Pool Party Project

Floreza Jensen, Sadia Bibi, Sune Hansen, Nicholas Hansen

Table of Contents

Description of How the System is Tested Brief Manual

To run the project click on the poolParty.gpj (project file)

To login to the system use the Username admin and the password admin (case sensitive)

We have 3 use cases for available. Create Member, search Member and Make Payment.

General Create Member info: Length and spacing in names or other fields are not an issue

Date of birth will have to be added in a ddmmyy format and *MUST* include a - at the end. This is a relic of modifying the program from using CPR and has not yet been adjusted.

After adding members to the system It is imperative that you click the Save Member List button in the administration main frame. (click cancel to close the current frame). Failure to do will mean the new data will never be written to the File and will be lost.

Search Member: Currently you can only search for a member based on their name. Beware that it is case sensitive and any spaces you included in the adding of the name will need to be added in the searching of it as well. The checkboxes will check themselves if a member is active or Elite. The delete functionality has not yet been added though the code for is available.

The members in the search can be seen in the member and elite.dat files. Alternatively you can search for the names of any us in the project (first name only)

Make Payment: To view the make payment use case you must first click on the Payment Info button inside you will find the Make Payment button. To draw up information about the person you want to pay for you can either search for them by name or by ID. This will bring up general information about them. If you are satisfied you have who you are looking for, you can click pay.

You are free to explore the rest of the program however most of it are just placeholders and will either not function or will display a basic GUI frame outlining the most important aspects of what will be added at a later date.

The make payment is currently a demo and is lacking persistence.

Group Contract

Each Member of the Group Promises to abide by the following rules
1. Keep in contact.
2. Inform the group if you will not be present in class
3. Be aware that parts of the project may be assigned to you if you are not in class to pick them.
4. View facebook group for most important updates
5. First Page of the google doc will include a to do list if you have finished your assigned project before the scheduled date.
6. Failure to contribute to the project will result in stern looks and moral condemnation

If a member continuously fails to contribute we shall have a talk with them and if nothing improves will have a brief talk with the teachers.

Work Distribution

Fully Dressed Use Cases:
Floreza: Create Member
Sadia: Make Payment
Nicholas: Submit swimming results

Brief use Cases: Sune

Ito: Sune + Nicholas + Sadia

Programing
GUI - Nicholas + Sadia
Create Member + Search Member = Floreza
Make Payment = Sune
Domain Model + Design Diagram = Sune

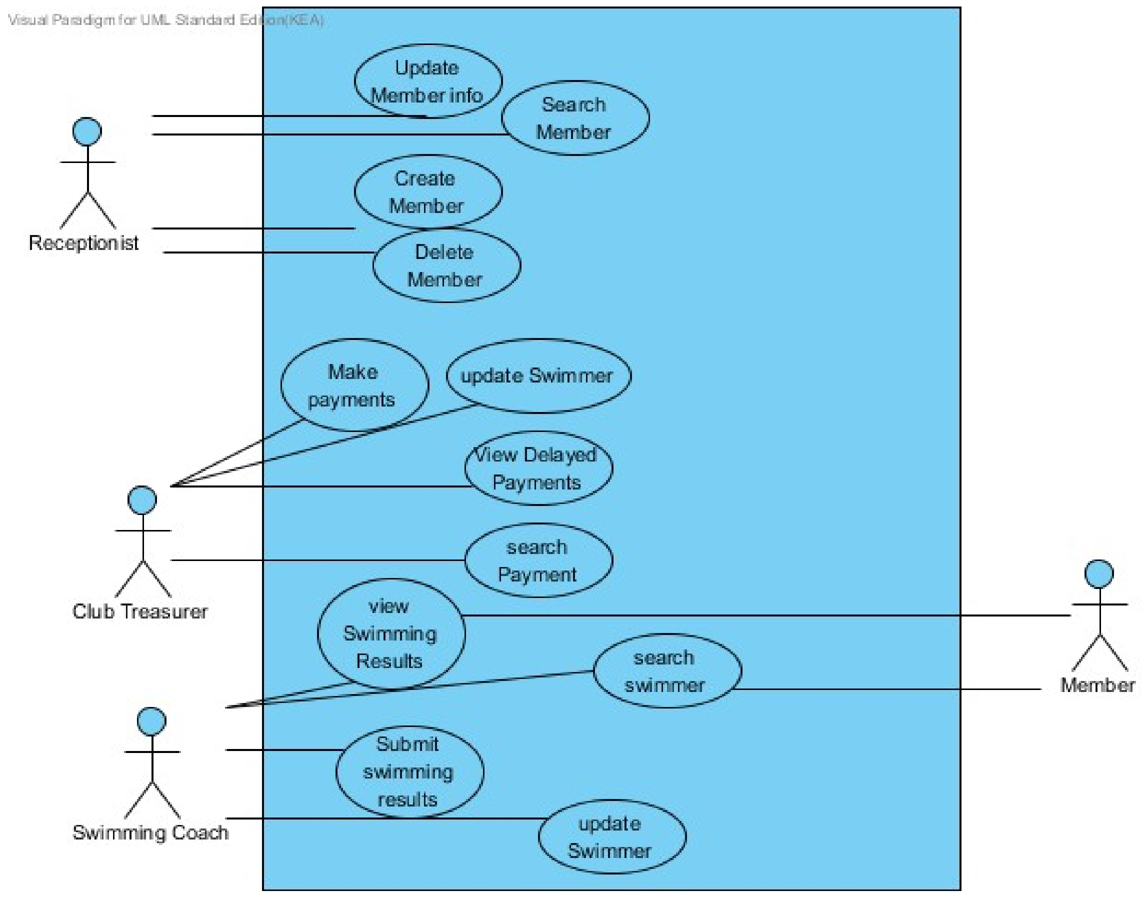
Programing supervisor = Sune
Report Supervisor = Nicholas

