TDA550 – Objektorienterad programvaruutveckling IT, forts. kurs
Laboration 3 – Refaktorering av spelramverket

Pelle Evensen

22 november 2011

Sammanfattning

I den här laborationen ska vi identifiera och i flera fall åtgärda tillkortakommanden i det spelramverk ni använde i laboration 1.

1 Allmänt

I strikt bemärkelse ska vi inte bara ägna åt oss *refaktorering*[1], då vi kan komma att ta bort vissa beteenden, metoder eller variabler som fanns i det gamla ramverket. Det gamla ramverket ska kunna simuleras med hjälp av det nya, givet att man skapar lämpliga hjälpklasser. Vi ska också använda några kända designmönster.

Vi ska punkt för punkt diskutera specifika problem med det gamla ramverket och stegvis försöka åtgärda några av dem.

Klasserna som ingår och berörs är:

CompositeTile	GameOverException	Main
Constants	GameTile	Position
CrossTile	GameView	RectangularTile
GameController	GoldModel	ReversiModel
GameFactory	GUIView	RoundTile
GameModel	IGameFactory	SquareTile

2 Implementation av ett nytt spel

För att få något konkret och ganska annorlunda jämfört med Snake-spelet ska vi använda Othello/Reversi[2] som första modell. Detta kommer mer konkret påvisa några av begränsningarna i det befintliga ramverket. Spellogiken för Reversi ligger i paketet `orig2011.v0`, klassen **ReversiModel**.

[1] Wikipedia ger följande definition av "refactoring": "Refactoring is the process of changing a computer program's source code without modifying its external functional behavior in order to improve some of the nonfunctional attributes of the software. Advantages include improved code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility."

[2] `http://en.wikipedia.org/wiki/Reversi`

3 Tips om tillvägagångssätt

3.1 Skapa ett nytt paket för varje deluppgift

För varje deluppgift, skapa ett nytt paket. Om något går fel så har ni då hela tiden en bra version att falla tillbaks på. När ni ändrar gör ni det *bara* i det nya paketet.

3.2 Programmet ska gå att köra

Efter ni genomfört varje deluppgift ska programmet ha (minst) den funktionalitet det hade innan ni löste uppgiften. Förvissa er om att programmet går att köra och beter sig som det ska innan ni påbörjar nästa deluppgift.

3.3 Att lägga till nya spel

För att lägga till spelet Reversi räcker det att lägga till en ny **Main**-klass och en ny implementation av en lämplig **GameFactory** (som ju implementerar **IGameFactory**).

3.4 Deluppgift 1

Det paket ni fått att utgå ifrån heter `orig2011.v0`. Skapa ett nytt paket, `orig2011.v1`. Markera klasserna **Main** och **GameFactory** i paketet `orig2011.v0` i Eclipse, välj ”Copy”. Högerklicka på det nya paketet `orig2011.v1` i er src-katalog i projektet och välj ”Paste”. Nu ska ni fått en kopia av just dessa klasser till det nya paketet. Kopiera *inte* direkt via filsystemet, då kommer ni behöva byta paketnamn på alla klasser manuellt.
Byt namn på **GameFactory** till **ReversiFactory** (i ert nya paket, `orig2011.v1`). Modifiera **ReversiFactory** så att det går att spela både Gold och Reversi. Vill ni lägga till Snake från lab 1 får ni givetvis göra det men då måste ni också genomföra de förändringar som så småningom kommer krävas även i Snake. Ett enklare alternativ är att lägga till Snake när ni är klara med hela labben. *Detta är dock frivilligt.*

Berörda klasser/interface: **Main** samt **ReversiFactory** (ny klass).

4 Abstrakta klasser kontra interface

Fördelen med abstrakta klasser i stället för interface är som bekant att man kan tillhandahålla en viss grundimplementation och sedan bara implementera de delar som varierar. En av nackdelarana (i alla fall i Java) är att man bara kan ärva

implementation från en klass åt gången (som i sin tur kan ärva från en annan klass, o.s.v.). Detta medför i sin tur att vi *tvingas* återanvända kod, även om den inte passar särskilt bra för uppgiften. I lab 2, om geometriska former, såg vi hur man kan komma ifrån denna begränsning, interfacet **GeometricalForm** implementerades av en abstrakt klass vilka de konkreta klasserna **Line**, **Circle**, et.c. ärvde från. Detta gör det möjligt att välja hur vår implementation ska se ut; endast *om* vi har nytta av implementationen som tillhandahålles av den abstrakta klassen använder vi den.

