Inlämningsuppgift 3

Grupp: 12
Simon Olofsson (siol0547)

Objektorienterad programmering
Höstterminen 2015
Kursansvarig: Henrik Bergström

1 Design

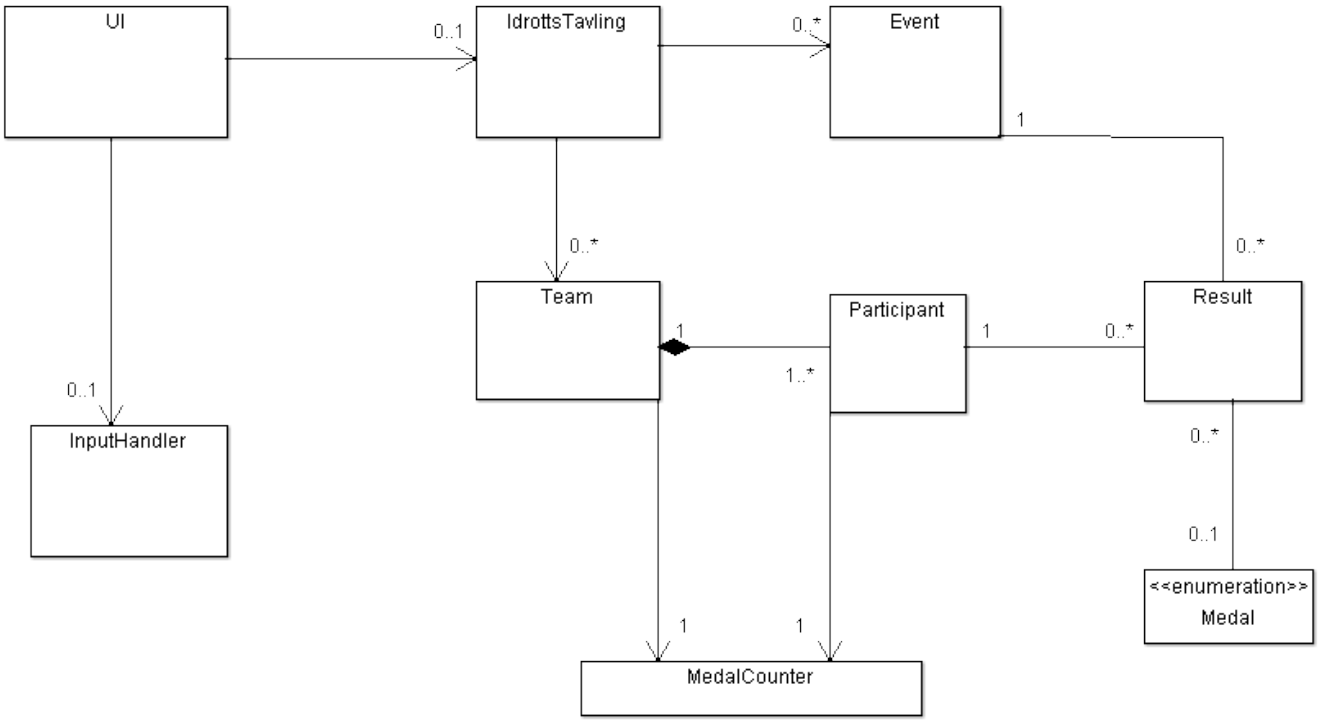

Figure 1: Översiktligt klassdiagram

Ovan ser vi ett översiktligt klassdiagram. På grund av problem med att få klassdiagramsbilderna till en vettig storlek har jag valt att inkludera mer detaljerade utsnitt av klassdiagrammet i ett appendix längst bak. Referenser till dessa diagram i appendixet kommer att ges i texten som följer. Detta gäller endast klassdiagrammet.

Den övergripande designen är framförallt ett resultat av ett försök att upprätthålla så hög cohesion och så låg coupling som möjligt. För det första finns en särskild UI-klass som endast sysslar med att ta in rådata och skicka till en controller som jag valt att benämna IdrottsTavling eftersom varje instans av den kommer att koordinera all data som behövs för en given tävling. Controllerns enda uppgift är att koordinera arbetet mellan de resterande klasserna och sedan returnera rådata till UI-klassen. Som ett resultat av detta bör det inte vara några problem att lägga till ett grafiskt användargränssnitt utan några större ändringar än att gränssnittet kodas och kopplas till controllern. Det är i alla fall visionen.

Jag valde att plocka ut alla metoder som läser in data och lägga dem i en egen klass kallad InputHandler. Den främsta motiveringen är att många av dem är metoder jag kommer att kunna använda i princip alla textbaserade program jag skriver, framförallt de metoder som läser in tal och sedan tar hand om den överblivna newlinen. För återanvändbarhetens skull är det därför smidigt att lägga dem i en egen klass. Det går självklart att lägga fram cohesionargument för valet också. UI-klassen sysslar med att skriva ut meddelanden och att bestämma när saker ska hända, InputHandlern med att processa användarinput så att den blir användbar. Att ha båda funktionerna i samma klass hade blivit tungrott och svåröverskådligt. UI, InputHandler och IdrottsTavling kan ses i figur 7 i appendix.

Controllern har bara direkt tillgång till klasserna Team och Event, detta eftersom kravspecifikationen gör det tydligt att både deltagare och grenar ska kunna läggas till oberoende av varandra. Det faktum att controllern inte har tillgång till Participant-klassen direkt motiveras av creator-mönstret som beskrivs av Craig Larman i Applying UML and Patterns. Larman menar att ansvaret för att instantiera objekt av en viss klass i första hand bör ligga hos den klass som "Contains or compositely aggregates" den klass som ska intantieras [1, p.292]. Eftersom ett lag (representerat av Team-klassen) utgörs av deltagare (representerat av Participant-klassen) och inte kan existera utan sina medlemmar kändes det logiskt att låta Team-klassen snarare än controllern organisera sina medlemmar. Controllern kommer sedan åt dem via Team-klassen. IdrottsTavling, Team och Event kan ses i figur 8 i appendix.

Jag valde att representera de medaljer som lagen ska sorteras efter genom en enum. I en tidigare version av programmet hade jag en regelrätt Medal-klass med associationer till både ett Team och ett Event, där Eventet hade ansvar för att fördela medaljer mellan de olika Teamen varje gång kommandot "teams" körs av användaren. Detta fungerade egentligen rätt bra, det var definitivt logiskt och koden inte nämnvärt mer komplicerad än vad den är nu. Det som störde mig var att klassen i princip bara var en samling associationer och en textsträng som representerade medaljens värde. Eftersom resultat indirekt kan kopplas till både Team och Event kändes det som en smartare lösning att låta medaljen utgöras av en enum med tre tillåtna värden som kopplas till ett specifikt resultat. Event kan på så sätt fortfarande vara ansvarig för att associera resultat med medaljer när kommandot "teams" körs. I slutändan blev koden likvärdig, ansvarsfördelningen densamma och antalet associationer mindre vilket gjorde att det kändes som ett lämpligt designval. Relationen mellan IdrottsTavling, Event, Result, Participant och Medal kan ses i figur 9 i appendix.

Både Participant och Team hade i början två helt separata metoder för att beräkna antal medaljer. Genom att lägga funktionaliteten i en separat klass kallas MedalCounter blev det möjligt att plocka ut vanliga operationer och lägga dem i egna metoder och på så sätt undvika väldigt mycket koduppprepning. Team, Participant och MedalCounter kan ses i figur 10 i appendix.