PHASE PLAN

Inception Monday 28April
Elaboration Iteration1 Thursday 1 May
Construction iteration1 Monday 5May
Transition iteration1, Elaboration Iteration2, Thursday 8May
Construction Iteration2, Transition iteration2 Thursday 12may
Report Tuesday 14 May
FINAL HAND IN 15May

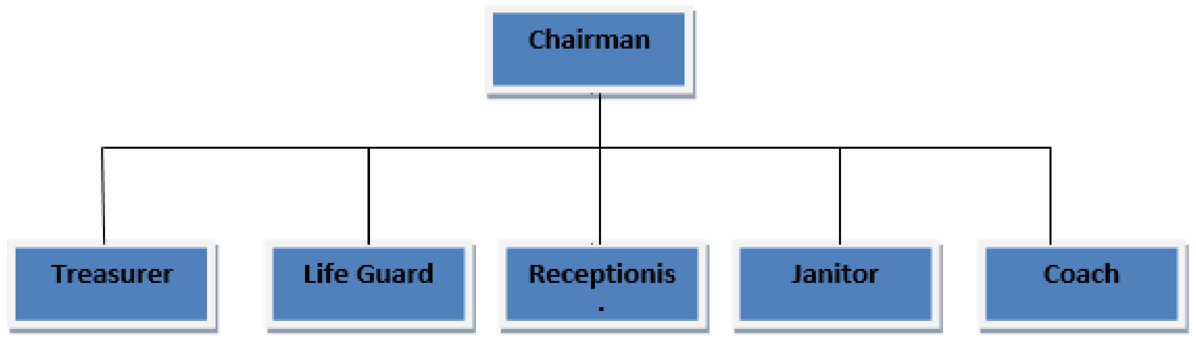
Much to our surprise our project followed the phase plan much more closely than we had expected. The construction part of the project encountered a number of unexpected problems and as such lasted several days longer than projected.

Use Case Diagram

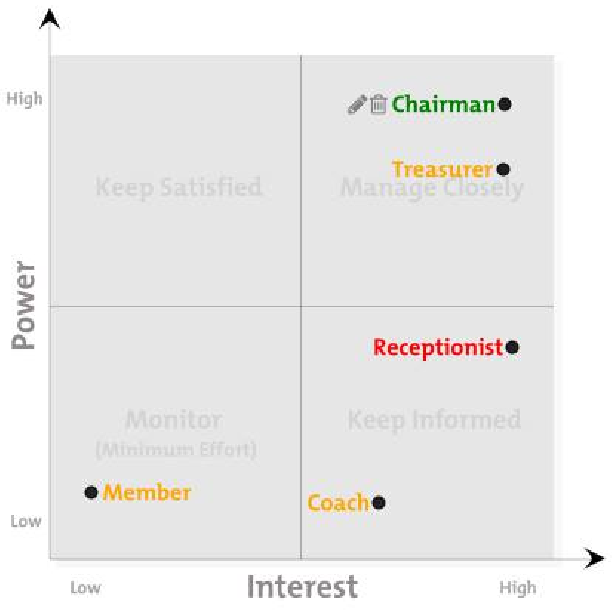


We started off with defining our use cases by referencing the requirements and coming up with 7 basic use cases that we knew had to be added. These use cases were create member, search member, update member, delete member, view swim times ,make payment as well as search Member. As the programing progressed we realised that we needed to add functionality to the program that we had not thought of. When doing the fully dressed use cases we realised that we would run in to problems if you could not update or edit the data you entered. This would be particularly problematic if we entered incorrect swim times in competitions or accidently marked one member who had yet to pay, as paid. We also had a small debate about using the term “manage” and grouping the add, edit and delete use cases into it.We would thus have Manage Member, Manage Swimmer and Manage Payment turning 9 use cases into 3. This would make for a more presentable use case diagram but would overly complicate our fully dressed use cases.

