**Computerspil 1**

I løbet af de kommende fem uger skal I programmere et computerspil, hvor spillerne rejser rundt mellem byer i forskellige lande og indsamler point. Der er en delaflevering hver uge, hvor I tilføjer nye ting til spillet – ofte baseret på ting, der er gennemgået i de nærmest foregående forelæsninger (og kapitler i BlueJ-bogen).

I denne første delaflevering skal I bl.a. modellere byerne og vejene imellem dem, samt de lande byerne ligger i. Når I er færdige, vil I have den første version af et computerspil, hvor den grafiske brugergrænseflade viser et netværk af udvalgte byer i Norden og vejene imellem dem.

Når I laver de forskellige klasser, skal I samtidig dokumentere dem, således at dokumentationen er helt i top og følger:

* retningslinjerne fra BlueJ bogen (herunder specielt afsnit 6.11, afsnit 9.7 og Appendix I),
* de eksempler fra ***Road*** klassen, der blev præsenteret i en forelæsning.

Hver klasse og hver metode/konstruktør skal have en passende kommentar (der begynder med /\*\* og slutter med \*/). Tænk på, at første sætning indsættes i dokumentationens *Method/Constructor Summary*, mens hele kommentaren indsættes i dokumentationens *Method/Constructor Detail*. I skal også indsætte *@author* og *@version* tags i jeres klasser, samt *@param* og *@return* tags i jeres metoder/konstruktører. For at undgå for mange sprogskift, anbefales det, at alle jeres kommentarer skrives på Engelsk (eller Amerikansk). Skift jævnligt til *Documentation view* for at kontrollere, at resultatet er fornuftigt og giver relevant og letlæselig information til brugere, der ikke kender implementationen af jeres klasser.

Der er forholdsvis meget at lave i Computerspil 1, men langt de fleste af tingene burde på nuværende tidspunkt være lette at implementere.

Download BlueJ projektet ***Nordic Traveller*** ([zip](https://users-cs.au.dk/dintprog/e20/opgaver/CG.zip)). Projektet indeholder klassen ***TestServer***, der bl.a. har nedenstående to klassemetoder:

* ***test*** metoden uploader jeres projekt til vores testserver. Ved afslutningen af hver opgave, kan I afprøve, om det, som I har lavet i opgaven, fungerer korrekt ved at kalde ***test*** med parameteren ”CG1-X”, hvor X er nummeret på den pågældende opgave. Testserveren afprøver konstruktøren og metoderne i opgave X ved hjælp af nogle regression tests, som vi har skrevet. Men det er ikke altid muligt at gentage gamle tests. Når man f.eks. i opgave 5 ændrer City klassens konstruktør, så den tager en ekstra parameter, kan man ikke længere køre den gamle test fra opgave 1 uden at denne fejler.
* ***download*** metoden skal I først bruge i opgave 7, hvor den vil downloade nogle nye klasser til jeres projekt. Hvis I kalder den tidligere, vil I ikke kunne oversætte jeres projekt, idet de nye klasser bruger nogle af de ting, som I endnu ikke har skrevet.

Der vil altid være små fejl i så kompliceret en opgave som computerspillet – det gælder både opgaveformulering, udleverede klasser og testserveren. Brug webboardet til at rapportere eventuelle fejl/problemer, som I måtte støde på. Så retter vi dem hurtigst muligt og uploader nye versioner, når det er nødvendigt. Af samme grund er det vigtigt, at I alle følger med på webboardet, således at I bliver opmærksomme på nye versioner eller hints til at omgå problemer.

Når I får en fejlrapport, bør I rette **alle** de fejl, der rapporteres og teste, at rettelserne er korrekte, før I atter forsøger at køre testserveren Hvis I blot retter en enkelt fejl ad gangen (uden selv at teste om rettelsen fungerer) kommer I let til at bruge alt for megen tid på at vente på, at testserveren genererer rapporter til jer (specielt hvis der er kø).

**Opgave 1**

Implementér ***City*** klassen, der repræsenterer en by, som man kan besøge. En by har et navn, der er repræsenteret ved en tekststreng, samt en værdi, der er repræsenteret ved et heltal.

Opret ***City*** klassen med tre feltvariabler ***name***, ***value*** og ***initialValue***.

Klassen skal have accessor metoderfor alle tre feltvariabler og en konstruktør på formen:

* ***City(String name, int value)***

hvor den første parameter bruges til at initialisere feltvariablen ***name***, mens den anden parameter bruges til at initialisere de to sidste feltvariabler.

Herudover skal klassen have nedenstående metoder:

* ***void changeValue(int amount)*** // Adderer ***amount*** til ***value***
* ***void reset()*** // Resetter ***value*** til ***initialValue***
* ***String toString()***

Den sidste metode returnerer en tekststreng på formen “London (60)”, hvor London er byens navn og 60 dens værdi.

Klassen skal implementere interfacet ***Comparable<City>***, dvs. at I skal lave en ***compareTo*** metode, der definerer byernes sorteringsrækkefølge (alfabetisk efter navn).

Til sidst skal I implementere ***equals*** og ***hashCode*** metoder for ***City*** klassen, idet I anvender de principper, der blev gennemgået i en forelæsning. To ***City*** objekter er ens, hvis og kun hvis de har samme navn.

Husk at bruge *@Override*, når I overskriver eksisterende metoder, såsom ***toString***, ***equals*** og ***hashCode***.

**Opgave 2**

Implementér ***Road*** klassen, der repræsenterer en vej mellem to byer. En vej har referencer til de to byer (***City*** objekter) den forbinder. Den har også en længde, der angiver, hvor langt der er mellem de to byer.

Opret klassen ***Road*** med tre feltvariabler ***from***, ***to*** og ***length***.

Klassen skal have accessor metoderfor alle tre feltvariabler og en konstruktør på formen:

* ***Road(City from, City to, int length)***

Herudover skal klassen have nedenstående metode:

* ***String toString()***

der returnerer en tekststreng på formen “London (60) -> Liverpool (40) : 8”, hvor 8 er afstanden mellem de to byer.

Klassen skal implementere interfacet ***Comparable<Road>***, dvs. at I skal lave en ***compareTo*** metode, der definerer vejenes sorteringsrækkefølge. Først sorteres efter de byer, hvor vejene starter. Hvis to veje starter i samme by sorteres efter de byer, hvor vejene slutter og dernæst efter længden (lavest til højest).

Til sidst skal I implementere ***equals*** og ***hashCode*** metoder for ***Road*** klassen. To ***Road*** objekter er ens, hvis og kun hvis de tre feltvariabler har ens værdier.

**Opgave 3**

Implementér ***Position*** klassen, der repræsenterer en spillers position. Klassen har fire feltvariabler:

* ***from*** // Den by spilleren kommer fra
* ***to*** // Den by spilleren er på vej til
* ***distance*** // Den afstand der er tilbage
* ***total*** // Den totale afstand mellem de to byer

Klassen skal have accessor metoderfor alle fire feltvariabler og en konstruktør på formen:

* ***public Position(City from, City to, int distance)***

hvor parameteren ***distance*** også bruges til at initialisere feltvariablen ***total***.

Herudover skal klassen have nedenstående metoder:

* ***boolean move()*** // Tæller ***distance*** 1 ned hvis ***distance > 0***// Returværdien angiver om ***distance*** ændres
* ***void turnAround()*** // Vender spillerens bevægelsesretning uden at spilleren flyttes
* ***boolean hasArrived()*** // Returnerer ***(distance == 0)***
* ***String toString()***

Den sidste metode returnerer en tekststreng på formen “London (60) -> Liverpool (40) : 3/8”, hvor 3 er den afstand, der mangler, mens 8 er den totale afstanden mellem de to byer.

Til sidst skal I implementere ***equals*** og ***hashCode*** metoder for ***Position*** klassen. To ***Position*** objekter er ens, hvis og kun hvis de fire feltvariabler har ens værdier.

