Projektbeskrivning

Monopol

2018-02-12

Projektmedlemmar:
Jacob Wahlman <jacwa448@student.liu.se>
Max Norling <maxno810@student.liu.se>

Handledare:
Mikael <handledare@ida.liu.se>

Table of Contents

Planering

1. Introduktion till projektet

Projektet är ett monopol spel utvecklat I Java. Spelet följer de regler som används I den officiella versionen av monopol, den är dock inspirerad av platser I Linköping e.g namn på områden. Det går att spela ett godtyckligt antal spelare. Spelet har även en förbättrad bank implementerad som tar hänsyn till den ekonomiska ställning spelaren är I och uppmuntrar till lån när spelaren inte köpt så mycket eller är allmänt fattig med låga räntekostnader och vise versa om en spelare med god ekonomi tar lån.

2. Ytterligare bakgrundsinformation

De regler som spelet går ut på går att återfinna här:
http://monopoly.wikia.com/wiki/Monopoly_Rules

Det enda som skiljer sig är hur banken fungerar.

3. Milstolpar

#	Beskrivning
1	Visa spelplanen grafiskt I ett fönster. Det ska även gå att stänga fönstret med hjälp av att kryssa fönstret.
2	Lägg till alla bilder som finns I monopol till spelplanen. e.g chans, community chest etc.
3	Implementera flyttbara pjäser som rör sig beroende på tärningskastet
4	Möjlighet att spela flera spelare.
5	Implementera pengasystemet.
6	Implementera möjlighet att köpa tiles och lägg in knapp för detta.
7	Implementera att sälja tiles och lägg in knapp för detta.
8	Möjlighet att köpa hus och visualisera detta.
9	Möjlighet att ta lån från banken.
10	Banken bör kunna säga nej till lån beroende på den ekonomiska ställning spelaren är I.
11	Klickbar spelplan som visar information om tiles.

12	Implementera speciella kort som chans och community chest kort som gör saker när man landar på dem.
13	Implementera fängelse funktionen.
14	Gör en turnsummering som visar vad som gjorts under en turn.
15	Allmänna spelregler.
16	Möjlighet att använda en egen board.
17	Möjlighet att skriva egna speciella kort.
18	Göra en meny för antal spelare och andra spelinställningar.
19	
20	
21	
22	
23	
...	

4. Övriga implementationsförberedelser

Spelet bör vara uppdelat på sådant vis att de skall gå att kopplas självständigt till en annan klass utan problem. Det vill säga att klassen för tiles skall kunna kopplas till spelare om man nu vill men kommer att kopplas till en board klass. Sedan allt som händer med en spelare bör ske I spelarklassen. Allt som är relaterat till banken bör ske I banken osv.

Man kan ibland spara mycket tid på att först ta en ordentlig funderare på t.ex. vad programmet ska göra, ungefär hur funktionaliteten kan delas upp mellan olika klasser, osv.

5. Utveckling och samarbete

Vi delar upp arbetet beroende på vad man känner för att arbeta med och vad som verkligen behövs göras. Vi arbetar dock för det mesta tillsammans så man kan diskutera om man stöter på problem eller vill diskutera hur något skall implementeras.

Alla gruppmedlemmar har förståelse för resterande del av koden då man förklarar vad man ändrat och varför. Vi behöver också förstå hur något annat fungerar då vi kommer att använda dessa delar I den kod som man själv skriver också. Exempelvis om en skriver klassen för att göra tiles, så kanske en annan skriver en implementation för att läsa in en textfil med brädesinformation. Då kommer den andre att behöva informationen från den förstes kod angående objektet för att se vad som krävs.

Jobbar helger utanför projekttid.

Betygsambitioner har vi båda 4.

Slutinlämning

6. Implementationsbeskrivning

6.1. Milstolpar

1-17. Genomfört.