2 Funktionen lägg till resultat

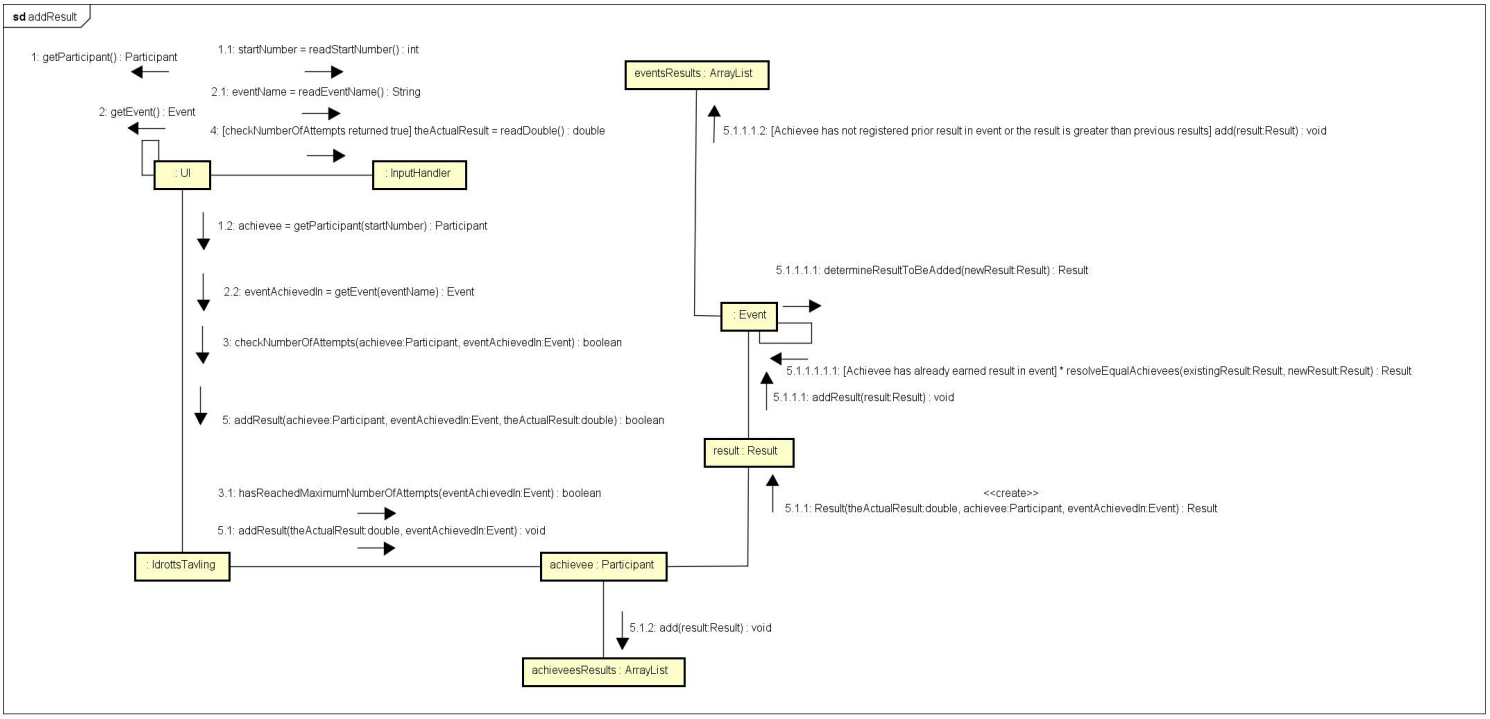

Figure 2: Kommunikationsdiagram över funktionen "Lägg till resultat"

Som vi kan se i kommunikationsdiagrammet ovan sker följande när ett resultat ska läggas till. Numreringen avser punkterna i kommunikationsdiagrammet:

1 - 2 UI anropar metoder på sig själv som i sin tur använder metoder hos InputHandler för att läsa in deltagarens startnummer och grenens namn i form av en int och en String. Sedan hämtar UI ett Participantobjekt samt ett Eventobjekt från IdrottsTavling med hjälp av denna information. Här kontrolleras alla sådana saker som tomma namn, icke giltiga deltagarnummer samt icke-existerande deltagare eller grenar.

3 - 3.1 UI kontrollerar så att inte deltagaren redan registrerat maximalt antal tillåtna försök i grenen. Detta sker genom ett metodanrop till IdrottsTavling som i sin tur gör ett metodanrop till Participantobjektet.

4 Om deltagaren fortfarande får registrera försök läses det aktuella resultatet in.

5 UI anropar addResult()-metoden hos IdrottsTavling och skickar med Participant- och Eventobjekt samt resultatet.

5.1 IdrottsTavling anropar Participantobjektets addResult()-metod.

5.1.1 Participantobjektet skapar ett nytt Resultobjekt och skickar med information om resultatet samt referenser till sig självt och det Eventobjekt som hämtats tidigare.

5.1.1.1 Resultobjektets konstruktor anropar addResult() hos Eventobjektet och skickar med en referens till sig självt.

5.1.1.1.1 Eventobjektet anropar en metod hos sig själv som kontrollerar ifall resultatet verkligen ska läggas till. Om deltagaren som försöker lägga till resultatet inte tidigare har registrerat ett resultat i grenen returneras det resultat som försökte läggas till.

5.1.1.1.1.1 Om den deltagare som försöker lägga till resultatet redan har registrerat ett resultat i grenen anropar Eventobjektet ytterligare en metod för att se vilket av resultaten som är störst. Ifall det gamla resultatet är störst görs ingenting förutom en nullreturn, annars sätts det existerande resultats medalj till null och tas bort ur Eventobjektets resultatlista. Det nya resultatet returneras.

5.1.1.1.2 Om de tidigare metoderna inte returnerade null lägger Eventobjektet till Resultobjektet i en ArrayList<Result>.

5.1.2 Participantobjektet lägger till Resultobjektet i en ArrayList<Result>.

Det enda av det ovanstående som inte är helt lätt att följa bör vara punkterna 5.1.1.1.1 - 5.1.1.1.2. Koden ser ut såhär:

```
1  private Result resolveEqualAchievees(Result existingResult, Result newResult) {
2
3    if(newResult.getResult() > existingResult.getResult()) {
4
5      existingResult.setMedal(null);
6      results.remove(existingResult);
7
8      return newResult;
9
10   }
11   return null;
12
13 }
14
15 private Result determineResultToBeAdded(Result newResult) {
16
17   for(Result existingResult : results) {
18
19     if(existingResult.getAchievee() == newResult.getAchievee()) {
20
21       return resolveEqualAchievees(existingResult, newResult);
22
23     }
24
25   }
26   return newResult;
27
28 }
29
30 public void addResult(Result result) {
31
32   Result resultToBeAdded = determineResultToBeAdded(result);
33
34   if((resultToBeAdded != null)) {
35
36     results.add(resultToBeAdded);
37
38   }
39 }
```

Listing 1: Metoder för att avgöra vilket resultat som ska läggas till hos ett Eventobjekt

Anledningen till att resultatets medalj sätts till null innan den tas bort på raderna 5-6 är att Participantobjektet fortfarande lagrar alla resultat som registreras. Ett resultat som är lägre än deltagarens högsta får dock inte användas för att räkna medaljer på. Medaljuträkning förklaras närmare nedan.