Organizational Chart



As is apparent from the above Organizational diagram we have made the assumption that our swim club would be quite small. It could be a small start up or a family business. As such there is no need for a large host of different employee. The chairman is the boss to whom everyone reports



A number of assumptions had to be made regarding how cooperative each person was likely to be. In this case we made the assumptions based off of their role in the project, their influence in the company, the theorized age of the people in various positions and how important these systems would be in being able to do ones job effectively.

StakeHolder	Stake in Project	Power/ Interest	What we need from them	Perceived Attitudes/ Risks	Stakeholder Management Strat.
Chairmen	System Admin Owner	High/High	Employee Knowledge & MemberShip Info	Enthusiastic but Unrealistic in expectations	Meeting with Project Leaders
Treasurer	Financial System Admin	High/High	Pricing info, Payment Status	Over Complicated System Data loss	Meeting with Project Leaders
Receptionist	New Member Registration	low/high	Contribute to process design and testing	Complicated System. Increase Workload	Involvement in User Groups
Coach	Enter and get Swim info	low/medium	Contribute to testing	Complicated System. Increase Workload	Involvement in User Groups
Members	Wants accurately filed info	low/low	n/a	n/a	n/a

StakeHolder Analysis

We have identified 6 stakeholders. Being a small organization the structure is very flat. The Chairman Being the founder has control over each individual function. Because of the variety of jobs creating extra layers of responsibility amongst so few staff would create extra unnecessary bureaucracy. With such a clear distinction between the different jobs every person is very clear on their own responsibilities and who they need to report to in the event of a problem. This gives the staff a certain degree of autonomy and makes the system relatively more efficient.

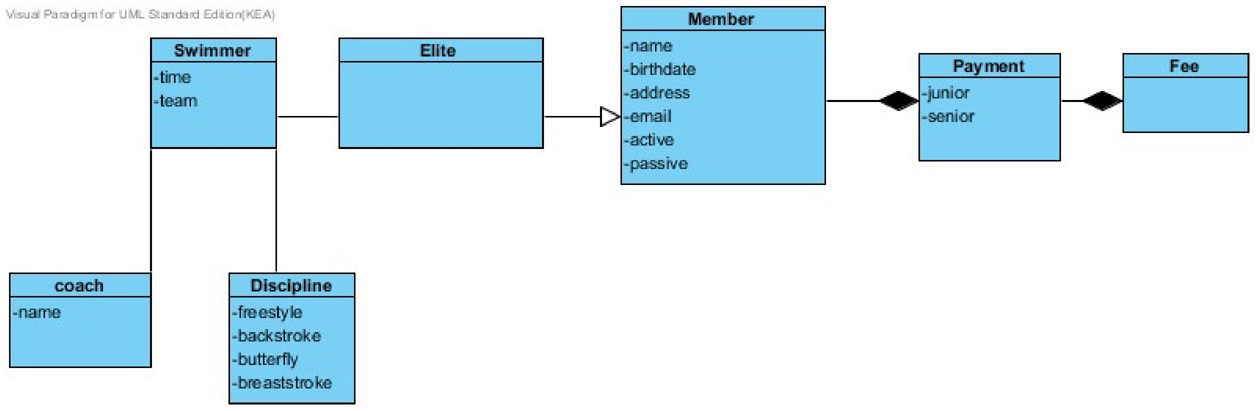
The chairman being on top of this organizational structure has a very high impact on the decisions made in the project and as the the figurehead of the organization sees the modernization of their administration system as his idea. Therefore he also has a high interest in the outcomes of the project. He is part of the project leaders group.

The other person involved in the Project leader group is the treasurer who also has a high interest and high impact on the project. As the treasurer deals with all financial matters he is given a lot of leeway to decide how the financial parts of the software are to be designed. While the chairman can veto any suggestion put forward by the treasurer, the treasurer's expertise is not questioned. His biggest concern is data loss which he emphasizes could bankrupt the organization if problems were to occur with the data stored.