18. Ej genomfört. Går att ställa in antal spelare I koden utan problem. Dock ingen meny för att göra detta I programmet.

6.2. Dokumentation för programkod, inklusive UML-diagram

Beskrivning

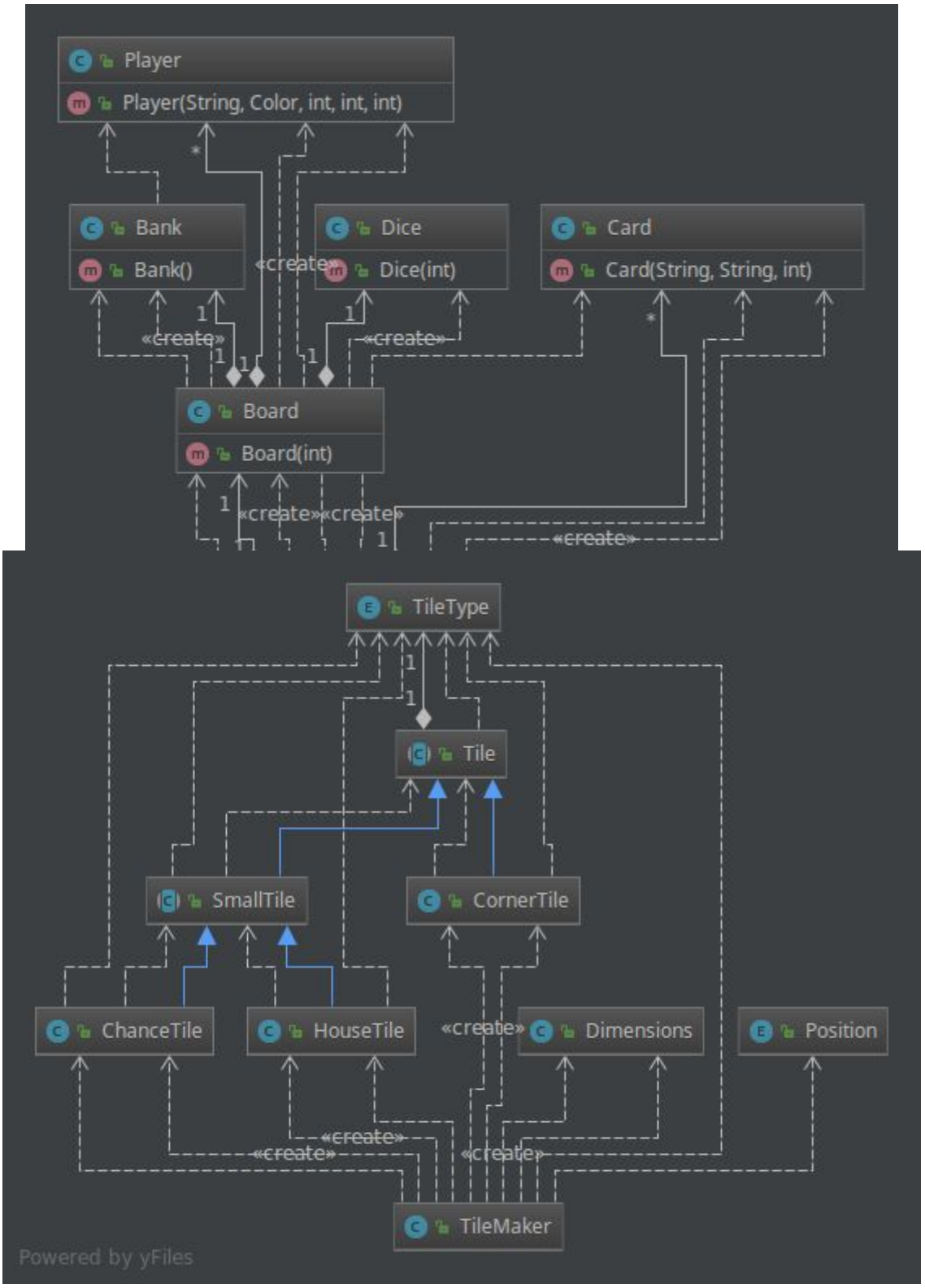
Programmet är en implementation av spelet monopol skrivet I Java. Spelet följer de allra flesta regler som satts I den officiella versionen, med undantag för några speciella egenskaper. Exempelvis ett verklighetsbaserat banksystem som lånar ut pengar beroende på den ekonomiska ställningen spelaren är I och anpassar räntan därefter. Spelet har också möjligheten att ladda in egna rutor och speciella kort med hjälp av csv (comma-separated values), vilket gör detta till en mycket enkel process som inte kräver något kodskrivande från användaren.

UML-Diagram

Ett top-level uml diagram finns under UML I projektmappen. Denna är alldeles för stor för att infoga I dokumentet då det inte går att se vad det står I den. Programmet är uppdelat I tre paket som består av **gamelogic, gui** och **tiles.** Dessa har I sin tur en samling av klasser. Dessa klasser är utformade sådana att de skall kunna förändras utan problem. Exempelvis om man vill förändra hur banken fungerar eller att lägga till flera tärningar.

Relationen mellan de olika klasserna som agerar under spellogiken. Board som är den centrala delen I spellogiken styr spelets gång. Alla klasser skapas en gång och kontrolleras från board, förutom spelare (Player) och kort (Card) då det kan finnas flera av dessa och vi vill att de skall på ett enkelt vis skall kunna hanteras separat. Detta gör också att det är enklare att kontrollera flödet I programmet och säkerheten kring att man vet var något kom ifrån. Exempelvis relationen mellan spelare och bank är riktad åt ett håll och tillåter banken att ta eller ge pengar till spelaren beroende på vad spelaren landar på eller gör under en runda,

