Inlämningsuppgift 3

Grupp: 325

Sanna Melke (same3047)

Sophie Lindberg (soli5539)

|  |
| --- |
| Objektorienterad programmering  Höstterminen 2015  Kursansvarig: Henrik Bergström |

# Design

Här ska ni beskriva designen av systemet i form av ett detaljerat klassdiagram kompletterad med en kort textuell beskrivning av vilka designbeslut ni tagit. Syftet är att visa att ni uppfyllt kravet på att systemet ska vara objektorienterat. Var också noga med att redovisa vilka delar ni har gjort själva och vilka som är tagna från Java eller från andra bibliotek. Det finns inga speciella krav på omfattning, men 1-2 sidor bör vara ganska lagom.

Detaljeringsnivån på klassdiagrammet ska vara samma som för hundklassen: allt ska finnas med. Om ni gjorde ett klassdiagram innan ni började får ni gärna ta med det också så att det går att se hur ni tänkte, och vad som blev annorlunda. Detta klassdiagram behöver naturligtvis inte vara lika detaljerat.

# Funktionen lägg till resultat

Vi valde att lägga upp funktionen *lägg till resultat* på det sätt att det var en helt egen klass. Det som vi gjorde var att vi skapade först en Command klass som hanterar alla de komandon som kommer in till systemet och som sedan så tillämpade vi arv på de övriga Command-klasserna så att dessa ärvde alla egenskaper från Command-klassen. Detta var på grund av att vi inte ville ha klasser med för mycket kod och långa metoder, detta för att göra koden mer lättläst och lätthanterlig.

Allt som vi gör i klassen CommandAddResult (alltså den klass där vi hanterar funktionen *lägg till resultat*) gör vi inom en execute metod. Det första som vi då gör är att fråga användaren efter det startnummer som deltagaren vars resultat vi vill lägga till har. Sedan så anropar vi en scanner för att kunna läsa det startnummer som användaren anger genom att skriva in detta på tangentbordet. Denna kod ser ut på följande sätt:

System.*out*.print("Startnumber: ");  
int startnumber = *scanner*.nextInt();  
*scanner*.nextLine();

Det som vi sedan vill göra är att kontrollera att det finns en deltagare med det angivna startnumret och detta gjorde vi på följande sätt:

Participant participant = tournament.findParticipantByStartnumber(startnumber);  
if (participant == null) {  
 System.*out*.println("There is no participant with that number");  
 return;  
}

Där anropar vi en finder-metod som finns i vår Tournament-klass som ser ut på följade sätt:

public Participant findParticipantByStartnumber(int startnumber) {  
 for (Team t : teams) {  
 for (Participant p : t.getParticipants()) {  
 if (p.getStartnumber() == startnumber) {  
 return p;   
 }  
 }  
 }  
 return null;

}

Vi anropar detta från Tournament-klassen då vi i den klassen har sparat en lista med alla lag som finns registrerade. Eftersom att alla deltagare måste tillhöra ett lag ansåg vi att det skulle vara lämpligt att då spara alla deltagare i en lista i dess lags instans av klassen Team. För att sedan kunna komma åt en deltagare utifrån dess startnummer, så måste vi då gå igenom alla lag och för varje lag måste vi sedan gå igenom alla dess deltagare för att hitta det unika startnummer som den deltagaren har. Om vi inte kan hitta en deltagare med det startnumret så returnerar vi null. Om null skulle returneras så hanteras detta i CommandAddParticipant-klassen. Eftersom att finder-metoden returnerar ett helt objekt, det vill säga en hel deltagare, så kan vi lätt kontrollera om detta är null och i så fall så skriver vi ut att det inte finns en deltagare med det angivna startnumret och returnerar, vilket tar användaren tillbaka till huvudmenyn och användaren får ange ett kommando igen. Om det finns en användare med det angivna startnumret så frågar programmet sedan efter vilken gren som resultatet gäller och detta löste vi på följande sätt:

String eventName = this.readString("Event: ");  
eventName = this.*normaliseName*(eventName);

Programmet måste även i CommandAddEvent-klassen spara vilket lag som den angivna deltagaren tillhör. Detta löste vi med hjälp av ännu en finder-metod. I detta fall så behöver vi dock inte oroa oss för att vi ska få något null-värde tillbaka eftersom att vi redan vet att laget finns, eftersom att vi annars inte hade kunnat hitta deltagaren i den tidigare finder-metoden. Detta gör att vi aldrig kommer att få ett null-värde och detta betyder att programmet inte behöver någon kod som hanterar eventuella null-värden.