**Opgave 4**

Implementér ***Country*** klassen, der repræsenterer et land med et netværk af byer og veje. Klassen har to feltvariabler:

* ***name*** // Landets navn
* ***network*** // Landets veje repræsenteret som en ***Map<City, Set<Road>>***

Netværket (network) er en afbildning (map), hvor hver by tilknyttes en mængde indeholdende alle de veje, der starter i den pågældende by.

Klassen skal have en accessor metode for den første feltvariabel og en konstruktør på formen:

* ***Country(String name)***

hvor ***network*** initialiseres til at være en tom afbildning (map) ved hjælp af følgende udtryk  
***new*** ***TreeMap<>()***. Bemærk, at typen for feltvariablen ***network*** og signaturerne for nedenstående metoder bruger typerne ***Map*** og ***Set***, der er interfaces*.* Vi bruger kun konkrete klasser, når vi instantierer et objekt via ***new*** operatoren.

Landet skal have nedenstående metoder:

* ***Set<City> getCities()*** // Returnerer landets byer
* ***City getCity(String name)*** // Returnerer byen med det angivne navn
* ***Set<Road> getRoads(City c)*** // Returnerer alle veje, der starter i byen c
* ***void reset()*** // Resetter alle landets byer
* ***String toString()*** // Returnerer landets navn

I den første metode kan man bruge en af ***Map*** interfacets metoder til at finde alle de byer, der forekommer som nøgler i netværket.

Den anden metode returnere ***null***, hvis der ikke findes en by i netværket med det angivne navn. Den kan implementeres ved hjælp af en af algoritmeskabelonerne, hvor man dog nu gennemløber en mængde i stedet for en liste.

Den tredje metode returnerer en tom mængde ***new TreeSet<>()***, hvis den angivne by ikke findes som nøgle i ***network***. Dette kan afgøres ved hjælp af en af ***Map*** interfaces metoder. Ellers returneres den mængde af veje, som er tilknyttet nøglen (byen).

Til sidst skal I implementere ***equals*** og ***hashCode*** metoder for ***Country*** klassen. To ***Country*** objekter er ens, hvis og kun hvis de har samme navn.

**Opgave 5**

Når en spiller ankommer til en by, modtager hun en bonus, der er en del af byens værdi (som samtidigt formindskes tilsvarende). Bonussens størrelse udregnes i ***Country*** klassen, idet vi senere vil introducere lande, hvor udregningen foretages på en anden måde.

Hver by skal derfor have en feltvariabel ***country***, der fortæller, hvilket land byen ligger i. Introducér denne feltvariabel (sammen med en accessor metode ***getCountry***) og tilføj en parameter til konstruktøren for ***City*** klassen, således at man kan angive, hvilket land en by ligger i:

* ***City(String name, int value, Country country)***

Modificér ***equals*** og ***hashCode*** metoderne i ***City*** klassen, således at de anvender den nye feltvariabel. To ***City*** objekter er ens, hvis og kun hvis de har samme navn og ligger i samme land.

I ***Country*** klassen implementeres nedenstående metode:

* ***int bonus(int value)*** // Beregner størrelsen af den bonus, som spilleren modtager

Hvis parameterværdien er positiv, returneres et tilfældigt heltal i intervallet ***[0, value]***. Ellers returneres ***0***.

Testserveren forudsætter, at I ikke laver unødvendige kald af metoderne i ***Random*** objektet. Hvis f.eks. ***bonus*** metoden laver to kald af ***nextInt*** for a beregne bonussen, vil de pseudo-tilfældige tal komme ud af sync, og testen vil fejle.

I ***City*** klassen implementeres en metode:

* ***int arrive()***// Fratrækker den udbetalte bonus fra byens værdi

der kaldes af ***Game*** klassen (der introduceres senere), hver gang en spiller ankommer til en by. Metoden kalder ***bonus*** metoden i det land, som byen tilhører (med en parameterværdi, der er lig med byens værdi). Den værdi, som ***bonus*** metoden returnerer, fratrækkes fra byens værdi og returneres som ***arrive*** metodens returværdi.

**Opgave 6**

Implementér nedenstående metoder i ***Country*** klassen:

* ***void addCity(City c)*** // Tilføjer en by til netværket
* ***void addRoads(City a, City b, int length)*** // Tilføjer veje mellem a og b

Den første metode indsætter den angivne by som nøgle i den afbildning som feltvariablen ***network*** peger på. Den tilknyttede værdi er en tom mængde ***new TreeSet<>()***.

Den anden metode opretter veje mellem de angivne byer, ved at indsætte dem i de mængder, der forekommer som værdier i den afbildning, som feltvariablen ***network*** peger på. Der oprettes kun veje der starter i det pågældende land. Hvis begge byer ligger i landet oprettes der veje i begge retninger. Hvis én af byerne ligger i landet oprettes kun den vej, der starter i landet. Hvis ingen af byerne ligger i landet oprettes der ingen veje. Der oprettes kun veje, der har positiv længde og forbinder to forskellige byer (afgjort ved hjælp af ***equals***).

Implementér nedenstående metoder i ***Country*** klassen:

* ***Position position(City city)*** // Returnerer byens position
* ***Position readyToTravel(City from, City to)*** // Gør klar til at påbegynde rejse

Den første metode returnerer byens position, dvs. et ***Position*** objekt på formen  
***new Position(city, city, 0)*** – svarende til at spilleren står i byen ***city*** uden at have valgt en vej mod en anden by. Dette er også tilfældet, hvis byen slet ikke ligger i landet.

Den anden metode returnerer et ***Position*** objekt, der repræsenterer, at spilleren er klar til at påbegynde rejsen fra ***from*** til ***to*** (men stadig står i ***from***). Hvis ***from*** og ***to*** er samme by (afgjort ved hjælp af ***equals***), returneres byens position. Hvis der ikke er en direkte vej fra ***from*** til ***to*** returneres ***from***’s position. Dette er også tilfældet, hvis ***from*** slet ikke ligger i landet.

**Opgave 7**

Kald klassemetoden ***download*** i ***TestServer*** klassen med parameteren ”CG1”. Dette vil downloade nedestående klasser og tilføje dem til jeres projekt. Mange af de nye klasser har oversætterfejl. Det skal I ikke blive forskrækket over. Det retter sig, når I laver de to rettelser, der er beskrevet nederst i opgave 7.

* ***Game*** styrer interaktionen mellem spillere, byer og lande.
* ***GUI*** indeholder den grafiske brugergrænseflade (hvor man kan se landene, byerne, vejene imellem dem og spillernes positioner).
* ***Settings*** gemmer de forskellige options, som brugeren kan sætte.
* ***Player*** repræsenterer de forskellige spillere.
* ***GUIPlayer*** repræsenterer den spiller, der styres af brugeren via musen og den grafiske brugergrænseflade.
* ***RandomPlayer*** repræsenterer en computerstyret spiller, der altid vælger en tilfældig vej.
* ***GreedyPlayer*** repræsenterer en computerstyret spiller, der altid vælger den vej, hvor destinationsbyen har højest værdi.
* ***SmartPlayer*** repræsenterer en computerstyret spiller, der vælger den vej, der på sigt giver flest penge.
* ***CGTest*** indeholder en Test Fixture til konstruktion af et simpelt netværk. Denne skal bruges i Computerspil 2.

Ovenstående klasser må I kun modificere, når I får eksplicit besked herom.

