object structure and their content) that corresponds to the table above, for the structure of **MatchResult**, **LeagueTable** and possibly **LeagueTableRow** objects.

d) A real football league table considers goals for and against and the results in mutual matches in the sorting of the table. Briefly describe with text and code how you can support this.

Part 3 – Generalisation and inheritance (20%)

Different sports have different rules for earning points in both matches and league tables and sorting the tables. In tennis a game when one participant has won 2 or 3 sets for women and men, respectively, so the match result 1-0 is not possible in tennis. In football you get 3 table points for victory, whereas in handball you get 2. In some sports the results of mutual games are considered before goals for and against when sorting, in others the priority is the opposite.

a) Explain with text and code how you can make sport-specific result classes, e.g. **TennisResult**, that inherits from **MatchResult** and implements logic for ensure the rules of this sport are followed, among others to prevent registering invalid results. Describe changes to **MatchResult** that you consider necessary.

b) Explain with text and code how you can make sport-specific league tables, e.g. **FootballLeagueTable** and **TennisTournamentTable**, that inherits from **LeagueTable** and implements logic according to this sport's rules for computing table points. Describe changes to **LeagueTable** that you consider necessary.

Part 4 – Various (15%)

a) Which general technique can be used for ensuring that a league table is automatically updated if a **MatchResult** object is modified after it has been added to the table? Explain with text and code how you would implement support for this. Focus on explaining how the general technique is used/adapted and not on getting the logic for computing points completely correct (can be a bit tricky in this case).

b) Explain how you would test that:
- the implementation of method **isWinner(…)** in 2 a) bullet 3 is correct
- comparison of table rows in 2 b) is correct
- the solution for 1 d) is correct

You don't need to write code, but the explanation must be detailed enough to illustrating the testing method.

NTNU
Institutt for datateknikk
og informasjonsvitenskap

Eksamensoppgåve i

TDT4100 – Objektorientert programmering

Fredag 20. mai 2011, kl. 09:00 - 13:00

Oppgåva er utarbeidd av faglærar Hallvard Trætteberg og kvalitetssikra av Trond Aalberg.
Kontaktperson under eksamen Hallvard Trætteberg (mobil 918 97263)

Språkform: Nynorsk
Tillatne hjelpemiddel: C
Éi valfri lærebok i Java er tillaten.
Bestemt, enkel kalkulator tillaten.

Sensurfrist: Fredag 10. juni.

Les oppgåveteksten nøye. Finn ut kva det er spurt etter i kvar oppgåve.

Dersom du meiner at opplysningar manglar i ei oppgåveformulering, gjer kort greie for dei føresetnadene som du finn naudsynte.

Del 1 – Innkapsling (25%)

a) Ein kan kategorisere innkapslingsmetodar som anten lese- eller endringsmetodar. Kva er den viktigaste oppgåva til endringsmetodane, bortsett frå å utføre sjølve endringa?

b) Gitt ein **Date**-klasse med metodar **getDay()** for å lese datoen sin dag (1-31), **getMonth()** for månad (1-12) og **getYear()** for år (0-99). Skriv kode for ein **Person**-klasse, med felt og innkapslingsmetodar for fødselsdato og kjønn. Kjønn skal berre kunne setjast ved oppretting av **Person**-objektet, medan det for fødselsdatoen ikkje er noka slik avgrensing.

c) Skriv kode for å lagre og innkapsle personnummer i **Person**-objekta. Du kan anta at fødselsdato og kjønn allereie er satt. Eit personnummer består grovt sett av fødselsdatoen, eit (vilkårlig) løpenummer og to kontrollsiffer. Kontrollsifra gjer det enklare å sjekke om eit personnummer er ekte. Meir spesifikt er reglane for personnummer som følgjer:

- Et personnummer består av 11 siffer, med følgjande struktur: **D**1**D**2**M**1**M**2**Y**1**Y**2**N**1**N**2**N**3**K**1**K**2 (fargen illustrerer siffergruppene).
- Dei seks første sifra, **D**1**D**2**M**1**M**2**Y**1**Y**2, tilsvarer fødselsdatoen sin dag (1-31), månad (1-12) og år (0-99).
- Dei tre neste sifra, **N**1**N**2**N**3, kan antakast å vere vilkårlige, men **N**3 må vere *partal* for kvinner og *oddetal* for menn.
- Dei to siste sifra, **K**1**K**2, er kontrollsiffer, som kvar for seg blir rekna ut frå dei førre sifra. Formelen for dei begge er **11 – (VS % 11)**[3], der **VS** (veid sum) for **K**1 er **D**1***F**1 + **D**2***F**2 + … + **N**2***F**8 + **N**3***F**9 og **VS** for **K**2 er **D**1***G**1 + **D**2***G**2 + … + **N**3***G**9 + **K**1***G**10. **F**'ane og **G**'ane er oppgitt i tabellen under. Dersom formelen gir eit eitt-sifra resultat for både **K**1 *og* **K**2 så er personnummeret gyldig, men gir formelen eit to-sifra resultat for **K**1 og/eller **K**2, så er personnummeret ugyldig.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
F	3	7	6	1	8	9	4	5	2	
G	5	4	3	2	7	6	5	4	3	2

Introduser og bruk gjerne hjelpemetodar for å gjere koden ryddigare.

d) Beskriv korleis du kan bruke ein såkalt *checked exception* for å hindre at fødselsdatoen blir endra etter at personnummeret er satt! Kva slags konsekvensar vil dette ha for kode som kallar endringsmetoden for fødselsdatoen?

