Inlämningsuppgift 3

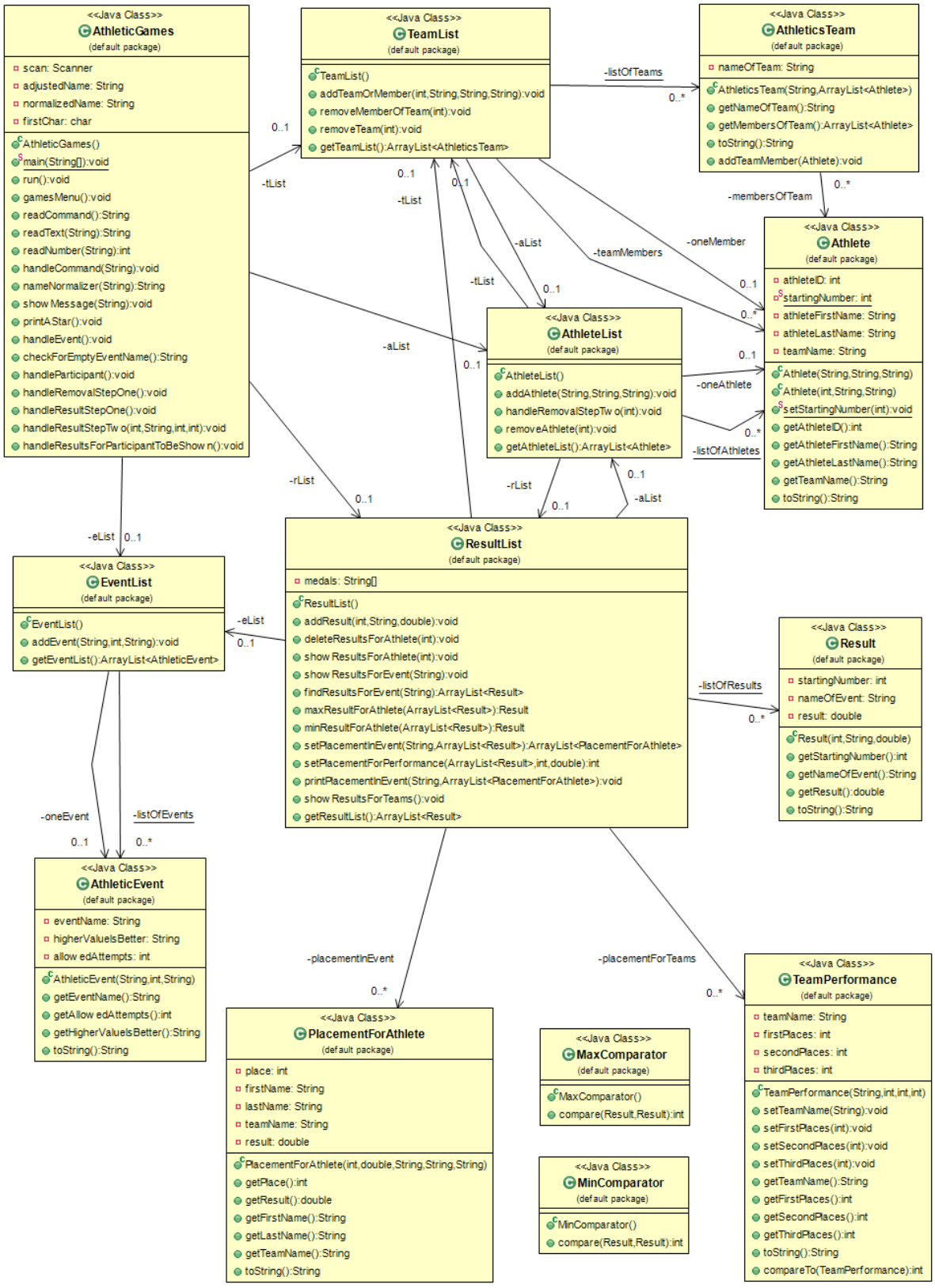
Grupp: G019
Daniel Knöös (dakn3990)
Nasimul Hasan (naha1113)
Reza Rohani Makvandi (rero6268)

