**System Architecture Design**

For a Track & Field Meet Server

Version 1.0

Submitted in partial fulfillment of the requirements of the degree of MSE

Tracy Marshall

CIS 895 – MSE Project

Kansas State University

# Table of Contents

[Table of Contents 2](#_Toc462169182)

[1. Introduction 3](#_Toc462169183)

[2. Architecture 3](#_Toc462169184)

[2.1. Component Design 3](#_Toc462169185)

[2.2. Component Interface Specification 3](#_Toc462169186)

[2.3. System Analysis 3](#_Toc462169187)

[2.4. High-Level Design 3](#_Toc462169188)

[3. Mid-Level Design 4](#_Toc462169189)

[3.1. XXX 4](#_Toc462169190)

[3.2. XXX 4](#_Toc462169191)

[3.3. XXX 4](#_Toc462169192)

[3.4. XXX 4](#_Toc462169193)

[4. Component Interaction 5](#_Toc462169194)

[4.1. Role Execution Sequence 5](#_Toc462169195)

[5. Role Models 6](#_Toc462169196)

[5.1. XXX 6](#_Toc462169197)

[6. USE/OCL Model 7](#_Toc462169198)

# Introduction

The purpose of this document is to provide information about the design for the Track & Field Meet Server (TFMS). Details will be provided laying out the component design and interface specification for the TFMS. The document will provide a high-level design as an overview of the system and mid-level design to detail each component. Sequence diagrams will be defined to describe the component interactions, but full interface specification for each component is not provided in this document.

# Architecture

The system architecture for the TFMS follows a very similar design to most webservers where a server, such as apache, provides an interface for browsers to use based on a standard protocol. The server then interfaces with other components. In ways the TFMS is a webserver that doesn’t communicate based on standard protocols. This lead the design to three major components and then the client component is pictured to communicate the full picture of operation. The three components are the communicator, database manager, and database.

## Component Design

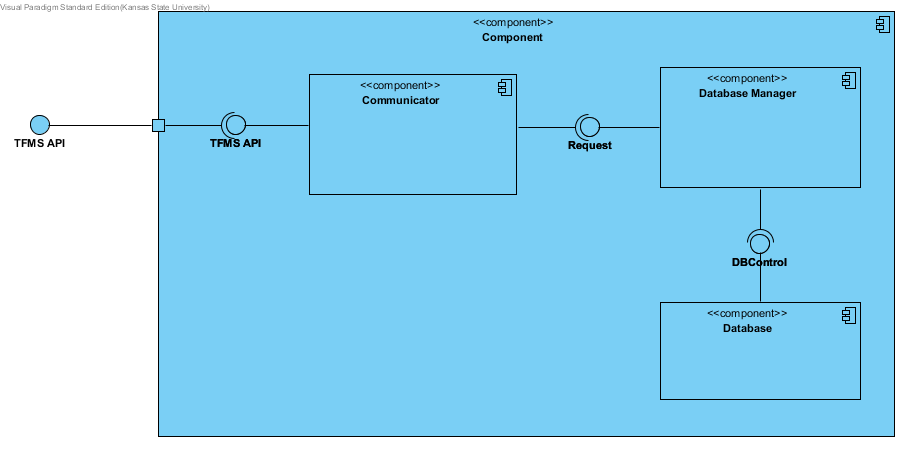


Figure 1 – Component Diagram

## Component Interface Specification

The interfaces that are shown in the component design are specified with more detail in the tables below. The interface specification takes a slightly abstracted view of the interfaces to be clear about what the capabilities are.