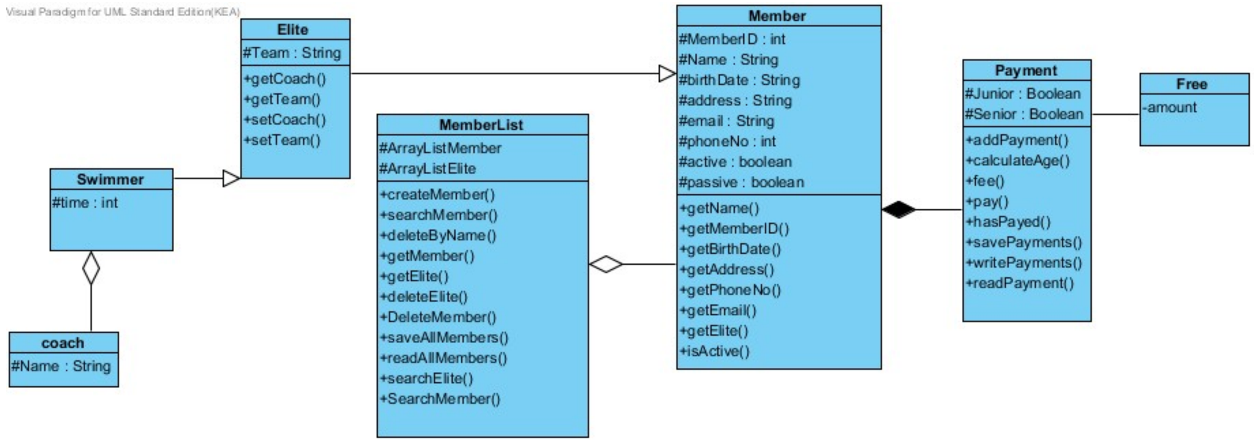
The receptionist is not part of the project leader group and has little impact on the direction of the project. His input in making the system user friendly is appreciated and is included in the user groups for testing of the project and has made his concerns clear where system complexity is concerned.

The coach is enthusiastic in getting electronic swim times and looks forward to the clarity the system could potentially provide in respect to the ranking of his swimmers. His impact is quite low and is likewise not included in the leader group. He echoes the concerns of the receptionist about complexity and wants to be able to edit the swim data in case mistakes had been made. He has also voiced concerns about the possible increase in workload he will be given as a result of the system.

Domain Model



Design Class Diagram



Some of the challenges in the Design Class Diagram included what classes we needed information from and which classes needed to be split or could be merged. For example Instead of choosing 3 classes corresponding to Member, Elite and Exercise we opted for two classes where Member would function as the exercise class. In addition we had initially opted to have a super class of Member and have sub classes of Junior and senior which would both also be split into elite and exercise. We however quickly realised that we could reduce redundancy and maintain high cohesion/low coupling by reducing the classes to 2. As you can see from the above diagram we have a class called memberList which does the File Handling for working with member information but we don't have a similar file handling class for dealing with payments. This is partially as a result of different coding styles of the people involved with these classes and partially a result of the quantity of information we had to process in the class. The Member class was filled with getter and setter methods dictating what information each new member would be assigned to. Adding File handling to the class would lead to unnecessary confusion when we would come to implementing the code. The payment class on the other hand only needs a small amount of information from the member class and including File handling in the class increased efficiency.

Fully Dressed Style

<u>Use Case Name</u>: Create member

Scope: The system under design

Level: user goal.

Primary Actor: Receptionist

Stakeholders and Interests:

- Member – Wants to be registered quickly.

- Club Chairman – Wants to accurate record transactions and satisfy the member interests. Wants the data to be registered quickly and without errors. Wants the system that works perfectly

- Receptionist – Wants to use the system without having any problems. Wants to store the members information in the system.

- Swimming coach –Register the scores of each elite member in the system and wants to know the top elite winners automatically.

- Club treasurer - who takes care of the all the payment transactions. He/She wants an accurate, fast entry and no payment errors.

Preconditions: Receptionist receives an activity form that is filled up By a new member.
Receptionist is logged into the system.

Postconditions: Information is stored. Membership card is printed. The files updated..

Main Success Scenario:

1. Receptionist click “create member” to register the new member in the system
2. System prompts a form.
3. Receptionist enters the member's information such as name, birth date, address, email phone number and status.
4. Receptionist specifies the member as active
5. Receptionist submits the form by clicking “submit button”.
6. System stored information to a file and prompts member is save to a file.

Extensions:
*1.a. At any time: System breaks down or returns an error

1.a.1The receptionist restart the system
1.a.2The receptionist logs in by username and password
1.a.3The receptionist contacts the chairman

4a. Member is passive
1. Receptionist leaves the active checkbox unchecked.