Objektorienterad programmering

Höstterminen 2015

Kursansvarig: Henrik Bergström

1 Design


<<Java Class>>
AthleticGames
(default package)
scan: Scanner
adjustedName: String
normalizedName: String
firstChar: char
AthleticGames()
main(String[]):void
run():void
gamesMenu():void
readCommand():String
readText(String):String
readNumber(String):int
handleCommand(String):void
nameNormalizer(String):String
show Message(String):void
printA Star():void
handleEvent():void
checkForEmptyEventName():String
handleParticipant():void
handleRemovalStepOne():void
handleResultStepOne():void
handleResultStepTw o(int,String,int,int):void
handleResultsForParticipantToBeShow n():void
<<Java Class>>
TeamList
(default package)
TeamList()
addTeamOrMember(int,String,String,String):void
removeMemberOfTeam(int):void
removeTeam(int):void
getTeamList():ArrayList<AthleticsTeam>
<<Java Class>>
AthleticsTeam
(default package)
nameOfTeam: String
AthleticsTeam(String,ArrayList<Athlete>)
getNameOfTeam():String
getMembersOfTeam():ArrayList<Athlete>
toString():String
addTeamMember(Athlete):void
<<Java Class>>
Athlete
(default package)
athleteID: int
startingNumber: int
athleteFirstName: String
athleteLastName: String
teamName: String
Athlete(String,String,String)
Athlete(int,String,String)
setStartingNumber(int):void
getAthleteID():int
getAthleteFirstName():String
getAthleteLastName():String
getTeamName():String
toString():String
<<Java Class>>
AthleteList
(default package)
AthleteList()
addAthlete(String,String,String):void
handleRemovalStepTw o(int):void
removeAthlete(int):void
getAthleteList():ArrayList<Athlete>
<<Java Class>>
ResultList
(default package)
medals: String[]
ResultList()
addResult(int,String,double):void
deleteResultsForAthlete(int):void
show ResultsForAthlete(int):void
show ResultsForEvent(String):void
findResultsForEvent(String):ArrayList<Result>
maxResultForAthlete(ArrayList<Result>):Result
minResultForAthlete(ArrayList<Result>):Result
setPlacementInEvent(String,ArrayList<Result>):ArrayList<PlacementForAthlete>
setPlacementForPerformance(ArrayList<Result>,int,double):int
printPlacementInEvent(String,ArrayList<PlacementForAthlete>):void
show ResultsForTeams():void
getResultList():ArrayList<Result>
<<Java Class>>
Result
(default package)
startingNumber: int
nameOfEvent: String
result: double
Result(int,String,double)
getStartingNumber():int
getNameOfEvent():String
getResult():double
toString():String
<<Java Class>>
EventList
(default package)
EventList()
addEvent(String,int,String):void
getEventList():ArrayList<AthleticEvent>
<<Java Class>>
AthleticEvent
(default package)
eventName: String
higherValueIsBetter: String
allow edAttempts: int
AthleticEvent(String,int,String)
getEventName():String
getAllow edAttempts():int
getHigherValueIsBetter():String
toString():String
<<Java Class>>
PlacementForAthlete
(default package)
place: int
firstName: String
lastName: String
teamName: String
result: double
PlacementForAthlete(int,double,String,String,String)
getPlace():int
getResult():double
getFirstName():String
getLastName():String
getTeamName():String
toString():String
<<Java Class>>
MaxComparator
(default package)
MaxComparator()
compare(Result,Result):int
<<Java Class>>
MinComparator
(default package)
MinComparator()
compare(Result,Result):int
<<Java Class>>
TeamPerformance
(default package)
teamName: String
firstPlaces: int
secondPlaces: int
thirdPlaces: int
TeamPerformance(String,int,int,int)
setTeamName(String):void
setFirstPlaces(int):void
setSecondPlaces(int):void
setThirdPlaces(int):void
getTeamName():String
getFirstPlaces():int
getSecondPlaces():int
getThirdPlaces():int
toString():String
compareTo(TeamPerformance):int
-listOfTeams
0..*
0..1
-tList
-membersOfTeam
-oneMember
-teamMembers
-aList
-oneAthlete
-listOfAthletes
-rList
-eList
-oneEvent
-listOfEvents
-listOfResults
-placementInEvent
-placementForTeams