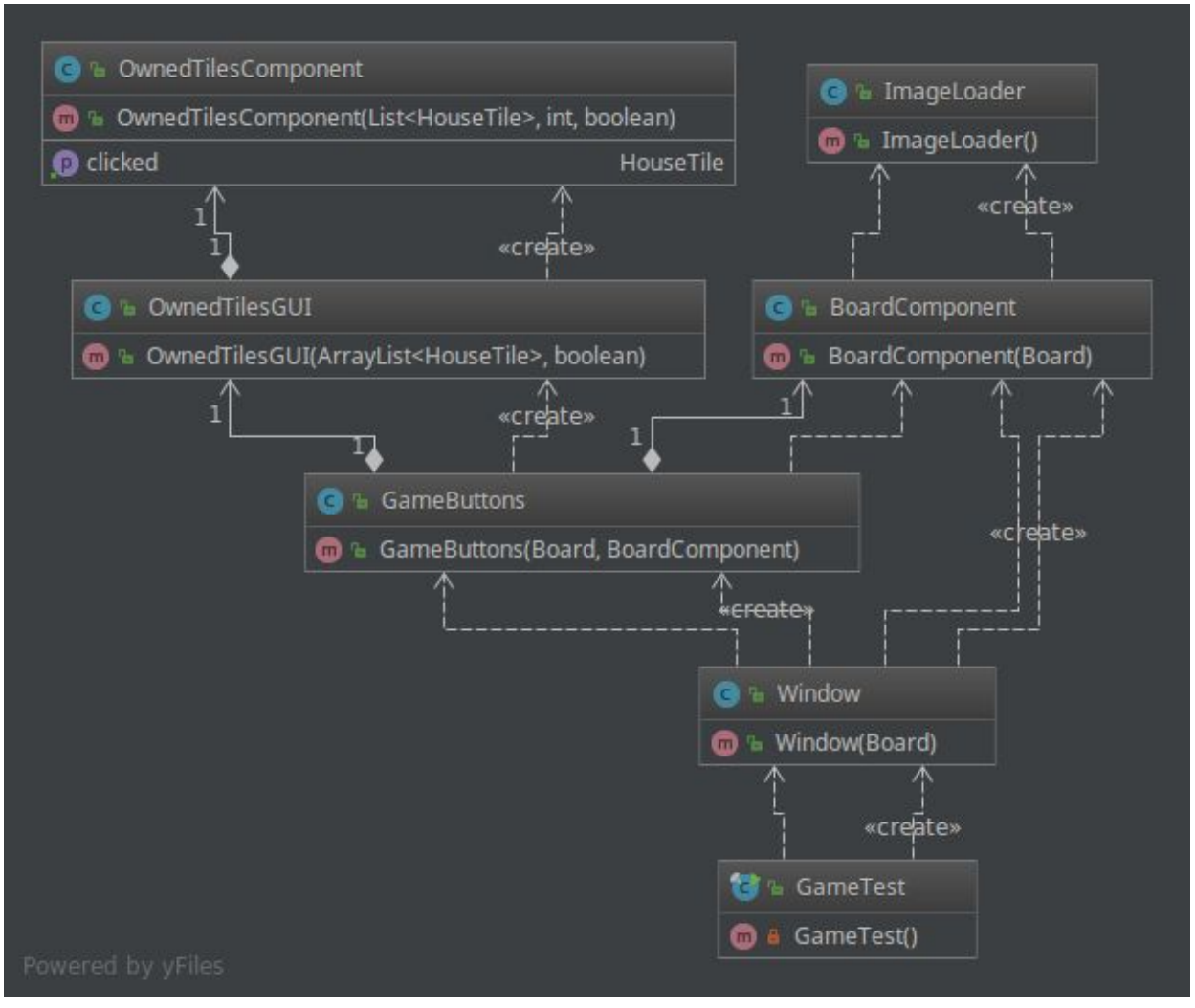
spelaren kan dock aldrig direkt interagera med banken direkt. Det gör det även möjligt att använda fler än en tärning då det bara skapa en ny instans av Dice och representera den. Vi kan också se att det finns en klass LoadBoard, denna klass är separerad och används för att låta användaren läsa in egna rutor med egen information som namn, pris, hyra etc. Den tillåter även att spelaren läser in egna speciella kort som chans.



Skapandet av de olika tiles eller rutor som representeras på spelbrädet. De rutor som representeras på spelbrädet är av olika typer, men skapas av samma klass TileMaker. Beroende på om de är speciella kort som chanskort, hörnrutor eller helt vanliga rutor som man kan köpa så har de olika egenskaper men samma storlek förutom hörnrutor. Detta gör att

TileMaker kan skapa alla och bara ärva nedåt de egenskaper som är samma för alla. Detta gör det också mycket enkelt att skapa egna rutor eller en helt ny sorts korttyp.

Spelet visualiseras genom en serie av klasser som ansvarar för olika delar. Window som är den centrala delen visualiserar de andra klasserna, exempelvis alla knappar för att genomföra händelser som att köpa eller ta lån etc. Detta gör det mycket enkelt att visualisera nya knappar eller skapa en ny klass att visualisera. Vi använder oss också av bilder för att visualisera bilder som hörnrutor etc. det gör det också mycket enkelt att byta ut till egna bilder.



6.3. Användning av fritt material

Vi har **inte** använt några externa bibliotek, vi har enbart använt oss av Java 8 I projektet. Vi har inte heller kopierat eller använt oss av kod som någon annan skrivit.

6.4. Användning av objektorientering

För varje betygsnivå måste ni peka ut *hur* ni har använt ett visst antal olika **typiska egenskaper hos just objektorienterade språk.** Ni kan välja bland följande, beroende på vad ni har använt – fråga examinatorn i förväg om ni har egna förslag!

- **Objekt/klasser**, som ett sätt att samla information **och** beteende / funktionalitet.
- **Konstruktorer**, som ett sätt att initialisera objekt
- **Typhierarkier**, där en variabel kan peka på ett objekt som tillhör en ”bastyp” (till exempel ett gränssnitt/interface) eller vilken av dess subtyper som helst.
- **Subtypspolymorfism** med sen/dynamisk bindning, för att anropa "rätt" implementation av en metod utan att känna till objektets exakta typ.
- **Ärvning med overriding**, specifikt för att *byta ut* existerande fungerande funktionalitet från en av era egna klasser i en av era egna subtyper.
- **Inkapsling (encapsulation)**, i betydelsen att fält och metoder bara kan användas från vissa klasser.

Några icke-specifika egenskaper hos objektorienterade språk:

- Även i andra språk kan man undvika att repetera kod.
- Även i andra språk kan man samla data i poster (record, struct) och till exempel skicka med detta som parametrar till metoder.
- Även andra språk har enum-typer.
- Även andra språk har serialisering av datastrukturer.

För varje egenskap ni beskriver ska ni diskutera följande. Se detta som **tentafrågor** där ni försöker ge tillräckligt med information för att verkligen visa upp era kunskaper!

