**Programvaruteknik – utveckling och underhåll, 15 hp och VT14**

Intrimning av nytt API och demoutveckling för pluggbara brädspel

av

Mattias Didriksson, Anders Blomkvist och Cecilia Tyboni

Akademin för teknik och miljö

Högskolan i Gävle

S-801 76 Gävle, Sweden

Innehåll

[Inledning 1](#_Toc388892235)

[Designmönster 2](#_Toc388892236)

[Testdriven design 2](#_Toc388892237)

[Unified modeling language 2](#_Toc388892238)

[Refactoring 2](#_Toc388892239)

[Ingen javadoc-notation 2](#_Toc388892240)

[Metod 3](#_Toc388892241)

[Scrum 3](#_Toc388892242)

[Git 3](#_Toc388892243)

[Start av projektet 3](#_Toc388892244)

[Unittestning 4](#_Toc388892245)

[Factory Pattern 5](#_Toc388892246)

[Strategy Pattern 6](#_Toc388892247)

[Observer Pattern 7](#_Toc388892248)

[Decorator 7](#_Toc388892249)

[Model-View-Controller(MVC) 8](#_Toc388892250)

[Resultat 8](#_Toc388892251)

[Solitär 9](#_Toc388892252)

[Fia med knuff 10](#_Toc388892253)

[Diskussion 12](#_Toc388892254)

[Att arbeta med Scrum 12](#_Toc388892255)

[Att arbeta med refactoring 12](#_Toc388892256)

[Att arbeta med Unittester 12](#_Toc388892257)

[Förbättringar av arbetet 13](#_Toc388892258)

[Fortsatt arbete 13](#_Toc388892259)

[Vad var bra/dåligt med projektet? 13](#_Toc388892260)

[Vad tyckte vi om detta arbete? 13](#_Toc388892261)

[Referenser 14](#_Toc388892262)

[Bilagor 15](#_Toc388892263)

# Inledning

Syftet med detta projekt är att skapa brädspel med hjälp av ett givet API, Gameboard. Detta API kommer utvecklas under projektets gång. De två brädspel som i detta projekt ska tas fram är ”Fia med knuff” och ”Solitär”.

Fia med knuff är ett brädspel där antalet spelare kan variera mellan två till fyra spelare. Genom att slå en tärning bestäms antal steg en spelare får förflytta sig åt gången runt spelplanen som visas i bild 1.

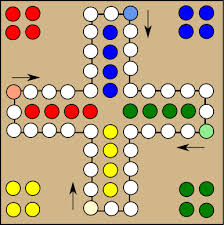


Bild - Spelplan för Fia med knuff

Varje spelare har fyra spelpjäser i sitt bo vid start. För att vinna måste en spelare flytta alla fyra spelpjäser ett varv runt spelplanen och därefter gå in i mål med dem. Om en spelare hamnar på samma ruta som en annan spelare redan står på kommer den andra spelarens spelpjäs att knuffas bort. Spelaren får då flytta tillbaka spelpjäsen till sitt bo och börja om med den spelpjäsen [1].

Solitär är ett brädspel för en spelare. För detta spel kan spelplanen variera och de olika varianterna kan ses i bild 2 nedan. I detta projekt används den engelska varianten av solitär.

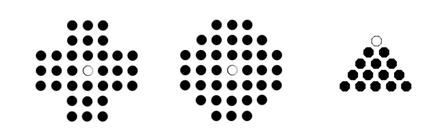


Bild - Olika spelplaner för Solitär, från vänster: engelsk solitär, fransk solitär och triangelsolitär

Vid spelets start är varje position på brädet fylld med en kula, förutom den mittersta. Spelet går ut på att plocka bort så många kulor som möjligt. Det görs genom att vertikalt eller horisontellt hoppa med en kula över en annan. Den kula som man hoppar över får plockas bort från spelplanen. [2]

I detta projekt ska minst två spelkärnor tas fram, i detta fall, Fia med knuff och Solitär. Till detta skulle det skapas två användargränssnitt, ett grafiskt och ett konsolbaserat. Dessa skulle med mindre modifikationer vara kompatibla med andra spelkärnor.

Ett av de mål som skulle nås med projektet var att få en djupare förståelse hur designmönster kan appliceras och användas under programvaruutveckling. Ulf Bilting beskriver i sin bok, “Designmönster för programmerare” [3] designmönster som arkitektur och används inom programutveckling. Dessa är en samling av mönster som kan hjälpa till med återkommande problem. Dessa mönster hjälper designers och programmerare att konstruera välstrukturerade, robusta och mer flexibla program. Ett designmönster är väl beprövat och löser ett välkänt existerande problem.

## Designmönster

De designmönster som använts vid detta projekt är Factory, Model-View-Controller, Strategy, Observer och Decorator. Utöver detta ligger vikt vid polymorfa lösningar genom hela projektet. För Fia med knuff används alla ovanstående designmönster, för solitär används bara de två första. Då fokus även låg på agil utveckling så användes ett välkänt arbetssätt som främjar detta, Scrum.

## Testdriven design

Testdriven design(TDD) är ett effektivt sätt att testa koden under implementationsprocessen. För att detta arbetssätt ska fungera optimalt bör ingen programkod skrivas utan att ett testfall har skrivits. Vid TDD testas varje kodblock enskilt utav automatiserade skrivna enhetstester.

## Unified modeling language

Unified modeling language(UML) är ett generellt objektorienterat språk för modellering av olika system. Genom att använda UML så går det att skapa en modell över systemet. Detta ger en överblick för hur systemet ser ut och därmed förenklar implementation det.

## Refactoring

Att arbeta med refactoring är att förbättra och “städa upp” koden utan att kodens beteende förändras. Vid refactoring kan exempelvis antal kodrader minska eller så kan metodnamn bytas ut till mer beskrivande metodnamn för att förklara vad själva metoden utför. Man brukar prata om att dela upp ansvaret så att en metod eller klass endast ska utföra **en** sak. Refactoring ska motverka “Smelly code” och ge en bättre struktur på programkoden.

## Ingen javadoc-notation

Valet av att inte använda oss utav förklarande kommentarer och javadoc-notation i koden var medvetet då det är en tankegång som uppkommer inom refaktorisering och som även förespråkas i Clean Code [4]. Man bör skriva klasser, metoder och variabelnamn så belysande samt förklarande att kommentarer känns överflödigt. Tycker man att kommentarer känns nödvändig så bör man se över koden och städa upp den istället.

# Metod

Här nedan beskrivs de metoder som använts i arbetet med detta projekt.

## Scrum

För att fördela arbete och få struktur på hur projektet genomfördes så beslutades att utvecklingsmetoden Scrum skulle användas. Det centrala i Scrum är att innan utvecklingsarbete börjar varje dag kort ge en uppdatering om hur man ligger till i sitt arbete. Enligt Mike Cohn, Mountain Goat Software [5], ska man svara på tre enkla frågor:

* Vad gjorde jag igår?
* Vad ska jag göra idag?
* Vad hindrar mig ifrån att göra det?

Genom att varje dag få denna snabba uppdatering ifrån alla så hålls gruppen informerade om hur projektet ligger till, samt så kan man tidigt upptäcka eventuella större frågor som måste tas upp.

Utöver morgonmötena så genomförs också ett större möte där det gemensamt tas fram och planeras vad som ska genomföras under nästkommande period, kallad Sprint. På en sprint så bestäms en uppskattad tid samt vem som har till uppgift att utföra den. Vid slutet av sprinten går det enkelt att utvärdera hur perioden har gått. Genom att regelbundet genomföra detta så blir man under loppet bättre på att uppskatta hur mycket tid en uppgift kan ta, samt ge en klarare uppfattning om vart man befinner sig i projektet. I större projekt kan detta vara ovärderligt om man har en deadline som måste uppehållas.