Designen har under arbetets gång förändrats en hel del. Ingen av oss tre killar som arbetade tillsammans på projektet hade någon större erfarenhet av programmering – två av oss var totala nybörjare på området. Detta tog sig till exempel uttryck i att vi i början inte förstod hur man skapar metoder i en klass som kan anropa andra metoder i andra klasser. Då är det inte lätt att lägga upp en objektorienterad strategi för projektet. Men i takt med att arbetet och studierna fortlöpte såg vi nya designmöjligheter och anpassade programmet därefter. Snart sagt varje huvudfunktion som kan sägas vara av eget slag har slutligen fått egen klass.

Huvudklassen (AthleticGames) har hand om all inläsning av data. Då fann vi det lämpligt att lägga metoden för normalisering av namn (nameNormalizer()) som läses in här. Då huvudklassen även sköter hanteringen av kommandona tyckte vi att meddelandefunktionen (showMessage()), som har viss koppling till kommandona, också borde ligga där. Sedan har vi fyra olika typer av huvudobjekt i hanteringen av idrottstävlingen – gren, deltagare, lag och resultat – som fått egna klasser (AthleticEvent, Athlete, AthleticsTeam och Result). De dataregister som ska lagra dessa objekt har vi sett som skilda objekt och därför även gett dem egna klasser (EventList, AthleteList, TeamList och ResultList). Alla metoder som utför operationer på dessa dataregister har förlagts dit. För avgörandet av vilket värde i deltagarnas prestationer som är bättre än annat, beroende på huruvida ett högre värde är bättre eller ej, har vi använt gränsnittet Comparator och stoppat in metoderna för detta i egna klasser (MaxComparator och MinComparator). Slutligen har vi funnit det användbart med två specialklasser. Den ena ser placeringen som en deltagare uppnår i en given gren som ett objekt och egen klass (PlacementForAthlete) utifrån vilken man enkelt kan skriva ut placeringslistor för grenen ifråga. Den andra håller reda på antalet medaljer som ett givet lag uppnår i tävlingen (TeamPerformance), från vilket man likaledes enkelt kan skriva ut lagresultaten. Metoderna som utför operationer på dessa klasser har stoppats in i klassen för resultatlistor (ResultList). Vårt program kom därför att bestå av totalt tretton klasser.

Vad gäller redovisningen av vilka delar vi ”har gjort själva och vilka som är tagna från Java eller från andra bibliotek” är vi lite osäkra på vad som efterfrågas. Alla förutnämnda klasser har vi skapat själva, men till dom har vi importerat ett antal klasser från Java. Klassen *ArrayList* från java.util kommer till användning i AthleteList, AthleticGames, AthleticsTeam, EventList, ResultList och TeamList. Klassen *Iterator* finns med i ResultList och TeamList. ResultList innehåller även klassen *Collections* från java.util. Klassen *Scanner* från java.util finns bara i AthleticGames. Här finns också klassen *Locale* från java.util. Därutöver används klassen *Comparator* från java.util i två klasser: MaxComparator och MinComparator. TeamPerformance-klassen implementerar även gränssnittet *Comparable*.

För namnen på klasserna och variablerna har vi förutom språket valt att prioritera klarhet rörande deras funktion snarare än korthet i textmässig längd.