- **Vilken egenskap** det gäller
- Exempel på **hur ni använt den**, och var. Ange exakta namn på klasser och metoder så att det lätt går att hitta exemplet/exemplen, och beskriv vad ni har åstadkommit genom att använda egenskapen (vilken funktionalitet ni ville införa, till exempel).
- Hur ni kunde ha löst detta (på ett rimligt sätt!) **utan objektorientering**, ifall ni hade använt ett språk där den aktuella egenskapen inte var tillgänglig.
- **Hur lösningarna skiljer sig åt**. Vad vinner (eller förlorar) ni på att använda den objektorienterade finessen istället för att lösa det på något annat sätt? Eller är lösningarna kanske likvärdiga, även om de ytligt ser olika ut? Här ska ni alltså jämföra med en någorlunda *rimlig* alternativ lösning, som man faktiskt kunde ha använt om man ville lösa samma "problem" i ett icke objektorienterat språk.

Numrerade svar:

1. Ärvning hos Tiles. Vi kan se I UML-diagrammet ovan hur TileMaker kan skapa alla Tiles och hur några av tiletyperna ärver från varandra. Utan arv hade vi behövt skriva alla funktioner som finns i förälder klassen i alla barns klasser så istället för bara en getX() i föräldern hade det behövts en för varje klass och då blir det onödigt mycket repetition av kod.
2. Board som skapar ett spelbrädes använder sig av en konstruktor för att bestämma startvärden. Startvärdena hade kunnat sättas som en attribut till klassen. Detta hade gjort det svårare att skapa bräden med olika attributer. Exempelvis brädstorlek etc. Vi vinner på att använda oss av konstruktorer för att lätt kunna ändra på olika attributer och använda oss av olika värden för olika objekt. Exempelvis så hade det varit mycket jobbigare att skapa olika tiles om de inte hade funnits konstruktorer då vi skulle behövt sätta varje attribut för sig själv. Exempellösningen är då att ett bräde har attributer som beskriver ett standardbräde, säg 250x250. Men om vi inte vill ha 250x250 utan spelaren bestämmer sig för att köra på 100x100 istället så måste en funktion som sätter dessa värden skrivas. Det gör också att vi inte skulle kunna ha flera bräden igång samtidigt utan vi skulle hela tiden bara kunna referera till ett och samma bräde under spelets gång, om ingen konstruktor skulle finnas.
3. Tile och SmallTile är abstrakta klasser som andra klasser ärver från.
4. TileMakers metoder som bestämmer storleken har blivit inkapslade och klassen kan bara refereras om man vill skapa en Tile.
5. HouseTile ärver med orverriding av landAction för att bestämma beteende hos rutan. Det skulle gå att definera en landAction för varje typ av Tile utan att “overrida” landAction, dock så finns det inget krav längre att implementera den. Vi vinner inte kanske prestandamässig utan mer strukturellt sätt på att ha en abstract metod som vi “overridar” eftersom alla typer av Tile är en subtyp av Tile och om Tile har en metod så bör då också alla subtyper ha den metoden.
6. Board har metoder som är inkapslade exempelvis movePlayer, eftersom vi vill kunna kontrollera vilken klass som flyttar spelaren. Det skulle ha gått att skippa att kapsla in movePlayer och låtit alla andra funktioner/metoder interagera med movePlayer. Detta hade dock lett till att det blir svårare att kontrollera vilken del som gör vad. Eftersom spelaren flyttar i princip varje gång så blir det enklare att kontrollera programkörningen för en spelares omgång samt enklare att felsöka då man alltid vet vart i programmet som spelaren flyttas.

6.5. Motiverade designbeslut med alternativ

När man har **flera rimliga alternativ** för hur programmet ska designas och struktureras, och väljer ett av alternativen, tar man ett **designbeslut**. För varje betygsnivå ska ni beskriva ett antal designbeslut ni har tagit. Anledningen är både att ni ska visa oss hur ni har tänkt (som en tentafråga) och att ni själva ska fundera och reflektera över era val.

Många studenter har kommenterat att de har lärt sig mycket på att tänka genom dessa beslut!

Vad är <u>inte</u> ett designbeslut?

- Att man väljer **int** istället för **long** för ett visst fält är snarare ett "kodningsbeslut". Vi är intresserade av design på en lite högre nivå!
- Sådant som **ska göras enligt projektkraven** är inte (gruppens) designbeslut, t.ex. att

de flesta fält ska vara privata.

- När det **saknas rimliga alternativ** har man inte tagit ett designbeslut.
    - Att dela upp koden i flera klasser är ett uppenbart beslut i ett projekt av denna storlek. Att samla allt i en enda klass är inte ett tänkbart alternativ man behöver fundera på, utan något som är uppenbart orimligt.
    - Att man använder konstruktorer direkt istället för att komplicera sitt program med designmönstret Abstract Factory är inget beslut – om det inte är så att något tydligt talar för att använda Abstract Factory, samtidigt som det finns starkare fördelar med att undvika mönstret.

För varje designbeslut ni beskriver ska ni diskutera följande. Se återigen detta som **tentafrågor** där ni vill ger tillräckligt med information för att visa upp era kunskaper!

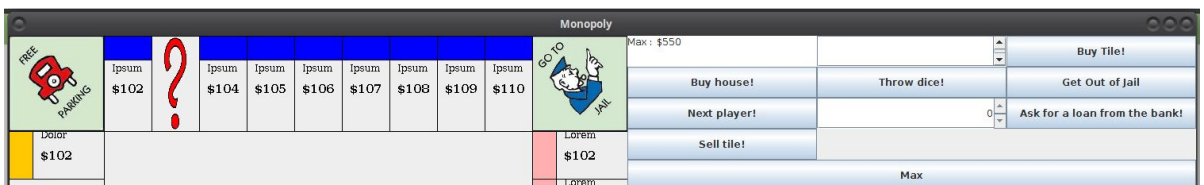
- Vad ni **ville åstadkomma**.
- **Hur** ni gjorde detta. Namnge klasser och metoder som är inblandade och förklara lösningen.
- En rimlig **alternativ lösning** som ni **kunde** ha använt istället.
- Varför den lösning ni valde var bättre än alternativet. (I vissa fall kanske ni inser att den lösning ni valde till slut blev *sämre* än alternativet – beskriv då varför.)

