Tentamen OOP med Java

Ville Wassberg

June 2023

1 Referenser

a) Ja, en klass utifrån kan ändra på högen, en annan klass kan referera till en instans av CardPile via getCards(), och därigenom ändra på korthögen som man önskar, utan att behöva kalla på metoderna definierade i klassen cardPile. Ett exempel på hur man på ett bättre sätt kan definiera klassen kan vara genom att använda sig av inkapsling. Så i konstruktorn för klassen skulle man kunna skriva $cards = new\ ArrayList <>$ (); för att skapa en ny lista som refererar till varje ny instans av CardPile. I metoden getCards skulle man också kunna returnera: $return\ new\ ArrayList <>$ ($cards$) för att returnera en kopia av listan, så att ändringar av högen måste ske via de metoder som finns i CardPile.

b) Ja, två variabler kan referera till samma objekt. Till exempel så skulle jag kunna instansiera en CardPile som referas till av en variabel CardPile hög1, och sedan låta en variabel CardPile hög2 referera till hög1. Om jag då ändrar något i hög 1 och sedan ska printa händelsen, så skulle jag kunna printa hög2, och få samma resultat. Det här fungerar eftersom att referenserna pekar på en allokerad plats i minnet, som objektet i fråga ligger i, så när man gör ändringar på objektet så skickar referenserna den nya datan oavsett genom vilken av referensvariablerna man vill kalla på objektet via.

c) Nej typomvandling av ett befintligt objekt är inte tillåtet i Java. Det beror på att typen av objektet bestäms när man instantierar objektet. Specialfall kan vara i polymorfism när till exempel en annan typ av hög ärver från CardPile, då kan CardPile-objektet typomvandlas till ett sådant. Eller som i Java Direkt, där ett objekt av typen Object skulle kunna omvandlas till ett annat objekt i och med att alla objekt ärver från Object, men konverteringen är farlig då man måste vara säker på att det verkligen är ett Object man försöker typomvandla.

2 OOP

a) Ett välskrivet objektorienterat program bör utgå från identifiera klasser som ska vara med genom att identifiera objekt eller koncept som ska användas i programmet; hur polymorfi ska användas, vad som är gemensamt mellan olika objekt och klasser. En klass bör göra vad den ska och inte ha ett för brett eller snävt ansvarsområde. Gemensamma metoder bör inte behöva definieras flera gånger. På så vis blir koden lättläst, rätt fram och lättare att redigera utifrån. Klasserna bör inte heller vara beroende av varandra i för hög utsträckning, så att en förändring i en klass påverkar en annan klass mer än nödvändigt. Dessutom bör det vara lätt att lägga till funktionalitet utan att man behäver göra förändringar i den befintliga koden.

   En klass bör ha ett väl beskrivande namn. Den bör ha en konstruktor som initialiserar diverse attribut för objektet som klassen representerar. Klassen bör använda sig av inkapsling för att definiera hur manipulation av objektet kan tillåstas och ske. Klassen bör använda modifierare frekvent så att åtkomsten till metoder och innehåll kan bestämmas.

   Metoder bör ha beskrivande namn som beskriver vad dens uppgift är. Uppgiften bör dessutom inte vara mer än en, utan man bör definiera flera metoder för flera uppgifter, så kan metoderna kallas på i en större uppgift. De bör vara klara och lättlästa med lagom mycket funktionalitet. Kommentarer som beskriver metoden är ett fint tillägg.

   En bör använda sig av arv när det finns en tydlig hierarki mellan klasser. Till exempel när en Spelare kan vara en dator elle människa, så kan det vara bra att låta datorn och människan ärva från klassen Spelare. Som ovan nämnt bör arvet användas så att metoder och attribut inte behöver definieras flera gånger; gemensamma metoder och attribut bör ärvas av superklassen, medan sådant som utmärker varje enskild klass bör definieras inom den klassen. Man bör definiera arvet tydligt, utan att göra det för invecklat; då kan det vara klokt att använda gränssnitt och multipelt arv istället.

b) Ett exempel på när ett program inte följer objektorienterad konvention kan vara när två klasser som ärver från Spelare namnges till Spelare1 och Spelare2. Spelarna kanske har helt skilda uppgifter, men det är inget som namnen på klasserna pekar på. En klass kan, likt exemplet i fråga 1, ha getter-metoder som skickar ut objektet i fråga så att det kan redigeras utifrån istället för att skicka ujt en referens till en kopia av objektets attribut. En metod som har flera uppgifter och kann returnera olika typer kan vara ett exempel på en metod som inte följer OOP-konventioner. Om man till exempel har en klass som beskriver djur, och några olika klasser

som beskriver enskilda djur som katt, hund etc, men även en klass Micro, som ärver från Djur och och ett attibut som ”görEttLjud()” ärvs, då bryter man mot arvskonventionen, då en micro knappast är ett djur. Man bör alltså inte låta bli att följa är-relationer i arv.