2 Funktionen lägg till resultat

Programmet är utformat så att när det körs efterfrågas först ett kommando angående vad man vill göra. För att lägga till ett resultat får man information från en meny att man ska skriva ”add result”. När man gör detta och därefter trycker på ”Enter” så kollar programmet först ifall den inmatade strängen överensstämmer med ett grennamn. Om den inte gör det hoppar programmet in i en switch-struktur där inmatningen matchas till det alternativ som är benämnt ”add result”. Här anropas en metod som kallas handleResultStepOne(). Denna metod sköter det mesta av inläsningen av datan härför och är förlagd i huvudklassen. Här ligger också nästa steg i hanteringen av resultaten, en metod med namnet handleResultStepTwo(). Denna sköter kontrollen för antalet försök som en deltagare har till förfogande i grenen ifråga och tar emot datan för själva prestationen. När all data för resultatet är insamlat anropas en sista metod, addResult(), som ligger i en annan klass som kallas ResultList. Här finns en ArrayList vid namn listOfResults som ska ta emot registreringen av resultatet. Första steget i datainläsningen är kodat på följande vis:

```
public void handleResultStepOne()
   {
      int startingNumber = readNumber("State starting number of participant
for which there is a result: ");
      ArrayList<Integer> athletes = new ArrayList<>();
      for (Athlete a : aList.getAthleteList()){
            if (!(aList.getAthleteList().isEmpty())){
                  athletes.add(a.getAthleteID());
            }
      }
      if ((!athletes.contains(startingNumber)) || startingNumber < 100){
            System.out.println("There is no participant with that number.");
            return;
      }
      String namedEvent = readText("State the name of the event: ");
      if (namedEvent.equals("")|| namedEvent.trim().isEmpty()){
            System.out.println("Error. You must state a name.");
            return;
      }
      String normalizedEventName = nameNormalizer(namedEvent);
      ArrayList<String> events = new ArrayList<>();
      for (AthleticEvent e : eList.getEventList()){
            events.add(e.getEventName());
      }
      if (!events.contains(normalizedEventName)){
            System.out.println("Error. You must state an event that is held
in these Athletic Games.");
            return;
      }
      int allowedAttempts = 0;
      for (AthleticEvent e : eList.getEventList()){
            if (e.getEventName().equals(normalizedEventName)){
                  allowedAttempts = e.getAllowedAttempts();
            }
      }
      ArrayList<Result> resultsForAthlete = new ArrayList<>();
      for (Result r : rList.getResultList()){
```

```
            if ((r.getStartingNumber() == startingNumber) && (r.getNam-
eOfEvent().equals(normalizedEventName))){
                  resultsForAthlete.add(r);
            }
      }
      int numberOfResultsInEvent = resultsForAthlete.size();
      handleResultStepTwo(startingNumber, normalizedEventName, al-
lowedAttempts, numberOfResultsInEvent);
   }
```

3 Funktionen resultatlista för lagen

Denna funktion är naturligtvis den mest omfattande i hela programmet och vars fulla redovisning följaktligen skulle uppta stor plats i denna rapport. Vi tror därför att det är tillräckligt att här försöka beskriva huvudlinjen däri. Resultatlistan för lagen tas huvudsakligen fram av en metod som genom kommandot ”teams” anropas från huvudklassen och som kallas showResultsForTeams(). Denna metod ligger i ResultList-klassen och har följande struktur:

```
public void showResultsForTeams()
  {
     ArrayList<String> eventsWithResults = new ArrayList<>();
     for (Result r : listOfResults){
        if (!eventsWithResults.contains(r.getNameOfEvent())){
            eventsWithResults.add(r.getNameOfEvent());
       }
    }
    for (AthleticsTeam aT : tList.getTeamList()){
        String aTeam = aT.getNameOfTeam();
        TeamPerformance tPerformance = new TeamPerformance(aTeam, 0, 0, 0);
            placementForTeams.add(tPerformance);
    }
    ArrayList<Result> resultsForEvent = new ArrayList<>();
    ArrayList<PlacementForAthlete> placementInEvent = new ArrayList<>();
    for (String e : eventsWithResults){
        resultsForEvent = findResultsForEvent(e);
        placementInEvent = setPlacementInEvent(e, resultsForEvent);
        for (PlacementForAthlete p : placementInEvent){
            String teamWithResult = p.getTeamName();
            for (TeamPerformance tP : placementForTeams){
                if (tP.getTeamName().equals(teamWithResult)){
                    if (p.getPlace() == 1){
                        int places = tP.getFirstPlaces();
                        tP.setFirstPlaces(places + 1);
                    }
                    else if (p.getPlace() == 2){
                        int places = tP.getSecondPlaces();
                        tP.setSecondPlaces(places + 1);
                    }
                    else if (p.getPlace() == 3){
                        int places = tP.getThirdPlaces();
                        tP.setThirdPlaces(places + 1);
                    }
                }
             }
        }
     }
     placementForTeams.sort((t1, t2) -> t2.compareTo(t1));
     System.out.println();

     for (int x = 0; x < medals.length; x++){
         System.out.print(medals[x] + " ");
     }
      System.out.println("TEAM");
```

```
        for (int i = 0; i < 25; i++){
            System.out.print("*");
        }
        System.out.println();
        for (TeamPerformance teamP : placementForTeams){
            int gold = teamP.getFirstPlaces();
            int silver = teamP.getSecondPlaces();
            int bronze = teamP.getThirdPlaces();
            String teamName = teamP.getTeamName();
            System.out.println(" " + gold + "  " + silver + "  " + bronze + " "
        + teamName);
        }
        placementForTeams.clear();
        System.out.println();
    }
```