|  |  |
| --- | --- |
| **Communicator** |  |
| **Signature** | register( u: User) |
| **Purpose** | Registers a users account and establishes identity |
| **Pre-Conditions** | User is not registered in the system, User is in the system if not a spectator |
| **Post-Conditions** | User is registered, users public key is saved |
|  |  |
| **Signature** | requestData(u: User, d: data) |
| **Purpose** | Allows user the ability to request data that is stored in the server's database |
| **Pre-Conditions** | User must be registered |
| **Post-Conditions** | Server is queued for a response |
|  |  |
| **Signature** | addData(u: User, d: data) |
| **Purpose** | Allows users the capability to add data to the server given they have the correct permissions |
| **Pre-Conditions** | User must be registered |
| **Post-Conditions** | Server is queued for a response, data is added if the users permissions are correct |
|  |  |
| **Signature** | modifyData(u:User, d:data) |
| **Purpose** | Allows users the capability to modify data in the server given they have the correct permissions |
| **Pre-Conditions** | User must be registered |
| **Post-Conditions** | Server is queued for a response, data is modified if the users permissions are correct |
|  |  |
| **Signature** | response(u:user, d:data) |
| **Purpose** | Provides the users feedback on their requests or provides them with data depending on the nature of the request |
| **Pre-Conditions** | User must be resisted, a request must have been made |
| **Post-Conditions** | Response is removed from the server queue |

|  |  |
| --- | --- |
| **Request** |  |
| **Signature** | isRegistered(u:User): Boolean |
| **Purpose** | Checks if a given user is registered in the system |
| **Pre-Conditions** | Request is made by a user |
| **Post-Conditions** | State of registration is returned |
|  |  |
| **Signature** | isAutheticated(u:User): Boolean |
| **Purpose** | Authenticates the user to prove identity |
| **Pre-Conditions** | User is registered |
| **Post-Conditions** | State of authentication is returned |
|  |  |
| **Signature** | hasPermissions(u:User, d:Data): Boolean |
| **Purpose** | Checks if the given user is allowed to make a given request to the server |
| **Pre-Conditions** | User is registered and authenticated |
| **Post-Conditions** | State of permissions are returned |
|  |  |
| **Signature** | queryDB(u:User, d:Data): Data |
| **Purpose** | Accepts a request to query the DB |
| **Pre-Conditions** | User is registered, authenticated, and has permissions |
| **Post-Conditions** | Data that was being queried is returned |
|  |  |
| **Signature** | updateDB(u:User, d:Data) |
| **Purpose** | Accepts a request to update data in the DB |
| **Pre-Conditions** | User is registered, authenticated, and has permissions |
| **Post-Conditions** | DB is queued for an update of the given data |
|  |  |
| **Signature** | addToDB(u:User, d:Data) |
| **Purpose** | Accepts a request to add data to the DB |
| **Pre-Conditions** | User is registered, authenticated, and has permissions |
| **Post-Conditions** | DB is queued to add the new data |

|  |  |
| --- | --- |
| **DBControl** |  |
| **Signature** | queueUpdate(d:Data) |
| **Purpose** | Adds an update to the DB queue so that when the DB is in a safe state the udpate will be made |
| **Pre-Conditions** | Data exists |
| **Post-Conditions** | Upadate is queued |
|  |  |
| **Signature** | dataExists(d:Data): Boolean |
| **Purpose** | Allows the ability to check existence of given data |
| **Pre-Conditions** | DB has initialized |
| **Post-Conditions** | The state of existence of the data is returned |
|  |  |
| **Signature** | queueAdd(d:Data) |
| **Purpose** | Adds an add to the DB queue so that when the DB is in a safe state the data will be added |
| **Pre-Conditions** | DB has initialized |
| **Post-Conditions** | Add is queued |
|  |  |
| **Signature** | getData(d:Data):Data |
| **Purpose** | Grabs the data in the DB described by d |
| **Pre-Conditions** | Data exists |
| **Post-Conditions** | Data is returned |

