Laboration 1 – Att använda ramverk

Labbgrupp: 15
Medlemmar:
Johan Brook (900720-0216)
Robin Andersson (900122-0574)

Datum: 2011-10-31

Svar på frågeställning

1. Vad är skillnaden på GoldModel och GameModel, och hur är de relaterade till varandra?

`GameModel` är en abstrakt klass som hanterar spelplanens tillstånd (storlek, setters och getters av enskilda plattor på X- och Y-positioner). `GoldModel` **ärver** `GameModel`, och implementerar den enda abstrakta metoden – `gameUpdate()`. Den metoden anropas gång på gång av tråden i `GameController`, och tar i princip hand om allt som ska hända när spelet uppdaterar sig. T.ex. kollar den om man har flyttat sig utanför spelplanen, träffat ett mynt, och uppdaterar planens tillstånd alltefter.

Vi tänker oss `GoldModel` som en utbytbar del i spelet – kod som beskriver specifika delar av ett spel – hur allt ska fungera.

2. Vilken klass utför själva ritandet av ”Gold coins” i spelet ”Gold Game”?

Metoden som ritar den grafiska representationen av ett enskilt mynt är implementerad i `RoundTile`. Metoden, `draw()`, är en överskuggning av metoden i superklassen (`GameTile`), och ritar en fylld cirkel på ritytan (referensen till `Graphics`-objektet fås via en parameter). Vidare anropas `draw()` på alla tiles i `GameViews` `paintComponent()`-metod, däribland mynten som vi här är intresserade av.

3. Vilken klass är det som anropar för detta ritande?

Se ovan.

4. Vad är syftet med GameFactory, och hur kan den utvidgas för att få med ett Snake-spel?

`GameFactorys` uppgift är att indexera och bestämma vilka spellägen som kan skapas. Med metoden `createGame` returneras/skapas en modell av det spel vars namn skickas med som parameter. I det här fallet så är parametern ett val i dropdown-listan i `GUIView`, som har fyllts med strängar (namn på spelmodeller) från just `GameFactory`. Det betyder att om man har skapat ännu ett spelläge kan man lägga till lägets namn i sträng-fältet i `getGameNames()`-metoden samt lägga till ytterligare en villkorssats i `createGame()` som returnerar det nya spelläget utifrån namnet.

Om vi inte hade haft tillgång till implementationen av `IGameFactory` så hade ett annat alternativ varit att skriva en ny klass som implementerar gränssnittet, d.v.s. som implementerar metoderna `getGameNames()` och `createGame()` så att de returnerar namn och en instans av vår nya modell.

5. Var återfinns beräkningen och kontrollen av ”gold eaters” rörelser?

I `GoldModel`. I metoden som uppdaterar spelet kontinuerligt (`gameUpdate()`) anropas diverse hjälpmetoder som bl.a. simulerar gold eaters rörelser[1], byter riktning på gold eater när spelaren trycker på piltangenterna, kollar om gold eater är innanför spelområdet, och kollar om man har träffat ett mynt.

6. Hur lagras ett spelbräde, och i vilken klass?

Man kan tänka sig spelbrädet som ett rutnät, en matris, med `GameTiles` där matrisens koordinater återspeglar spelbrädets koordinater. I klassen `GameModel` finns detta implementerat, och accessmetoder används för att uppdatera och returnera spelbrädets tillstånd i en specifik koordinat.

7. Beskriv arbetsflödet i programmet.

I `Main` skapas huvudvyn `GUIView` med en referens till ett `GameFactory`. I `GUIView` byggs gränssnittet upp med element, bl.a. en startknapp vars lyssnare (inre klass) skapar ett nytt spel baserat på vad man valde för spelläge i dropdown-listan. Detta med hjälp av `GameFactory` (se fråga 4). I `GUIView` skapas även den övergripande kontrollern (`GameController`) med en referens till spelbrädets vy (`GameView`). Nedan gås de olika klasserna och deras uppgifter igenom i den ungefärliga ordning som de anropas och används allteftersom spelet körs. Vi delade upp dem under rubriker så det blir lättare att läsa.