I skal nu modificere ***Country*** klassen. Tilføj en ny feltvariabel ***game*** af type ***Game*** og implementér nedenstående accessor og mutator metoder:

* ***Game getGame()***
* ***void setGame(Game game)***

Modificér dernæst ***bonus*** metoden, så de tilfældige værdier beregnes via det ***Random*** objekt, der er skabt af det ***Game*** objekt, som ***game*** refererer til. Dette kan fra ***Country*** klassen tilgås via metodekaldet:

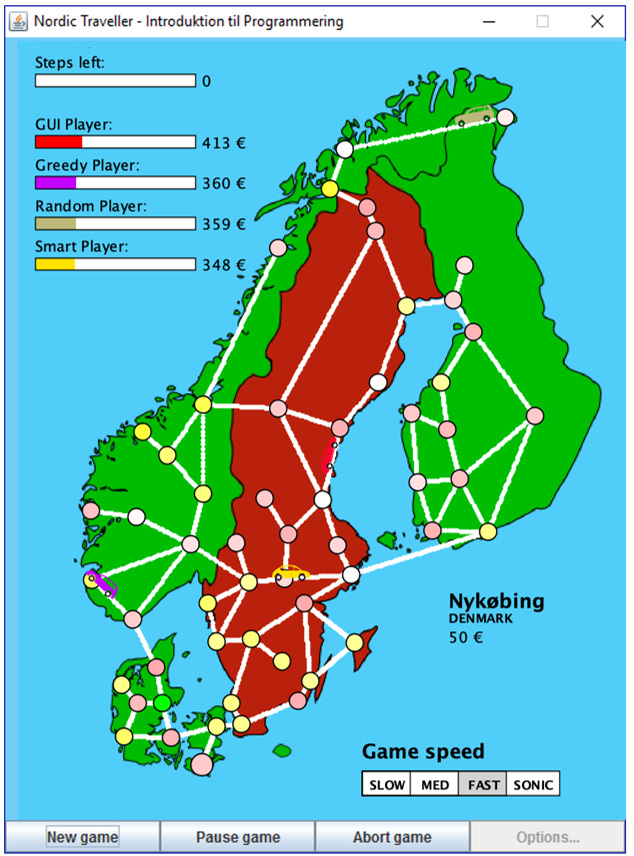
* ***game.getRandom()***

Hvis I har kodet jeres fire klasser korrekt, kan det samlede projekt nu kompileres.

**Opgave 8**

Kald klassemetoden ***createGameBoard*** i ***GUI*** klassen. Metoden opretter de nødvendige lande, byer og veje, og I vil nu se nedenstående vindue med et kort over Norden (nogle af vejene er færgeruter).

* Man starter et nyt spil ved at trykke på ***New game*** knappen.
* Spillet kan på ethvert tidspunkt pauseres ved at trykke på ***Pause game*** knappen. Teksten på knappen skifter så til ***Resume game***, og et nyt tryk på knappen genoptager spillet. Spillet kan afbrydes ved at trykke på ***Abort game*** knappen.
* Spillets hastighed (dvs. hvor hurtigt spillerne bevæger sig) kan styres ved at klikke på ***Game speed*** knapperne i nederste højre hjørne.

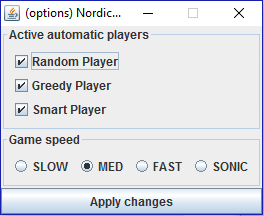


* Cirklerne repræsenterer byer. Farven i cirklen angiver byens værdi med følgende indkodning:



Når musen placeres oven på en by, kan man (over ***Game speed*** knapperne) se navnet på byen, samt dens præcise værdi.

* Stregerne mellem byerne angiver veje (eller færgeruter). Længden er en grov indikation af, hvor mange skridt, det vil tage at følge vejen (lang streg ≈ lang vej ≈ mange skridt).
* Bilerne angiver spillernes placering. Ude til venstre kan man se nogle aflange rektangler, der angiver, hvor mange penge, hver spiller har indsamlet. Farverne på rektanglerne matcher farverne på spillernes biler.
* Det mørkeblå rektangel i øverste venstre hjørne viser, hvor mange skridt, der er tilbage i spillet. Når man når ned på 0, kan ingen spillere længere bevæge sig, og vinderen er den med flest penge.
* Den røde spiller (GUI Player) styres af den person, der spiller, mens de øvrige spillere styres af programmet. Man styrer den røde spiller ved at klikke på byer. Hvis den røde spiller befinder sig mellem to byer, vil et klik på en af disse byer, få spilleren til at bevæge sig mod den angivne by. Hvis den røde spiller befinder sig i en by, vil et klik på en naboby, få spilleren til at bevæge sig mod den angivne by (forudsat, at der findes en direkte vej mellem de to byer). Alternativt kan man anvende piletasterne, idet et tryk på en af disse vælger den by, der bedst matcher pilens retning (samtidigt tryk på to piletaster giver den retning, der ligger midt mellem de to pile).
* ***Options*** knappen i nederste højre hjørne åbner nedenstående dialogboks, hvori man kan foretage forskellige valg. De aktuelle valg huskes fra spil til spil (også selv om programmet lukkes ned i mellemtiden). Man kan blandt andet vælge, hvilke automatiske spillere man vil kæmpe imod. Random Player vælger en tilfældig vej, Greedy Player vælger den vej, hvor destinationsbyen har højest værdi, mens Smart Player vælger den vej, der på sigt giver flest penge. Man kan ikke trykke på ***Options*** knappen, mens et spil er i gang.



Spil nogle spil og afprøv de forskellige faciliteter i spillet. Prøv at finde nogle gode strategier for at samle flest mulige penge sammen.

**Opgave 9**

I skal nu afprøve om de ting, som I har lavet Computerspil 1, fungerer korrekt ved at kalde klassemetoden ***test*** i ***TestServer*** klassen med parameteren ”CG1”.

Testserveren afprøver hver enkelt af de klasser, som I selv har skrevet, sammen med vores løsning – således at I kan se, hvilke klasser, der er fejl i. Afprøvningen foretages ved at udføre en række regression tests for konstruktørerne og metoderne i klassen.

Testserveren forudsætter, at I ikke laver unødvendige kald af metoderne i ***Random*** objektet. Hvis f.eks. ***bonus*** metoden laver to kald af ***nextInt*** for at beregne bonussen, vil de pseudo-tilfældige tal komme ud af sync, og testen vil fejle.

Hvis testserveren finder fejl, skal l gennemgå jeres kode og forsøge at rette dem.

I de næste afleveringer skal I:

* Lave automatiske regression tests for alle konstruktører og metoder (Computerspil 2).
* Bruge nedarvning til at strukturere jeres kode, således at der er flere forskellige slags lande og flere forskellige slags byer (Computerspil 3).
* Udvide den grafiske brugergrænseflade med nogle ekstra knapper, labels og tekstfelter samt en menubar (Computerspil 4).
* Optage spil ved hjælp af en ***Log klasse***, hvis objekter kan gemmes i filsystemet, for sidenhen at blive genindlæst og afspillet (Computerspil 5).

**Computerspil 2**

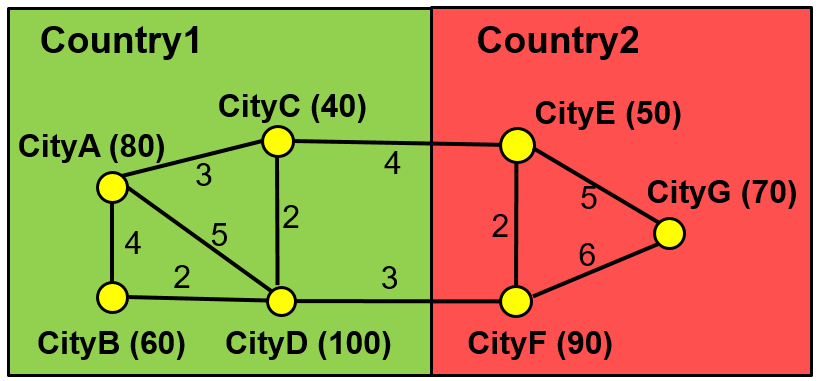
I denne anden delaflevering skal I bruge de ting, som I har lært om automatiske regression tests, på de klasser, som I har implementeret i den første delaflevering. Herudover skal I rette de fejl og mangler, som instruktoren har påpeget i jeres første delaflevering.

I BlueJ bogens kapitel 9 og Appendix G er det beskrevet, hvordan man bør afteste sine klasser, herunder hvordan man (ved hjælp af JUnit testing frameworket) laver automatiske regression tests, der let kan gentages, hver gang man har modificeret sin kode. Bemærk at det er langt hurtigere (og nemmere) selv at skrive Java koden for de enkelte tests (i stedet for at optage testene interaktivt).