Team team = tournament.findTeamByParticipantStartnumber(startnumber);

Även denna finder-metod finns i Tournament-klassen och denna ser ut på följande sätt:

public Team findTeamByParticipantStartnumber(int startnumber) {  
 for (Team t : teams) {  
 for (Participant p : t.getParticipants()) {  
 if (p.getStartnumber() == startnumber) {  
 return t;  
 }  
 }  
 }  
 return null;  
}

Programmet behöver även kontrollera att det finns en gren med det angivna grennamnet och detta problem löste vi på följande sätt:

Event event = tournament.findEventByName(eventName);  
if (event == null) {  
 System.*out*.println("There is no such event");  
 return;  
}

Denna finder-metod finns även den i Tournament-klassen och ser ut på följande sätt:

public Event findEventByName(String name) {  
 for (Event e : events){  
 if (e.getName().equals(name)) {  
 return e;  
 }  
 }  
 return null;  
}

Precis som i det tidigare fallet då programmet frågade efter ett startnummer och inte sedan hittade en deltagare med det angivna startnumret så returneras null även i detta fall om det skulle vara så att grenen inte finns registrerad. Detta gör det även lätt för oss att lösa problemet med att det eventuellt inte skulle finnas en gren med det angivna namnet genom att kontrollera om det finns null-värden. Om grenen som returnerades skulle vara null skriver programmet ut att det inte finns någon gren med det angivna grennamnet och returnerar vilket tar användaren tillbaka till startmenyn.

Om både deltagaren och grenen redan är registrerade så vill vi att programmet ska fråga användaren efter vad resultatet är. Dock så vill vi inte att programmet ska acceptera resultat som är mindre än 0 vilket vi löste genom en while-loop.

float score = -1;  
while (score < 0) {  
 System.*out*.print("Results for " + participant.getFullName() + " from "  
 + team.getName() + " in " + eventName + ": ");  
 score = *scanner*.nextFloat();  
 *scanner*.nextLine();  
  
 if (score < 0) {  
 System.*out*.println("Error, result must be 0 or bigger. Please try again.");  
 }  
}

Vi valde då att sätta den float som sparar resultatet till (-1), och detta gjorde vi för att vi ville komma in i while-loopen. Så länge som resultatet som användaren försöker registrera är mindre än 0 så kommer while-loopen att fortsätta. Detta gör att programmet först kommer att skriva ut ”Results for *deltagarens för- och efternamn* from *lagets namn* in *grenens namn*:” och sedan sparar det decimaltal som användaren skriver in i variablen score. Om det angivna resultatet är 0 eller större tar detta programmet ur while-loopen och programmet fortsätter med nästa steg, men om resultatet är mindre än 0 så kommer programmet att skriva ut ”Error, result must be 0 or bigger. Please try again.” och loopen börjar om och programmet skriver ytterligare en gång ”Results for *deltagarens för- och efternamn* from *lagets namn* in *grenens namn*:” och fortsätter så tills det angivna resultatet är minst 0.

När ett godkänt resultat är angivet så måste programmet spara undan detta resultat. Detta gör vi genom att skapa en ny instans av klassen Result.

Result result = new Result(startnumber, eventName, score);

Programmet måste även kontrollera att användaren inte försöker spara undan fler resultat för en gren än vad som är max antal tillåtna resultat. Detta kontrollerar vi på följande sätt:

if (!event.addResult(result)) {  
 System.*out*.println("Too many attempts!");  
 return;  
}

För att kunna förstå vad som händer i denna if-sats så måste vi gå in i Event-klassen och se vad som händer där. Det som händer där är följande:

public boolean addResult(Result result) {  
 int i = 0;  
 for (Result r: this.results){  
 if (r.getStartnumber() == result.getStartnumber()){  
 i++;  
 }  
 }  
 if (i<tries) {  
 results.add(result);  
 return true;  
 } else {  
 return false;  
 }  
   
}

Detta är en boolean som antingen kommer att returnera true eller false. Den kommer att returnera true om det gick att lägga till resultatet i listan (det vill säga om inte antalet försök är lika med maximalt antal försök) eller false om det inte gick. Detta kommer att medföra att om false returneras så kommer programmet att säga till användaren att maximalt antal försök redan är uppnått.

# Funktonen resultatlista för lagen

Samma som ovan, fast för funktionen *resultatlista för lagen*.