## High-Level Design

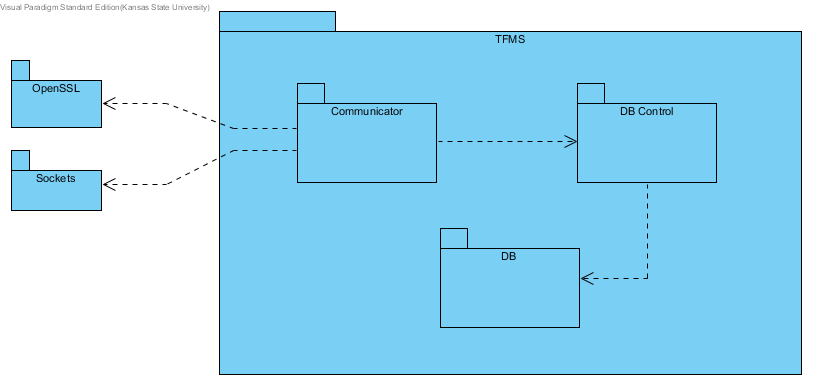


Figure 2- High Level Package View

The TFMS is a fully self-contained product and is not dependent on other applications to function. Due to this fact the system analysis and high level designs ended up looking the exact same so I did not replicate them. The communicator class is really the connection to the outside world which will allow developers to develop client side apps to work with the TFMS. The classes will be refined in the mid-level design as they are put into perspective of the components in the system.

# Mid-Level Design

## Communicator

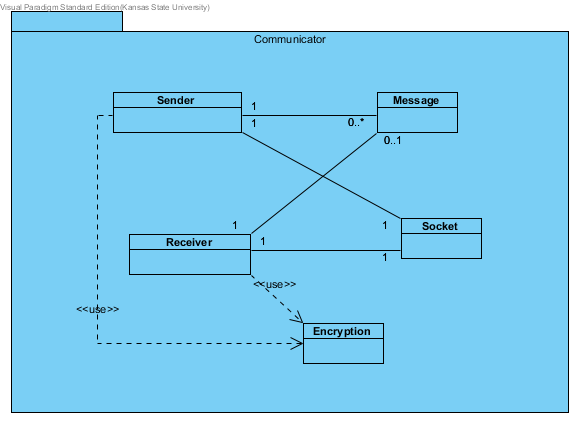


Figure 3- Communicator Package

The communicator package is responsible for communication with all of the clients of the server. This means that it has to be able to receive and send messages. To do this the package builds upon existing socket packages and utilizes encryption packages to do the symmetric and asymmetric encryption/decryption necessary to keep data secure.

## Database Manager

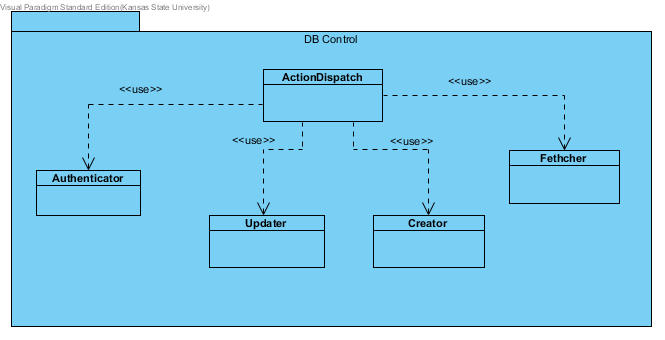


Figure 4- DB Control Package

The database manager package is responsible for being the intelligent one in the system. This package contains the interpreter of the ICD and is able to determine what certain requests from clients mean. On top of that the manager makes determinations about user’s privileges so that not just anyone can modify and add information to the server. The controller dispatches the requests from the client into queries to the database.

