Laboration - Winning Calculator

Denna laboration kan lösas individuellt eller i par (max 2 personer).

Bakgrund

Företaget Winning INC har utvecklat en ny programvara som är tänkt att hjälpa användaren att räkna fram hur stor chans hen har att lyckas.

Implementationen är nästan helt klar, men i slutskedet av utvecklingen så hoppade företages enda utvecklare BumliDev54 av projektet.

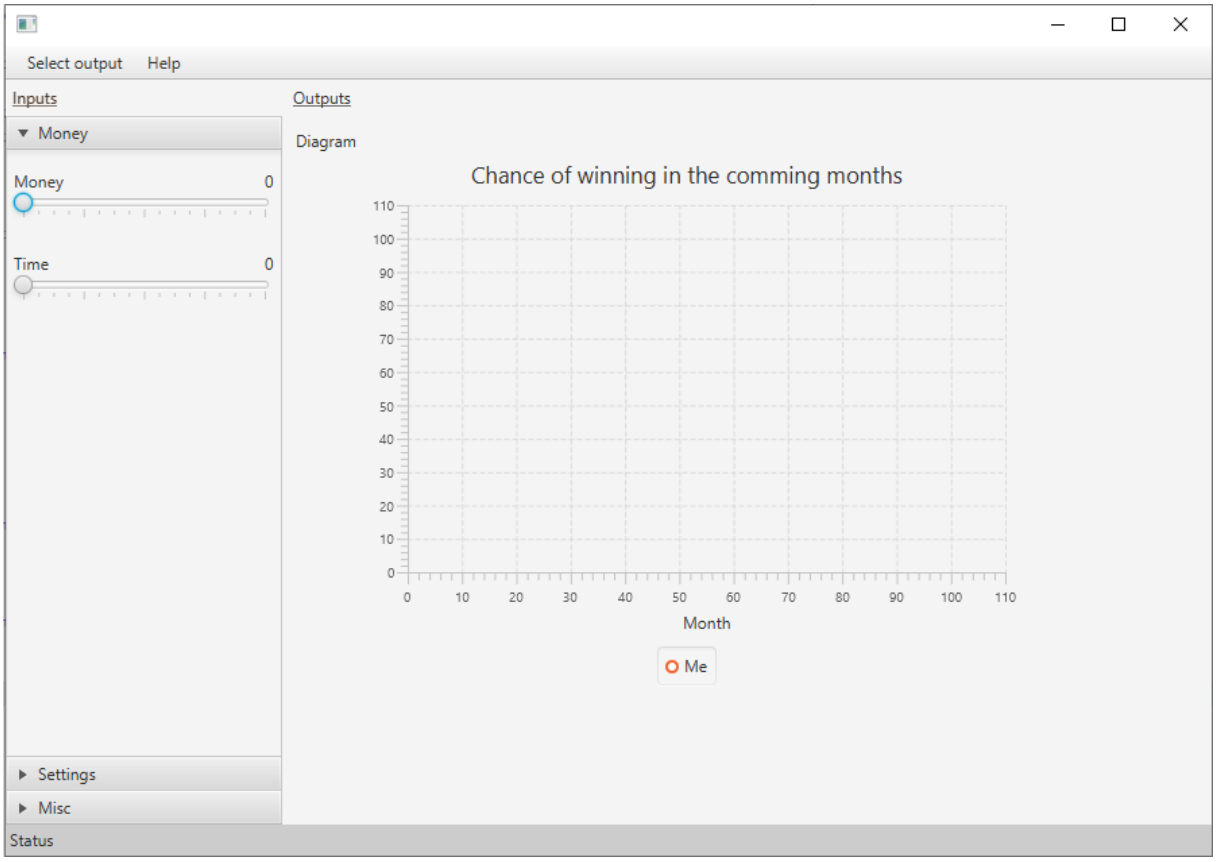
Winning INC behöver alltså nu din hjälp för att få sista pusselbiten på plats innan publicering av applikationen kan ske.

Projektet

Kod att utgå ifrån hittas här: https://bitbucket.org/margob/exam_winning-calculator_a-d/src
(git clone https://margob@bitbucket.org/margob/exam_winning-calculator_a-d.git)

Att göra

När du startar applikationen så möts du av följande:



I vänstra delen av applikationen finns det möjlighet att ange input till programmet (ex: Money och Time).