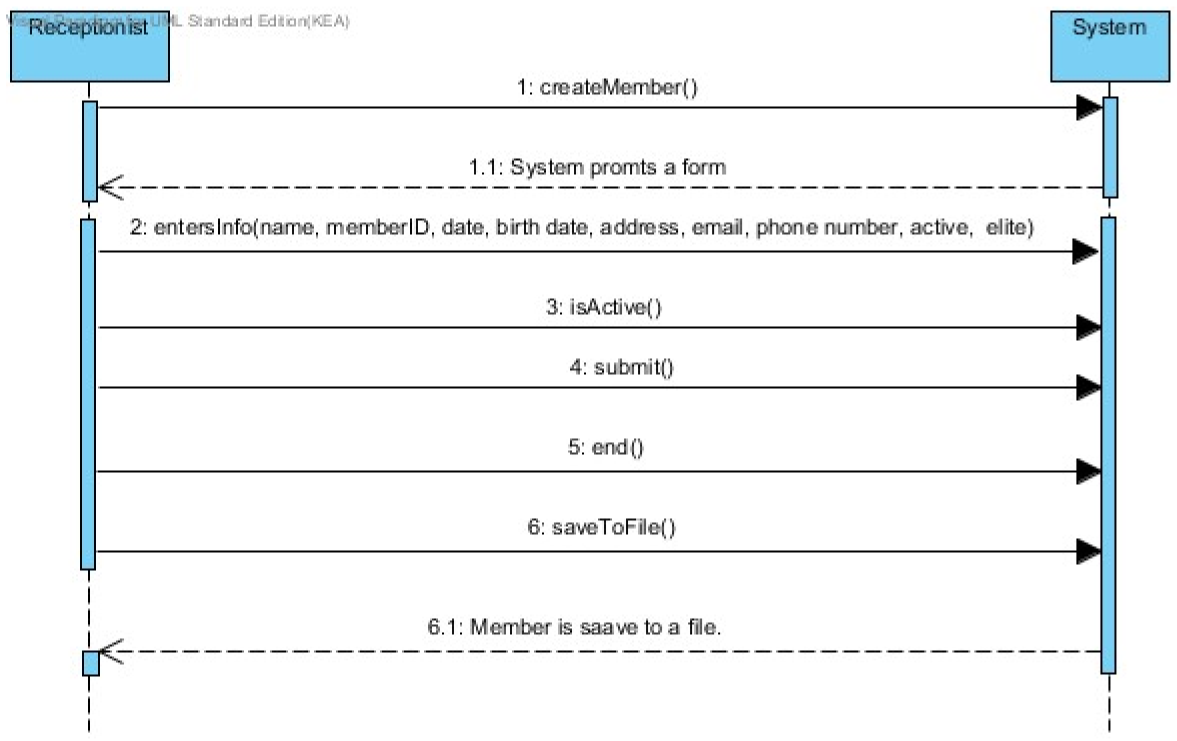
4b. Member is an elite
1. Receptionist checks the "elite checkbox"

6a. Receptionists submits incomplete form
1. System displays an error
2. Receptionist fills in the missing fields

6b. Receptionist inputs information incorrectly
1.Receptionist consults update member use case

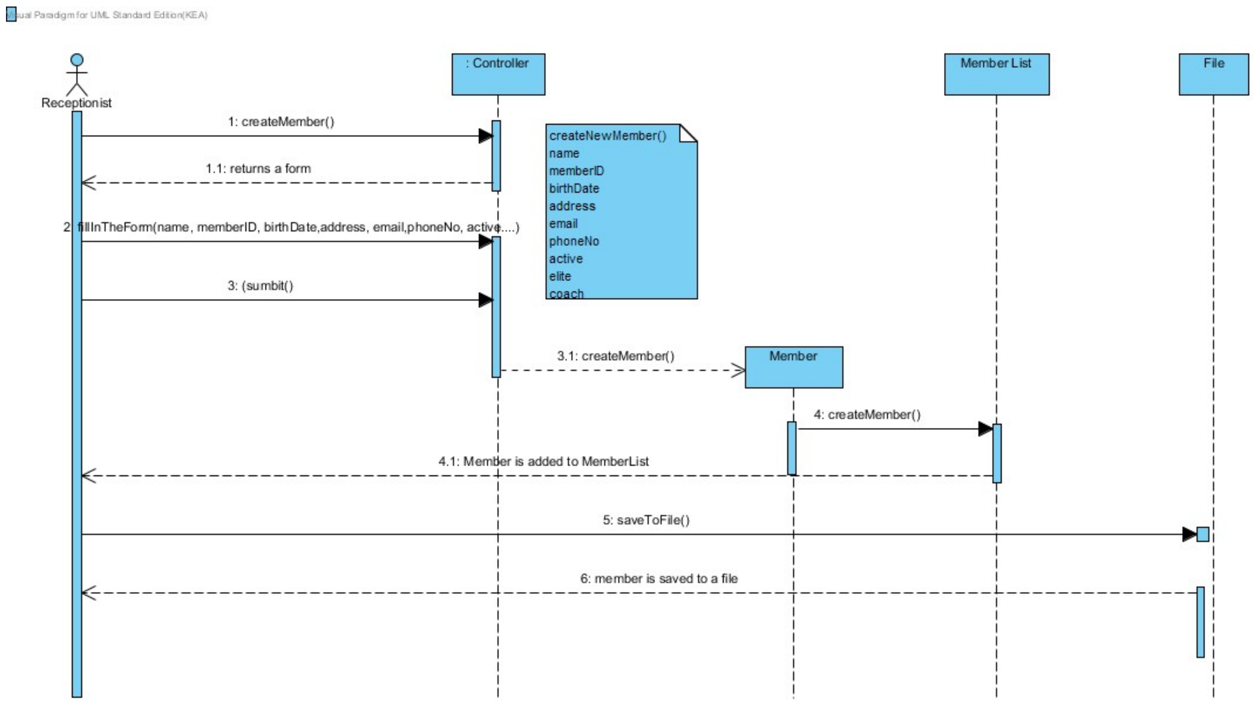
Technology and Data Variations List:
Membership card must be registered by a card reader.