## Git

För versionshantering så användes Github. Git är en distribuerad lösning vilket innebar att alla har ett lokalt repo på sin dator. Detta synkas sedan upp till Github. Genom användning av denna lösning kunde man göra ändringar och tillägg utan att vara rädd för att förstöra något permanent [6].

## Start av projektet

Vid start av projektet bestämdes vilka spel som skulle skapas. Fia med knuff och Solitär har liknande spelplaner och därmed passar de bra för projektarbete. Arbetssättet vid start av detta projekt har varit att jobba parallellt med de större delarna så som det grafiska gränssnittet(hädanefter refererat till som GUI) samt motsvarande konsolversion för Solitär och Fia med knuff.

Till en början så implementerades grova regler och funktioner i kärnklassen för spelet, LudoGameState.java respektive SolitarGameState.java. Vartefter att funktionalitet började implementeras så valdes att tidigt skapa en simpel konsolversion för att få en grafisk överblick och feedback.

Desto längre projektet fortlöpte så började man fokusera på mer specifika detaljer i spelen, såsom en specifik regel, sällsynta buggar eller specialfall som att avsluta spelet när inga fler drag kan göras. Det var också vid detta stadie som Unittestning användes framförallt. Det märktes ganska omgående att när ett visst specifikt problem uppräcktes kunde spelplanen enkelt återskapas - framförallt i Solitär som har en betydligt enklare struktur och uppbyggnad av spelplanen - i ett test och utgå ifrån detta. Detta gjorde att det snabbt gick att få feedback vid en ändring och kunde se ifall buggen var åtgärda.

Konsolen för Solitär skapades först för att den inte är lika komplex som Fia med knuff. Dock så kan spelplanen byggas upp på liknande sätt som fia med knuff. Detta gör då att när Solitär fungerar så kan en del av grundkoden återanvändas i Fia med knuff. De större förändringar som måste göras är reglerna för spelet eftersom Solitär har mindre komplicerade regler.

## Unittestning

En viktig del och ett utav målen med projektet var Unittestning, enhetstestning på svenska, som förekommer inom TDD som vi skrev tidigare och är en mjukvaruutvecklingsprocess där man granskar och testar enskilda funktioner/kodblock, detta innebär endast de egenskaper som är avgörande för resultatet är för den enhet som ska testas. För oss som utvecklare skapar detta en säkerhet vid ändringar av befintlig kod och implementering av nya funktioner [6].

När det kommer till testerna för detta arbete så har den största vikten lagts på tester för GameStateklassernai de båda spelen och RulerControllersamt BaseControllerför Ludo*.*Här nedan ser vi ett litet utdrag från ett test som görs på proposeMoveför SolitarGameState*.*

|  |
| --- |
| *@Test*  ***public******void*** *proposeMoveTest() {*  *assertTrue(gameState.proposeMove(****new*** *Move(gameState.getPlayerInTurn(),*  *SolitarHelpMethods.getBoardLocationFromCoordinate("24",*  *gameState.getBoard()), SolitarHelpMethods.getBoardLocationFromCoordinate(*  *"44", gameState.getBoard()))));*  *assertEquals(****null****,SolitarHelpMethods.getBoardLocationFromCoordinate("24",*  *gameState.getBoard()).getPiece());*  *assertFalse(gameState.proposeMove(****new*** *Move(gameState.getPlayerInTurn(),*  *SolitarHelpMethods.getBoardLocationFromCoordinate("24",*  *gameState.getBoard()), SolitarHelpMethods.getBoardLocationFromCoordinate(*  *"34", gameState.getBoard()))));*  *assertFalse(gameState.proposeMove(****new*** *Move(gameState.getPlayerInTurn(),*  *SolitarHelpMethods.getBoardLocationFromCoordinate("14",*  *gameState.getBoard()), SolitarHelpMethods.getBoardLocationFromCoordinate(*  *"34", gameState.getBoard()))));*  *}* |

Källkod - Unittest för proposeMove

I det första fallet kollar man om det går att göra en förflyttning från koordinat 24 till 44. Som vi nedan kan se så är detta möjligt att göra.

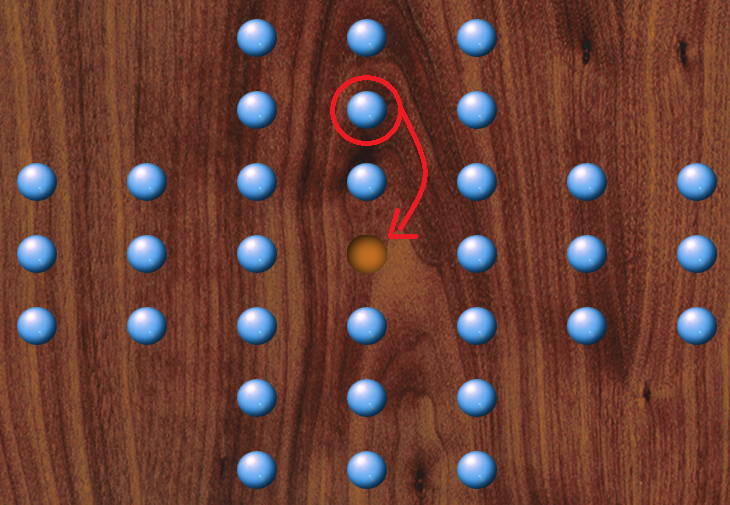


Bild - Förflyttning i Solitär

Fall nummer *två* kollar om *första värdet(det förväntade värdet)* stämmer överens med det *andra värdet(det faktiska värdet)*. I vårt fall så kollar vi om koordinat 24 inte har en kula på sig, alltså om det är *null*, vilket det är om eftersom vi flyttade bort kulan i fall ett.

På samma vis sker i fall tre och fyra där vi försöker göra otillåtna drag och eftersom vi använder assertFalse kommer även dessa två att gå igenom och returnera ett grönt resultat.

## Factory Pattern

En factory är en klass som endast skapar ett viss sorts objekt. Enligt Bilting [3] ska den som anropar factoryn kunna lita på att factoryn gör allt rätt. Den som anropar factoryn ska inte behöva veta om detaljerna som ligger i factoryn, enligt vanlig inkapslingsordning. För Solitär har en factory skapats för att skapa ett bräde. När denna factory anropas ska ett bräde skapas med boardLocations på, alltså olika bestämda positioner. Brädets olika positioner fylls med kulor. Istället för att låta SolitariGameStateskapa ett bräde valdes att låta en factory hantera detta och SolitariGameStatefår vid start av ett nytt spel anropa factoryn för att få ett bräde. Detta ger en kod som är lättare att läsa och förstå. Genom att flytta ut denna kod från SolitarGameStategör att den klassen får färre kodrader och slipper veta om detaljerna för hur ett bräde skapas. Enligt kodexempel nedan visas hur factoryn för att skapa Solitärs bräde ut.

|  |
| --- |
| **public void execute() {**  **this.board = new Board(createBoardLocations());**  **putAllTheBeadsOnTheBoard();**  **}**  **public Board getBoard() {**  **return board;**  **}** |

Källkod - Utdrag från SolitarBoardFactory

För Fia med knuff finns det också en factory, BoardAndPlayerFactory. Denna factory har fått ansvaret att skapa ett bräde med fia med knuff-positioner och fyra olika spelare med fyra pjäser var. Factoryn lägger till dessa spelare till det skapade brädet. Tidigare låg koden som nu ligger i factoryn denna kod i LudoGameState för Fia med knuff.