I högra delen av programmet visas en där det skall vara möjligt att se sin chans att lyckas över en given tidsperiod. Denna chans är då meningen att den skall bero på de inputs som anges. Om du testar att dra i de sliders som finns så händer det inget i grafen. Detta för att det för närvarande saknas koppling emellan klasserna som ansvarar för de olika delarna i applikationen.

Det är här du kommer in, din uppgift är att lösa så vi får en koppling emellan klasserna. Denna koppling skall göras med hjälp av designmönstret Observer (Observer pattern).

Konkret:

- Få önskat resultat i applikationen genom att implementera Observer pattern*.

- Bifoga i din inlämning en .pdf som besvarar följande fråga:
  1. Följer din implementation av Observer pattern pull eller push metodiken för mönstret?

** Det är endast tre klasser som du behöva göra ändringar i för att lösa uppgiften, dessa är: StateStore, OutputDiagramController samt OutputTableController. Börja med att identifiera vilken klass som bör vara Subjekt-klassen och vilka klasser som blir observers (subscribers).*

Du kommer troligen behöva skapa och lägga till klasser och/eller interfaces för att lösa uppgiften. Om du vill får du självklart göra ändringar i andra klasser.

OBS Javas native lösning för observer får ej nyttjas.

Förberedelser

- Föreläsningar och laborationer på sal vecka 1.
- Kursbok, kap 1 och 2 *(kap 4 förekommer i lösningen sedan tidigare)*.

Inlämning

Du laddar upp en .zip innehållandes:

1. Koden för en fungerande lösning av applikationen där Observer pattern är implementerat.

2. En .pdf med efterfrågat innehåll.

Om du har jobbat i par så skickar ni båda in samma fil och inkluderar då även:

3. En .txt som innehåller *”Vi (”namn1” och ”namn2”) har arbetat i par med denna lösning. Och försäkrar här med att vi samtliga har aktivt deltagit samt nu anser att vi kan redogöra för lösningen om så skulle bli aktuellt”.*

Betyg

Inlämningen kan ge något av betygen komplettera (U) eller godkänd (G).

Betygskriterier

När vi betygsätter inlämningen tittar vi på om:

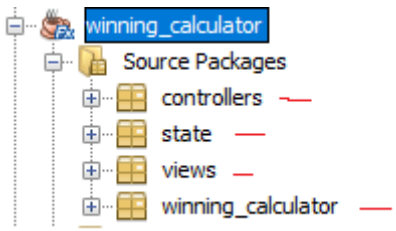
- Efterfrågat mönster finns implementerat.
- Frågan besvaras korrekt givet din/er implementation.
- Kvalitet på lösningen.

Hjälpmedel

Förväntat resultat i applikationen kan ses här: https://youtu.be/Ans4fxr-eaE

Genomgång av klasserna i projektet:

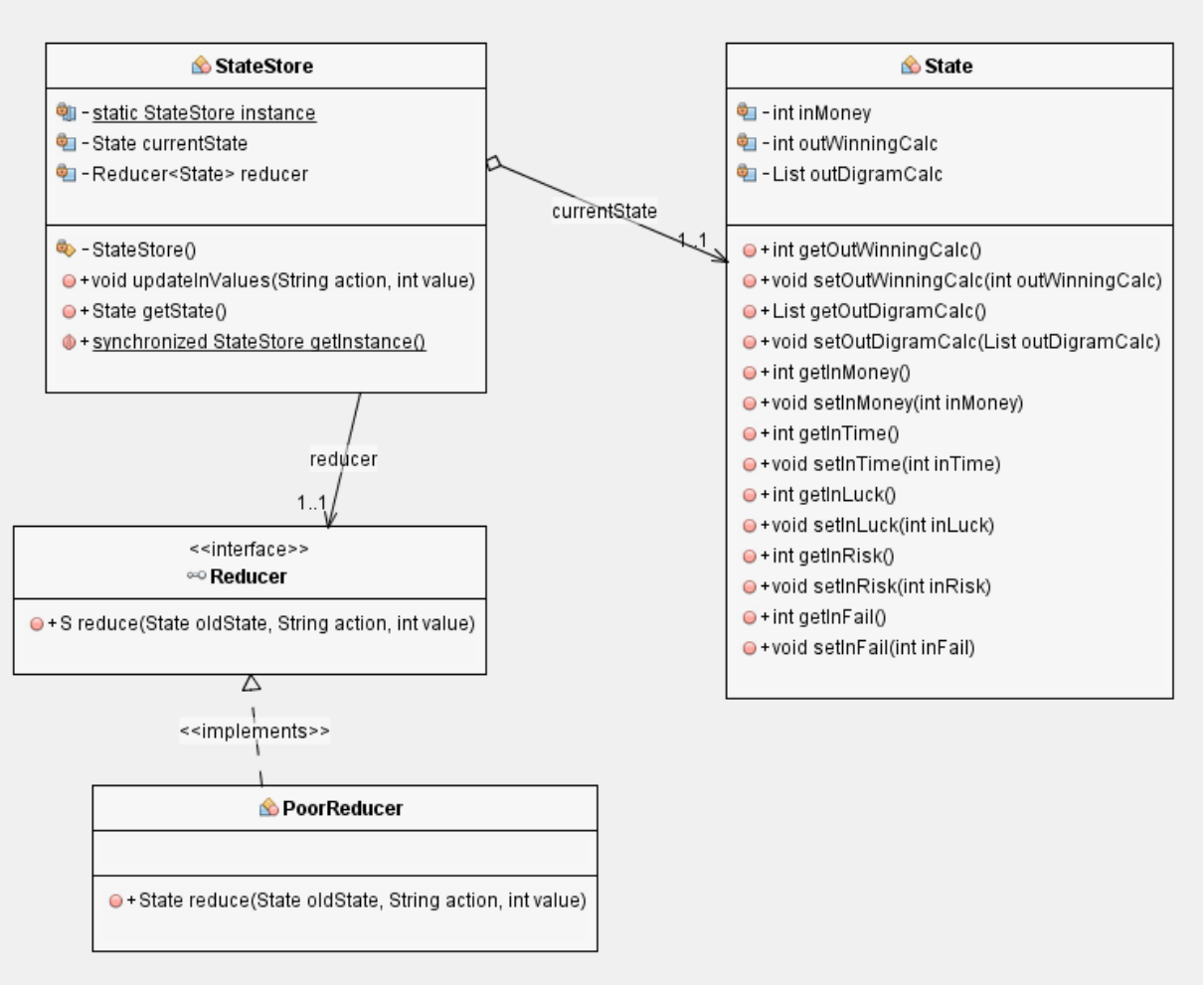
I applikationen återfinns fyra moduler (paket/packages):



I detta dokument kommer varje paket gås igenom med huvudansvar samt delar som kan vara viktiga för dig för att lösa uppgiften.

State paketet

State paketet innehåller logik rörande hantering av applikationens tillstånd (den data som den för närvarande använder). Paketet innehåller 3 klasser och ett interface:



Klassen State: Ansvarar för att hålla reda på data, alltså en ren modellklass där de värden som behövs i applikationen sparas.

Klassen StateStore: Ansvarar för att hålla reda på applikationens nuvarande tillstånd (State-klassen). Det är även denna klassen utomstående klasser kan anropa för att göra ändringar på ”tillståndet”. Detta görs då med anrop till metoden **updateInValues.** updateInValues tar två parametrar, action och value.

Denna metoden anropas i de olika kontroller klasserna som börjar med Input, exempel: updateInValues("money", 50);