Klassen som hanterar resultatlistan är en relativt kort klass och det ända som sker i klassen CommandTeams är att den kallar på två metoder som finns i Tournament-klassen. Detta är eftersom att all information som krävs för att få ut en resultatlista för lagen finns sparade i Arrayer i Tournament-klassen. Detta är hela CommandTeams-klassen:

public void execute (Tournament tournament) {  
 tournament.updateMedalCounters();  
 tournament.printMedalStandings();  
  
 return;  
   
}

Den första metoden räknar ut hur många medaljer varje lag har fått. Detta gör vi genom en for-loop som går igenom varje lag. Eftersom att detta är en metod som ska uppdatera alla medaljer så måste programmet börja med att nollställa alla counters.

public void updateMedalCounters() {  
 for (Team t : this.teams) {  
 t.resetMedals();  
 for (Event e : this.events) {  
 ArrayList<CalculatedResult> calcResults = findCalculatedResultsForEventName(e.getName());  
 for (CalculatedResult c : calcResults) {  
 int startnumber = c.getParticipant().getStartnumber();  
 if (c.getCalculatedPosition() == 1 && t.equals(findTeamByParticipantStartnumber(startnumber))) {  
 t.addGold();  
 }  
 if (c.getCalculatedPosition() == 2 && t.equals(findTeamByParticipantStartnumber(startnumber))) {  
 t.addSilver();  
 }  
 if (c.getCalculatedPosition() == 3 && t.equals(findTeamByParticipantStartnumber(startnumber))) {  
 t.addBrons();  
 }  
 }  
 }  
 }  
}

Alla medaljer sparas sedan under i intar som finns i klassen Team. Metoden resetMedals ser ut på följande sätt:

public void resetMedals() {  
 this.bronsCounter = 0;  
 this.silverCounter = 0;  
 this.goldCounter = 0;  
}

Det som sedan sker är att för varje gren så gås en till for-loop igenom där alla resultat gås igenom och för varje första, andra och tredje plats så läggs detta till i dess respektive intar. Denna kod ser ut på följande sätt:

public void addGold() {  
 this.goldCounter++;  
}  
  
public void addSilver() {  
 this.silverCounter++;  
}  
  
public void addBrons() {  
 this.bronsCounter++;  
}

När alla lag och resultat har gåtts igenom så börjar nästa metod i CommandTeams-klassen körs. Den

# Normalisering av namn

Eftersom att en del av uppgiften gick ut på att vi ville undvika kodupprepning valde vi att lösa detta genom att skapa en metod som hanterar normalisering av namn. Denna normalisering använde vi oss av i samtliga fall förutom när programmet skulle skriva ut ett meddelande. Normaliseringsmetoden som vi skrev ser ut på följande sätt:

protected static String normaliseName(String originalName){  
 String trimmedName = originalName.trim();  
 String firstLetter = trimmedName.substring(0, 1);  
 String firstLetterUpperCase = firstLetter.toUpperCase();  
 if (originalName.length()==1) {  
 return firstLetterUpperCase;  
 }   
 String theRest = trimmedName.substring(1);  
 String theRestLowerCase = theRest.toLowerCase();  
 return firstLetterUpperCase + theRestLowerCase;   
}

Denna metod skrev vi i Command-klassen och eftersom att vi endast använder oss av normalisering av namn i de övriga command-klasserna och alla dessa ärver av Command-klassen så gjorde vi denna till protected istället för public.

Det första som metoden gör är att ta bort begynnande och avslutande mellanslag. Det som den sedan gör är att använda sig av substring och genom detta göra den första bokstaven till en versal. Efter detta så används substring ytterligare en gång och genom detta kan programmet ändra den resterande delen av strängen till små bokstäver. Sedan returneras dessa två nya strängar tillsammans vilket resulterar i dett normaliserat namn.