Numrerade svar:

1. Alla “Tiles” som finns på spelplanen ärver från en och samma Tile klass eftersom alla “tiles” behöver ha x och y koordinater en höjd och bredd och något som händer när spelaren landar på brickan och ett enkelt sätt och kolla om spelaren klickar på brickan så istället för att behöva implementera de funktionerna i de enskilda klasserna finns de i en klass.
2. Vi har en Dimensions klass som håller variablerna som behövs för att skapa de olika sorters brickorna som finns alltså de med bilder på och de med bara en färgad rektangel. Utan den klassen så hade vi behövde spara de variablerna någon annanstans som t.ex i en Array och då är det betydligt svårare att förstå vad vilken variabel ska göra.
3. Vi har en enum klass med de olika positionerna brickorna kan ha alltså UP, DOWN, LEFT och RIGHT så vi kan kalla på en och samma funktion för att skapa brickor som i sin tur väljer rätt funktion för de olika positionerna annars hade vi behövt fyra stycken olika “makeChanceTile” och “makeHouseTile” funktioner en för varje position.
4. Alla Tiles måste ha en TileType variabel som är en enum. Den används i PaintComponent som finns i BoardComponent för att rita ut de olika Tile sorterna, vissa tiles ska ha bilder medans andra bara ska ha en färgad rektangel. Vi skulle ha kunnat kolla vilken klass en Tile ärver ifrån med instanceof men det kändes enklare att de bara hade en variabel som alltid fanns där.

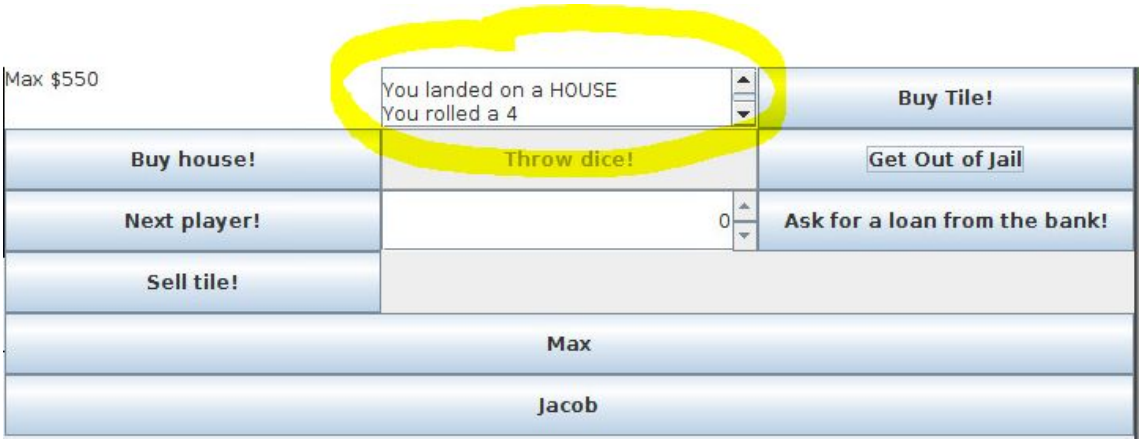
7. Användarmanual

Spelet startas genom att klicka på 'Run' I GameTest klassen under gui paketet. Spelet öppnas I



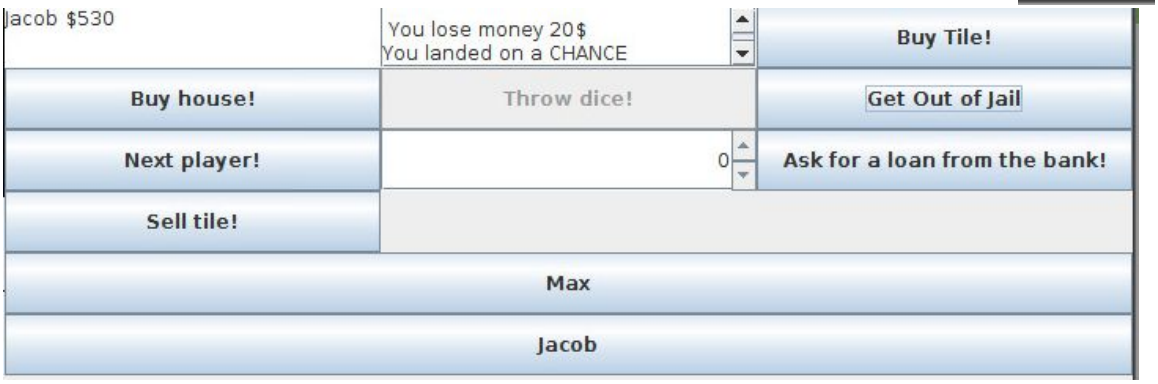
ett nytt fönster och bör se ut enligt följande,

Högst upp I vänstra hörnet så visas vems tur och hur mycket pengar de har. För att börja spelet så trycker man på throw dice som visualiserar I mitten av skärmen. Beroende på vart du hamnar så kan det ske lite olika saker dessa sammanfattas I textfältet till höger om informationen om spelaren och ser ut såhär,