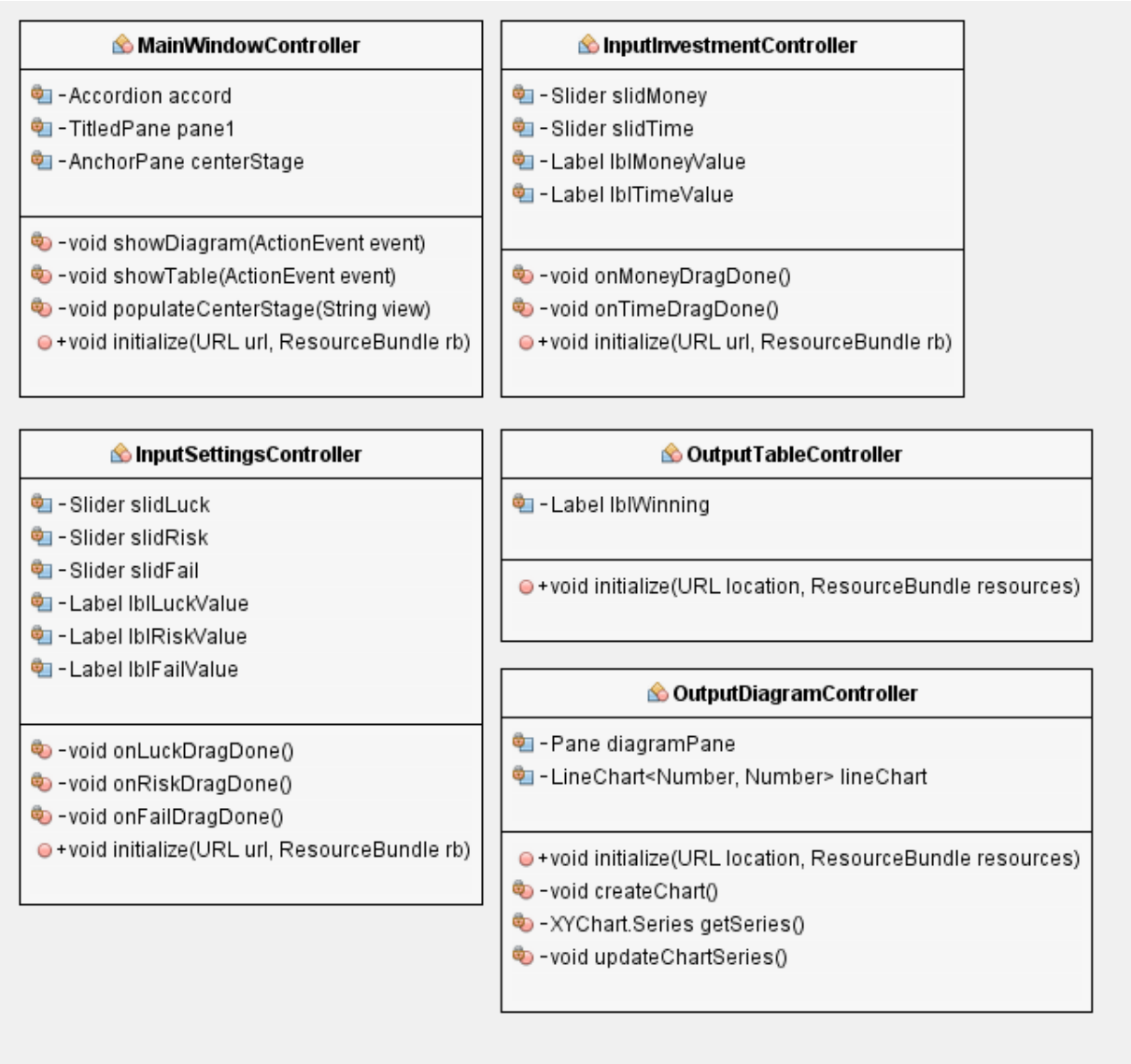
För att säga att vi vill uppdatera money till 50.

Interface Reducer samt klassen PoorReducer:

Iden med en reducer är kunna ta en behållare med data (State) och utföra en modifikation (action) på den. När det är gjort ges ett nytt tillstånd tillbaka där modifikationen nu återspeglas.

Detta finns här implementerat genom ett interface som definierar hur Reducer's får se ut (API). Vi har sedan en PoorReducer som utför själva jobbet. Namnet Poor föreslår att detta är en fattig implementation av mönstret (state-reducer-pattern).

Controllers paketet:



Controllers paketet innehållet klasser med logik kopplat till varje vy (det användaren ser). Vyerna är indelade i avsnitt där varje avsnitt har sin egna kontroller klass.

MainWIndowController:
Huvudfönstret som innehåller de andra vyerna. Sätter vid start de sub-vyer som skall finnas. Input-vyer sätts i pane1 respektive pane2 (de utgör vänstra delen av applikationen). I metoden populateCenterStage så sätts vyn som skall synas till höger i applikationen (output data). Vid start av applikationen anges diagramvyn som output. I menyn går det sedan att ändra så man istället ser resultatet som en tabell.

InputSettingsController + InputInvestmentController:
Kontrollerar främst när någon drar i dem sliders som finns i inputvyerna, när det sker så kommuniceras det vidare till State-paketet i applikationen att användaren vill uppdatera applikationens tillstånd.

OutputDiagramController:
Kontrollerar Output data för vyerna. Metoden updateChartSeries kan anropas för att säga till klassen att hämta och skapta data till diagrammet. Denna data hämtas då ifrån State-paketet. Ingen ny kod behövs för själva diagrammet, men det som efterfrågas i uppgiften är när denna updateChartSeries skall köras.

OutputTableController:
Bör när du gjort din lösning uppdatera sin Label så det för användaren redovisas det som återfinns när man anropar getOutWinningCalc() metoden som återfinns i State-klassen.