## Strategy Pattern

Vid införandet av fler och fler regler började koden att bli rörig och svårhanterlig. För att lösa detta så valdes att implementera en form utav Strategy Pattern. Istället för att använda enumerationer för att hålla koll på de olika tillstånd som ett drag kunde föra med sig så valdes att skapa klasser som implementerade gränssnittet MoveStrategy. Då kunde koden som utvärderade tillståndet separeras från den kod som faktiskt utförde själva förflyttningen. Detta gav en mycket mer lättläst kod samt att nu används ett väl beprövat designmönster som kommer att underlätta vid införandet av eventuella förändringar av reglerna. På bilden under kan man se ett tidigt stadie utav proposeMove()som sedan fick ytterligare tillskott av tillstånd och regler. Med hjälp av StrategyPattern[7] fick man lösningen som ni kan skåda på efterföljande bild.

|  |
| --- |
| @Override  **public** Boolean proposeMove(Move move) {      LudoMoveResult result = ruler.isValidMove(move);  **switch** (result) {  **case** *MOVE\_VALID*:  **if** (needToPush(move))              ruler.pushOtherPiece(move.getDestination().getPiece());          message = "";                 executeAndMakeSureThatNoPieceWillBeDeleted(move);          nextPlayer();  **return** **true**;  **case** *MOVE\_LAPSED*:          message = "Lapsed!";  **return** **false**;  **case** *MOVE\_NOGAMEPIECE*:          message = "No game piece located in source.";  **return** **false**;  **case** *MOVE\_INCORRECTNUMBEROFSTEPS*:         message = "You can't move to this position. Please try again.";  **return** **false**;  **case**       *MOVE\_IN\_BASE\_DID\_NOT\_GET\_THE\_CORRECT\_EYES\_ON\_THE\_DICE\_TO\_MOVE\_OUT*:         message = "You need to get 1 or 6 in order to move out of base.";         nextPlayer();  **return** **false**;  **case** *MOVE\_VALID\_INBASE\_TWO\_PIECES*:  **if** (needToPush(move))             ruler.pushOtherPiece(move.getDestination().getPiece());         message = "";         move.execute();                moveSecondPieceToStartPosition(move);                nextPlayer();  **return** **true**;  **case** *MOVE\_PIECE\_IN\_TO\_GOAL*:         move.execute();  **return** **false**;  **default**: {        message = "";  **return** **false**;     }     }   } |

Källkod - Innan införandet av Strategy

|  |
| --- |
| **public** Boolean proposeMove(Move move) {      MoveStrategy result = ruler.evaluateMove(move);  **return** result.execute(move, gameState);   } |

Källkod - Efter införandet av Strategy

## Observer Pattern

Målet med detta designmönster enligt ”Head First – Design Patterns” [7],är att bryta kopplingen mellan en datamodell och dess intressenter. Detta genomförs genom att man låter de objekt som vill ha en uppdatering vid förändring själva ansvara för att registrera sig. Detta ger ett flertal fördelar gentemot att låta objekten inneha en direkt länk mellan vy och modell. I projektet så förekommer detta på två ställen. Dels använder sig API’t av detta efter att en uppdatering har skett i GameState, då kommer funktionen notifyListernersOfMove()att kontaktas som pushar ut det nuvarande tillståndet hos GameState, samt informerar alla intressenter att en uppdatering har skett. Vid en uppdatering kommer metoden publish()kontaktas på de registrerade lyssnarna som sedan kan uppdatera sin grafiska presentation därefter.

I detta arbete valdes att implementera detta i klassen LudoGUIOutputunit.java som hanterar det grafiska som förmedlas ut till användaren. Istället för att hårdkodat in ett beroende till en klass som hanterar inmatning så valdes att låta dessa själva ansvara för att registrera sig. Genom denna enkla ändring så togs ett direkt beroende bort och där av fås en direkt mer flexibel lösning.

|  |
| --- |
| **public** **void** registerListener(ActionListener listener) {     inputUnits.add(listener);   } |

Källkod - LudoGUIOutputUnit

Registreringen av lyssnaren hanterades i klassen, LudoGUIIOFactory.java.

## Decorator

En decorator gör att man dynamiskt kan lägga till nya beteenden för en klass. Enligt Bilting [3]”ska dessa utökningar läggas till och tas bort vid olika tillfällen och förekomma i kombinationer. En användare av ett dekorerat objekt skall uppleva att objektet fungerar som förut och att utökningen är rena tillägg.”

I detta fall har en decorator används för att skapa en LudoPlayeristället för bara en player. Nedan visas koden för att pusha en annan pjäs för hur en del koden såg ut innan decoratorn skapades.

|  |
| --- |
| **public** **void** pushOtherPiece(GamePiece piece) {  String name = getPlayerName(piece);  **if** (name.equals("Red")) {  putInBase(LudoStaticValues.***REDHOME***, piece);  }  **else** **if** (name.equals("Blue")) {  putInBase(LudoStaticValues.***BLUEHOME***, piece);  } **else** **if** (name.equals("Yellow")) {  putInBase(LudoStaticValues.***YELLOWHOME***, piece);  }  **else** {  putInBase(LudoStaticValues.***GREENHOME***, piece);  }  } |

Källkod - Innan införandet av decorator i RuleControllerLudo

Nästa kodexemple visar hur koden ser ut efter skapandet av decoratorn, LudoPlayer. I den nya koden är det i metoden pushOtherPiece som ger tar emot en boardlocation.

|  |
| --- |
| **public** **void** pushOtherPiece(BoardLocation destination) {     GamePiece piece = destination.getPiece();     LudoPlayer player = getPlayerFromPiece(piece);     baseController.putInBase(player.getHomePositions(), piece);     destination.clear();  } |

Källkod - Efter införandet av decorator

## Model-View-Controller(MVC)

I detta projekt så var användandet av MVC mer eller mindre obligatoriskt då det API som arbetades emot nyttjade denna lösning. Tanken är att modellen inte behöver ha någon som helst kännedom om det som hanterar det grafiska eller inmatning ifrån användaren. Genom en sådan lösning så kan man enklare göra vyn flexibel och mer utbytbar mot andra lösningar [7].

Ett exempel på detta kan vara utifall det beslutades att en androidversion av Solitär skulle släppas, då skulle koden för modellen inte behöva ändras någonting i sin kärna, utan man kan bara fokusera på det som faktiskt skiljer sig mellan plattformarna.

# Resultat

För att enkelt kunna navigera mellan spelen så är GUI:t skapat så att det ger möjlighet att välja mellan de olika spelen, där antingen Runnerklassen för Solitär eller för Ludo kontaktas.

För att hålla beroendena så lågt som möjligt så skapas både instanser av Input- och OutputUnit som skickas in i Runnerobjektet.  
Nedan ser vi ett utdrag från Runnerklassen för Solitär.

|  |
| --- |
| **public** BoardGamesSolitarGUIRunner() {   mainPanel = **new** JPanel();   mainPanel.setBackground(Color.*BLACK*);   in = **new** SolitarGUIInputUnit();   out = **new** SolitarGUIOutputUnit(in);  **new** Runner(**new** SolitarGameState(), **new** SolitarGUIIOFactory(in, out))   .run();   setNewMidPanel(out);  **this**.add(mainPanel);  } |

Källkod - Runner för Solitär GUI

## Solitär

Ett av kraven var att ett spel skulle ha två olika gränssnitt, ett grafiskt och ett konsolbaserat. Här under kan man se exempel på vår konsolbaserade version av Solitär.