Ett exempel på när denna metod används är i CommandAddEvent-klassen:

name = Command.*normaliseName*(name);

# Arrayer och ArrayList

Den datasamlingsklass som vi använt oss av mest är ArrayList. Denna ansåg vi var lämpligast i de flesta fall eftersom att vi i ytterst få fall vet i förväg hur lång listan kommer att kunna bli och eftersom att det krävs att man anger ett maxvärde vid användning av vanliga Arrayer ansåg vi att dessa inte var lämpliga i de flesta fall. Vi använder oss exempelvis av en ArrayList i klassen Team. När vi skapar en ny instans av klassen Team så skapar programmet automatiskt med hjälp av en konstruktor en ny ArrayList där programmet sparar undan alla lagets deltagare och detta ser ut på följande sätt:

private ArrayList<Participant> participants = new ArrayList<Participant>();

Sedan använder programmet sig av denna ArrayList varje gång som den måste komma åt deltagare. Detta sker ofta genom att vi skapat en get-metod som ser ut på följande sätt:

public ArrayList<Participant> getParticipants() {  
 return this.participants;  
}

Denna metod returnerar hela listan med alla det specifika lagets deltagare. Programmet var även tvunget att kunna ta bort och lägga till deltagare i denna lista och detta lösta vi med följande metoder:

public void addParticipant(Participant participant){  
 this.participants.add(participant);  
}  
  
public void removeParticipant(Participant participant){  
 this.participants.remove(participant);  
}

Vi ansåg att eftersom att det inte finna något sätt att veta hur pass många deltagare användaren kommer att vilja registrera i programmet så var det lättare att välja att arbeta med ArrayList i detta fall än en vanlig Array. Detta för att undvika en bug med att *index out of bounds* om en användare plötsligt väljer att registrera fler deltagare än vad som finns tillgängligt i den specifika Arrayen.

Vi resonerade på liknande sätt när vi valde att programmet skulle hantera data i ArrayLists i alla de övgriga fallen utom ett. Den klass då vi valde att programmet skulle hantera datan i en Array istället för en ArrayList var i klassen Event i metoden removeResultsByParticipantStartnumber. Denna metod ser ut på följande sätt:

public void removeResultForParticipantByStartnumber (int startnumber) {  
 int[] matchingIndexes = new int[tries];  
 int counter = 0;  
 for (int i = 0 ; i < results.size() ; i++) {  
 Result r = results.get(i);  
 if (r.getStartnumber() == startnumber){  
 matchingIndexes[counter] = i;  
 counter++;  
 }  
 }  
 for (int i = counter - 1 ; i > -1 ; i--) {  
 results.remove(matchingIndexes[i]);  
 }  
}

I klassen Event så finns det en variabel som vi har kallat för tries. Denna variabel är en int som sparar undan det maximalt antal försök som är tillåtet för den specifika grenen. Eftersom att programmet inte kommer att tillåta att en användare försöker spara fler resultat för en deltagare än det maximalt tillåtna antalet försök så vet vi att programmet maximalt kommer att behöva spara undan data i en Array som är maximalt lika stor som antalet maximalt tillåtna försök för den specifika grenen. Detta är på grund av att programmet kommer att behöva ta bort lika många resultat som är det maximalt antal försök som är tillåtna eller färre.

# Statiska variabler och metoder

Vi använde oss av en del statiska metoder utöver den obligatoriska i main-metoden. Alla statiska metoder som vi skrivit förutom main-metoden finns i Command-klassen. Detta är på grund av att det inte är logiskt att vi ska vara tvungna att skapa nya instanser av Command-klassen eftersom att allt denna gör är att hantera de olika kommandona som programmet får in och returnerar nya instanser av specifika klasser för varje kommando. Ett exempel då vi använder oss av en statisk-metod är:

public static Command getCommand(Tournament tournament) {  
 System.*out*.print("Command> ");  
 String commandString = *scanner*.nextLine();  
 commandString = *normaliseName*(commandString);  
  
   
 if (commandString.equals("Add event")) {  
 return new CommandAddEvent();  
 } else if (commandString.equals("Exit")) {  
 return new CommandExit();  
} else if (commandString.equals("Add participant")) {  
 return new CommandAddParticipant();  
} else if (commandString.equals("Add result")) {  
 return new CommandAddResult();  
} else if (commandString.equals("Participant")) {  
 return new CommandParticipant();  
} else if (commandString.equals("Remove participant")) {  
 return new CommandRemoveParticipant();  
} else if (commandString.equals("Teams")) {  
 return new CommandTeams();  
} else if (commandString.equals("Reinitialize")) {  
 return new CommandReinitialize();  
} else if (commandString.startsWith("Message")) {  
 return new CommandMessage(commandString);  
} else if (commandString.equals(*getEventUsingString*(commandString, tournament))) {  
 return new CommandResultsPerEvent(commandString);  
}  
  
 return new CommandUnknown();  
 }

Att vi valde att göra denna metod till statisk medförde att flera andra metoder var tvunga att vara statiska då man inte kan anropa en icke-statisk metod från en statisk metod. Ett exempel på detta är följande:

public static String getEventUsingString(String commandString, Tournament tournament) {  
 String normalisedName = *normaliseName*(commandString);  
 Event event = tournament.findEventByName(normalisedName);  
 if (event != null) {  
 String eventName = event.getName();  
 return eventName;  
 }  
 return null;  
}