GameController

Den övergripande ”spelmotorn” som lyssnar på tangenttryck, ritar om spelbrädets vy, och uppdaterar spellägets modell. För att få motorn att ”röra på sig” läggs omritningen och uppdateringen i en tråd som man pausar i ett visst antal millisekunder (en uppdateringsfrekvens). Flödet blir alltså ”uppdatera, rita om, pausa -> repetera”. Om en `GameOverException` fångas medan spelet är igång avslutas körningen. `GameOverException` kastas inifrån spellägets modell (här `GoldModel`).

Kontrollern har alltså en referens till en spellägesmodell (en `GameModel`) och låter även spelbrädets vy få den referensen genom att anropa metoden `setModel()` på vyn.

GameView

Vidare i `GameView` ritas spelbrädet upp med dimensioner från klassen `Constants` och alla relevanta brickor (`GameTiles`). Själva ritningen av en enskild bricka delegeras vidare till dennes egna `draw()`-metod. I och med att vyn känner till spellägesmodellen kan den komma åt enskilda brickor i matrisen (se fråga 6) genom `GameModels getGameBoardState()`-metod.

---

[1] Genom att uppdatera spelbrädesmatrisen med en blank bricka på gold eaters nuvarande position och sedan lägga gold eater-brickan på en ruta bredvid i matrisen (beror självklart på den nuvarande riktningen).

GoldModel
Tillbaka i `GameController` uppdateras som sagt spellägets modell genom att anropa dess `gameUpdate()`-metod. `gameUpdate()` måste implementeras, eftersom den är deklarerad som abstrakt i `GoldModels` superklass (`GameModel`). I `GoldModel` är gold-spelets egentliga logik implementerad, vilket innebär att spelbrädets tillstånd och vad som ska hända vid tangenttryck – det är i själva verket `GameController` som lyssnar och skickar med en numerisk representation av det senaste tangenttrycket, vid varje ”uppdateringsfas”, till `gameUpdate()`-metoden. Piltangenterna bestämmer gold eaters riktning, övriga tangenter ignoreras.

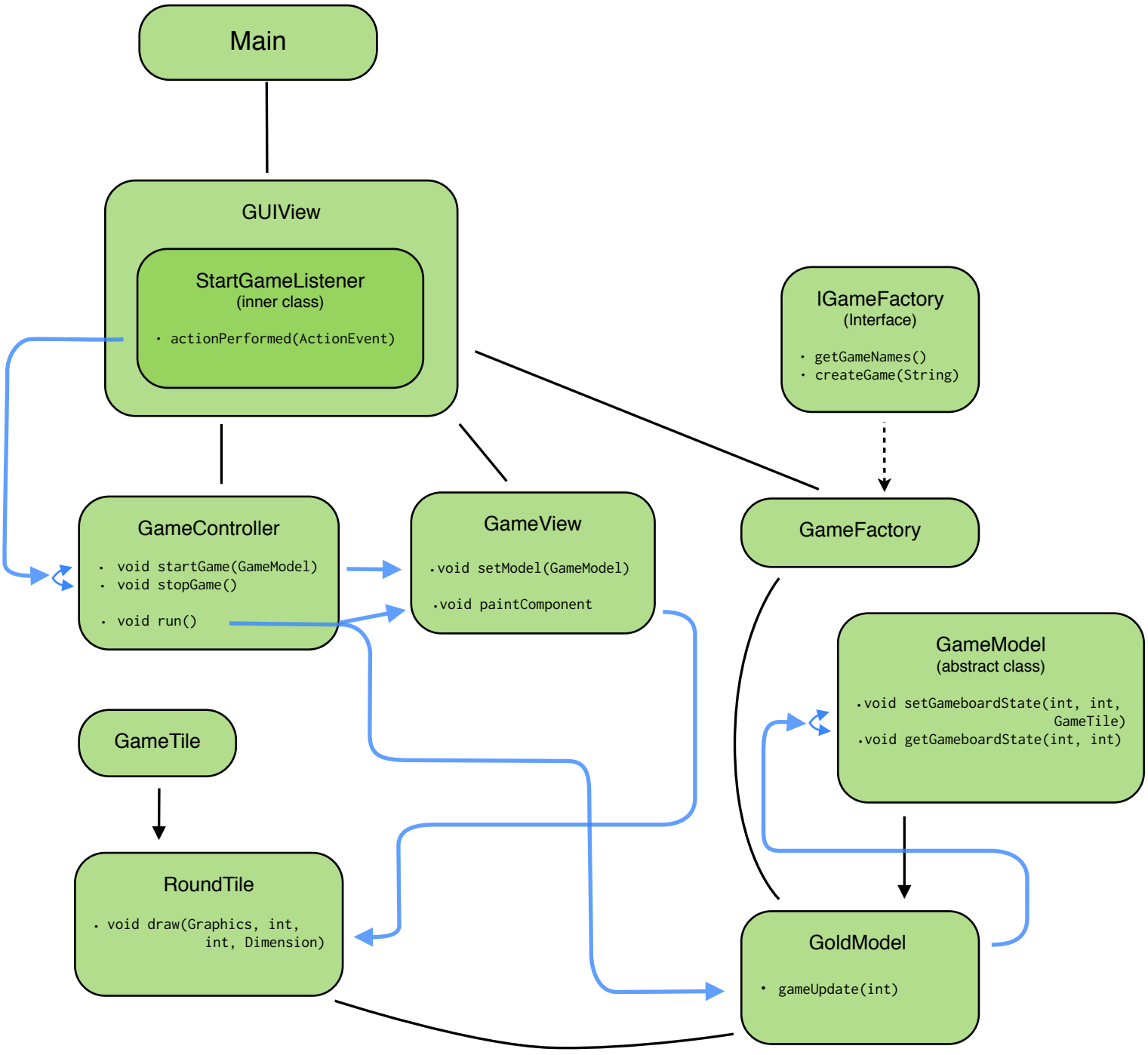
När `GoldModel` instansieras läggs mynt och gold eater ut på spelbrädet. Vidare uppdateras gold eaters och myntens positioner (se fråga 5). Mynten finns i en lista och är grafiskt representerade av en bricka av typen `RoundTile`. Om gold eater träffar ett mynt på spelbrädet ökas poängen och myntet tas bort ur listan. Spelet avslutas om alla mynt är borta ur listan eller om gold eater flyttar sig utanför spelplanen.