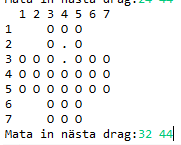


Bild Konsolbaserade gränssnittet för Solitär

För att representera en spelplan så finns det ett koordinatsystem från 1 till och med 7. För att representera om en position på brädet är fylld så är det markerat med en nolla, respektive “.“ utifall den är tom. För att kunna göra ett drag krävs det att man först matar in en position som man vill flytta ifrån, till den position som man vill flytta till. Exempelvis: “22 44” betyder - ta kulan på position 22 och flytta till position 44.

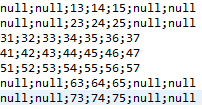


Bild Koordinaterna för Solitärs spelplan

Hela brädet kommer att läsas in via fil och ser ut enligt bild till höger. För att representera en tom ruta i ett rutnät så skriver man helt enkelt null. Denna lösning är vald utifall man vid senare tillfälle ville byta ut formationen på brädet.

Själva kärnan i Gameboard-API och det som är den absolut viktigaste komponenten är implementationen utav GameState. I fallet Solitär är det inte så komplext då man aldrig byter spelare, och inte heller någon tärning förekommer. Det som valts att göra för att hålla Gamestate så ren som möjligt är att flytta ut funktionaliteten för regler till en separat klass, kallad SolitarRuleController.java*.* Majoriteten av det som skapar spelet ligger i denna klass, med undantag för en metod kallad removeBeadInBetween().Denna metod är egentligen inte en del utav reglerna utan hanterar eftereffekten utav att ett drag genomförs och tar bort den kula som man valt att hoppa över.

Den metoden som är mest krävande och svårast att implementera är en metod som hanterade logiken för om spelet var över eller inte. Eftersom reglerna i Solitär är att man får spela så länge man har möjlighet att göra ett drag så krävdes en funktion som itererade hela brädet för att kontrollera om ett drag var möjligt. Om inga drag gick att genomföra så var spelet helt enkelt slut. Denna funktion testades genom att skicka in olika fall i Unittestningen för hasEnded().

Största skillnaden mellan den grafiska implementationen och konsolversionen var att både inputunit och outputunit är mer omfattande. Konsolversionen har en simpel inputunit som läser ifrån en BufferedReader. För att kunna presentera ett Move så har en hjälpmetod skapats, getBoardLocationFromCoordinate(). Denna metod tar emot en koordinat och returnerar en referens till motsvarande boardlocation. Dessa boardlocation används sedan för att presentera ett move.

I den grafiska versionen utav spelet så fungerar det i grunden likadant. InputUnitlyssnar på ett event som användaren genererar och lagrar denna referens i antingen sourceClickeller destionationClick beroende på om det var första gången användaren klickar. Utifall både sourceClickoch destinationClickhar fått ett värde kommer ett move att skapas på liknande vis som i konsolversionen.

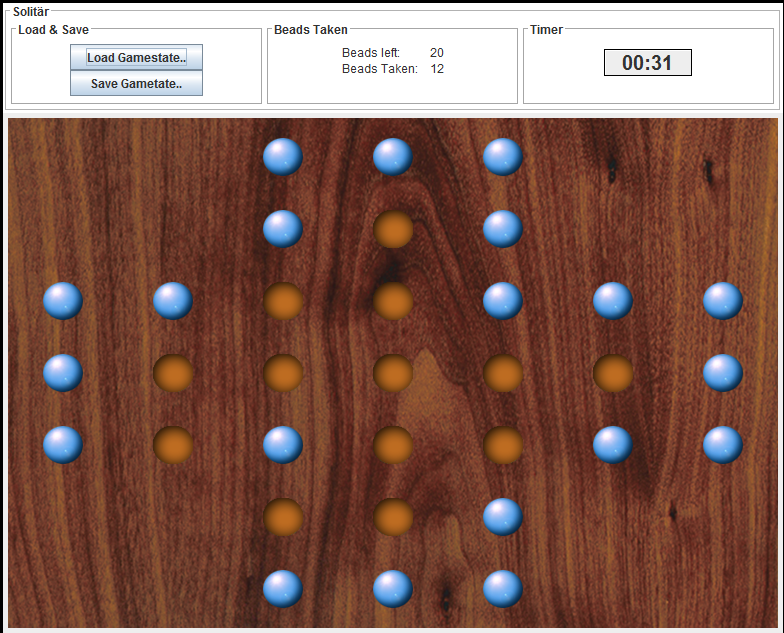


Bild - Solitärs grafiska gränssnitt

Även OutputUnitär väldigt lik konsolversionen i grunden, men blir både mer komplex och svårläst på grund utav användandet av Swing. OutputUnithar dock som grundtanke att presentera information för användaren, och låter InputUnitsjälv ta hand om händelsehantering, såsom att användaren klickat. InputUnitregistrerar sig själv som lyssnare i samband med att vi skapar spelet i BoardGamesLudoGUIRunner. För att ge spelaren lite ytterligare feedback har vi valt att inkludera en timer.

## Fia med knuff

Mycket av funktionaliteten ifrån Solitär kunde återbrukas här, och framförallt kunde vi ta med oss kunskaperna om API’t vi lärt oss ifrån Solitär. Vi kan se att InputUnit och OutputUnit är väldigt lika i grunden som hos Solitär, även om det är mer av allt. En mindre detalj som vi tvingades ändra ifrån Solitär var att koordinaterna inte kunde vara numrerade, utav vi fick istället använda oss av A-Z. Exempel på vår konsolversion kan ses på bilden nedan.

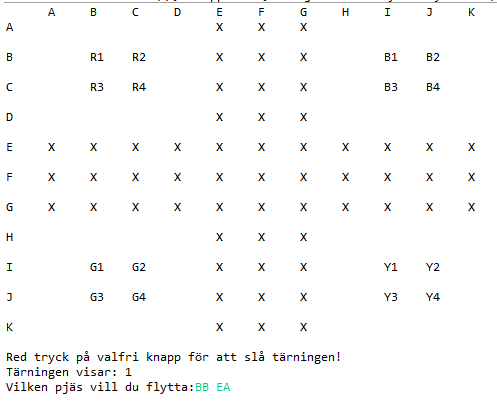


Bild - Konsolbaserade gränssnittet för Fia med knuff

    Den absolut största skillnaden mellan Solitär och Fia med Knuff var att vi istället för att läsa in brädet ifrån en fil som ett tvådimensionellt bräde så valde vi att istället läsa in den som en platt lista. Detta val gjorde vi eftersom vi då själva kan välja vägen som spelaren kommer att ta.

För att eliminera onödig kod där vi ständigt kontrollerar färg på pjäs och spelare så valde vi att istället knyta dessa data närmare spelarobjektet. Vi använde oss då av designmönstret decorator. Genom att skapa klassen LudoPlayer.java kunde vi knyta fält som var nära knutet till en viss spelare till rätt objekt. Istället för långa case/switch där vi kontrollerar en färg och utför ett kommando - kan vi nu helt enkelt säga t.ex. player.getFinishLineCoordinates(). Något som ytterligare underlättade vid implementeringen var att vi nu låter varje spelare ha sin personliga lista av boardlocations, det gör att vi kan ta bort all form av modulering utav listan samt att vi kunde radera alla kontroller såsom “Har spelaren gått ett varv?” och kunde behandla målrakan som vilken annan position som helst på brädet

Som tidigare nämnt så använde vi oss också utav designmönstret Strategy. Detta för att reglerna som implementerades fortsatte att göra koden mer och mer svårläst. Nu har vi separerat utförandet av ett drag med utvärderingen av ett drag. Det underlättar dessutom vid framtida ändringar eller ifall man vill addera ytterligare regler i framtiden.

    Här var vi även tvungna att lägga till funktionalitet för tärningen, vilket var väldigt enkelt då det stöds på API-nivå.