3 Resultatlista för lagen

Eftersom kravspecifikationen är ganska otydlig på hur denna funktion ska implementeras och jag inte har någon riktig fackunskap om hur sådant här sköts i verkligheten har jag gjort några mer

eller mindre godtyckliga designval. För det första bestäms resultatlistan för lagen när "teams"-kommandot körs - då delas alla medaljer ut och placeringen bestäms. Den hade kunnat bestämmas när ett nytt resultat läggs till, men då hade man behövt lägga in metodanrop till alla relevanta funktioner på alla ställen där man gör någonting med deltagarna eller resultaten. I nuläget är detta endast när man tar bort deltagare, men skulle man vilja lägga till funktionalitet senare kan det bli bökigt och/eller lätt att glömma bort sig. På det här sättet är funktionaliteten inkapslad och jag kan vara säker på att den tillgängliga informationen alltid gäller för det tillfälle då användaren vill se resultatlistan.

För det andra nollställs alla medaljer och placeringar innan de uppdateras. Detta var det enklaste sättet att se till att inga gamla medaljer/placeringar ligger och skräpar.

Det finns en tydlig ansvarsfördelning mellan Event, som delar ut medaljer enligt vissa regler, och varje Participantobjekt som räknar ihop sina egna medaljer samt Teamobjekten som räknar ihop det totala antalet medaljer för alla sina Participantobjekt. På det här sättet blir det väldigt lätt att ändra på hur medaljer delas ut eller att ha flera alternativa sätt beroende på gren. Det finns exempelvis många olika sätt att lösa likaplaceringar på. Det tog mig lång tid att förstå testutskrifterna på ilearn, länge hade jag en helt annan lösning på hur likaplaceringar skulle lösas men det var väldigt lätt att ändra på eftersom medaljutdelningsalgoritmen är skiljd ifrån ihopräknandet.

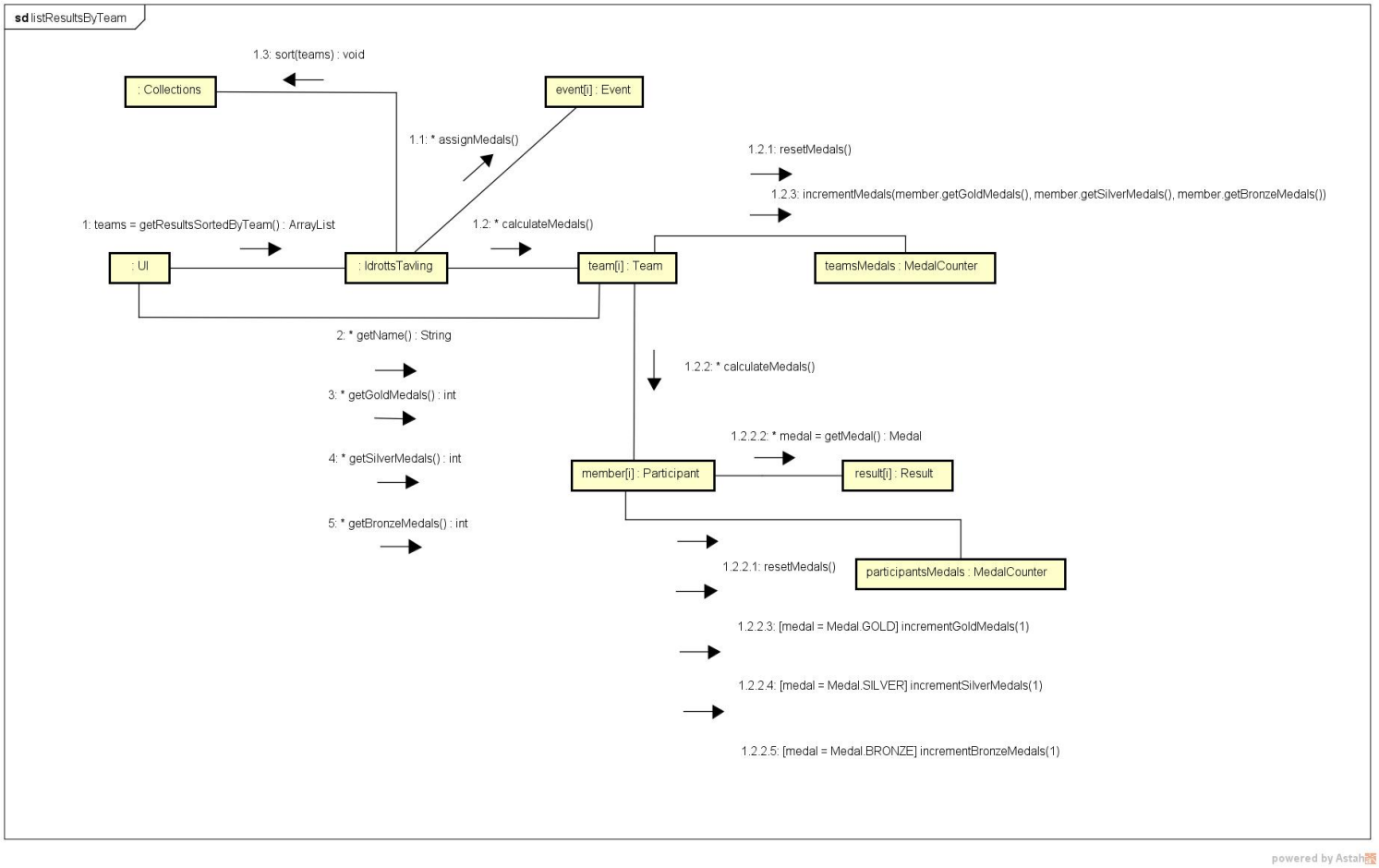

Figure 3: Kommunikationsdiagram för funktionen "Resultatlista för lagen"

Meddelandeflödet när resultat ska listas efter lag ser ut som följer. Numreringen avser punkterna i kommunikationsdiagrammet ovan:

1 UI anropar IdrottsTavlings getResultsSortedByTeam().

1.1 IdrottsTavling itererar över sin ArrayList<Event> och låter varje gren dela ut medaljer baserade på de resultat som uppnåtts i den. Jag har valt att inte modellera upp den metoden i detta kommunikationsdiagram eftersom det inte blir särskilt lättläst. Metoden behandlas istället i ett aktivitetsdiagram nedan.

1.2 IdrottsTavling itererar över sin ArrayList<Team> och anropar calculateMedals() på varje lag.

1.2.1 Teamobjektet sätter alla sina medaljer till noll.

1.2.2 Teamobjektet itererar över sin ArrayList<Participant> och anropar calculateMedals() på varje lagmedlem.

1.2.2.1 Participantobjektet sätter alla sina medaljer till noll.

1.2.2.2 Participantobjektet itererar över sin ArrayList<Result> och anropar getMedal() på varje objekt för att ta reda på vilken medalj som är associerad med resultatet.

1.2.2.2.3 - 1.2.2.2.5 Beroende på vilken typ medaljen är av ökar Participantobjektet antingen antalet guld-, silver- eller bronsmedaljer med hjälp av anrop till sitt MedalCounterobjekt.