Studera klassen **ReversiModel**. Denna klass tvingas att använda det befintliga spelbrädet på ett mycket konstlat sätt. Mer specifikt är inte ett fält av fält av **GameTile** något bra sätt att hålla reda på tillståndet på brädet då vi ska utföra beräkningar eller förändringar. Det fungerar däremot bra för *presentation* av spelet.

Betydligt bättre är att istället bara använda instansvariabeln **ReversiModel**.`board`. Vad anrop till `getGameBoardState` resulterar i ska kunna härledas från instansvariablerna i **ReversiModel** som representerar:

- brädet (`board`)
- markörpositionen (`cursorPos`)
- vems tur det är (`turn`)

Ingen annan lagring av brädet får ske än den som redan finns i dessa tre variabler. Detta inkluderar eventuella variabler i superklassen (**GameModel**). Som **ReversiModel** ser ut nu uppdateras både `board` och det ärvda spelbrädet (från **GameModel**). Detta är dålig stil; när vi duplicerar data blir det svårare att hålla dem i fas, all logik som rör brädet måste dubbleras.

I första hand bör man försöka härleda/beräkna värden från befintlig information snarare än att lagra den. Se gärna riktlinje på sid. 91 i [Skr09]. Undantag från denna princip kan göras om klassen är icke-muterbar *och* beräkningarna utgör ett mätbart prestandaproblem.

4.1 Deluppgift 2

Bygg om ramverket så att **GameModel** blir ett interface i stället. Den (mycket lilla) funktionalitet den befintliga *klassen* **GameModel** tillhandahåller bör man lägga över i en hjälpklass, lämpligt namn kan vara **GameUtils**. Klassen **GameUtils** får inte ha något tillstånd (instans-/klassvariabler). Metoderna ska direkt manipulera objekt av typen **GameTile**[][].
Om **GameModel** ska vara ett interface, ska då någon av de metoder som har synlighet `protected` finnas med? Interfacet bör endast ha 3 (eller möjligen 4) metoder deklarerade.

Berörda klasser/interface: **GameModel**, **GoldModel**, **IGameFactory**, **Main**, **ReversiFactory**, **ReversiModel** samt **GameUtils** (ny klass).

4.2 Deluppgift 3

Klassen **GameTile** lider av samma problem som **GameModel** gör.
Bygg om så att klassen **GameTile** istället blir ett interface. Här är det befogat att skapa en ny klass som har samma beteende som konkreta instanser av **GameTile** har. Kalla denna nya klass **BlankTile** (den ska givetvis också vara en **GameTile**).
Då denna deluppgift är klar bör ni diskutera vad ni gjort och hur ni gjort det med en handledare.

Berörda klasser/interface: **BlankTile** (ny klass), **CompositeTile**, **CrossTile**, **GameTile**, **GoldModel**, **RectangularTile**, **RoundTile** och **SquareTile**.

5 Olämpliga kopplingar och brist på flexibilitet

Då vi prövar att implementera något annat än Snake med ramverket blir några begränsningar ganska tydliga;

1. Det är svårt eller omöjligt att på ett elegant sätt lägga till fler komponenter som kan visa status. I reversi-fallet hade det varit trevligt att kunna se vems tur det är och vad poängställningen är.

2. Det enda sättet att få en spelplan med olika utseende på underlag och spelpjäser är genom att göra mer komplicerade spelbrickor. Detta finns som en

nödlösning i **CompositeTile**.

3. Det är långt i från alla spel som har nytta av en timer som skickar uppdateringsanrop.

4. Timern går på fasta tidsintervall.

Vi ska i tur och ordning se vilka förändringar och strukturförbättringar vi kan genomföra för att lösa problemen ovan.