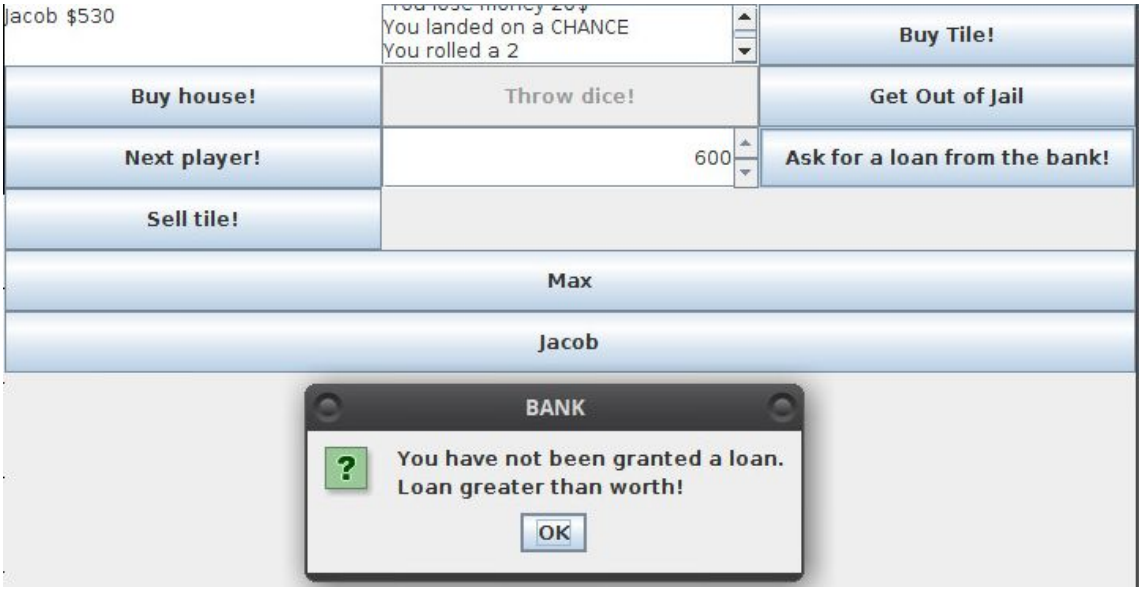


Eftersom spelet går ut på att köpa rutor så gör man det genom att trycka på buy tile när man står på den ruta man vill köpa, rutan kommer om man lyckats köpa den få en kant med samma färg som spelaren.

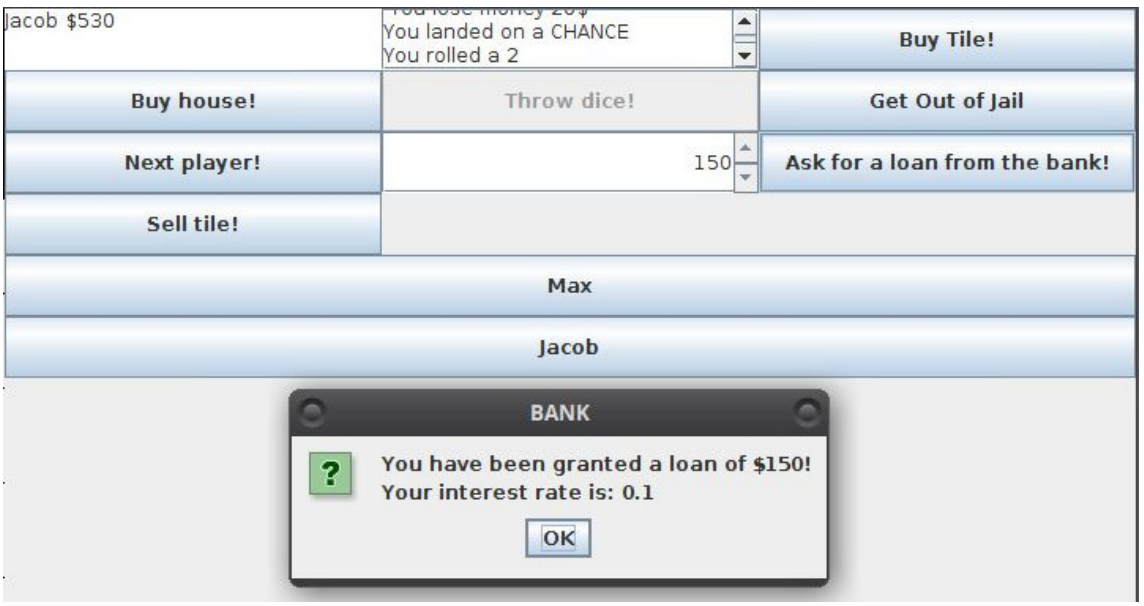
Om en spelare landar på en chans eller community chest så kommer spelaren få ett kort av brädet som har olika betydelser exempelvis så kan spelaren förlora pengar,