1.2.3 Teamobjektet ökar sitt totala antal medaljer med det antal medaljer som Participantobjektet hade.

1.3 IdrottsTavling sorterar sin ArrayList<Team> efter antalet medaljer och returnerar den sedan till UI.

2 - 5 UI itererar sedan över den returnerade samlingen, hämtar namn och antalet medaljer för varje lag och skriver ut dem på skärmen.

Team och Participant har varsin referens till ett MedalCounterobjekt på vilket de anropar metoder för att öka antalet medaljer de har. MedalCountern har en enkel array med tre platser där 0 representerar guld, 1 silver och 2 brons. Genom att "förvara" arrayen i en klass som specificerar beteende för hur medaljer får läggas till leder dessa indexkodade placeringar inte till några problem. Under en lång period använde jag en HashMap<String, Integer> (där strängen var en nyckel för valören) men det kändes till slut onödigt eftersom jag bara behövde precis tre platser och beteendet för hur medaljer läggs till och tas bort är hårt styrt på ett enda ställe. På sätt och vis har jag nu skrivit en klass som genom sina metoder fyller precis den funktion som nycklarna i HashMapen gjorde.

3.1 Event.assignMedals()

Som jag tolkat de instruktioner som finns på ilearn delas medalj ut per placering. Hur många deltagare som helst kan få samma placering om de har samma resultat, men de efterföljande placeringarna "äts upp". Finns det tre guldmedaljörer kommer ingen silvermedaljör eller bronsmedaljör att koras, däremot får alla med högst resultat guldmedalj. Jag hoppas att detta är en korrekt tolkning.

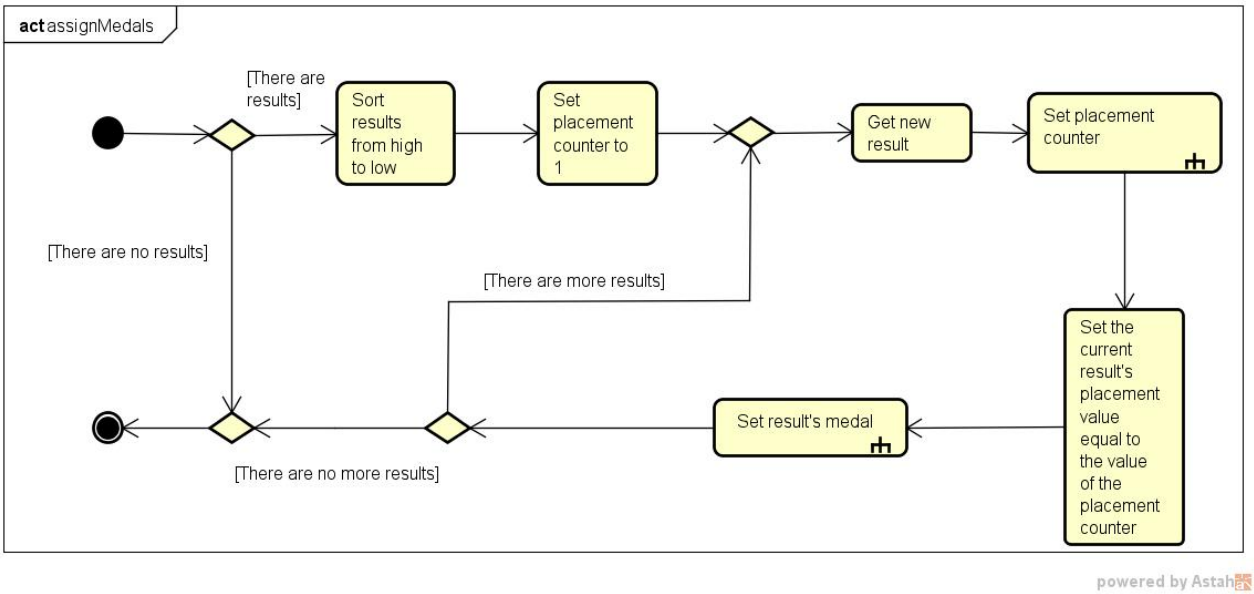

Figure 4: Aktivitetsdiagram för Event.assignMedals()

I aktivitetsdiagrammet ovan ser vi de huvudsakliga stegen i algoritmen:

- Kontrollera om det finns resultat
- Sortera resultaten
- Initiera placeringsindex
- Hämta resultat
- Bestäm om placeringsindexet ska ändras för resultatet eller inte
- Sätt resultatets placering till placeringsindexet
- Fortsätt om det finns fler resultat, annars avsluta

Det enda tricket är egentligen att sätta placeringsindexet. Som vi snart kommer att se i koden använder metoden en traditionell for-loop för att iterera över resultaten. Detta ger ett iterationsindex utöver placeringsindexet. Aktivitetsdiagrammet ser ut som följer:

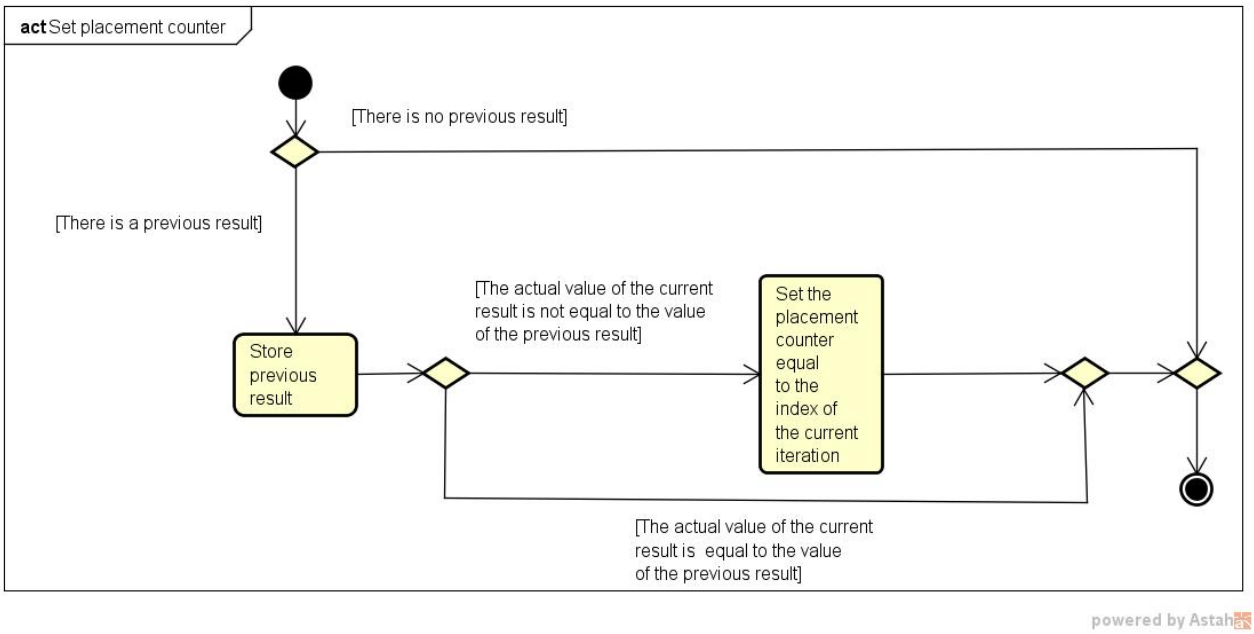

Figure 5: Bestäm placeringsindex

Så länge som det aktuella resultatets värde är det samma som värdet hos resultatet innan det i listan kommer resultatets att ges samma placering som det tidigare resultatet. När ett resultat med ett annat värde dyker upp sätts placeringsindexet till att vara lika med indexet för iterationen. Finns det inget tidigare resultat görs ingenting. Detta kommer bara att vara sant under den första iterationen, då placeringsindexet ändå satts till ett. Såhär ser koden ut:

```
1
2  int placementCounter = 1;
3
4  for(int i = 0; i < results.size(); i++) {
5
6    Result result = results.get(i);
7    Result lastResult = i == 0 ? null : results.get(i - 1);
8
9    boolean thereIsAPreviousResult = lastResult != null;
10   boolean currentResultIsNotEqualToLastResult = thereIsAPreviousResult &&
11                           (result.getResult() != lastResult.getResult());
12
13   if(currentResultIsNotEqualToLastResult) {
14
15     placementCounter = i + 1;
16
17   }
18
19   result.setPlacement(placementCounter);
20
21   //...
22 }
```