Eftersom det kan finnas fler grenar registrerade i tävlingen än det finns resultat för börjar metoden med att samla in alla grenar för vilka det finns resultat och listar dessa i en ArrayList vid namn *eventsWithResults*. Detta utan dubletter då resultaten för given gren bara ska visas en gång.

Nästa steg är att skapa en lista för alla deltagande lag som ska hålla reda på deras medaljskörd i tävlingarna. Den listan benämns *placementForTeams* och innehåller sålunda objekt från TeamPerformance-klassen.

Sedan gäller det att samla in resultaten för varje gren man har tävlat i. Detta sker med hjälp av metoden findResultsForEvent(). Datan man får fram på detta sätt läggs i en annan ArrayList vid namn *resultsForEvent*. För att avgöra placeringarna för deltagarna i grenen ifråga skickas sistnämnda lista med som parameter tillsammans med namnet på grenen till metoden setPlacementInEvent(). Denna metod lägger placeringarna i en tredje ArrayList kallad *placementInEvent*, som innehåller föremål av annan typ än de föregående, nämligen objekt ur PlacementForAthlete-klassen. Objekten i denna senare lista ligger alltså i den ordning som deltagarna ifråga har hamnat med sina resultat, med den bästa prestationen på index 0, vars attribut av datatypen int för *place* därmed har satts till 1.

Därefter körs en loop med en formel för att hitta alla lag med placeringar på första, andra och tredje plats enligt placementInEvent-listan för varje gren och räkna upp alla sådana positioner i *placementForTeams*-listan. När denna räkning är färdig återstår att sortera lagresultaten primärt efter antalet guld, sekundärt efter antalet silver och sist enligt antalet brons. Detta görs helt enkelt med hjälp av ett lambda-uttryck för *placementForTeams*-listan, som begagnar sig av en compareTo()-metod i TeamPerformance-klassen.

Resten av koden här har huvudsakligen med utskriften av lagresultaten att göra. Medaljdesignationerna hämtas från en Array vid namn *medals*, vilka skrivs ut som rubriker på en rad, följt av rubriken ”TEAM”. Därefter skrivs en linje med stjärnor ut och därunder listas lagens medaljskörd efter antalet ”GOLD”, ”SILVER” och ”BRONZE”, enligt den sorterade *placementForTeams*-listan. Det sista som händer är att *placementForTeams*-listan töms på sitt innehåll då användaren efter ha sett en första resultatlista för lag kanske önskar ta bort någon deltagare och vederbörandes resultat. Detta kan påverka resultatlistan för lag som därför måste kunna räknas ut på nytt från noll för lagens vidkommande.