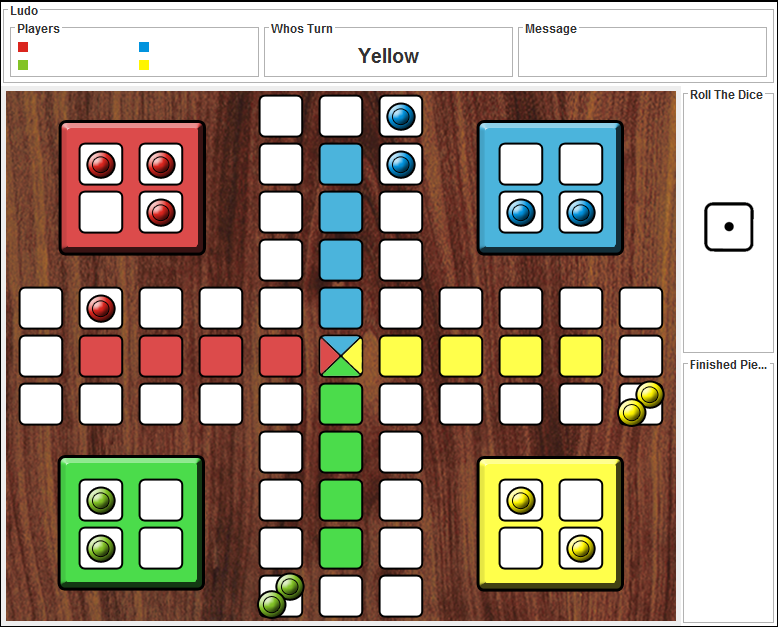


Bild - Fia med knuff grafiska gränssnitt

# Diskussion

## Att arbeta med Scrum

I början av projektet fick vi i uppgift att använda Scrum. Eftersom ingen av oss i projektgruppen hade någon större erfarenhet av Scrum fick vi till en början prova oss fram. När vi i början jobbade med sprintarna hade vi svårt för att bryta ner och tidsätta delarna av arbete. Genom att vid varje arbetsstund ägna en liten del av tiden på att tänka igenom vad som hade gjorts, hur lång tid det tog och vad som borde göras härnäst gjorde att vi började komma in i arbetssättet Scrum lite mer för varje gång. I slutet av projektet har sprintarna blivit längre, mer specifika för vad som ska göras och det har gått bättre med tidsuppskattningen.

## Att arbeta med refactoring

Att arbeta med refactoring har visat för oss vilken skillnad det kan bli på koden efter en refactoring. Då vi lade till designmönstret Strategy som beskrivs under metod minskade antalet kodrader märkbart. Vi har även fått en bättre förståelse för användandet och betydelsen av flera olika designmönster.

## Att arbeta med Unittester

Att arbeta med tester är något som är väldigt nytt för oss och är den del som kommer ta ett tag att vänja sig vid. Omställningen till att först skriva tester för att sedan implementera funktionalitet känns just nu inte naturligt. Vi har dock insett fördelen med att göra klasserna testbara eftersom det för med sig en lösare koppling mellan de olika klasserna. Något som vi bör ha implementerat i ett tidigt stadie är en funktion för att generera tester utifrån brädets tillstånd. Detta hade underlättat väldigt mycket för att åtgärda buggar. Nu tvingades vi försöka återskapa brädet manuellt vilket var både tidskrävande och mödosamt.

## Förbättringar av arbetet

Vi bör som tidigare nämnt ha implementerat en funktion för att spara ner ett brädes tillstånd för mer precisa tester. Detta är dock något som borde gjorts vid ett tidigare stadie av arbetet. Vi känner också att vi borde ha gett testdriven utveckling ett större fokus genom projektet. Vi känner att mycket tid gick åt för att manuellt testa om en diverse buggar var lösta, eller om en funktion/funktioner var implementerad korrekt. Detta hade kunnat ske automatiskt via tester och på så vis hade vi kunnat fokusera på mer tongivande delar av projektet.

## Fortsatt arbete

Om vi kontrollerar den väldigt grova backlog som vi satte upp innan projektet kan man se att vi har en funktion som inte blivit implementerad, nämligen multiplayer för Fia med Knuff. Något som vi känner hade kunnat förbättra spelupplevelsen hade varit en funktion för att spara/ladda tillståndet av spelet. Vi diskuterade dock detta vid ett tidigare stadie under projektet och kom fram till att detta är en funktion som bör erbjudas på API-nivå.

## Vad var bra/dåligt med projektet?

Något vi känner har påverkat dels hur vi har arbetat och troligtvis också vår slutprodukt är att vi tillskillnad från de andra grupperna, endast var tre medlemmar. Detta har fört med sig att vi troligtvis arbetat tätare än om vi varit flera. Självklart hade en person till antagligen varit till en fördel på så vis att vi får ytterligare en person som kunnat bidra. Då hade det även varit enklare att utnyttja oss av par programmering. Det hade även tvingat oss till att ha hårdare fokus på Scrum.

## Vad tyckte vi om detta arbete?

Avslutningsvis så tycker vi att det har var lärorikt att jobba mot ett *API*. Samtidigt som det har varit kul att vara med att påverka dess funktionalitet. Däremot tyckte några av oss att det var tråkigt med brädspel, även om vi förstår att själva slutprodukten inte var det väsentliga i projektet. Vi känner att om ett spel nödvändigtvis skulle ha utvecklats så vore det roligare att jobba med ett redan etablerat API, såsom Unityeller libGDX.

# Referenser

|  |  |
| --- | --- |
| [1] | ”Spelregler,” [Online]. Available: http://www.spelregler.org/fia-med-knuff/. [Använd 2014]. |
| [2] | ”Alga spel,” Alga, [Online]. Available: http://www.algaspel.se/~/media/Alga/Files/Rules/se/Solitar\_SE.ashx. [Använd 2014]. |
| [3] | U. Bilting, Desingmönster för programmerare, Malmö: Studentlitteratur AB, 2011. |
| [4] | R. C. Martin, in *Clean Code*, Boston, MA, Prentice Hall, Pearson Education, 2012. |
| [5] | M. Cohn, ”Mountain Goat Software,” [Online]. Available: http://www.mountaingoatsoftware.com/agile/scrum/daily-scrum/. [Använd 2014]. |
| [6] | ”JUnit,” [Online]. Available: http://junit.org/. [Använd 2014]. |
| [7] | E. R. B. B. K. S. Eric Freeman, Head First Design Patterns, O'Reilly Media, 2004. |
| [8] | ”Github,” [Online]. Available: https://github.com/. [Använd 2014]. |