I behøver ikke at lave testmetoder for trivielle accessor og mutator metoder, der blot returnerer/ ændrer en enkelt feltvariabel uden at gøre andet, og I behøver heller ikke at lave testmetoder for ***compareTo***, ***equals*** og ***hashCode*** metoderne. Det bør man normalt gøre, men da det kan være lidt besværligt og tidskrævende, har vi besluttet at lade jer slippe for det.

Dokumentationen for jeres testmetoder kan holdes på et minimum. Testmetoder har ingen parametre og returnerer intet, så @param og @return tags giver ikke mening. Herudover vil testmetodens navn normalt fortælle, hvilken metode den tester. I kan derfor nøjes med at lave @author og @version tags for testklasserne samt indsætte // kommentarer udvalgte steder i jeres kode.

I jeres aftestning kan I bruge den Test Fixture, der er indeholdt i klassen ***CGTest***. Den indeholder et ***Game*** objekt, to ***Country*** objekter, syv ***City*** objekter og et antal ***Road*** objekter, der realiserer nedenstående netværk, hvor tallene i parenteserne angiverne byernes initiale værdi, mens tallene på vejene angiver deres længde:



Test Fixturen kan kopieres til jeres egne testklasser via følgende skridt:

* Højreklik på ***CGTest*** og vælg *"Test Fixture to Object Bench"*.
* Højreklik på den ønskede testklasse og vælg *"Object Bench to Test Fixture"*.

Ved afslutningen af hver opgave, kan I afprøve, om det, som I har lavet, fungerer korrekt. Dette gøres ved at kalde klassemetoden ***test*** i ***TestServer*** klassen med parameteren ”CG2-X”, hvor X er nummeret på den pågældende opgave. Det afprøves, at jeres regression tests:

* *ikke* finder fejl i et korrekt projekt,
* finder *alle* ”oplagte” fejl i nogle forkerte projekter.

**Opgave 1**

I en forelæsningen blev der vist et eksempel på en testklasse ***RoadTest***, der tester konstruktøren og de ikke trivielle metoder i ***Road*** klassen. Kopiér indholdet af testklassen til jeres projekt, og udfyld de manglende dele.

Jeres regression tests skal kontrollere, at konstruktøren initialiserer alle feltvariabler korrekt. Herudover skal de kontrollere at ***toString*** metoden returnerer en tekststreng på den specificerede form. Som nævnt tidligere, behøver I ikke at lave regression tests for ***compareTo***, ***equals*** og ***hashCode*** metoderne.

Udfør testene ved at højreklikke på ***RoadTest*** klassen og vælge *TestAll*, eller ved at trykke på knappen *Run Tests* i venstre side af vinduet med BlueJ’s klassediagram (hvis knappen ikke er synlig trykkes først på den lille trekantede knap).

Hvis man bruger *Run Tests* knappen, skal man huske at tjekke, om nogle af testklasserne har skrå grå striber i den nederste del, hvilket viser, at de skal kompileres før man kan udføre dem (ellers ignoreres de pågældende testklasser).

Hvis en eller flere tests fejler, skal I forsøge at finde ud af, hvad der er galt, og rette dette. Det kan både være jeres implementation af ***Road*** klassen og jeres implementation af testene i ***RoadTest*** klassen, der er forkert. Men hvis testdriveren tidligere har godkendt jeres ***Road*** klasse er det mest sandsynligt, at fejlen ligger i ***RoadTest***.

**Opgave 2**

I en forelæsning blev der vist et eksempel på en testklasse ***PositionTest***, der tester konstruktøren og nogle af de ikke trivielle metoder i ***Position*** klassen. Kopiér indholdet af testklassen til jeres projekt, og udfyld de manglende dele.

Jeres regression tests skal kontrollere, at konstruktøren initialiserer alle feltvariabler korrekt. Herudover skal de kontrollere at ***move***, ***turnAround***, ***hasArrived*** og ***toString*** metoderne fungerer korrekt, dvs. foretager de forventede opdateringer af feltvariablerne og har korrekte returværdier. Husk at afprøve metoderne i de forskellige situationer, der kan opstå, inklusiv diverse grænsetilfælde, f.eks. at feltvariablen ***distance*** er lig 0.

Som nævnt tidligere, behøver I ikke at lave regression tests for ***equals*** og ***hashCode*** metoderne.

Udfør testene ved at højreklikke på ***PositionTest*** klassen og vælge *TestAll*, eller ved at trykke på knappen *Run Tests* i venstre side af vinduet med BlueJ’s klassediagram. Hvis en eller flere tests fejler, skal I forsøge at finde ud af, hvad der er galt, og rette dette. Hvis testdriveren tidligere har godkendt jeres ***Position*** klasse, er det mest sandsynligt, at fejlen ligger i ***PositionTest***.

**Opgave 3**

Lav en testklasse ***CityTest*** med velvalgte regression tests for ***City*** klassen. Det er vigtigt, at I både tester, hvad der normalt sker, og hvad der sker i specialtilfælde og tæt ved grænseværdier. Husk også, at der kan være både positive og negative tests.

Når I skal teste ***arrive*** metoden i ***City*** klassen kan det, som vist i en forelæsning, være nyttigt at kunne kontrollere den seed værdi som ***Random*** objektet bruger. På den måde kan I finde ud af, hvad ***bonus*** metoden returnerer, når den kaldes inde fra ***arrive*** metoden.

Jeres regression tests skal kontrollere, at konstruktøren initialiserer alle feltvariabler korrekt. Herudover skal de kontrollere at ***changeValue***, ***reset***, ***toString*** og ***arrive*** metoderne fungerer korrekt, dvs. foretager de forventede opdateringer af feltvariablerne og har korrekte returværdier. Husk at afprøve metoderne i de forskellige situationer, der kan opstå, inklusiv diverse grænsetilfælde, f.eks. hvad der sker med ***arrive*** metoden, når feltvariablen ***value*** er 0.

Som nævnt tidligere, behøver I ikke at lave regression tests for ***compareTo***, ***equals*** og ***hashCode*** metoderne.

Udfør testene ved at højreklikke på ***CityTest*** klassen og vælge *TestAll*, eller ved at trykke på knappen *Run Tests* i venstre side af vinduet med BlueJ’s klassediagram. Hvis en eller flere tests fejler, skal I forsøge at finde ud af, hvad der er galt, og rette dette. Hvis testdriveren har godkendt jeres ***City*** klasse, er det mest sandsynligt, at fejlen ligger i ***CityTest***.

**Opgave 4**

Lav en testklasse ***CountryTest*** med velvalgte regression tests for ***Country*** klassen. Det er vigtigt, at I både tester, hvad der normalt sker, og hvad der sker i specialtilfælde og tæt ved grænseværdier. Husk også, at der kan være både positive og negative tests.

Når I tester ***bonus*** metoden, skal I kalde metoden mange gange og derefter tjekke at middelværdien er som forventet, samt at alle de forventede værdier returneres (og ingen andre). Det svarer helt til det, som I gjorde, da I testede ***roll*** metoden i Raflebæger 4. Som vist i en forelæsning, bør testet gentages for mange forskellige seed værdier. I bør også teste grænsetilfældene, hvor ***bonus*** kaldes med parameterværdierne 0 og 1.

Jeres regression tests skal kontrollere, at konstruktøren initialiserer alle feltvariabler korrekt. Herudover skal de kontrollere at ***getCities***, ***getCity***, ***getRoads***, ***reset***, ***toString***, ***bonus, addCity, addRoads, position og readyToTravel*** metoderne fungerer korrekt, dvs. foretager de forventede opdateringer af feltvariablerne og har korrekte returværdier. Husk at afprøve metoderne i de forskellige situationer, der kan opstå, inklusiv diverse grænsetilfælde, herunder specielt dem, der er nævnt i den forklarende tekst under de forskellige metoder.

I testene af konstruktøren og metoder som ***getCities*** og ***addRoads***, er det tilstrækkeligt, at I tjekker antallet af byer/veje. I behøver ikke at tjekke, at det er de rigtige byer/veje, der ligger i network/returneres.