## Database

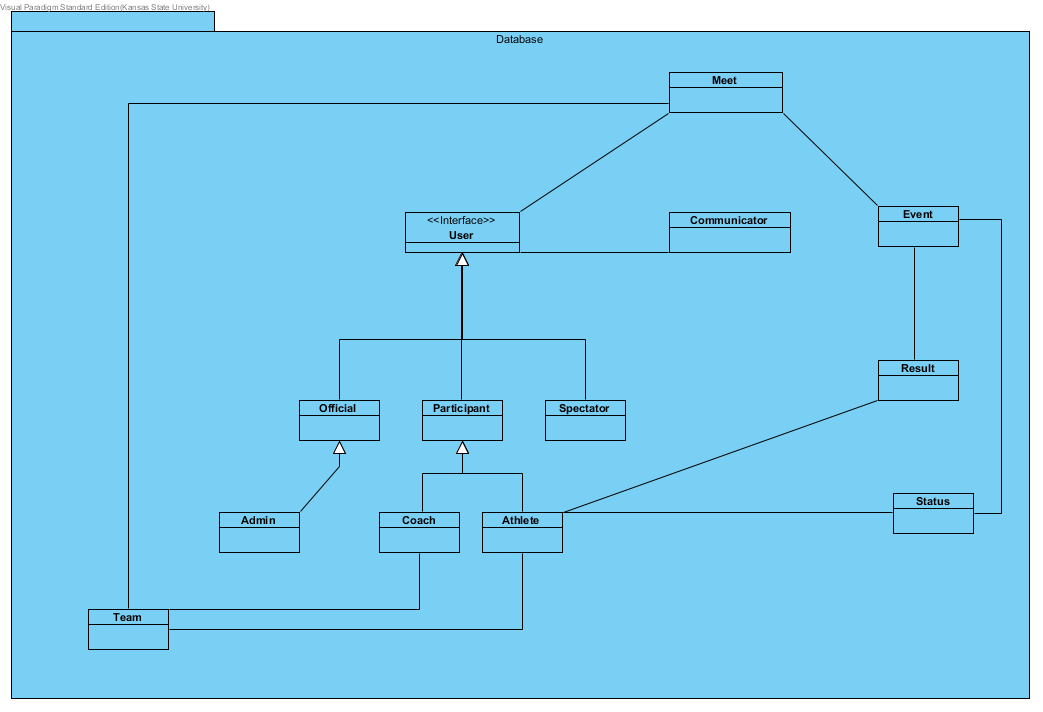


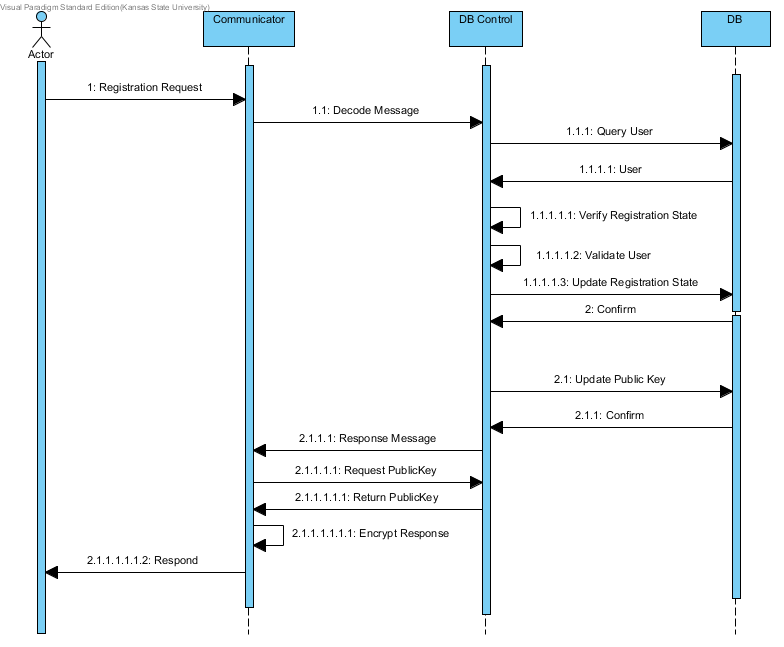
Figure 5- Database Package

The database package is the heart of the system and it contains all of the data that makes the server so important. This package is responsible for managing the actual data objects and providing interfaces to looking up data as well as adding and modifying data in the system.