Frequency of Occurrence:
Could be nearly continuous.


Visual Paradigm for UML Standard Edition(KEA)
Receptionist
System
1: createMember()
1.1: System promts a form
2: entersInfo(name, memberID, date, birth date, address, email, phone number, active, elite)
3: isActive()
4: submit()
5: end()
6: saveToFile()
6.1: Member is saave to a file.

Create Member System sequence Diagram
We use system sequence diagram to show how the receptionist interacts directly with the system. These ordering of events is based in our main success scenario.
In this use case, when the receptionist clicks the ”create member” button, the receptionist is requesting the pool party system to add a new member on the system and the system will prompts a form for the receptionist to fill in. The receptionist will also need to specify whether the member is active otherwise member is passive. When the receptionist submits the form, the new member will be added to the ArrayList. When the receptionist click the ”Save to file” then the member will be added to the file and the system displays ”Member is saved to a file”.

We had a small debate as to whether or not to set the members as being active by default or whether or not we wanted to mark them as active. Ultimately we went with the functionality of being able to mark them as passive when registering. This gives us flexibility in registering members that might be coming from out of state (or country) and would prevent annoying paperwork right after moving. They would be able to use the pools almost immediatly after moving.



GRASP Responsibilities

Controller: In createMember use case, the Controller Object is the creator because the Controller creates the member instance by passing some basic information such as name, memberID, birthDate, address, phoneNo, email etc from the GUI textFields.

Information Expert: The Controller object can also be the information expert as it contains all the relevant information needed for creating a member. The MemberList class can also be seen as information experts as it contains all the operations for creating a new member in the ArrayList. The File can also be an information expert as it stores all the information to a file. In sum all process contain information vital to the adding of members to the system.

Low Coupling and High Cohesion:
Create Member use case has relatively low coupling and high cohesion because it is easy to maintain. There are proper delegation of tasks between classes. We have a Member class who has all the information of a member. We have an Elite class who has extra information for elite but inherit everything from the members. We have a MemberList that contains all the operation to create a new member in the ArrayList. We have Controller objects which delegates to the MemberList class.

Fully dressed style

The Swimming club (The Dolphin)

Use case name: Make Payment

Scope: The system under design

Level: User goal

Primary Actor: Club treasurer:

Stakeholders and Interests:

Member: Wants to make a payment. Wants a confirmation receipt.

Club treasurer: Wants to know the type of membership. Wants to know who paid

Preconditions: Club treasurer is logged into the system.

Postconditions: Fee is paid successfully.

Main Success scenario:

1. Treasure click the button payment Info.
2. System displays payment Info screen.
3. Treasure click the button Make payment
4. System prompts a payment form
5. Club treasurer enter member name or id.
6. Club treasurer click search tab.
7. System shows description of member.
8. Club treasurer click pay button.
9. System displays payment details and makes the payment
10. System prints out a receipt.