Listing 2: Att sätta placeringsindex

Nästa steg är att dela ut medaljer. Aktivitetsdiagrammet ser ut som följer:

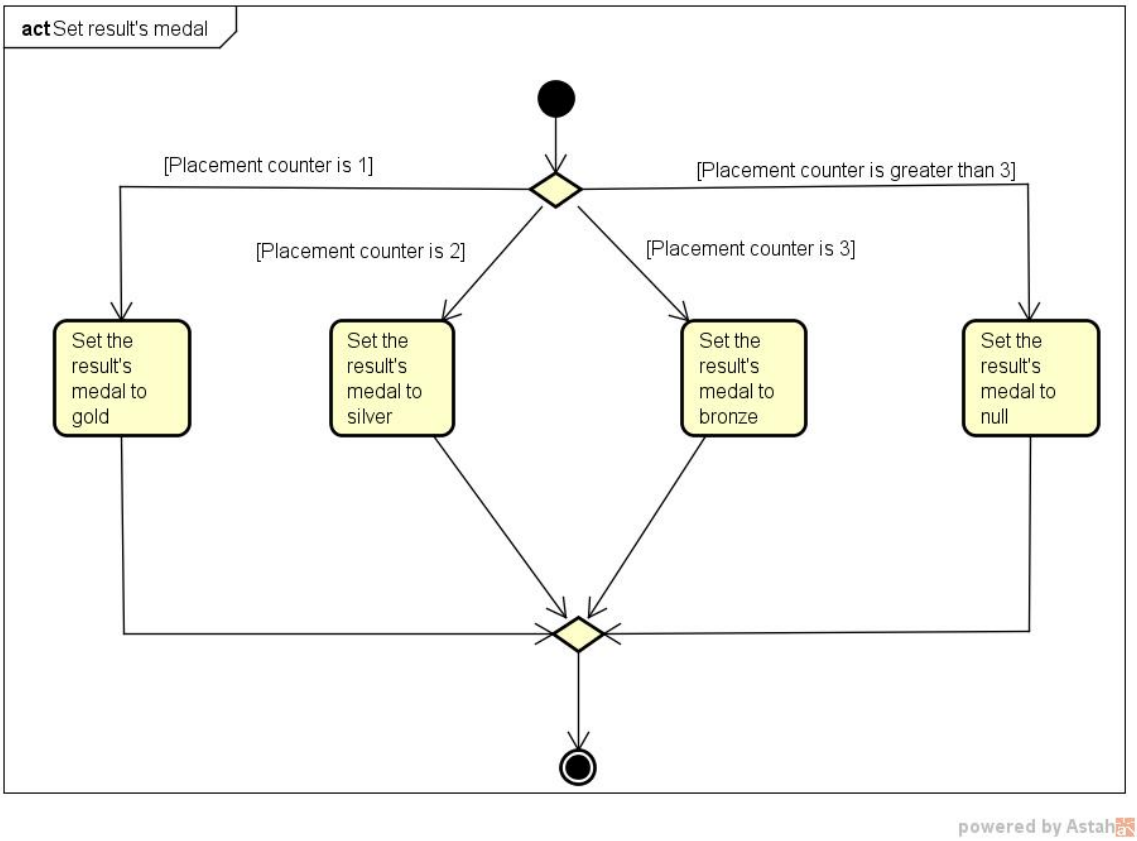

Figure 6: Att dela ut medaljer

Så länge som placeringsindexet är lika med ett kommer guldmedaljer att delas ut, när det är lika med två delas silvermedaljer ut och när det är lika med tre delas bronsmedaljer ut. Detta sköts med ett enkelt switch-statement.

```
switch(placementCounter) {

case(1):
  result.setMedal(Medal.GOLD);
break;
case(2):
  result.setMedal(Medal.SILVER);
break;
case(3):
  result.setMedal(Medal.BRONZE);
break;
default:
  result.setMedal(null);

}
```

Listing 3: Att dela ut medaljer

4 Normalisering av namn

För detta ändamål skrevs en metod hos InputHandler som ser ut på följande sätt:

```
public String normalizeString(String stringToBeNormalized) {

  stringToBeNormalized = stringToBeNormalized.trim();
```

```
4   stringToBeNormalized = stringToBeNormalized.toLowerCase();
5
6   try {
7
8     String upperCaseFirstLetter = stringToBeNormalized.substring(0, 1).toUpperCase();
9     String stringToBeNormalizedWithoutFirstLetter = stringToBeNormalized.substring(1)
      ;
10
11    String normalizedString = upperCaseFirstLetter +
      stringToBeNormalizedWithoutFirstLetter;
12
13    return normalizedString;
14
15  } catch(IndexOutOfBoundsException e) {
16
17    return null;
18
19  }
20
21 }
```

Metoden tar en sträng, tar bort all överflödig whitespace och gör alla bokstäver till gemener med hjälp av metoder hos java.lang.String. Efter detta plockar metoden ut den första bokstaven i strängen med hjälp av Strings substring-metod, gör den till versal och sparar den i en ny variabel. Sedan plockar den ut alla bokstäver utom den första med hjälp av en överlagrad variant av samma metod och sparar i ytterligare en variabel. Till sist skapas en ny variabel där den stora första bokstaven och resten av strängen konkateneras. Denna sträng returneras.

Eftersom normalizeString nästan enbart används i metoder som läser in namn kommer strängen som skickas till den i princip aldrig att vara tom. Try/catch-blocket motiveras endast av designen av handleCommand-metoden hos UI. För att kolla om användaren skrivit in ett grennamn gör den ett anrop till sin getEvent() metod för att se om den returnerar ett event baserat på det användaren skrivit in, på följande sätt:

```
1 private void handleCommand(String command) {
2
3     if(getEvent(command) != null) {
4
5     listResultsByEvent(command);
6
7   } else if(command.matches("message.+")) {
8
9     printMessage(command);
10
11  } else {
12
13    switch(command.toLowerCase()) {
14
15    case "add event":
16        addEvent();
17        break;
18
19    //...
20
21    default:
22        System.out.println("Unknown command.";
23  }
24
25 }
```

Listing 4: UI.handleCommand()

Metodanropet sker på rad 3. Skulle användaren skriva in ett tomt kommando skulle getEvent(command) kasta ett IndexOutOfBoundsException. Man skulle kunna ha haft en kontroll av detta innan detta metodanrop men det kändes mer logiskt att hantera tomma kommandon med default-raden i switch-statementet vilket resulterade i denna lösning. Att ha felhanteringen i normalizeString-metoden gör ju också att koduppreprepning minskar - felhanteringen sker på ett enda ställe.