# Bilagor

#### Källkod 1 – SolitairGameStateTest.java

|  |
| --- |
| **public** **class** SolitairGameStateTest {  SolitarGameState gameState;  @Before  **public** **void** setup() {  gameState = **new** SolitarGameState();  }  @Test  **public** **void** proposeMoveTest() {  *assertTrue*(gameState.proposeMove(**new** Move(gameState.getPlayerInTurn(),  SolitarHelpMethods.*getBoardLocationFromCoordinate*("24",  gameState.getBoard()), SolitarHelpMethods  .*getBoardLocationFromCoordinate*("44",  gameState.getBoard()))));  *assertEquals*(  **null**,  SolitarHelpMethods.*getBoardLocationFromCoordinate*("24",  gameState.getBoard()).getPiece());  *assertFalse*(gameState.proposeMove(**new** Move(gameState.getPlayerInTurn(),  SolitarHelpMethods.*getBoardLocationFromCoordinate*("24",  gameState.getBoard()), SolitarHelpMethods  .*getBoardLocationFromCoordinate*("34",  gameState.getBoard()))));  *assertFalse*(gameState.proposeMove(**new** Move(gameState.getPlayerInTurn(),  SolitarHelpMethods.*getBoardLocationFromCoordinate*("14",  gameState.getBoard()), SolitarHelpMethods  .*getBoardLocationFromCoordinate*("34",  gameState.getBoard()))));  }  @Test  **public** **void** hasEnded() {  *assertFalse*(gameState.hasEnded());  **for** (BoardLocation b : gameState.getBoard().getLocations()) {  **if** (b != **null**)  b.setPiece(**null**);  }  *assertTrue*(gameState.hasEnded());  **int** index = 0;  **for** (BoardLocation b : gameState.getBoard().getLocations()) {  **if** (b != **null**) {  **if** (index % 2 == 0)  b.setPiece(**null**);  **else**  b.setPiece(**new** GamePiece("O"));  }  }  *assertTrue*(gameState.hasEnded());  **for** (BoardLocation board : gameState.getBoard().getLocations()) {  **if** (board != **null**)  board.setPiece(**null**);  }  gameState.getBoard().getLocations().get(4).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(15)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(20)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(23)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(27)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(32)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(34)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(46)  .setPiece(**new** GamePiece("O"));  *assertTrue*(gameState.hasEnded());  **for** (BoardLocation board : gameState.getBoard().getLocations()) {  **if** (board != **null**)  board.setPiece(**null**);  }  gameState.getBoard().getLocations().get(2).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(14)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(17)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(20)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(25)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(27)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(30)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(33)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(34)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(44)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(45)  .setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(46)  .setPiece(**new** GamePiece("O"));  *assertFalse*(gameState.hasEnded());  **for** (BoardLocation board : gameState.getBoard().getLocations()) {  **if** (board != **null**)  board.setPiece(**null**);  }      gameState.getBoard().getLocations().get(16).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(17).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(18).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(19).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(20).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(25).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(26).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(27).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(28).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(32).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(33).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(34).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(38).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(39).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(44).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(45).setPiece(**new** GamePiece("O"));  gameState.getBoard().getLocations().get(46).setPiece(**new** GamePiece("O"));    *assertFalse*(gameState.hasEnded());  }  } |

#### Källkod 2 – SolitarBoardFactory.java

|  |
| --- |
| **public** **class** SolitarBoardFactory {  Board board;  **public** **void** execute() {  **this**.board = **new** Board(createBoardLocations());  putAllTheBeadsOnTheBoard();  }  **public** Board getBoard() {  **return** board;  }  **private** **void** putAllTheBeadsOnTheBoard() {  **for** (BoardLocation b : board.getLocations()) {  **if** (b != **null**)  b.setPiece(**new** GamePiece("O"));  }  board.getLocations().get(24).setPiece(**null**);  }  **private** List<BoardLocation> createBoardLocations() {  List<BoardLocation> boardLocations = **new** ArrayList<BoardLocation>();  List<String> listOfRows = **null**;  **try** {  listOfRows = SolitarFileHandler.*readCoordinate*();  } **catch** (IOException e) {  e.printStackTrace();  }  **while** (!listOfRows.isEmpty()) {  boardLocations.addAll(getBoardLocationFromRow(listOfRows.get(0)));  listOfRows.remove(0);  }  **return** boardLocations;  }  **private** List<BoardLocation> getBoardLocationFromRow(String row) {  String[] locations = row.split(";");  List<BoardLocation> boardLocations = **new** ArrayList<BoardLocation>();  **for** (String s : locations) {  **if** (s.equals("null"))  boardLocations.add(**null**);  **else**  boardLocations.add(**new** BoardLocation(s));  }  **return** boardLocations;  }  } |

#### Källkod 4 – LudoMoveController.java

|  |
| --- |
| **public** **class** LudoMoveController {  LudoGameState gameState;  LudoRuleController ruler;  MoveValidImplementation moveValidExec;      **public** LudoMoveController(LudoGameState gameState) {  **this**.gameState = gameState;  ruler = **new** LudoRuleController(gameState);  moveValidExec = **new** MoveValidImplementation(LudoStaticValues.***GOAL***, ruler);  }  **public** Boolean proposeMove(Move move) {  MoveStrategy result = ruler.evaluateMove(move);  **return** result.execute(move, gameState);  }  } |