4 Normalisering av namn

För att undvika koduppрepning i normaliseringen av namn har vi helt enkelt skapat en särskild nameNormalizer()-metod placerad i huvudklassen (AthleticGames) som anropas var gång ett namn behöver normaliseras. Detta sker omkring ett halvdussin gånger. Metoden för normaliseringen ser ut så här:

```
public String nameNormalizer(String inputText)
	{
		adjustedName = inputText.trim().toLowerCase();
		firstChar = Character.toUpperCase(adjustedName.charAt(0));
		normalizedName = firstChar + adjustedName.substring(1);
		return normalizedName;
	}
```

Normaliseringen kan på detta sätt i varje enskilt fall ske med hjälp av blott två kodrader, vilket är exemplifierat i svaret på fråga 2.

5 Arrayer och ArrayList

I vårt program har bara använt oss av *en* array. Detta för att minimera problematiken med förändring av storleken på datasamlingar när exempelvis element ska läggas till eller tas bort från dem. Även kodningen för att kontrollera innehållet i en tom array (inför exempelvis en dublettkontroll) fann vi knepig. Vi såg därför bara naturlig användning av en array när innehållet av datasamlingen, efter det att den hade initierats, är oföränderligt. I detta fall namnen på de olika valörerna av medaljer. Arrayen för dessa har således förlagts i den klass som vi har satt att hålla reda på resultatlistorna: ResultList.

ArrayLists, däremot, har vi begagnat flitigt. Denna typ av datasamling är betydligt enklare att arbeta med an arrayer, så de förekommer på ett flertal ställen i vårt program. Inte bara som ursprungliga datasamlingar vars innehåll är föränderligt, utan även när nya behöver skapas med innehåll som utgör delar av de ursprungliga. Som exempel kan vi nämna en ArrayList, här kallat *chosenResultList*, som samlar alla resultat gjorda i en given gren. Dessa har sammanställts från en ”ursprunglig” ArrayList kallad *listOfResults* (från ResultList-klassen) där alla grenars resultat registreras vid inmatning av resultaten i tävlingen. Datasamlingen *chosenResultList* har sedan använts som utgångspunkt för att bestämma värdet eller sorteringen av resultaten i grenen ifråga.

6 Statiska variabler och metoder

Vi har använt oss av fem statiska variabler och en statisk metod förutom main i vårt program. En av de statiska variablerna är ett attribut av datatypen int vid namn *startingNumber* i Athlete-klassen. Den alstrar ett nytt ID-nummer för varje ny deltagare som registreras. Då denna håller reda på hur många instanser av klassen som har skapats har den en funktion som är kopplad till klassen som helhet och inte någon enskild instans. Den har därför gjorts statisk. För att komma åt denna variabel behövdes en metod som vi har kallat getStartingNumber(), också placerad i Athlete-klassen. Eftersom variablen är statisk har metoden för att komma åt den också gjorts statisk.

Övriga statiska variabler är alla av datasamlingstypen ArrayList: *listOfEvents*, *listOfAthletes* *listOfTeams* och *listOfResults*. Dessa är förlagda i klasser vars huvudfunktion är att utgöra datasamlingar, vilket också indikeras genom dessa klassers namn: EventList, AthleteList, TeamList respektive ResultList. Det skapas bara en instans av vardera registersort här. Vi fann det därför passande att definiera just dessa ArrayLists som en klassfunktion och därmed som statiska. Funktionen av många metoder i programmet underlättas uppenbarligen också av att dessa register är modifierade på det sättet. Annars får man problem som skulle kräva mer kod att hantera.

7 Reflektion

Som tidigare sagt hade ingen av oss tre killar som arbetade tillsammans på uppgiften rörande idrottstävlingen någon större erfarenhet av programmering innan projektet startade. Största problemet var nog att få grepp om hur metoder fungerade och hur man skapade dom. Svårigheter tillkom när vi inte hann färdigställa programmet under ordinarie projektperiod, för då kolliderade fortsättningen med andra kurser under vårterminen. Här hann man glömma bort en del saker man behövde kunna för att lösa uppgiften. Så det dröjde till ”sommarlovet” innan alla bitar föll på plats.