I vilket fall som helst används normalizeString() sedan av ett flertal metoder i både InputHandler och UI.

```
1  public String readName(String leadText) {
2
3    String name = readString(leadText);
4
5    while(name.trim().length() == 0) {
6
7      System.out.println("Names can't be empty!");
8      name = readString(leadText);
9
10   }
11
12   name = normalizeString(name);
13
14   lastString = name;
15
16   return name;
17
18 }
```

Listing 5: InputHandler.readName(). Variabeln på rad 14 är en instansvariabel som kan användas av UI för att komma åt information användaren skrivit in även i funktioner som p.g.a. scope inte skulle ha tillgång till informationen. Detta motverkar koduppreping. Se exempelvis UI.addResult().

```
1  private Event getEvent(String eventName) {
2
3    eventName = inputHandler.normalizeString(eventName);
4
5    return idrottsTavling.getEvent(eventName);
6
7  }
```

Listing 6: UI.getEvent()

```
1  private void listResultsByEvent(String eventName) {
2
3    eventName = inputHandler.normalizeString(eventName);
4    ArrayList<Result> eventsResults = idrottsTavling.getResultsByEvent(eventName);
5
6    System.out.println("Results for " + eventName + ":");
7
8    for(Result result : eventsResults) {
9
10     System.out.println(result.getPlacement() + " " + result.getResult() + ", " +
       result.getAchievee().getFullName());
11
12   }
13
14 }
```

Listing 7: UI.listResultsByEvent()

5 Arrayer och ArrayList

ArrayLists har använts på alla ställen där det inte i förhand går att veta hur mycket data som ska sparas. Detta gäller på de flesta ställen. IdrottsTavling har ArrayLists för Teamobjekt och Eventobjekt medan Participant och Event har varsin ArrayList för resultat. Vad gäller arrayer har jag redan nämnt den array som finns i MedalCounter-klassen. Den enda arrayen jag använt utöver denna finns i UI-klassens printMenu()-metod. Metoden använder arrayen för att lagra möjliga menyval och sedan iterera över den för att skriva ut dem på skärmen. Vilka möjliga menyval det finns kommer alltid att vara känt innan körning och att lagra dem i en array för att sedan skrivas ut

gör att nya kommandon kan läggas till utan att någon ändring i själva utskriftskoden görs. Metoden ser ut som följer:

```
private void printMenu() {

  String[] commands = {"add event",
              "add participant",
              "remove participant",
              "add result",
              "participant",
              "teams",
              "EVENTNAME",
              "reinitialize",
              "message MESSAGE",
              "print menu"};

  System.out.println("Menu:");
  for(String command : commands) {

    System.out.println("* " + command);

  }

}
```

Listing 8: UI.printMenu()

6 Statiska variabler och metoder

Det enda användandet av static i mitt program, undantaget mainmetoden, är konstanten MESSAGE_BOX_WIDTH i UI-klassen, som anger hur bred ramen för meddelanden ska vara.

7 Reflektion

Det har varit en rolig uppgift. För min egen del har svårighetsgraden legat på en sådan nivå att jag fått tänka en del kring hur jag ska få de olika programfunktionerna att fungera men framförallt lyckats få fram ett fungerande program ganska snabbt och därefter kunnat fundera över designval. Jag har framförallt fokuserat på att organisera klasserna på ett vettigt sätt och har lyckats få programmet mer objektorienterat än vad jag har lyckats med i tidigare program. Som jag nämnde i första punkten har målet varit att avgränsa klassernas ansvarsområden så mycket som möjligt och därmed minska antalet associationer, men eftersom vissa av klasserna ändå är rätt stora skulle jag förmodligen kunna ta det ännu längre och introducera många fler. MedalCounter var ett väldigt sent påfund för att flytta ut en del av ansvaret med att räkna medaljer till en gemensam plats, och man skulle till exempel kunna fortsätta med att introducera en ResultList-klass och föra över ansvaret att kontrollera vilka resultat som ska få läggas till dit, för att nämna en sak. Det har jag dock inte hunnit med.

Jag har försökt vara så konkret, tydlig och förklarande som möjligt i mitt val av namn på såväl variabler som metoder och klasser, därför har det dykt upp en del långa namn som definitivt gör koden mer lättläst men kanske inte alltid de olika diagrammen (de blir så fantastiskt stora). Skyddsnivåerna bör vara ganska självförklarande. Mig veterligen är allt som inte behöver kommas åt av andra klasser satt till private och resten till public.

Det svåraste har nog varit att få resultatlistan för lagen och grenarna att fungera ordentligt. Jag skrev den grundläggande funktionaliteten innan det fanns någon information om hur medaljer och placeringar skulle räknas ut. Algoritmen för detta fick jag ändra väldigt sent när jag upptäckte att det lagts till information. Det positiva var att jag i alla fall kapslat in funktionaliteten så pass mycket att jag kunde ändra utdelningsalgoritmen utan att påverka räknandet av medaljerna eller något annat för den delen heller. Eftersom endast medaljerna nämndes i början, inga placeringar, har jag tänkt i termer av medaljer hela tiden och lagt på placeringstänket ovanpå detta. I detta fall

hade det förmodligen varit smidigare att utgå helt från placeringarna, kanske ha en placeringsklass istället för medaljen, men koden fungerar.

Jag tror att jag var rätt väl förberedd för uppgiften redan när jag satte igång och jag känner verkligen att jag har haft hjälp av kursen vad gäller designtänk, med vilket jag menar objektorientering och hur metoder ska designas. Jag hade hyfsad koll på hur man får ett program att fungera innan, det vill säga hur man använder loopar, if-satser, switch-statements, arrayer, maps och så vidare men väldigt mycket sämre koll på hur man organiserar upp det på ett bra sätt. Där känner jag att jag har lärt mig mycket av den här kursen.

References

[1] Craig Larman. *Applying UML and Patterns*. Addison Wesley Professional, 2004. ISBN: 0-13-148906-2.