Del 2 – Klassar (40%)

Du skal implementere klassar for å representere informasjon ifm. gjennomføring av seriespill i idrettar som fotball, handball, volleyball osb. Klassen **MatchResult** skal representere informasjon om ein kamp og resultatet, medan klassen **LeagueTable** skal representere eit sett kamper og rekne ut serietabellen basert på kampresultata. I del 2 skal vi avgrense oss til tabellar for *ferdigspelte fotballkampar*. Eit eksempel på to kampresultat og ein tabell er vist nedanfor:

```
Rosenborg - Lillestrøm: 4 - 4
Lillestrøm - Rosenborg: 1 - 5
```

[3] % er modulo-operatoren

```
Rosenborg   4
Lillestrøm  1
```

Som vi ser har Rosenborg fått totalt 4 poeng, 3 poeng for siger og 1 poeng for uavgjort, medan Lillestrøm har fått 1 poeng for den eine uavgjortkampen. Tabellen utelet informasjon om mål for og imot og heime- og bortekampar.

Tenk på heilskapen i løysinga før du går i gong med kvar del. Kanskje kan det vere lurt å byrje med c) før du skriv koden for a) og b).

Det kan vere lurt å ha i bakhovudet at i del 3 skal koden generaliserast til å handtere idrettar med andre reglar for å rekne ut tabellar og i del 4 skal koden handtere at kamper kan leggjast inn i tabellen før dei er ferdigspelt. Men la ikkje dette gjere koden i del 2 unaudsynleg komplisert.

a) Implementer **MatchResult**:
- Representasjon og innkapsling av informasjon om namna på heime- og bortelaga. Namna skal ikkje kunne endrast etter at eit **MatchResult**-objekt er laga.
- Representasjon og innkapsling av informasjon om kampresultatet, dvs. talet på heime- og bortemål.
- Metodane **isParticipant(String participant)** for å spørje om (laget) **participant** spelar i denne kampen, **isDraw()** for å spørje om resultatet blei uavgjort og **isWinner(String participant)** for å spørje om (laget) **participant** vann denne kampen.

b) Implementer **LeagueTable**:
- Representasjon av (namna til) alle laga. Namna skal ikkje kunne endrast etter at **LeagueTable**-objektet er laga.
- Representasjon av informasjon om kampresultat, dvs. **MatchResult**-objekt. Det skal berre vere lov å leggje inn nye kampar, ikkje fjerne dei. Det skal ikkje vere lov å leggje inn kampar for lag som ikkje er med i tabellen.
- Metoden **getParticipantPoints(MatchResult matchResult, String participant)** som returnerer talet på poeng som (laget) **participant** fekk for resultatet **matchResult**. Ta høgde for at **participant** faktisk ikkje er med i **matchResult**.

Tabellpoeng blir summert basert på kampresultata, og tabellen sortert etter talet på poeng, med laget med flest poeng øvst. Når eit kampresultat blir lagt inn, skal tabellen oppdaterast.

Merk at du sannsynlegvis vil trengje ein klasse for å representere kvar rad i tabellen, t.d. kalla **LeagueTableRow**, som lagrar data om eit lag og summen av poenga laget har fått for siger og uavgjort i sine kampar. Denne klassen kan implementere relevante grensesnitt ifm. sortering.

c) Teikn eit objekt/instansdiagram (figur som viser instansane i ein objektstruktur og deira innhald) som tilsvarer tabellen over, for strukturen av **MatchResult**-, **LeagueTable**- og evt. **LeagueTableRow**-objekt.

d) I ein ordentleg fotballtabell så spelar også totalt talet på mål for og imot og resultat i innbyrdes kampar inn på tabellen. Beskriv kort med tekst og kode korleis du vil støtte dette.

Del 3 – Generalisering og arv (20%)

Ulike sportar har ulike reglar for poenggjeving i både kampar og serietabellar og sortering av tabellane. I tennis er kampen slutt når éin deltakar har vunne 2 eller 3 sett for hhv. kvinner og menn, så sluttresultatet 1-0 finst ikkje i tennis. Fotball gir 3 tabellpoeng for siger, medan handball gir 2. I somme idrettar tel innbyrdes oppgjer før målforskjell ved sortering, medan det for andre er omvendt.

a) Forklar med tekst og kode korleis du kan lage sport-spesifikke resultat-klassar, t.d. **TennisResult**, som arvar frå **MatchResult** og implementerer logikk for å handheve sporten sine reglar for poenggjeving, m.a. for å unngå registrering av ugyldige resultat. Forklar også naudsynte endringar av **MatchResult**.

b) Forklar med tekst og kode korleis du kan lage sport-spesifikke tabell-klassar, t.d. **FootballLeagueTable** og **TennisTournamentTable**, som arvar frå **LeagueTable** og implementerer logikk iht. sporten sine regler for å rekne ut tabellpoeng. Beskriv også naudsynte endringar av **LeagueTable**.

Del 4 – Diverse (15%)

a) Kva slags generell teknikk kan ein bruke for å sikre at ein serietabell automatisk blir oppdatert dersom eit **MatchResult**-objekt blir endra etter at det er lagt inn i tabellen? Forklar med tekst og kode korleis du vil implementere støtte for dette. Fokusér på å forklare korleis den generelle teknikken blir brukt / tilpassa, ikkje på å få logikken for poengutrekning heilt riktig (kan vere litt intrikat i dette tilfellet).

b) Forklar korleis du vil teste at:
- metoden **isWinner(…)** i 2 a) pkt. 3 er riktig implementert
- innbyrdes samanlikning av tabellrader i 2 b) er korrekt
- løysinga i 1 d) er korrekt

Du treng ikkje skrive kode, men forklaringa må vere detaljert nok til å illustrere testemetoden.