Som nævnt tidligere, behøver I ikke at lave regression tests for ***equals*** og ***hashCode*** metoderne.

Udfør testene ved at højreklikke på ***CountryTest*** klassen og vælge *TestAll*, eller ved at trykke på knappen *Run Tests* i venstre side af vinduet med BlueJ’s klassediagram. Hvis en eller flere tests fejler, skal I forsøge at finde ud af, hvad der er galt, og rette dette. Hvis testdriveren tidligere har godkendt jeres ***Country*** klasse, er det mest sandsynligt, at fejlen ligger i ***CountryTest***.

**Opgave 5**

I skal nu afprøve om de ting, som I har lavet Computerspil 2, fungerer korrekt ved at kalde klassemetoden ***test*** i ***TestServer*** klassen med parameteren ”CG2”. Det afprøves, at jeres regression tests:

* *ikke* finder fejl i et korrekt projekt,
* finder de*fleste* fejl i nogle forkerte projekter.

Hvis testserveren finder fejl, skal l gennemgå jeres kode og forsøge at rette dem.

I de kommende delafleveringer, vil I modificere jeres kode. Når I gør det, skal I opdatere jeres dokumentation og jeres regression tests. Når I tilføjer nye klasser og nye metoder/konstruktører, skal I forsyne disse med dokumentation og regression tests.

**Computerspil 3**

I denne tredje delaflevering skal I bruge nogle af de ting, som I har lært om nedarvning og ”dynamic method lookup” til at strukturere jeres kode, når vi indfører flere forskellige slags lande og flere forskellige slags byer. Herudover skal I rette de fejl og mangler, som instruktoren har påpeget i jeres anden delaflevering.

Husk at holde jeres dokumentation og regression tests opdaterede, så de afspejler de ændringer/ tilføjelser, som I laver i jeres kode. Når I laver nye klasser og nye metoder/konstruktører, skal I lave dokumentation og regression tests for disse. I behøver dog ikke at teste konstruktørerne i de nye klasser. Der tilføjes ikke nye feltvariabler, hvorfor konstruktørerne er ret trivielle.

Ved afslutningen af hver opgave, kan I afprøve, om det, som I har lavet, fungerer korrekt. Dette gøres ved at kalde klassemetoden ***test*** i ***TestServer*** klassen med parameteren ”CG3-X”, hvor X er nummeret på den pågældende opgave. Afprøvningen består af to ting:

* Nogle regression tests for konstruktørerne og metoderne i jeres klasser (som i Computerspil 1).
* Afprøvning af jeres egne regression tests (som i Computerspil 2).

**Opgave 1**

Implementér ***BorderCity*** klassen, der repræsenterer en grænseby. Klassen skal være en subklasse af ***City*** klassen. Husk at konstruktøren skal kalde superklassens konstruktør via det reserverede ord ***super***.

Subklassen skal overskrive superklassens ***arrive*** metode (jvf. opgave 5 i Computerspil 1), således at spilleren, udover at modtage den normale bonus, også skal betale told, såfremt spilleren ankommer fra et andet land (hvilket afgøres ved hjælp af ***equals*** metoden). Tolden udgør en procentdel af spillerens formue inden tillæg af bonus (nedrundet til nærmeste heltal).

For at kunne finde spillerens formue og det land, som spilleren kommer fra, er det nødvendigt, at ***BorderCity*** klassens ***arrive*** metoden har et ***Player*** objekt som parameter. Da metoden kun kan overskrive en metode, der har de samme parametre, skal I tilføje nedenstående metode til ***City*** klassen:

* ***public int arrive (Player p) { return arrive(); }***

Som det ses kalder den nye ***arrive*** metode blot den gamle (og ignorerer sin parameter).

I ***BorderCity*** klassens ***arrive*** metode anvendes den nye parameter til at kalde accessor metoderne ***getMoney*** og ***getFromCountry*** i ***Player*** klassen. Den første metode returnerer spillerens formue, mens den anden returner det land, som spilleren kommer fra.

Den procentdel af formuen, der skal betales i told, kan hentes via nedenstående metodekald, der returnerer et heltal i intervallet [0,100].

* ***getCountry().getGame().getSettings().getTollToBePaid()***

Den betalte told adderes til byens værdi, som dermed forøges. Bonus minus told returneres som returværdien af ***arrive*** metoden.

Til aftestning af ovenstående metoder, kan I bruge den Test Fixture, der stilles til rådighed via ***CGTest*** klassen (beskrevet i Computerspil 2). I skal blot ændre klassens ***setUp*** metode, således, at ***CityC***, ***CityD***, ***CityE*** og ***CityF*** kommer til at tilhøre klassen ***BorderCity*** i stedet for klassen ***City***.

Som vist i en forelæsning, skal l lave to testmetoder for ***arrive*** metoden i ***BorderCity***. Den første tjekker, at der betales korrekt told (dvs. 20%), når spilleren kommer fra et andet land, mens den anden tjekker, at der ikke betales told, når spilleren kommer fra samme land.

**Opgave 2**

Implementér ***CapitalCity*** klassen, der repræsenterer en hovedstad. Klassen skal være en subklasse af ***BorderCity*** klassen (idet alle hovedstæder i Norden har direkte forbindelser til udlandet). Husk at konstruktøren skal kalde superklassens konstruktør via det reserverede ord ***super***.

Subklassen skal overskrive superklassens ***arrive*** metode, således at spilleren udover at modtage bonus (og eventuelt betale told), bruger en del af sin formue (idet hovedstæder har mange dyre fristelser). Forbruget beregnes som et tilfældigt heltal i intervallet ***[0,money]***, hvor ***money*** er spillerens formue (efter tillæg af bonus og fradrag af eventuel told). Husk at anvende det ***Random*** objekt, der er skabt af ***Game*** objektet. Dette kan hentes via ***getGame*** metoden i ***Country*** klassen.

De forbrugte penge adderes til byens værdi, som dermed forøges. Bonus minus told og forbrug returneres som returværdien for ***arrive*** metoden.

Til aftestning af ovenstående metoder, kan I bruge Test Fixturen fra ***CGTest***. I skal blot ændre klassens ***setUp*** metode, således, at ***CityD*** og ***CityE*** bliver hovedstæder.

I skal lave to testmetoder for ***arrive*** metoden i ***CapitalCity***. Den første tjekker, hvad der sker når spilleren kommer fra et andet land, mens den anden tjekker, hvad der sker, når spilleren kommer fra samme land. Testene ligner de to testmetoder fra ***BorderCity***, idet I dog nu også skal beregne, hvor mange penge spilleren bruger, og tage hensyn til dette i jeres assertions.

**Opgave 3**

Implementér ***MafiaCountry*** klassen, der repræsenterer et land, hvor man har risiko for at blive berøvet hver gang man besøger en by. Klassen skal være en subklasse af ***Country*** klassen. Husk at konstruktøren skal kalde superklassens konstruktør via det reserverede ord ***super***.

Subklassen skal overskrive superklassens ***bonus*** metode, således at man risikerer at blive berøvet i stedet for at få en bonus. Dette opnås ved at subklassens ***bonus*** metoden nogle gange kalder superklassens ***bonus*** metode, mens den ellers returnerer et negativt tal, der angiver, hvor meget spilleren mister ved røveriet.

Den procentvise risiko for at blive røvet (angivet som et heltal) kan hentes via metodekaldet:

* ***getGame().getSettings().getRisk()***

mens tabet ved røveriet kan hentes (som et positivt heltal) via metodekaldet:

* ***getGame().getLoss()***

En spiller kan selvfølgelig ikke miste flere penge end hun pt. har. Det sørger ***Game*** klassen for, så det behøver I ikke at bekymre jer om.

Sverige er et Mafialand, hvilket ses ved, at det er farvet rødt. Undskyld til svenskerne for dette påhit!

Til aftestning af ovenstående metoder, kan I bruge Test Fixturen fra ***CGTest***. I skal blot ændre klassens ***setUp*** metode, således at ***country2*** bliver et Mafialand.