Vi har lärt oss mycket om programmering på denna uppgift. Vi kan konstatera att om vi hade haft den kunskap på området vi har idag hade vi kunnat lösa uppgiften inom ordinarie tid. Men det tog oss alltså längre tid än vad som fanns tillgängligt under ordinarie period att nå denna nivå.

En sak som inverkat menligt på detta var ett misstag vi gjorde i studieupplägget. Vi anser att i just programmering är inspelningarna av föreläsningarna i datautbildningen av särskilt betydelsefullt värde. Detta inte enbart för att dessa föreläsningar ofta ger tips som kan vara användbara för att lösa inlämningsuppgifterna, utan även för att de konkretiserar vad ett i förstone tämligen abstrakt och svårgripbart programmeringsspråk innebär och därmed snabbar upp inlärningsprocessen. Att enbart *läsa* en rad förklaringar till hur det fungerar enligt en kursbok ger inte alls samma grepp om saken. Dessutom kan man med inspelningarna när som helst pausa uppspelningen när det är något man inte uppfattade eller förstod i undervisningen, spola tillbaka, fundera i lugn och ro, liksom testa idéerna på egen dator, tills man hänger med igen. Detta förbättrar inlärningsprocessen ytterligare. Men enligt studierådet man fick på denna kurs tycktes värdet på föreläsningarna komma efter värdet av både eget arbete framför dator och kursboken. Därför prioriterade vi dessa studiemedel framför föreläsningarna. Och i alla andra typer av kurser hade denna studiestrategi även räckt långt. Kursböckerna kan här ofta vara bättre än föresläsningarna ur pedagogisk synpunkt. Men för programmering är praktisk demonstration ett effektivare läromedel än läsning av kursbok. Detta insåg vi alltför sent. Inlärningsprocessen gick därför långsammmare än nödvändigt.

Ett annat problem var kopplat till de testfiler som skulle användas. Dessa filer kräver en del funktioner som inte nämns i de officiella kravspecifikationerna. Testfilerna kan exempelvis i en fråga mata in flera felaktiga värden efter varandra för att avsluta med ett korrekt, vilket kräver en while- eller do/while-klausul för att hanteras; men i en annan fråga kan de stoppa in ett enda felaktigt värde för att sedan lämna frågan och ge ett nytt kommando, vilket kräver en enkel if-sats med return. Exakt vilken funktion som testfilerna begärde i ett givet läge kunde man inte lista ut enbart från typen av fråga eller genom de exceptions som ”felaktig” kodning genererade - särskilt knepigt kunde detta vara vid fall där programmet verkade fungera enligt kravspecifikationerna med manuell inmatning. Man var tvungen att studera innehållet i testfilerna själva.

Men denna uppgift var också osedvanligt omfattande och komplext för en grundkurs i något ämne. Och i andra kurser är man i alla fall vanligen bekant med det språk man ska använda för att lösa uppgifterna. Chansen för totala nybörjare i programmering att själva klara av idrottstävlingen under ordinarie period tycks relativt liten. Vilket tillsammans med förvånansvärt avancerade frågor (i relation till exemplen i kursboken och på föreläsningarna) i framför allt kodförståelse på tentamina lär återspeglas i genomströmningen på kursen.

Så för framtida studenters skull hoppas vi att kursledningen dels starkare understryker värdet av de inspelade föreläsningarna, dels detaljerar kravspecifikationerna med hänsyn till testfilerna

på ett tydligare sätt, dels skär ner storleken och/eller komplexiteten på den sista inlämningsuppgiften för kursen lite grann. Alla normalbegåvade studenter ska ha en rimlig chans att slutföra den med normal insats i studietid. Det är första gången under år av högskolestudier i fler ämnen än datavetenskap som vi känner att det finns ett klart behov av att man ser över upplägget för en kurs på universitetet. All utbildning i komplexa ämnen ska, ifall man löper hela linan ut, förr eller senare nå avancerade stadier. God utbildning har nu en väl avvägd takt med vilken man når de olika stadierna. På denna punkt kan OOP, bland de grundkurser i datautbildning vi har tagit del av hittills, förbättras mest.

Detta var våra reflektioner.