Extensions:

*A.At any time: System breaks down

1. Club treasurer restart the system.

*B At any time: Incorrect data has been entered

1. Treasurer consults update payment use case

2. Treasurer deletes payment info and repeats this use case

5.a. The system couldn't find member
1.a Treasurer contacts receptionist

8. a. Club treasurer selects "pay" button

8.a.1. Treasurer pays member fee
1. System displays a screen with input fields by card or by cash
2. Club treasurer selects one desired by member

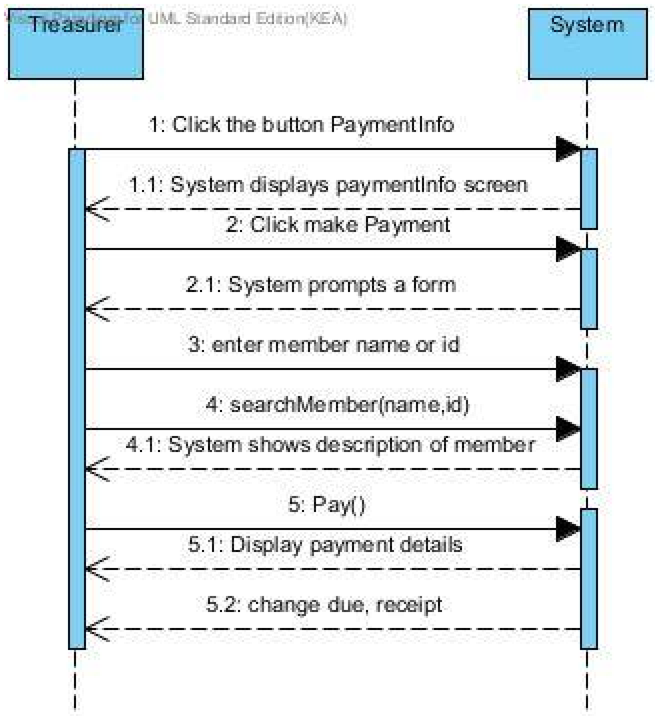
8.a.2 Treasurer pays late summation fee
1. system displays a screen with input fields by card or by cash
2. club treasurer selects one desired by member

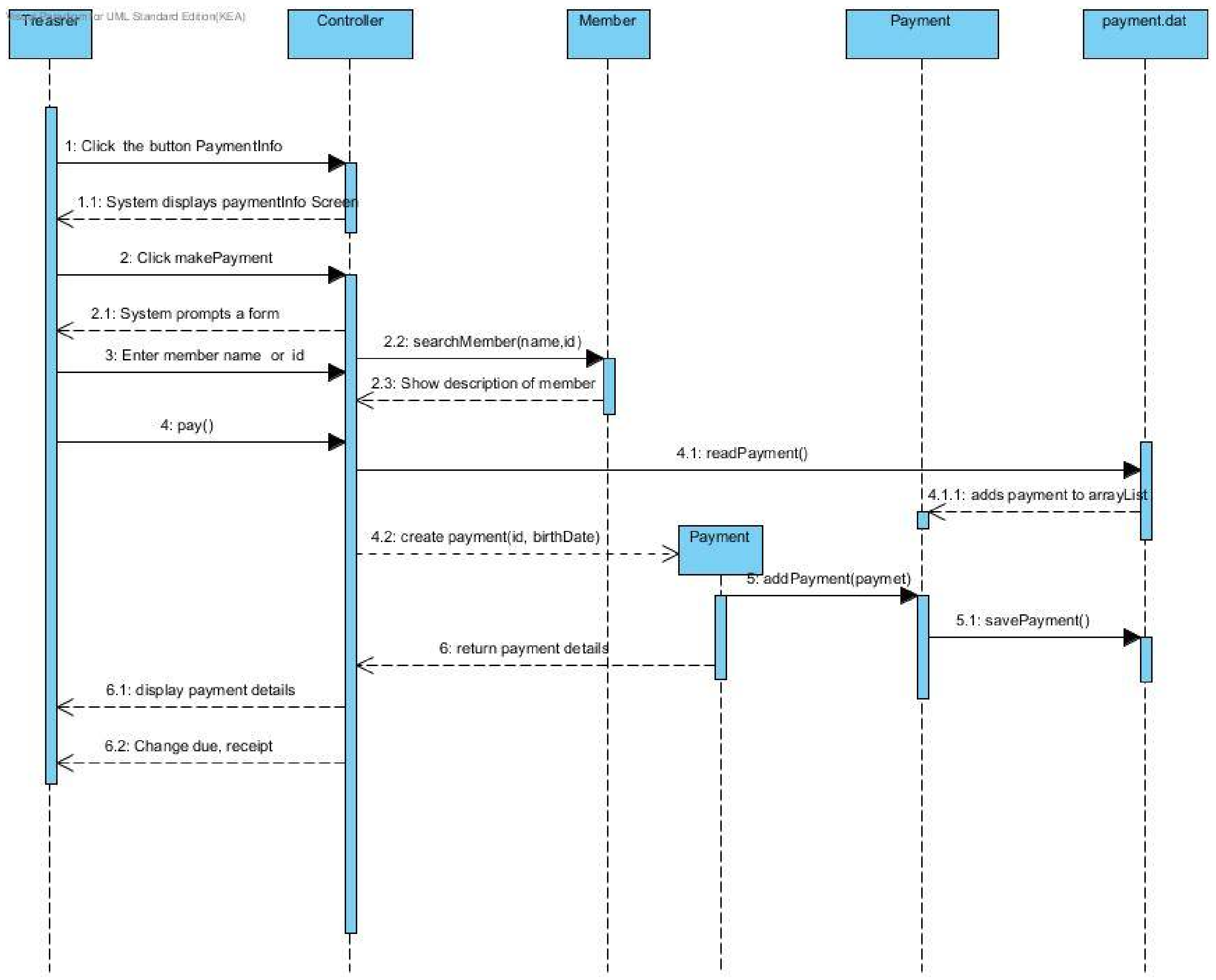
b.The member enter wrong Pincode
1. System display an error
2. Member tries again
3. System displays a confirmation to the Club treasurer