#### Källkod 5 – LudoGUIOutputUnit.java

|  |
| --- |
| **public** **class** LudoGUIOutputUnit **extends** JPanel **implements** OutputUnit {  **private** BackGroundLabelLudo backgroundLabel;  **private** FinishPopUpWindow finishWindow;  **private** LudoDiceChooser diceB;  **private** String redURL = "src\\boardgames\\img\\red.jpg";  **private** String blueURL = "src\\boardgames\\img\\blue.jpg";  **private** String yellowURL = "src\\boardgames\\img\\yellow.jpg";  **private** String greenURL = "src\\boardgames\\img\\green.jpg";  **private** String winnerRedIconURL = "src\\boardgames\\img\\redWINNER.png";  **private** String winnerBlueIconURL = "src\\boardgames\\img\\blueWINNER.png";  **private** String winnerYellowIconURL = "src\\boardgames\\img\\yellowWINNER.png";  **private** String winnerGreenIconURL = "src\\boardgames\\img\\greenWINNER.png";  **private** List<ActionListener> inputUnits;  **private** ImageIcon winnerIcon;  **private** BackGroundButtonIDLudo button[][];  **private** ButtonGroup buttonGroup;  **private** JCheckBox redC, blueC, yellowC, greenC;  **private** JPanel midPanel, topPanel, topPanelPlayers, eastPanelFinished,  topPanelWhoPlay, topPanelMessage, eastPanel, eastPanelDice, easttopFinishPanel, eastbottomFinishPanel;  **private** JLabel eastbottomWinnerLabel;  **private** JTextField textFieldWhosTurn, winnerTextField;  **private** JTextArea textAreaMessage;  **private** String player1, player2, player3, player4;  **private** **int** players;  **public** LudoGUIOutputUnit() {  createComponent();  // howManyPlayerAndSetName();  settingUpComponents();  }  **private** **void** createComponent() {  inputUnits = **new** ArrayList<ActionListener>();  backgroundLabel = **new** BackGroundLabelLudo(LudoStaticValues.***ROWS***,  LudoStaticValues.***COLS***);  winnerIcon = **new** ImageIcon();  textFieldWhosTurn = **new** JTextField();  textAreaMessage = **new** JTextArea();  winnerTextField = **new** JTextField();  topPanel = **new** JPanel(**new** GridLayout(1, 4));  topPanelPlayers = **new** JPanel(**new** GridLayout(2, 2));  topPanelWhoPlay = **new** JPanel(**new** GridLayout(1, 1));  topPanelMessage = **new** JPanel(**new** GridLayout(1, 1));  eastPanel = **new** JPanel(**new** GridLayout(2, 1));  eastPanelDice = **new** JPanel(**new** GridLayout(1, 1));  eastPanelFinished = **new** JPanel(**new** GridLayout(4, 4));  easttopFinishPanel = **new** JPanel();  eastbottomFinishPanel = **new** JPanel();  eastbottomWinnerLabel = **new** JLabel();  midPanel = **new** JPanel();  buttonGroup = **new** ButtonGroup();  redC = **new** JCheckBox();  blueC = **new** JCheckBox();  yellowC = **new** JCheckBox();  greenC = **new** JCheckBox();  }  **private** **void** howManyPlayerAndSetName() {  players = Integer.*parseInt*(JOptionPane.*showInputDialog*(**null**,  "How many players? (2-4)"));  **if** (players >= 1 && players <= 4) {  **if** (players == 2) {  player1 = JOptionPane.*showInputDialog*("Player 1 (Red) Name:");  player2 = JOptionPane.*showInputDialog*("Player 2 (Blue) Name:");  } **else** **if** (players == 3) {  player1 = JOptionPane.*showInputDialog*("Player 1 (Red) Name:");  player2 = JOptionPane.*showInputDialog*("Player 2 (Blue) Name:");  player3 = JOptionPane.*showInputDialog*("Player 3 (Green) Name:");  } **else** **if** (players == 4) {  player1 = JOptionPane.*showInputDialog*("Player 1 (Red) Name:");  player2 = JOptionPane.*showInputDialog*("Player 2 (Blue) Name:");  player3 = JOptionPane.*showInputDialog*("Player 3 (Green) Name:");  player4 = JOptionPane  .*showInputDialog*("Player 4 (Yellow) Name:");  }  } **else**  JOptionPane.*showMessageDialog*(**null**, "Wrong amount of players!!!");  }  **private** **void** settingUpComponents() {  redC.setIcon(**new** ImageIcon(redURL));  blueC.setIcon(**new** ImageIcon(blueURL));  greenC.setIcon(**new** ImageIcon(greenURL));  yellowC.setIcon(**new** ImageIcon(yellowURL));  redC.setOpaque(**false**);  blueC.setOpaque(**false**);  greenC.setOpaque(**false**);  yellowC.setOpaque(**false**);  redC.setText(player1);  blueC.setText(player2);  greenC.setText(player3);  yellowC.setText(player4);  buttonGroup.add(redC);  buttonGroup.add(blueC);  buttonGroup.add(greenC);  buttonGroup.add(yellowC);  textFieldWhosTurn.setFocusable(**false**);  textFieldWhosTurn.setEditable(**false**);  textFieldWhosTurn.setOpaque(**false**);  textFieldWhosTurn.setBorder(BorderFactory.*createEmptyBorder*());  textFieldWhosTurn.setHorizontalAlignment(JLabel.***CENTER***);  textFieldWhosTurn.setFont(**new** Font("Arial", Font.***CENTER\_BASELINE***, 20));  winnerTextField.setFocusable(**false**);  winnerTextField.setEditable(**false**);  winnerTextField.setOpaque(**false**);  winnerTextField.setBorder(BorderFactory.*createEmptyBorder*());  winnerTextField.setHorizontalAlignment(JLabel.***CENTER***);  winnerTextField.setFont(**new** Font("Arial", Font.***CENTER\_BASELINE***, 15));  textAreaMessage.setFocusable(**false**);  textAreaMessage.setEditable(**false**);  textAreaMessage.setOpaque(**false**);  textAreaMessage.setLineWrap(**true**);  textAreaMessage.setWrapStyleWord(**true**);  textAreaMessage.setFont(**new** Font("Arial", Font.***CENTER\_BASELINE***, 10));  topPanel.setBorder(BorderFactory.*createTitledBorder*(  BorderFactory.*createEtchedBorder*(), "Ludo", TitledBorder.***LEFT***,  TitledBorder.***TOP***));  topPanel.setBackground(Color.***WHITE***);  topPanelPlayers.setBorder(BorderFactory.*createTitledBorder*(  BorderFactory.*createEtchedBorder*(), "Players",  TitledBorder.***LEFT***, TitledBorder.***TOP***));  topPanelPlayers.setBackground(Color.***WHITE***);  topPanelPlayers.add(redC);  topPanelPlayers.add(blueC);  topPanelPlayers.add(greenC);  topPanelPlayers.add(yellowC);  topPanelWhoPlay.setBorder(BorderFactory.*createTitledBorder*(  BorderFactory.*createEtchedBorder*(), "Whos Turn",  TitledBorder.***LEFT***, TitledBorder.***TOP***));  topPanelWhoPlay.add(textFieldWhosTurn);  topPanelWhoPlay.setBackground(Color.***WHITE***);  topPanelMessage.setBorder(BorderFactory.*createTitledBorder*(  BorderFactory.*createEtchedBorder*(), "Message",  TitledBorder.***LEFT***, TitledBorder.***TOP***));  topPanelMessage.add(textAreaMessage);  topPanelMessage.setBackground(Color.***WHITE***);  eastPanelDice.setBorder(BorderFactory.*createTitledBorder*(  BorderFactory.*createEtchedBorder*(), "Roll The Dice",  TitledBorder.***LEFT***, TitledBorder.***TOP***));  eastPanelDice.setBackground(Color.***WHITE***);  eastPanelFinished.setBorder(BorderFactory.*createTitledBorder*(  BorderFactory.*createEtchedBorder*(), "Finished Pieces",  TitledBorder.***LEFT***, TitledBorder.***TOP***));  eastPanelFinished.setBackground(Color.***WHITE***);  easttopFinishPanel.setOpaque(**false**);  eastbottomFinishPanel.setOpaque(**false**);  easttopFinishPanel.add(winnerTextField);  eastbottomFinishPanel.add(eastbottomWinnerLabel);  eastPanelFinished.setLayout(**new** BorderLayout());  eastPanelFinished.add(easttopFinishPanel, BorderLayout.***NORTH***);  eastPanelFinished.add(eastbottomFinishPanel, BorderLayout.***SOUTH***);  topPanel.add(topPanelPlayers);  topPanel.add(topPanelWhoPlay);  topPanel.add(topPanelMessage);  eastPanel.add(eastPanelDice);  eastPanel.add(eastPanelFinished);  midPanel.add(backgroundLabel);  **this**.setLayout(**new** BorderLayout());  **this**.add(topPanel, BorderLayout.***NORTH***);  **this**.add(midPanel, BorderLayout.***CENTER***);  **this**.add(eastPanel, BorderLayout.***EAST***);  }  @Override  **public** **void** publish(GameState gameState) {  **char** cordRow = 'A';  **char** cordCol = 'A';  backgroundLabel.removeAll();  eastPanelDice.removeAll();  topPanelMessage.removeAll();  textFieldWhosTurn.setText(gameState.getPlayerInTurn().getName());  textAreaMessage.setText(gameState.getMessage());  button = **new** BackGroundButtonIDLudo[LudoStaticValues.***ROWS***][LudoStaticValues.***COLS***];  diceB = **new** LudoDiceChooser(gameState);  **for** (**int** rows = 0; rows < LudoStaticValues.***ROWS***; rows++) {  **for** (**int** cols = 0; cols < LudoStaticValues.***COLS***; cols++) {  String coordinate = Character.*toString*(cordRow)  + Character.*toString*(cordCol++);  BoardLocation location = LudoHelpMethods  .*getBoardLocationFromCoordinate*(coordinate,  gameState.getBoard());  button[rows][cols] = **new** BackGroundButtonIDLudo(location,  coordinate);    **for**(ActionListener listener : inputUnits){  button[rows][cols].addActionListener(listener);  }  diceB.addActionListener(diceB);  eastPanelDice.add(diceB);  diceB.addActionListener(diceB);  eastPanelDice.add(diceB);  topPanelMessage.add(textAreaMessage);  easttopFinishPanel.add(winnerTextField);  textAreaMessage.repaint();  backgroundLabel.add(button[rows][cols]);  getWinner(gameState);  }  cordCol = 'A';  cordRow++;  }  **if** (gameState.hasEnded()) {  finishWindow = **new** FinishPopUpWindow();  finishWindow.displayWindowLudo();  **if** (finishWindow.getReturnValue() == 1) {  gameState.reset();  publish(gameState);  }  }  backgroundLabel.revalidate();  eastPanelDice.revalidate();  eastPanelFinished.revalidate();  topPanelMessage.revalidate();  }  **private** **void** getWinner(GameState gameState) {  **if**(gameState.getWinner() != **null**){  String winner = gameState.getWinner().getName();  **switch**(winner){  **case** LudoStaticValues.***REDPLAYER*** :  winnerIcon = **new** ImageIcon(winnerRedIconURL);  **break**;  **case** LudoStaticValues.***BLUEPLAYER*** :  winnerIcon = **new** ImageIcon(winnerBlueIconURL);  **break**;  **case** LudoStaticValues.***YELLOWPLAYER*** :  winnerIcon = **new** ImageIcon(winnerYellowIconURL);  **break**;  **case** LudoStaticValues.***GREENPLAYER*** :  winnerIcon = **new** ImageIcon(winnerGreenIconURL);  **break**;  }  winnerTextField.setText(winner);  winnerTextField.repaint();  eastbottomWinnerLabel.setIcon(winnerIcon);  }  }  **public** **void** registerListener(ActionListener listener){  inputUnits.add(listener);  }  } |