5.1 Problem: Hårda kopplingar mellan modell, vy och kontrollklass

Flera av kopplingarna mellan modell, vy och kontroller sätts upp då de olika objekten instantieras. Här finns ett utmärkt tillfälle att bygga om så att vi får lösare kopplingar. Det arkitekturella mönstret *Model-View-Controller* (MVC) kan tillämpas.

Dessa tre abstrakta entiteter (varje del av M, V och C kan bestå av fler än en klass) har en ansvarsfördelning som följer;

- **Modell**
  - Svarar på frågor om tillstånd (oftast genom anrop från vyn)
  - Uppdaterar systemets datatillstånd (oftast genom anrop från kontrollern)
  - I händelsedrivna system (typiskt applikationer med grafiska gränssnitt) gör den sina *observatörer* (en eller flera vyer) uppmärksamma på att de ska reagera.
- **Vy**
  - Presenterar information från modellen eller modellerna på ett lämpligt sätt. Oftast en grafisk komponent i fönsterbaserade system.
  - Flera (olika) vyer kan finnas för en och samma modell.
- **Controller**
  - Omvandlar användarinteraktion till lämpliga anrop till modellen.
  - I *aktiv MVC*, anropar vyn om att den ska uppdateras.

MVC implementeras vanligen genom någon av dessa varianter:

- *Passiv MVC* – Modellen ansvarar för att just rätt information skickas till vyn. Detta kan också kallas för strikt ”push”; modellen ”knuffar” ut rätt data till vyn. Här är alltså *vyn* mer passiv, den agerar på de data den fått men frågar inte säkert efter mer.

  Anropskedjan är då typiskt **C** $\rightarrow$ **M** $\rightarrow$ **V**.

- *Aktiv MVC* – Kontrollern (eller modellen, indirekt) ansvarar för att vyn får information om att något uppdaterats, vyn frågar därefter modellen efter den information som behövs. *Vyn* intar här en mer aktiv roll – den avgör själv vilka data som behövs för att presentationen ska förändras.

  Anropskedjan blir typiskt **C** $\rightarrow$ **M, C** $\rightarrow$ **V** $\rightarrow$ **M**.

  Aktiv MVC är i grova drag den struktur ramverket har från början:

  - **GameController** känner till (och anropar) **GameModel** och **GameView**.
  - **GameView** känner till och anropar **GameModel**.
  - **GameModel** känner inte till **GameController** och **GameView** är, men kan svara på anrop från dem genom sina publika metoder. **GameModel** signalerar till alla sina *observatörer* då något spännande händer. **GameView** ska komma att observera **GameModel**.

En central idé i MVC-mönstret är att modellen ska ha *lösa kopplingar* till sina vyer. Detta verkar gå dåligt ihop med passiv MVC men här är tricket att den bara känner till lyssnare eller observatörer. Dessa är en *abstraktion*, modellen har inte dessa förutbestämda utan det blir upp till varje observatör att tala om för modellen att de vill lyssna efter förändring.

Flera av defekterna i listan i sektion 5 på sida 4 ska vi börja komma till rätta med genom att mer strikt tillämpa (passiv) MVC.

Vi ska göra så att **GameModel** alltid kan få observatörer tillagda. Skälet att vi inte ska använda **java.util.Observable** är att Javas standardklasser för detta inte är så genomtänkta som man kan önska. **java.util.Observable** *är en klass och inte ett interface*. En konsekvens är att man då inte kan vara observerbar med mindre än att man ärver implementationen!

5.2 Deluppgift 4

Skapa ett nytt interface, **IObservable**. Detta bör endast ha två metoddeklarationer, `addObserver(PropertyChangeListener observer)` och `removeObserver(PropertyChangeListener observer)`.
Gör **GameModel** observerbar. Börja med att låta interfacet **GameModel** utöka interfacet **IObservable**. Vi vill att alla klasser som är spelmodeller också ska kunna observeras.
Lättast är att delegera de metoder som behövs till en instans av **PropertyChangeSupport** i **ReversiModel**.
Tänk på att lämpliga metoder i erat **PropertyChangeSupport**-objekt måste anropas när vi vill göra observatörerna uppmärksamma på detta.

Berörda klasser/interface: **GameModel**, **GoldModel**, **ReversiModel** och **IObservable** (nytt interface).