Testmetoden for ***bonus*** metoden i ***MafiaCountry*** kan udformes som den tilsvarende testmetode i ***Country*** klassen. I skal tjekke, at spilleren bliver røvet ca. 20 % af gangene, at tabet i gennemsnit udgør 30€, og at det kan antage alle heltalsværdier i intervallet [10,50]. Hvis I ønsker at ændre risikoen for røveri, kan dette gøre via *Options* knappen. Herudover skal I teste, at bonussen beregnes korrekt, når man ikke bliver røvet.

**Opgave 4**

Som beskrevet i forrige opgave, kan bonussen, ved ankomsten til en by i et Mafialand, blive negativ (hvilket indikerer, at spilleren har været udsat for røveri).

Modificér ***arrive*** metoden i ***City*** klassen. Hvis ***bonus*** metoden returnerer en positiv værdi, skal alt fungere som før, men hvis værdien er negativ, returneres denne uden at ændre byens værdi (idet de røvede penge går i Mafiaens lommer og dermed er tabt for spillet).

Undersøg om der også er behov for at ændre ***arrive*** metoderne i ***BorderCity*** og ***CapitalCity*** og implementér eventuelle ændringer, der er nødvendige.

Modificér testmetoden for ***arrive*** i ***CityTest***, således at den tager hensyn til ovenstående og tester metoden på to byer, hvoraf den ene ligger i et Mafialand, mens den anden ikke gør det.

**Opgave 5**

I skal nu afprøve om de ting, som I har lavet Computerspil 3, fungerer korrekt, men først skal nogle af projektets klasser opdateres, hvilket sker ved at I kalder klassemetoden ***download*** i ***TestServer*** klassen med parameteren ”CG3”.

Kald klassemetoden ***test*** i ***TestServer*** klassen med parameteren ”CG3”. Det er kun de nye klasser, der testes, og afprøvningen består af to ting:

* Nogle regression tests for konstruktørerne og metoderne i jeres klasser (som i Computerspil 1).
* Afprøvning af jeres egne regression tests (som i Computerspil 2).

Testserverenforudsætter, at I ikke laver unødvendige kald af metoderne i ***Random*** objektet. Hvis f.eks. ***bonus*** metoden i ***MafiaCountry*** kalder ***getLoss*** for a beregne tabet i en situation, hvor spilleren slet ikke bliver røvet, vil de pseudo-tilfældige tal komme ud af sync, og testen vil fejle.

Hvis testserveren finder fejl, skal l gennemgå jeres kode og forsøge at rette dem.

**Afprøv spillet**

Spil nogle spil og afprøv de forskellige faciliteter i spillet. Prøv at finde nogle gode strategier for at samle flest mulige penge sammen. Hvor dyrt er det at rejse fra et land til et andet, og hvor vigtigt er det at undgå hovedstæder og mafialande?

**Computerspil 4**

I denne fjerde delaflevering skal I bruge nogle af de ting, som I har lært om grafiske brugergrænse­flader til at tilføje nogle ekstra knapper, labels og tekstfelter til ***GUI*** klassen. Herudover skal I tilføje en menubar. Endelig skal I rette de fejl og mangler, som instruktoren har påpeget i jeres tredje delaflevering.

Husk at holde jeres dokumentation opdateret, så den afspejler de ændringer/tilføjelser, som I laver i jeres kode. I behøver ikke at lave regression tests, der afprøver jeres GUI objekter, men de skal selvfølgelig testes manuelt, efterhånden som I laver dem.

Testserveren bruges ikke i denne opgave.

**Opgave 1**

Studér hvordan ***createButtonPanel*** metoden i ***GUI*** klassen skaber de knapper, som vises under kortet.

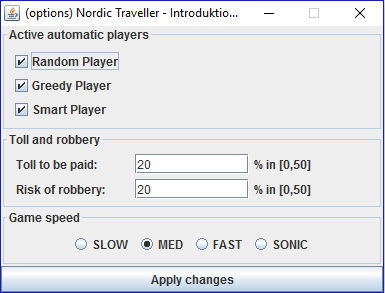
Skab to nye knapper, således at nederste del af vinduet får nedenstående udseende (hvilket bl.a. kræver, at I ændrer parametrene for det ***GridLayout***, der bruges):



I kan med fordel kopiere store dele af koden for de eksisterende knapper. De to nye knapper skal bruges i Computerspil 5. Pt. skal I blot lade lamda’erne i ***addActionListener*** kaldene mappe ***ActionEvent*** ***e*** til hhv. ***testPlayButton()*** og ***testSaveButton()***. Begge metoder ligger i ***GUI*** klassen (ikke i ***Game*** klassen).

**Opgave 2**

Når I trykker på ***Options*** knappen (hvilket kun kan gøres, når spillet er standset, f.eks. efter et tryk på ***Abort*** knappen) åbnes nedenstående dialogboks, som I nu skal udvide lidt:

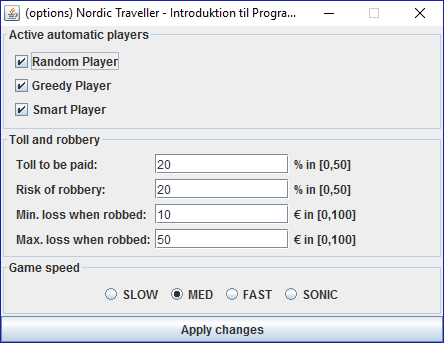


Start med at tilføje to nye feltvariabler ***minLossTextField*** og ***maxLossTextField*** af type ***JTextField*** til GUI klassen (under de eksisterende feltvariabler ***tollSizeTextField***, ***robberyTextField***).

Studér hvordan ***createOptionsDialogBox*** metoden i ***GUI*** klassen, for hvert af de to eksisterende valg i det midterste panel, skaber:

* En ***JLabel*** med beskrivende tekst (Toll to be paid og Risk of robbery).
* Et ***JTextField***, hvori den aktuelle værdi vises og kan ændres (20). Værdien i tekstfeltet gemmes i en feltvariabel i ***JTextField*** objektet.
* En ***JLabel***, der angiver enheden og det tilladte heltalsinterval (% in [0,50]).

Tilføj dernæst to nye valg til det midterste panel, således at dialogboksen får nedenstående udseende:



I kan med fordel kopiere store dele af koden for de eksisterende valg. Som det ses, kan de to nye valg begge antage heltalsværdierne i intervallet [0,100]. Defaultværdierne skal være henholdsvis 10 og 50. Det ***JPanel*** objekt, hvor I skal tilføje de nye valg, benytter et ***GridLayou***. Husk at modificere konstruktøren til det pågældende ***GridLayout*** objekt, så der er plads til de nye komponenter (se evt. Java API).

**Opgave 3**

I skal nu tilføje funktionalitet til de to valg, som I lige har tilføjet. Dette sker i metoden ***applyOptions***.

Aflæs værdierne i jeres to ***JTextField*** objekter, tjek at de begge er heltal i intervallet [0,100], og at minimumsværdien er mindre end eller lig maksimumsværdien. Såfremt der er fejl i tekstfelterne, skal alle fejl vises samlet i én dialogboks. Det vil sige, at I også skal ændre i koden for de andre to tekstfelter. I skal akkumulere alle fejl ved hjælp af ***StringBuilder*** klassen, og vise den samlede tekst ved hjælp af ***showMessageDialog*** metoden. Husk at lave gode fejlbeskeder, så brugeren let kan forstå, hvad der gik galt.

Hvis værdierne er ok, opdateres de pågældende feltvariabler i ***Game*** klassen via metodekaldet:

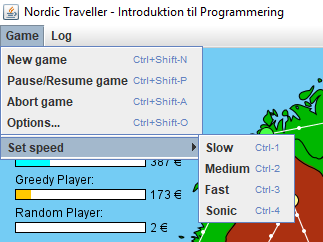
* ***game.getSettings().setMinMaxRobbery(min, max)***

hvor ***game*** er en eksisterende feltvariabel i ***GUI*** klassen.