8 Appendix

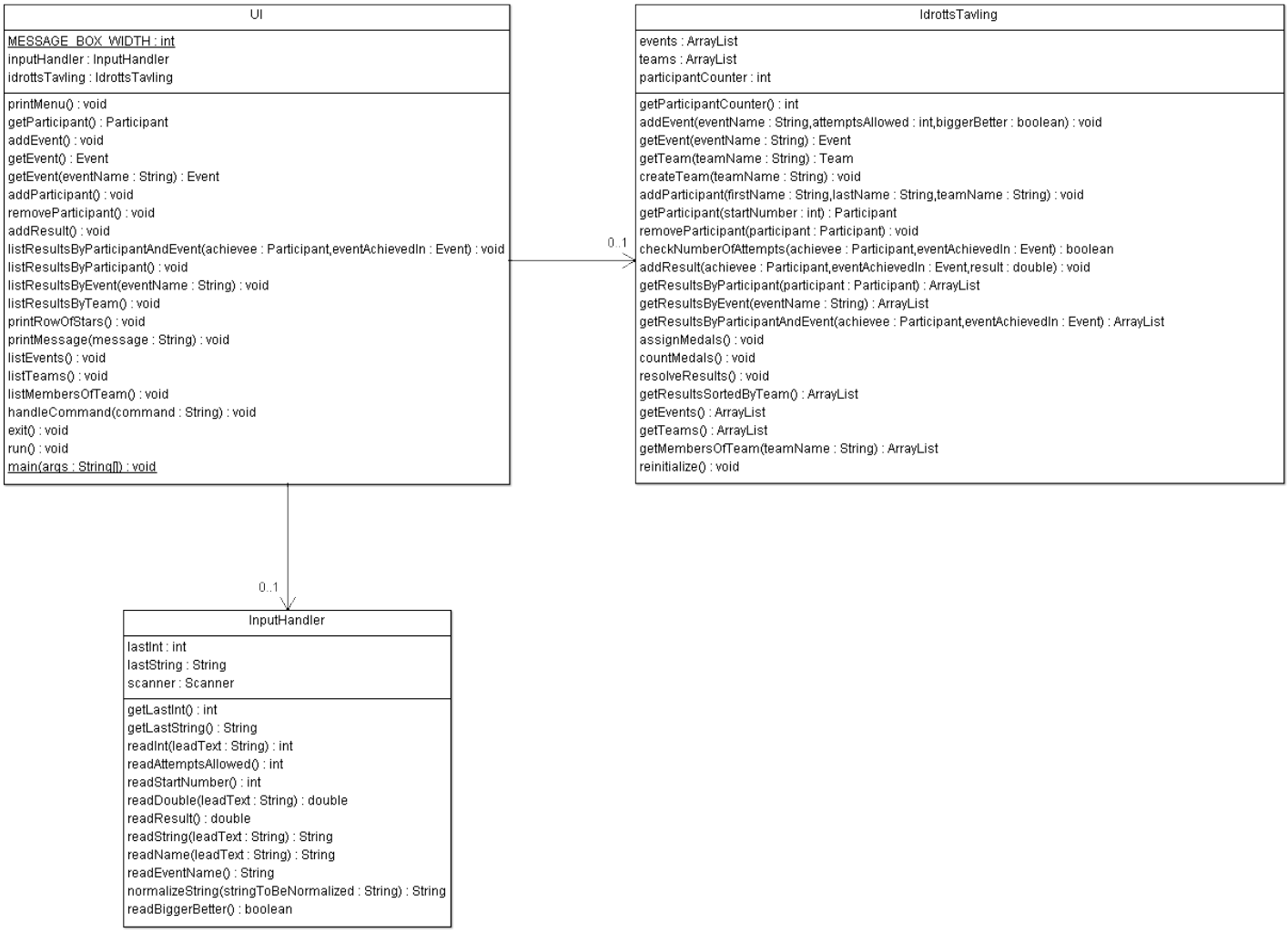

Figure 7: Klasserna UI, InputHandler och IdrottsTavling

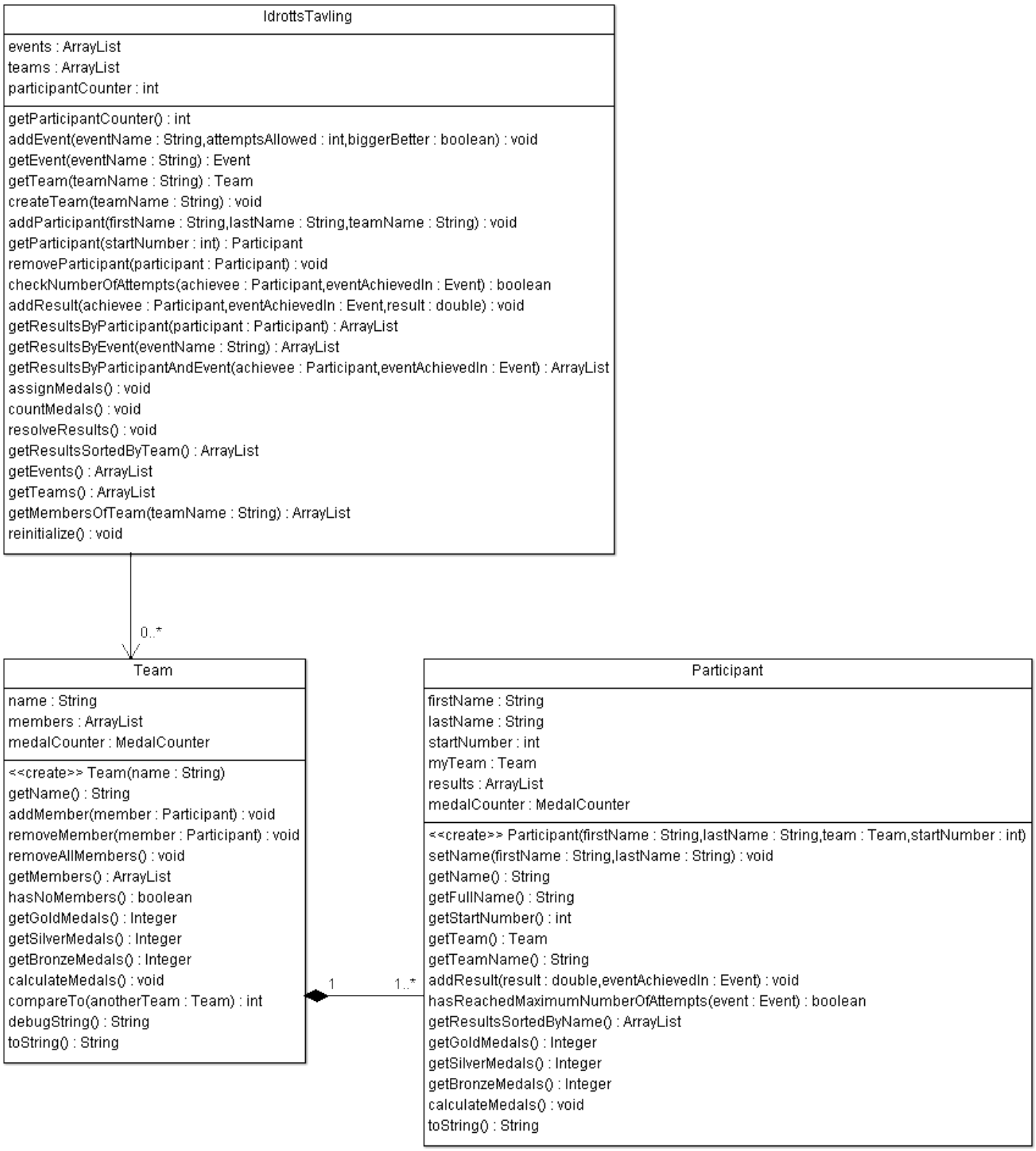

Figure 8: Klasserna IdrottsTavling, Team och Participant