5.3 Deluppgift 5

När **GameModel** blivit observerbar bör vi pröva och se att uppdateringar till vyer inträffar när de ska.
Ordna så att **GameView** implementerar **PropertyChangeListener**. Varje **GameView**-objekt ska lyssna på sin egen respektive modell. När något händer (från modellen) räcker det med att anropa `repaint` i **GameView**.

Berörd klass: **GameView**

Nu återstår att låta kontrollern släppa taget om vyn så att alla uppdateringar av vyerna endast sker genom att modellen anropar sina lyssnare.

Många av klasserna i Swing-paketet har något dubbla roller. Dels kan de presentera information grafiskt, men de kan också lyssna efter användarhändelser. **GameView** ärver **JComponent**. Det är denna **JComponent** som ska lyssnas på för att kontrollern ska kunna skicka händelserna vidare.

Det enda **GameController** ska göra med vyn är att lägga till sin tangentbordslyssnare. I övrigt ska den lämna vyn i fred.

5.4 Deluppgift 6

Låt uppdateringarna till vyn gå genom modellen, utan att kontrollern direkt anropar vyn.
Nu har vi ett utmärkt tillfälle att också göra olika uppdateringsintervall möjliga. Om vi lägger till metoden `getUpdateSpeed()` till modellen kan den själv avgöra hur ofta eventuella tidsstyrda händelser ska ske från kontrollern. Givet att vi betraktar hastigheten som en modellegenskap (rimligt då den kan hänga ihop med spellogiken) är det en klar vinst att kontrollern inte bestämmer över detta.
Här bör vi också i kontrollern möjliggöra direkta tangentbordstryckningar. Detta införs lättast genom att `enqueueKeyPress` anpassas så att den skickar tryckningarna direkt till modellen om `getUpdateSpeed()` antar särskilda värden (t.ex. `getUpdateSpeed() < 0`).
Glöm inte att pröva att det fortfarande går att byta eller starta om spel, även om man har en speltyp utan uppdateringsintervall.

Berörda klasser/interface: **GameController**, **GameModel**, **GUIView**, **GoldModel** och **ReversiModel**.

5.5 Fler och mer specifika vyer

Som vi sett tidigare har vi inte kunnat välja hur vi vill presentera eventuella poäng från spelen; den enda möjligheten har varit att låta modellen direkt sköta det. När vi har möjlighet att observera modellen blir detta mindre komplicerat.

5.6 Deluppgift 7

I **ReversiFactory**, lägg till en lyssnare av typen **ReversiScoreView** när en reversi-modell skapas. För en specifik lyssnare är det tillåtet att gå på klasspecifika egenskaper. Alltså kan **ReversiScoreView** anropa `getBlackScore()` och `getWhiteScore()`. Det är efter lämpligt test (`evt.getSource().getClass() ...`) tillåtet att göra antagandet att den som genererat uppdateringen är av typen **ReversiModel**.
ReversiScoreView kan ha en mycket enkel `propertyChange`-metod, låt den bara skriva ut svarts och vits poäng samt vems tur det är att spela.

Berörda klasser/interface: **ReversiFactory** och **ReversiScoreView** (ny klass).

Nu har vi löst problem 1, 3 och 4 från vår lista! Bortsett från att reversi-spelet nu inte köar massa tangentbordstryckningar ser det ungefär likadant ut. Om vi däremot skulle välja att implementera ett nytt spel, t.ex. Tetris, kommer vi troligen att upptäcka att det blir betydligt enklare. För just Tetris behöver vi minst två funktioner som det ursprungliga ramverket inte hade: möjlighet att ändra fördröjningen mellan händelser samt möjlighet att visa nästa bit och/eller poängställning på ett snyggt sätt.

5.7 Frivillig uppgift

Det finns givetvis fler både funktionsmässiga och strukturella defekter både i ramverket och våra modeller.
I mån av tid, identifiera så många problem som möjligt (förutom dem vi redan nämnt) och diskutera med en handledare.
Åtgärda gärna så många fel som möjligt men försäkra er om att ni har en version av koden som bara innehåller lösning fram till uppgift 7.

Berörda klasser/interface: ?

Referenser

[Skr09] Dale Skrien. *Object-Oriented Design using Java.* McGraw-Hill, international edition, 2009.