Viktigt att notera är att `gameUpdate()` **anropas och körs gång på gång** inifrån `GameControllers` callback-metod för tråden (`run()`).

GameModel
Är en abstrakt klass som lagrar spelbrädet och dess brickor (se fråga 6). Alla spelläges-modeller bör ärva `GameModel` då de därmed har tillgång till getter- och settermetoderna för spelbrädets tillstånd, och även utomstående klasser som använder spelläges-objekt kan använda sig av dessa accessmetoder.

RoundTile (ärver GameTile)
Vy för enskilda brickor, tiles, såsom mynten och gold eater. Ritar upp sig själv med givna färger, mått och på en position på ritytan (vars referens, ett `Graphics`-objekt, fås via en parameter och som därmed kommer ifrån `GameViews paintComponent()`-metod).

Flödesdiagram



= Programflöde/metodanrop

= Skapar/använder

= Implementeras av

= Ärvs av

8. Ge en plan för hur du tänker fortsätta med deluppgift 2. Vilka klasser kommer du att skriva, och vilka kommer du att återanvända?

För att implementera ännu ett spelläge i Snake-stil bör man för det första skapa en ny klass (kanske `SnakeModel`) som får ärva `GameModel`. I denna modell ska logik implementeras, såsom

- ✓ att när masken träffar en frukt på spelbrädet ska frukten tas bort, masken ska växa i längd, poäng ska läggas till, och en ny frukt ska slumpas ut på spelbrädet.
- ✓ att om masken kolliderar med sig själv eller med spelbrädets väggar ska spelet avslutas.

Vi tänker oss att många idéer kan hämtas från gold-spelet, med tanke på hur lika de två spellägena kommer att bli: styrning med piltangenter, kollisionshantering, uträkning av positioner, mm.

Nya klasser för brickor som blir visuella representationer av frukter och själva masken måste skapas. I `GameFactory` ska vi lägga till det nya spelläget i listan över tillgängliga spel, samt anpassa metoden `createGame` att returnera en instans av vår `SnakeModel` om en sträng ”snake” eller liknande används som parameter.

I övrigt kan klasserna i det befintliga ramverket användas och utnyttjas där det är möjligt.