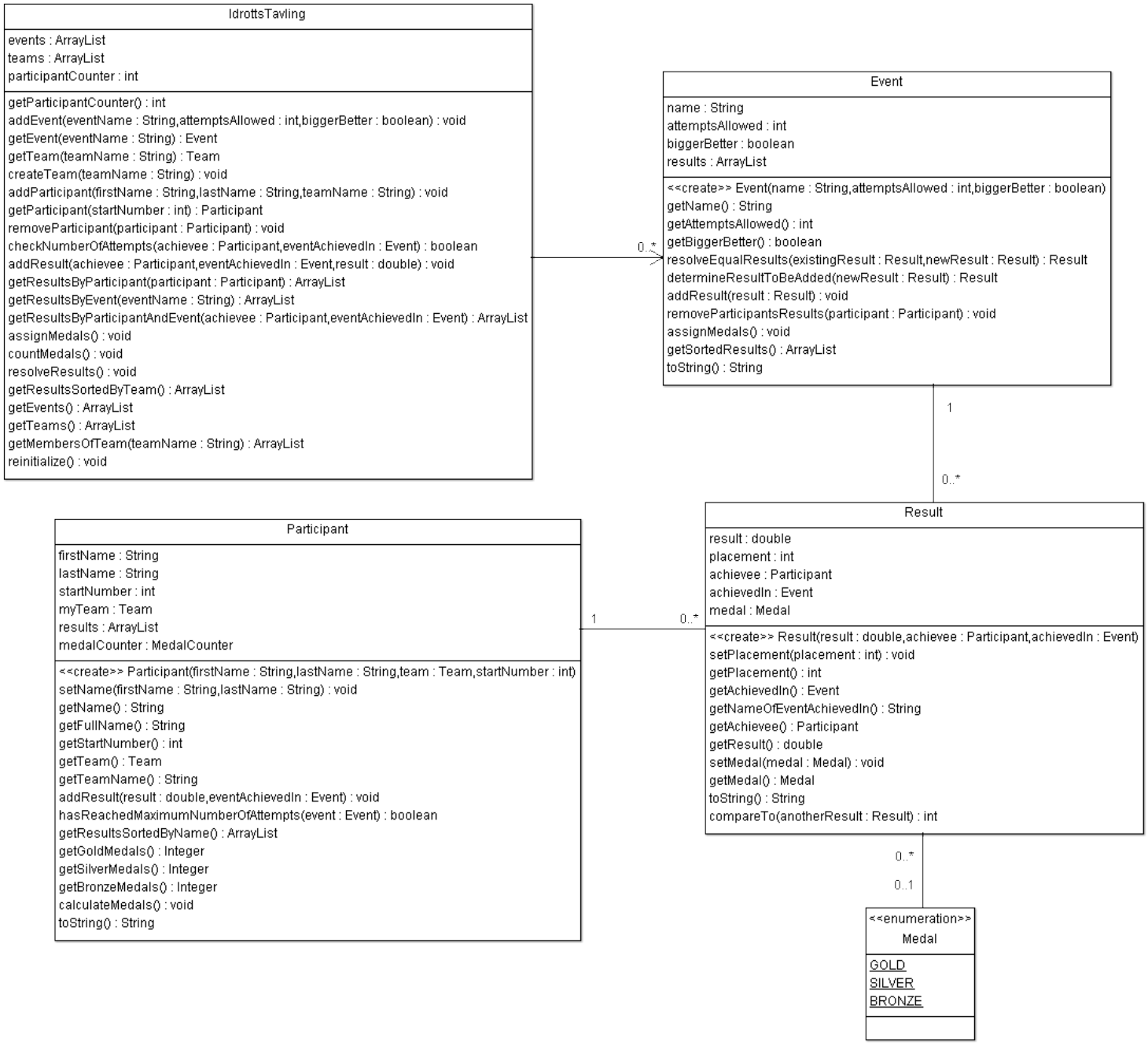

Figure 9: Klasserna IdrottsTavling, Event, Result och Participant samt enumen Medal

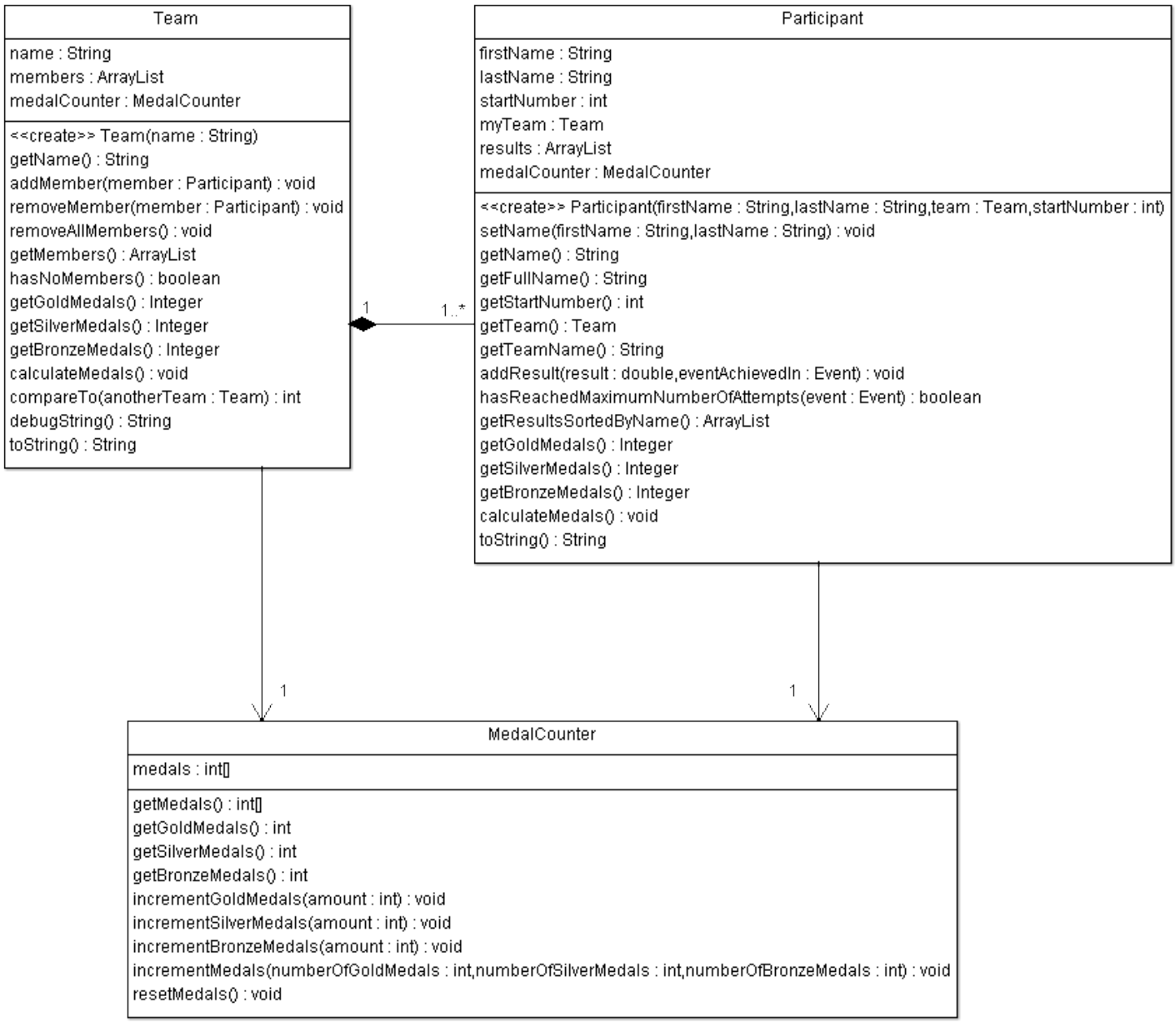

Figure 10: Klasserna Team, Participant och MedalCounter