Når brugeren trykker på ***Options*** knappen, kan hun, i den dialogboks, der kommer op, se de eksisterende valg. Studér hvordan metoden ***applyExistingSettings*** i ***GUI***-klassen virker, og modificér metoden, så den også håndterer de to tekstfelter, som I lige har tilføjet.

**Opgave 4**

I skal nu skabe en menubar, hvor *Game* menuen har de viste indgange, mens *Log* menuen har to indgange *Play log…* og *Save log…* De enkelte menuindgange skal have samme effekt, som de tilsvarende knapper nederst i vinduet. Der vil altså nu være to måder, hvormed brugeren kan vælge disse operationer. Ved forelæsningerne er det beskrevet, hvordan man får menuindgangen  
*New* *game* til at have samme effekt som *New game* knappen.



Menubaren tilføjes til ***mainFrame*** i ***GUI*** klassen ved hjælp af ***setJMenuBar*** metoden.

Når I skal implementere event håndtering, kan I med fordel læse dokumentationen for ***GUI*** klassen. Hastigheden af spillet ændres ved hjælp af den private metode ***setSpeed(int speed)***, hvor 1=slow, 2=medium, osv.

Dernæst skal I tilføje de viste tastaturgenveje til menuindgangene. Ved forelæsningerne er det beskrevet, hvordan dette gøres for menuindgangen *New game*. *Play log*… skal have genvejen *Ctrl+Shift-L*, mens *SaveLog...* skal have genvejen *Ctrl+Shift-S*.

I behøver ikke at få *Pause/Resume game* menuindgangen til at alternere mellem *Pause game* og *Resume game*, sådan som den tilsvarende knap gør. Men har I lyst til at prøve kræfter med det, er I selvfølgelig velkomne.

**Computerspil 5**

I denne femte og sidste delaflevering skal I bruge nogle af de ting, som I har lært om exceptions og fil-baseret input/output til at udvide programmet, så det kan optage et spil og gemme det på en fil, der så senere kan genindlæses og afspilles. Herudover skal I tilføje et tekstfelt, der lister de træk som brugeren laver under spillet. Hertil bruges *Decorator* designmønsteret, som gennemgås i kursets sidste forelæsning. Endelig skal I rette de fejl og mangler, som instruktoren har påpeget i jeres fjerde delaflevering.

Husk at holde jeres dokumentation opdateret, så den afspejler de ændringer/tilføjelser, som I laver i jeres kode. I behøver ikke at lave regression tests for ***Log*** klassen og jeres ændringer i ***GUI*** klassen. ***Log*** Klassen testes via testserveren, som beskrevet i opgave 1. Ændringerne i ***GUI*** klassen tester I manuelt, efterhånden som I laver dem.

**Opgave 1**

Implementér klassen ***Log***, der repræsenterer et gemt spil. Klassen har tre feltvariabler ***seed***, ***settings*** og ***choices***, der henholdsvis gemmer den initiale seed-værdi (for det ***Random*** objekt, der bruges i spillet), indstillingerne i ***Options*** dialogboksen og de valg, som brugeren laver (ved at trykke på byer med musen eller bruge piletasterne). Brugerens valg gemmes i en afbildning, der mapper fra skridt (heltal) til navnet på den by, der blev valgt i de pågældende skridt. Hvis brugeren trykker på flere byer i samme skridt, er det kun det seneste valg, der gemmes.

Klassen skal have en konstruktør på formen:

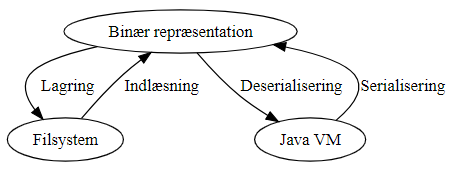
* ***Log(int seed, Settings settings)***

samt nedenstående metoder:

* ***int getSeed()***// Returnerer den initale seed-værdi
* ***Settings getSettings()*** // Returnerer indstillingerne i Options dialogboksen
* ***String getChoice(int step)*** // Returnerer navnet på byen, der blev valgt i skridtet
* ***void add(int step, City city)*** // Brugeren trykkede på byen city i skridtet step

Den tredje metode returnerer ***null***, hvis der ikke blev trykket på nogen by i det angivne skridt.

I de næste opgaver skal I gemme et ***Log*** objekt i filsystemet og efterfølgende læse det ind igen. For at gøre det, skal der skabes en binær repræsentation af ***Log*** objekt. Heldigvis har Java indbygget funktionalitet til det formål. Nogle objekter afhænger af variabler i operativsystemet (f.eks. et id på en proces) og kan derfor ikke genskabes fra sin binære repræsentation. En klasse angiver, at den er *serialisérbar* (dvs. at dens instanser kan gemmes i binært format og genskabes herfra) ved at implementere ***Serializable*** interfacet, der er et *tagging* interface, hvilket betyder, at det ikke indeholder metoder, men blot angiver, at klassen er serialisérbar (se BlueJ-bogen side 526-527).



Lad ***Log*** klassen implementere ***Serializable*** interfacet.

I skal nu afprøve om de ting, som I har lavet i opgave 1, fungerer korrekt, men først skal nogle af projektets klasser opdateres, hvilket sker ved at I kalder klassemetoden ***download*** i ***TestServer*** klassen med parameteren ”CG5”.

Kald dernæst klassemetoden ***test*** i ***TestServer*** klassen med parameteren ”CG5”.

Det tjekkes at ***Log*** klassen implementerer ***Serializable*** interfacet og at konstruktøren/metoderne virker, som de skal.

Hvis testserveren finder fejl, skal l gennemgå jeres kode og forsøge at rette dem.

**Opgave 2**

I operativsystemer er filer og mapper angivet via en unik sti. I Windows systemer er harddisken, hvor operativsystemet er installeret, typisk identificeret ved ***"C:\"***. Windows er så installeret i mappen ***"C:\Windows\"*** og hvis brugeren ***phmadsen*** har filen ***picture.jpg*** liggende i mappen ***Images*** på sit skrivebord vil denne fil have stien:

* ***Windows****:* ***"C:\Users\phmadsen\Desktop\Images\picture.jpg".***

I Linux og Mac OS er navngivningen lidt anderledes, og man bruger / som skilletegn i stedet for \:

* ***Linux****:* ***"/home/phmadsen/Desktop/Images/picture.jpg".***
* ***Mac******OS****:* ***"/Users/phmadsen/Desktop/Images/picture.jpg".***

I opgave 1 i Computerspil 4 skabte I en ***Save log*** knap og tilføjede en ***ActionListener***, der blot åbner en dialogboks, der viser at knappen fungerer. I skal nu implementere den rigtige funktionalitet for ***Save log*** knappen, dvs. gemme ***Log*** objektet på en fil.

Når brugeren trykker på ***Save log*** knappen, skal der vises en dialogboks, hvori brugeren kan specificere, hvad filen skal hedde, og hvor den skal gemmes. Til dette formål har ***GUI*** klassen en feltvariabel, ***fileChooser*** (af typen ***JFileChooser***). Feltvariablen initialiseres i ***GUI*** klassens konstruktør, så det behøver I ikke at tænke på. Find klassen ***JFileChooser*** i Java API’en. I klassen kan I finde en metode, som tillader brugeren at åbne en dialogboks, der kan gemme en fil. Kald denne metode på ***fileChooser*** (som parameter kan I bruge feltvariablen ***mainFrame***). Metoden returnerer et heltal. Brug dette heltal (og en passende klassevariabel i ***JFileChooser***) til at tjekke, om brugeren accepterede den valgte fil eller lukkede vinduet uden at vælge en fil.

Hvis brugeren valgte en fil og accepterede denne ved at trykke på ***Save/Gem*** knappen, skal ***Log*** objektet, som I får fat i via metodekaldet

* ***game.getLog()***

gemmes på den valgte fil ved hjælp af en ***ObjectOutputStream***. I kan finde et eksempel på brug af ***ObjectOutputStream*** i BlueJ-bogen på side 527. For yderligere information, se Java API’en om objekt output streams og file output streams.