# Reflektion

I det hela gick vårt projekt väldigt bra. Det var en ytterst lärorik uppgift och vi har lärt oss mycket. Det var en väldigt svår uppgift, framförallt att komma igång med men när vi väl hade kommit igång ordentligt så gick det väldigt bra. Självklart så fastnade vi på flera delar och det var vissa gånger som koden inte fungerade som den skulle, men detta medförde även att vi lärde oss mycket då detta tvingade oss att söka upp information om vad som hade blivit fel och hur vi skulle kunna lösa det. En sak som vi tror skulle ha kunnat vara bra skulle vara om uppgiften med hundregistret hade varit lite större och denna lite mindre.

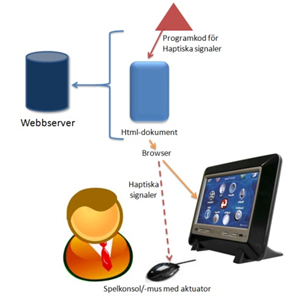
Vi valde att lägga upp vårt program i relativt många klasser för att vi ville ha en tydligare kod som var lätt att hitta i. Vi funderade på att lägga upp alla kommandon i Tournament-klassen men vi bestämde oss att vi ville ha en tydligare kod som skulle vara lättare att hitta i även fast detta gjorde att koden blev mer komlicerad att skriva och det var mycket som vi var tvunga att söka information om innan vi kom igång ordentligt.

Vi valde även att använda oss av arv i alla kommando-klasserna trotts att detta inte ingick i kursen för att vi ville kunna göra en del metoder som användes i många kommando-klasser till protected istället för public. Ett exempel på detta var i metoden där programmet normaliserar strängar. Eftersom att de strängar som vi ville att programmet skulle normalisera endast kom in till systemet genom någon av kommando-klasserna så var detta en bra metod som skulle vara protected istället för public. Dock så hade vi kunnat göra de metoder som nu är protected till private om vi hade hanterat alla kommandon i en klass. Dock så ansåg vi att det var bättre att ha flera klasser och därför göra koden mer lättöverskådlig.

När det gäller namngivningen så försökte vi döpa alla metoder och variabler till namn som var lätta att förstå. Detta hjälpte när vi arbetade med att få tag på information från andra klasser eller ta ut information från ArrayLists. Eftersom att vi hade tydliga namn på alla get-metoder så gjorde det kodningen lättare då detta medförde att det var lätt att hitta rätt metod som vi ville använda i just det specifika fallet.

# Mallanvändningen

Detta avsnitt tar upp några vanliga frågor/problem med mallandvändningen. Det ska naturligtvis plockas bort i den version ni lämnar in.



1. Systemillustration

## Bilder

Bilder ska tillföra något till texten, vara tydliga, ha en förklarande bildtext, och sitta på ett vettigt ställe i förhållande till texten. De bör också refereras till i texten.

### Rubriknivå 3

Rubriknivå tre och neråt bör man normalt undvika att använda. Ibland går det inte, men ofta är det ett tecken på att man delar upp sitt dokument i för små bitar.

## Referenser

De senaste Word-versionerna har ett referenshanteringssystem inbyggt. Det är lite rudimentärt, men fungerar hyggligt, och om man inte har tillgång till något bättre så bör man naturligtvis använda det. På OOP använder vi IEEE:s format med numrerade referenser inom hakparanteser. En referens till kursboken ska alltså se ut så här: [1], eller så här om man vill ha med sidanvisningar: [1, p. 4].

Om ni inte refererar till några källor ska ni plocka bort källförteckningen.

## Kod

Kod i dokumentet ska vara läsbar och helst formaterad som i en utvecklingsmiljö. Ett tips är att inte använda skärmdumpar. De blir nästan alltid oläsliga, och det går inte att få ut källkoden från dem om man skulle vilja.

## Avsnitt i mallen

Den text som finns i mallen från början är avsedd bara för att illustrera användningen och ska plockas bort innan ni lämnar ni.

Källförteckning

|  |  |
| --- | --- |
| [1] | J. Lewis och W. Loftus, Java Software Solutions, sjunde upplagan red., Pearson Education, 2012. |