# Sequence Diagram

xxx

## Registering User Sequence Diagram



Prerequisites

1. Client has obtained the servers public key
2. Client has obtained user ID

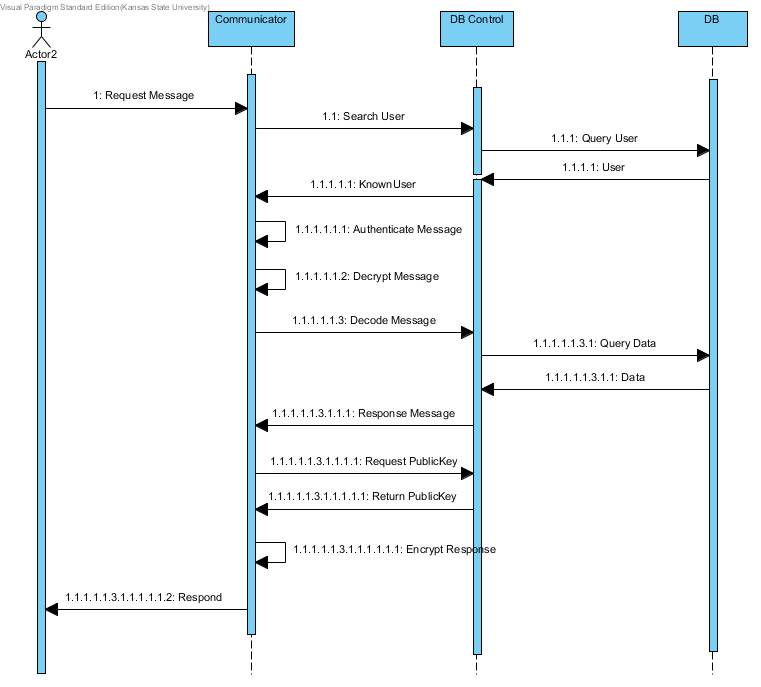
Sequence of Events

1. Client sends a registration request
2. The message is sent to the control to be decoded
3. Database is queried for a matching user object
4. The user is returned to the controller
5. Registration state is verified for the user by checking the public key on record
6. Validate that the user credentials received match with the user credentials that the database knows about
7. Update the database with the registration state
8. Database confirms the update
9. Update the public key of the user
10. Database confirms the update
11. A response message is formed and given to the communicator
12. Users public key is requested
13. The Public Key is returned
14. The public key is used to encrypt the response
15. Response is sent back to the client