Hvis der opstår exceptions, skal brugeren gøres opmærksom på dette via en dialogboks.

**Opgave 3**

I opgave 1 i Computerspil 4 skabte I en ***Play log*** knap og tilføjede en ***ActionListener***, der blot åbnede en dialogboks, der viser at knappen fungerer. I skal nu implementere den rigtige funktionalitet for ***Play log*** knappen, dvs. indlæse et ***Log*** objekt fra en fil og dernæst udføre det spil, som ***Log*** objektet repræsenterer.

Når brugeren trykker på ***Play log*** knappen, skal der vises en dialogboks, hvori brugeren kan specificere hvilken fil, der skal indlæses. Dette gøres på tilsvarende vis som i opgave 2, men nu skal i åbne den fil, som brugeren vælger (i stedet for at gemme den) og indholdet af filen bruges til skabe et ***Log*** objekt. Dette sker i et antal skridt, der minder meget om skridtene i opgave 2. Hvis der opstår exceptions, skal brugeren gøres opmærksom på dette via en dialogboks.

Endelig skal loggen afspilles via metodekaldet:

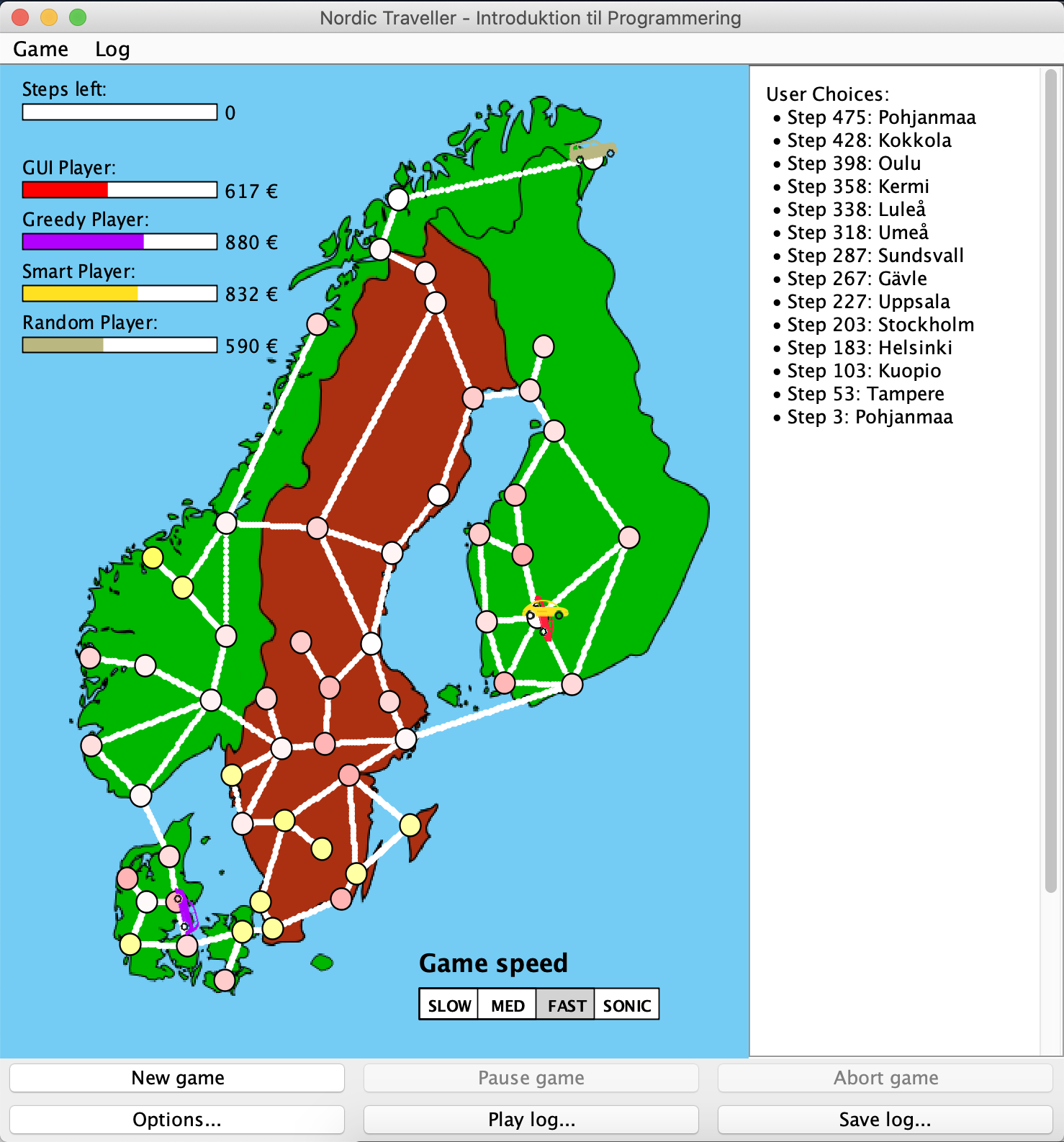
* ***game.playLog(log)***

hvor ***game*** er en eksisterende feltvariabel i ***GUI*** klassen og ***log*** er den log, der skal afspilles.

**Opgave 4**

Denne opgave er valgfri, forstået på den måde, at afleveringen kan godkendes, uden at I har løst opgave 4, men så kan I højst få 2,5 point.

Indsæt et tekstfelt, hvori I lister de træk, som brugeren laver. Da der kan være mange træk, skal tekstboksen forsynes med scrollbarer, så den får nedenstående udseende.



Nedenfor følger nogle vink til, hvad I skal gøre.

Opret en ny feltvariabel ***choiceText***. I konstruktøren sættes feltvariablen til at pege på et nyt tekstfelt vedhjælp af nedenstående kode:

* ***choiceText = new JTextArea(50,15);***

***choiceText.setText("User Choices:");***

***choiceText.setMargin(new Insets(10,10,10,10));***

Første linje angiver tekstfeltets størrelse. Anden linje indsætter startlinjen i vores tekst. Tredje linje laver nogle passende marginer. Tilføj en linje, der gør, at tekstfeltet ikke kan editeres.

Dernæst bruges*Decorator*designmønsteret fra kursets sidste forelæsning, og der indsættes nedenstående i konstruktøren:

* ***JScrollPane userChoices = new JScrollPane(choiceText,***

***JScrollPane.VERTICAL\_SCROLLBAR\_ALWAYS,***

***JScrollPane.HORIZONTAL\_SCROLLBAR\_NEVER);***

Første parameter angiver, at det er tekstfeltet ***choiceText***, der dekoreres med scrollbarer, mens de to andre parametre specificerer, at vi ønsker en lodret scrollbar, men ikke en vandret.

Find det sted i ***GUI*** klassens konstruktør, hvor der laves en anonym indre klasse, som er en subklasse af ***MouseAdapter*** klassen (beskrevet i forelæsningen om grafiske brugergrænseflader). Efter metodekaldet ***game.clickCity(c)*** insættes nedenstående metodekald, som bevirker at der skrives en linje i vores tekstfelt, hver gang brugeren vælger en by ved hjælp af musen:

* ***choiceText.append("\n • Step ” + game.getStepsLeft() + ": " + c.getName());***

Find det sted i ***GUI*** klassens konstruktør, hvor der er følgende metodekald ***game.clickCity(best).*** Efter dette metodekald indsættes nedenstående metodekald, som bevirker, at der udskrives en linje i vores tekstfelt, hver gang brugeren vælger en by ved hjælp af piletasterne:

* ***choiceText.append("\n • Step ” + game.getStepsLeft() + ": " + best.getName());***

Find det sted i ***GUI*** klassens konstruktør, hvor layoutet for ***superpanel*** sættes til ***BoxLayout*** og ret det til ***BorderLayout****.* Kortet (som hedder ***panel***) indsættes i ***CENTER****,* de seks knapper (som hedder ***buttons***) i ***SOUTH*** og det nye tekstfelt ***userChoices*** i ***EAST****.*