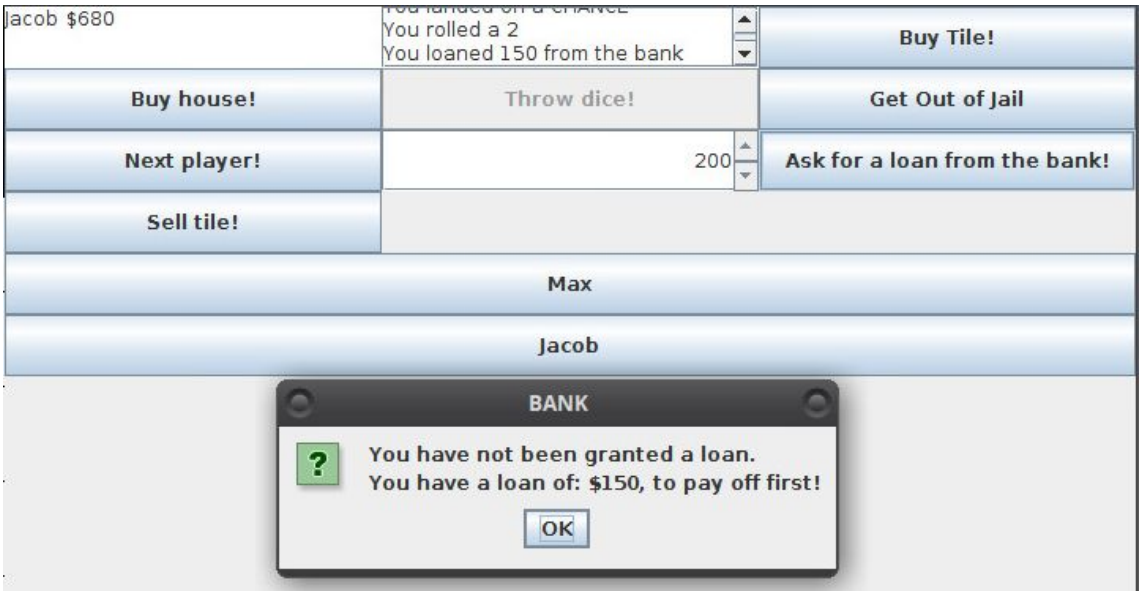
Banken I spelet kan låna ut pengar till en spelare genom att spelaren specificerar belopp I spinnern till vänster om låna knappen. Beroende på hur mycket spelaren vill låna så kan spelaren bli nekad till lån av olika skäl. Beslutet visas I ett fönster,



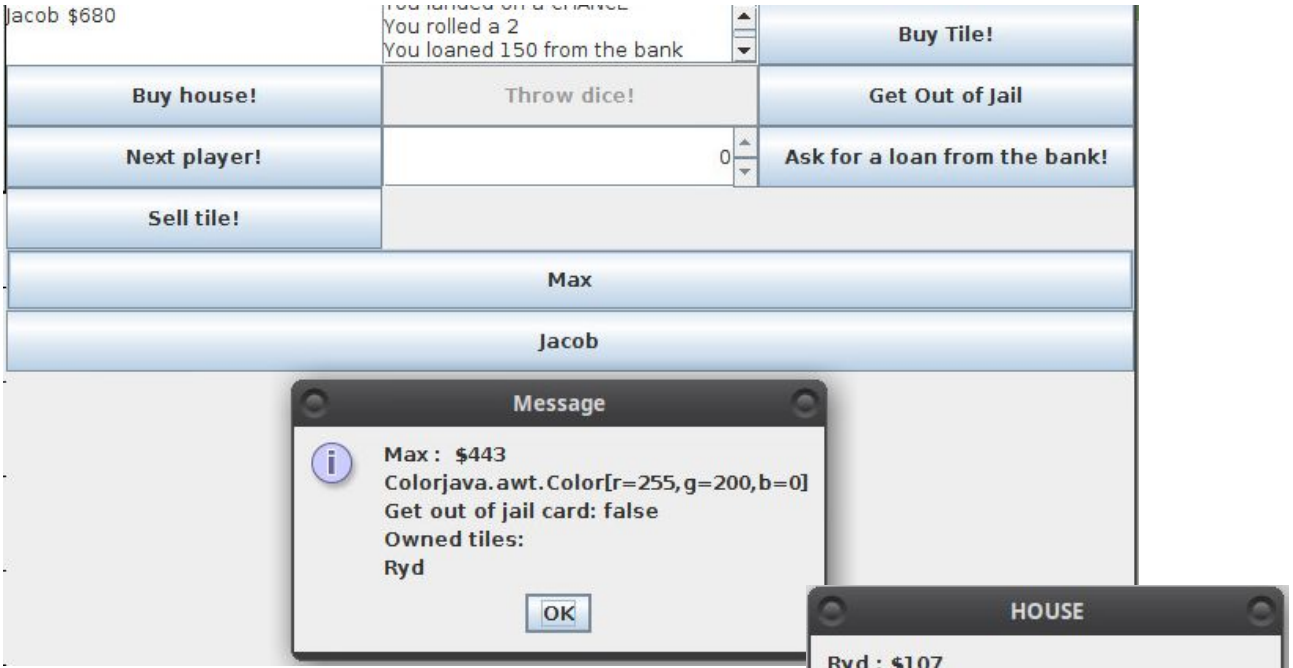
eller,



Som vi kan se ovan så fick spelaren ett lån. Räntan baseras på hur mycket spelaren är värd, alltså mängden likvida tillgångar och det totala värdet som spelarens rutor är värda. Amorteringen tas varje gång spelaren avslutar sin runda. Om spelaren försöker ta ett till lån så kommer spelaren att bli nekad då det inte är tillåtet att ta flera lån samtidigt.