#### Källkod 7 – LudoRuleController.java

|  |
| --- |
| **public** **class** LudoRuleController {  LudoGameState state;  LudoBaseController baseController;  **public** LudoRuleController(LudoGameState state) {  **this**.state = state;  baseController = **new** LudoBaseController(state, **this**);  }  **public** MoveStrategy checkAndReturnValidMoves(Move move) {  BoardLocation source = move.getSource();  **if** (source.getPiece() == **null**)  **return** **new** MoveNoGamePieceImplementation();  **if** (baseController.checkIfPieceInbase(move))  **return** baseController.checkValidMoveFromBase(move);  **if** (checkIfPlayerStepsIsNotCorrect(move))  **return** **new** MoveIncorrectStepsImplementation();  **if** (playerIsTryingToLapseHisOwnPiece(move, getNumberOfStepsFromDice()))  **return** **new** MoveLapseOwnPieceImplementation();  **if** (playerIsTryingToLapseAblock(move, getNumberOfStepsFromDice()))  **return** **new** MoveInvalidBlockImplementation();  **if** (destinationIsAlreadyOccupado(move)) {  **return** **new** MoveInvalidBoardLocationAlreadyOccupied();  }  **else** {  **return** **new** MoveValidImplementation(LudoStaticValues.***GOAL***, **this**);  }  }  **public** MoveStrategy evaluateMove(Move move) {  **if** (!LudoHelpMethods.*doesPlayerHaveAnyPiecesOnTheBoard*(  move.getPlayer(), state.getBoard())) {  **if** (!checkIfDiceIsSIXorONE()) {  **return** **new** MoveInvalidBaseCantMoveOut();  }  }  **if** (playerCantMakeAMove(move))  **return** **new** MoveNoMovesAvailableImplementation();  **return** checkAndReturnValidMoves(move);  }  **private** **boolean** playerCantMakeAMove(Move move) {  List<GamePiece> playersPieces = move.getPlayer().getPieces();  LudoPlayer player = state.getLudoPlayerFromPlayer(move.getPlayer());  **if** (baseController.canPlayerMakeAMoveFromBase(player))  **return** **false**;  **if** (canAnyPieceMakeAMoveOnouterBoardArea(playersPieces, player))  **return** **false**;  **return** **true**;  }  **private** **boolean** canAnyPieceMakeAMoveOnouterBoardArea(  List<GamePiece> playersPieces, LudoPlayer player) {  **for** (GamePiece piece : playersPieces)  **if** (canPieceMove(piece, player))  **return** **true**;  **return** **false**;  }  **private** **boolean** canPieceMove(GamePiece piece, LudoPlayer player) {  BoardLocation source = LudoHelpMethods.*getBoardLocationFromPiece*(piece,  state.getBoard());  BoardLocation destination;  List<BoardLocation> playerListOfBoardLocations = player.getBoardList();  **int** sourceIndex = playerListOfBoardLocations.indexOf(source);  **int** destinationIndex = sourceIndex + getNumberOfStepsFromDice();  **if** (destinationIndex >= playerListOfBoardLocations.size())  **return** **false**;  destination = playerListOfBoardLocations.get(destinationIndex);  **if** (source == **null** || destination == **null**)  **return** **false**;  Move move = **new** Move(player.getPlayerObject(), source, destination);  **if** (checkIfValidResult(move))  **return** **true**;  **return** **false**;  }  **private** **boolean** checkIfValidResult(Move move) {  **return** checkLudoMoveResultsForValidMove(move);  }  **private** **boolean** checkLudoMoveResultsForValidMove(Move move) {  **return** checkAndReturnValidMoves(move) **instanceof** MoveValidImplementation;  }  **private** **boolean** destinationIsAlreadyOccupado(Move move) {  **if** (move.getDestination().getPieces().size() > 1)  **return** **true**;  **else**  **return** **false**;  }  **private** **boolean** playerIsTryingToLapseAblock(Move move, **int** dice) {  LudoPlayer player = state.getLudoPlayerFromPlayer(move.getPlayer());  **int** sourceIndex = player.getBoardList().indexOf(move.getSource());  **for** (**int** i = 1; i < dice; i++) {  BoardLocation destination = player.getBoardList().get(  sourceIndex + i);  **if** (destination.getPieces().size() > 1)  **return** **true**;  }  **return** **false**;  }  **private** **boolean** playerIsTryingToLapseHisOwnPiece(Move move, **int** dice) {  LudoPlayer player = state.getLudoPlayerFromPlayer(move.getPlayer());  **int** sourceIndex = player.getBoardList().indexOf(move.getSource());  **for** (**int** i = 1; i < dice; i++) {  BoardLocation destination = player.getBoardList().get(  sourceIndex + i);  **if** (move.getPlayer().hasPiece(destination.getPiece()))  **return** **true**;  }  **return** **false**;  }  **private** **boolean** checkIfDiceIsSIXorONE() {  **int** dice = state.getDieRollFactory().getLastRoll().getResult();  **if** (dice == 6 || dice == 1)  **return** **true**;  **else**  **return** **false**;  }  **private** **boolean** checkIfPlayerStepsIsNotCorrect(Move move) {  **return** stepsPlayerMoves(move) != getNumberOfStepsFromDice();  }  **private** **int** stepsPlayerMoves(Move move) {  LudoPlayer player = state.getLudoPlayerFromPlayer(move.getPlayer());  List<BoardLocation> playersList = player.getBoardList();  **int** destinationValue = playersList.indexOf(move.getDestination());  **int** sourceValue = playersList.indexOf(move.getSource());  **int** stepsPlayerMoves = destinationValue - sourceValue;  **return** stepsPlayerMoves;  }  **public** **void** pushOtherPiece(BoardLocation destination) {  GamePiece piece = destination.getPiece();  LudoPlayer player = getPlayerFromPiece(piece);  baseController.putInBase(player.getHomePositions(), piece);  destination.clear();  }  **private** LudoPlayer getPlayerFromPiece(GamePiece pieceToPush) {  **for** (Player p : state.getPlayers()) {  **if** (p.getPieces().contains(pieceToPush)) {  **return** state.getLudoPlayerFromPlayer(p);  }  }  **return** **null**;  }  **public** **boolean** needToPush(Move move) {  GamePiece destinationPiece = move.getDestination().getPiece();  **if** (move.getPlayer().hasPiece(destinationPiece))  **return** **false**;  **else**  **return** move.getDestination().getPiece() != **null**;  }  **private** **int** getNumberOfStepsFromDice() {  **return** state.getDieRollFactory().getLastRoll().getResult();  }  } |

#### Backlog

|  |
| --- |
|  |