3 Solitaire

Klassen Card beskriver varje enskilt kort och dess attribut. Där definieras metoderna för att måla upp korten, kolla om en punkt är inom kortets plats, om ett kort är uppvänt eller inte, flytta kortet med andra tillhörande metoder. Klassen Control är en panel som tar hand om knapparna som läggs till på panelen för nytt spel, ett förprogrammerat nytt spel, samt en avsluta-knapp; dess konstruktor tar in vilket spel och vilken panel som den ska referera till. Klassen DeckPile ärver från Pile, och representerar talongen i spelet. Klassen DiscardPile ärver alla egenskaper från Pile, och definierar inga egna attribut förutom dess egen konstruktor som gör detsamma som i klassen Pile. Klassen Game är klassen som beskriver spelet och initierar korthögarna och spelets utformning; där definieras också kortens värden och färg, och även semantiken och reglerna för spelet. Klassen Pile representerar allt det gemensamma för korthögarna; hur ett kort tas från en hög och flyttas till en annan, om en hög innehåller en punkt och hur högarna ritas upp. Klassen Shadow ritar upp en rektangel som är utsidan på korten; som syns om en korthög är tom till exempel; rektangel fungerar som en skugga till korten när de flyttas. Klassen Sol ärver från JFrame och fungerar som ramen för hela speler där panelen för spelet läggs till, och var kontrollpanelen samt panelen för spelet läggs till och där main-metoden finns, där spelet körs igång. Klassen SolListener beskriver vad som ska hända när en händelser med musen sker: när något blir klickat på, när musen blir dragen, när den släpps. Klassen SolPanel beskriver panelen som håller spelets layout; Bakgrund, var lyssnare läggs till, kallar på många metoder som återfinns i klassen Game. SuitPile är en hög som korten läggs i när talongen klickas på. Den ärver all från klassen Pile. TablePile ärver också allt från Pile och beskriver de högarna som finns nere i spelet.

Arv används framförallt i korthögarna, där TablePile och DeckPile ärver direkt från klassen Pile och är alltså dess under/subklasser. Piles metoder fungerar alltså gemensamt för dessa högar.

Polymorfism används i SolListener där metoder från MouseAdapter och MouseMotionListener omdefinieras. Även där paintComponent används, använder sig programmet av polymorfism, där varje målar-metod har sin egen funktionalitet.

4 Interaktion

a) När användaren klickar på talongen tas ett kort därifrån och läggs i discardPilen intill, där kortets framsida visas. Det görs genom att mouseClicked anropas i SolListener, där verifieras det om talongen är klickad på, om den är det och den är inte tom så vänds det översta kortet och tas från talongen och läggs på discardPile och aätts som senaste kortet i panelen. Om talongen är tom, då tas alla korten från discardPile och läggs så att översta kortet hamner underst.

b) När användaren klickar i spelfönstret letas det upp om en korthög innehåller klicket. Om det inte finns någon korthög som gör det eller om kortleken inte är talongen, så målas panelen bara om på nytt och ingenting sker.

c) Om en korthög är klickad på och den inte är tom då letas kortet upp i den högen genom findCard metoden som definieras i Game-klassen och återdefinieras i panelen. Om kortet inte är null räknas avståndet mellan kortets koordinater och det klickets koordinater, och skugga ritas upp där kortet flyttas. Allting är enligt spelets regler då enskilda regler är definierade genom metodernas syntax. Ett kort kan alltid flyttas, men om det kan läggas på sin plats kollas genom mouseReleased-metoden, där kollas vilken korthög som tar emot mussläpept, det dragna kortet, om inte null, flyttas dit genom moveCard-metoden där det valda korte, högen och nya koordinaterna är argumenten. I moveCard-metoden är reglerna för om kortet kan släppas där, eller inte, definierade beroende på vilken typ av korthög det är, om den är tom, som också kollas via pileCanTake-metoden.

d) Ett kort läggs så att det inte täcker underliggande kort när korthögen är av typen TablePile. Programmet avgör detta genom att kolla om högen är TablePile genom att använda det logiska predikatet instance of, då anropas addFanned-metoden som är definierad i klassen Pile som flyttar kortet till högen med ett avstånd till toppen av högen på konstanten "fanned".

e) Ett kort vänds upp om högen är av typen TablePile och den inte är tom, när ett kort tas därifrån och läggs på en annan hög, vilket metoden moveCard i klassen Game visar på.

5 Polymorfi

a) Några metoder som kan introduceras är följande: canTake, som är abstrakt i Pile som varje hög skriver över själv; canRelease som också kan vara abstrakt och som returnerar false om högen inte är DiscardPile eller TablePile.

b) I klassen Game behöver metoderna pileCanTake och pileCanRelease tas bort, samt vissa anrop på de metoderna ändras så att de anpassas till de

nya metoderna som finns i pile-klasserna. Jag ändrar även i SolListener i mouseClicked, för att ta bort instance of DeckPile, så jag skapar en metod i Pile och DeckPile, fillDeckPile() där Pile inte gör något men DeckPile fyller upp korten som ligger i discardPile. Jag ändrar även i mouseDragged där jag låter den kalla på received-högens canTake()-metod istället. Jag väljer också att ändra på addFanned-metoden i Pile-klassen och skriver över den i TablePile-klassen, så att om den inte är TablePile läggs kortet bara till, men om det är TablePile så läggs kortet till nedskjutet fanned distansenheter. På grund av den här förändringen satte jag this.x = x och this.y = y i TablePile's konstruktor. (På grund av att högarna ritades upp fel gjorde jag om lite här och skrev över addFanned i alla klasser utom TablePile istället). Jag lägger också till en abstrakt metod showCard, som ersätter kodsnutten i moveCard i Game som visar kortet om högen är en instans av TablePile.

c) Jag bifogar det redigerade programmet i en Zip-fil tillsammans med denna pdf.