Post-Conditions

1. Client is registered

## Requesting Data Sequence Diagram



Prerequisites

1. Client has registered

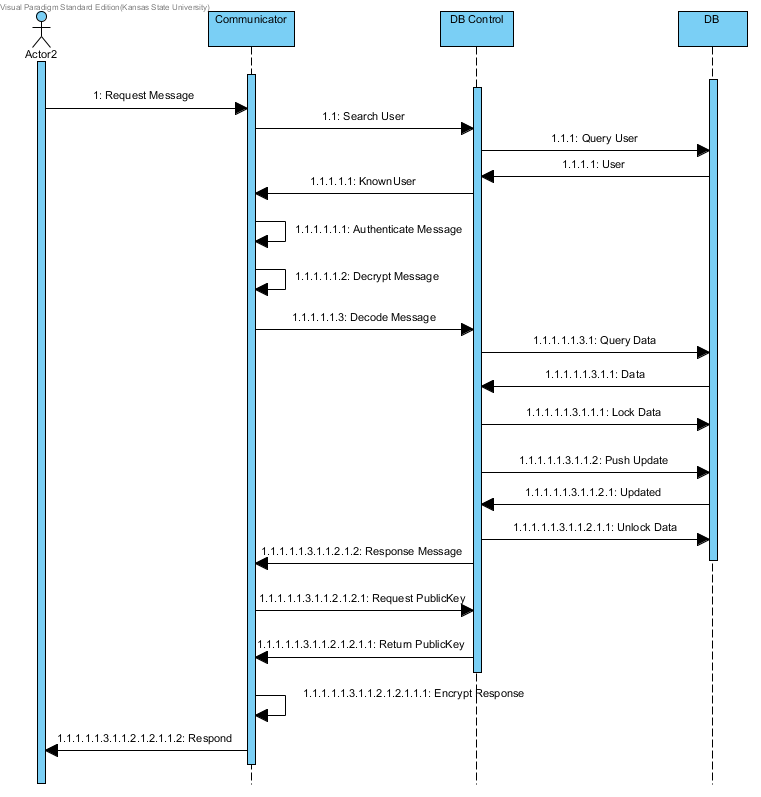
Sequence of Events

1. Client sends a request message
2. The user info is retrieve and sent to the controller to search the user
3. A database query is made for the user
4. The user object that matches the user data is returned
5. Known user status and data is returned to communicator
6. Message is authenticated
7. Message is decrypted
8. Message is sent to controller for decoding
9. Decoded message triggers a query to the database
10. Data is returned to the controller
11. A response message is formed and given to the communicator
12. Communicator request the users public key
13. The public key is returned
14. The response is encrypted using the users public key
15. The response is sent back to the client

Post-Conditions

1. Data requested by the client is returned in a response message

## Updating Data Sequence Diagram



Prerequisites

1. Client has registered

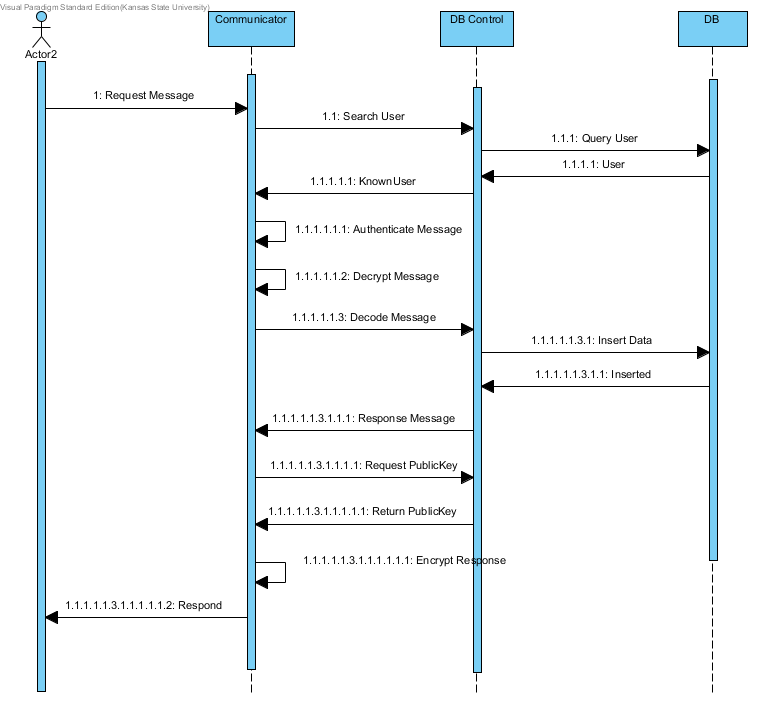
Sequence of Events

1. Client sends a request message
2. The user info is retrieve and sent to the controller to search the user
3. A database query is made for the user
4. The user object that matches the user data is returned
5. Known user status and data is returned to communicator
6. Message is authenticated
7. Message is decrypted
8. Message is sent to controller for decoding
9. Decoded message triggers a query to the database
10. Data is returned to the controller
11. Controller commands the database to lock the data that was requested
12. Controller sends updated data to the database
13. Database sends a data updated response
14. The controller unlocks the data
15. A response message is formed and given to the communicator
16. Communicator request the users public key
17. The public key is returned
18. The response is encrypted using the users public key
19. The response is sent back to the client