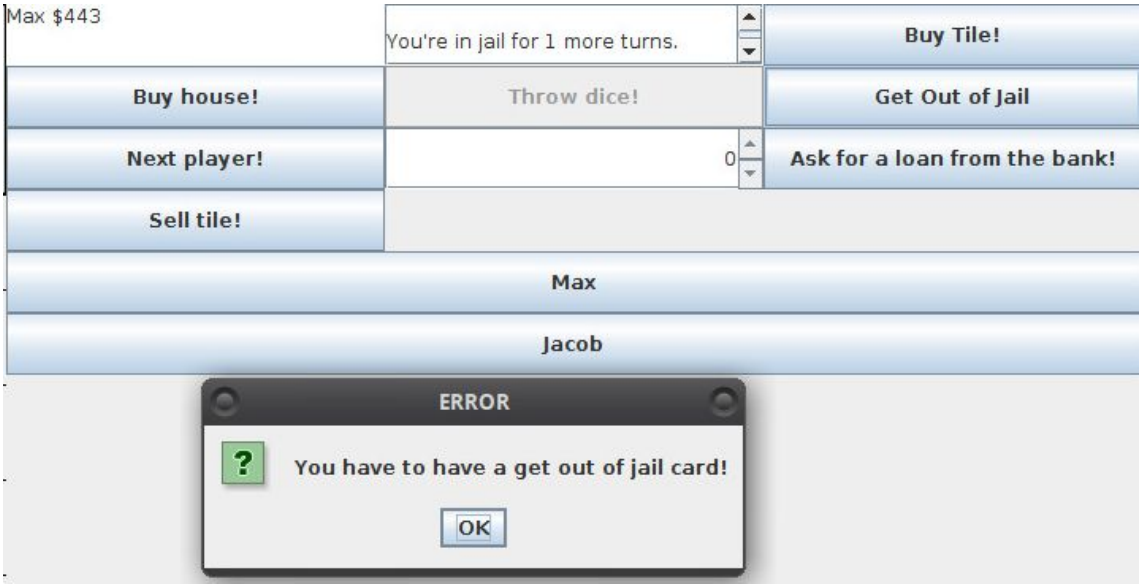


Om man klickar på ett av spelarnamnen längst ned så får vi upp information om spelaren I ett nytt fönster.



Samma gäller även om vi klickar på en ruta på spelbrädet.

Spelaren kan även hamna I fängelse genom att landa på hörnrutan där man blir skickad till fängelse eller får ett chans kort som skickar en. Man kan då använda ett get out of jail card om man fått det tidigare I spelet. Vi kan se här hur det ser ut annars,



Om man inte fått ett sådant kort tidigare så kan vi se I summeringen ovan att man kan vänta ut det också genom att vänta tre rundor.

Det finns vissa restriktioner på vad spelaren får göra och inte göra. Om spelaren gör något den inte får göra, exempelvis försöker köpa hus även fast man inte äger alla rutor av samma färg så kommer spelet att visa ett felmeddelande likt ovan och informera om det.

Sedan så fungerar resten på liknande sätt som beskrivit ovan. Spelet följer också standard spelregler så inga oväntade händelser kommer att ske som behöver beskrivas I detalj.

8. Slutgiltiga betygsambitioner

Vi siktar på betyg 4.

9. Utvärdering och erfarenheter

- *Vad gick bra? Mindre bra?*
    - *Det som gick bra var att implementera själva spellogiken och alla de delar som bygger upp spelet I sig. Det som gick mindre bra var att bygga upp GUI för spelet, exempelvis hade vi problem med knappar och en sammanställning av rundan. Vi antog att detta inte skulle ta så lång tid och nedprioriterade detta vilket gjorde att det blev svårt att hinna med att göra sedan.*
- *Vilket material och vilken hjälp har ni använt er av? Har ni gått på föreläsningar? Läst boken?*

Letat på nätet? Gått på handledda labbar? Ställt många frågor? Vad har "hjälpt" bäst? Vi vill gärna veta för att kunna vidareutveckla kurs och kursmaterial åt rätt håll!

- *Det material vi använt oss av var regler för monopol och letat upp problem som vi fått på StackOverflow. Vi gick inte på några labbar då vi kände att vi inte hade några frågor som vi behövde fråga om specifikt.*

- *Har ni lagt ned för mycket/lite tid?*
  - *Lite för lite tid I början vilket gjorde att det blev stressigare mot slutet och slarvigare kod blev skriven som vi behövt göra om.*
- *Var arbetsfördelningen jämn? Om inte: Vad hade ni kunnat göra för att förbättra den?*
  - Arbetsfördelningen var relativt jämn. En skrev lite mer då den var mer van med Java sedan innan men den andra fick också chansen att skriva kod och implementera olika funktioner I programmet. Men all kod vi skrev diskuterade vi med den andra och förklarade så vi båda visste vad den andra tänkt och hur programmet fungerar.
- *Har ni haft någon nytta av projektbeskrivningen? Vad har varit mest användbart med den? Minst?*
  - *Inte speciellt då vi glömde bort den lite I och med att vi skrev programmet. Vi hade också en tydligt ide om hur vi skulle lägga upp programmet. Dock så hade det varit bra om vi använt oss mer av milstolpar när vi skrev programmet.*
- *Har arbetet fungerat som ni tänkt er? Har ni följt "arbetsmetodiken"? Något som skiljer sig? Till det bättre? Till det sämre?*
  - Det har fungerat bra.
- *Vad har varit mest problematiskt, om man utesluter den programmeringstekniska delen? Alltså saker runt omkring, som att hitta ledig tid eller plats att vara på.*
  - Nej, inget sådant vi satt antingen hemma och skrev eller I cyd-poolen.
- *Vilka tips skulle ni vilja ge till studenter i nästa års kurs?*
  - Skriv projektbeskrivningen innan ni börjar.
- *Har ni saknat något i kursen som hade underlättat projektet?*
  - Nej.
- *Har ni saknat något i kursen som hade underlättat er egen inlärning?*
  - Nej.