Post-Conditions

1. Data is updated in the database

## Adding Data Sequence Diagram



Prerequisites

1. Client has registered

Sequence of Events

1. Client sends a request message
2. The user info is retrieve and sent to the controller to search the user
3. A database query is made for the user
4. The user object that matches the user data is returned
5. Known user status and data is returned to communicator
6. Message is authenticated
7. Message is decrypted
8. Message is sent to controller for decoding
9. Decoded message identifies new data and an insert is sent to the database
10. Database sends back an inserted resposne
11. A response message is formed and given to the communicator
12. Communicator request the users public key
13. The public key is returned
14. The response is encrypted using the users public key
15. The response is sent back to the client

Post-Conditions

1. Data is added to the database

# USE/OCL Model

-------------------------------------------------------------------------------

-- Track & Field Meet Server Architecture

--

-- This file contains a formal specification of the invariants maintained by

-- the getEventsToFire method from the RoleLevelGoalModel

--

-- File: GMoDSAgentArchitecture.use

-- Author: Kyle Hill

-- Date: June 20, 2011

-------------------------------------------------------------------------------

model TFMSArchitecture

-------------------------------------------------------------------------------

-- Classes

-------------------------------------------------------------------------------

class User

attributes

PublicKey : String

Name : String

end

class Official < User

attributes

ID : Integer

end

class Admin < Official

end

class Participant < User

attributes

ID : Integer

end

class Athlete < Participant

attributes

Gender : String

end

class Coach < Participant

end

class Spectator < User

end

class Team

attributes

Name : String

Score : Integer

Classification : String

Gender : String

end

class Event

attributes

Name : String

StartTime : Integer

Current : Integer

InProgress : Boolean

end

class Result

attributes

Value : Float

Attempt : Integer

Heat : Integer

Final : Boolean

Place : Integer

end

class Status

attributes

Ready : Boolean

Conflict : String

end

class Meet

attributes

Name : String

Location : String

Date : String

end

class Communicator

attributes

Port : Integer

end

-------------------------------------------------------------------------------

-- Associations

-------------------------------------------------------------------------------

association AllUsers between

Meet [1] role meetAt

User [1 .. \*] role users

end

association AllTeams between

Meet [1] role meetCompeting

Team [1 .. \*] role teams

end

association AllEvents between

Meet [1] role meetOccured

Event [1 .. \*] role events

association Communication between

Communicator [1] role connectedWith

User [1] role connectedTo

end

association TeamMembers between

Team [1] role onTeam

Athlete [1 .. \*] role teamMemembers

end

association TeamStaff between

Team [1] role coachesTeam

Coach [1 .. \*] role coaches

end

association EventResults between

Event [1] role eventOccured

Result [1 .. \*] role results

end

association AthletesResult between

Result [1] role performed

Athlete [1] role performedBy

end

association AthleteStatus between

Athlete [1] role statuser

Status [1 .. \*] role statuses

end

association EventStatus between

Event [1] role statusedEvent

Status [1 .. \*] role athleteStatuses

end

-------------------------------------------------------------------------------

-- Constraints

-------------------------------------------------------------------------------

constraints

-- Only one track meet

context Meet

inv OnlyOneMeet:

Meet.allInstances->size = 1

-- Coaches only coach for one teamMemembers

context t:Team

inv CoachesForOneTeam:

Team.allInstances->t.coaches->intersection(Team.allInstances->excluding(t).coaches)->size = 0

-- Athletes are only on one team

context t:Team

inv AthletesOnOneTeam

t.teamMemembers->intersection(Team.allInstances->excluding(t).teamMemembers)->size = 0

-- Athletes have one status per event

context a:Athlete

inv OneStatusPerEvent

a.statuses->size = Event.allInstances->intersection(a.statuses.statusedEvent)->size

-- Correct Gender Athlete on Correct Gender Team

context a:Athlete

inv GenderMatch

a.Gender = a.onTeam.Gender