9.a. Receipt is not printed
1. Club treasurer checks if the system has paper for printing

Special Requirements:

User friendly. Language option


Visual Paradigm for UML Standard Edition(KEA)
Treasurer
System
1: Click the button PaymentInfo
1.1: System displays paymentInfo screen
2: Click make Payment
2.1: System prompts a form
3: enter member name or id
4: searchMember(name,id)
4.1: System shows description of member
5: Pay()
5.1: Display payment details
5.2: change due, receipt


Visual Paradigm for UML Standard Edition(KEA)
Treasrer
Controller
Member
Payment
payment.dat
1: Click the button PaymentInfo
1.1: System displays paymentInfo Screen
2: Click makePayment
2.1: System prompts a form
2.2: searchMember(name,id)
3: Enter member name or id
2.3: Show description of member
4: pay()
4.1: readPayment()
4.1.1: adds payment to arrayList
4.2: create payment(id, birthDate)
Payment
5: addPayment(paymet)
5.1: savePayment()
6: return payment details
6.1: display payment details
6.2: Change due, receipt

To create persistence it is important that the content of the file be loaded to the arrayList and the new payment to be added to the complete arrayList before the information is printed to the file. Our project does not append to a file when we print it and instead overwrites the information. For this reason reading the file and adding the data to an arraylist is very important.

Grasp responsibilities

Creator: the controller creates the payment instance passing ID, birthDate

Information Expert: The Controller object can also be the information expert as it contains all the significant information needed for creating a payment. The payment class can also be seen as information experts as it contains all methods for creating a new payment. The File can also be an information expert as it stores all the information to a file. In sum all processes contain information essential to the adding of payment to the system.

Low Coupling: This use case has relatively low coupling as all classes involved are independent.

High Cohesion:

It has high cohesion as all the elements of the class are strongly related to each other. You don’t need to call multiple methods to achieve one goal.

Controller: The Payment class is the controller.

Fully dressed style

The Swimming club (The Dolphin)

Use Case Name: Submit Swim results

Scope: The system under design

Frequency of Occurrence: continuous

Level: User goal

Primary Actor: Swimming Coach

Stakeholders and interests:

Swimming Coach: Wants to be able to submit the data quickly and with minimal effort.

Elite Swimmers: Want their swim times to be entered accurately and quickly so they can see their results moments after each race.

Chairman: similar interests as swimming coach

Preconditions:

A race or timed training has been completed.

Coach is logged into account

Success Guarantee: Time, date, result and discipline have been recorded for the relevant swimmers.

Main success scenario

1. Coach clicks swim Info button.

2. System returns swim info frame.

3. Coach clicks add swim time button

4. System returns add swim time form

5. Coach enters swim data (member ID, swim times, race, discipline)

6. Coach prompted to confirm data

Extensions

A*At any time, System Fails

1.Coach restarts system

2. Coach calls Chairman

*At any time data has incorrectly been added

1. Consult update swimmer use case
2. Notify chairman

1.a Button is unresponsive

1.a.1 Coach tries again.

1.a.2 coach restarts system

1.a.3 coach contacts Chairman

1.b. button redirects to a different Frame

1.b.1 Coach tries again.

1.b.2 coach restarts system

1.b.3 coach contacts Chairman

2.a Different frame is returned

2.a.1 Refer to step 1.b

5.a Coach incorrectly enters swim data

5.a.1 coach consults update swimmer use case.

5.b. System can’t find swimmer

5.b.1 Double check swimmer spelling

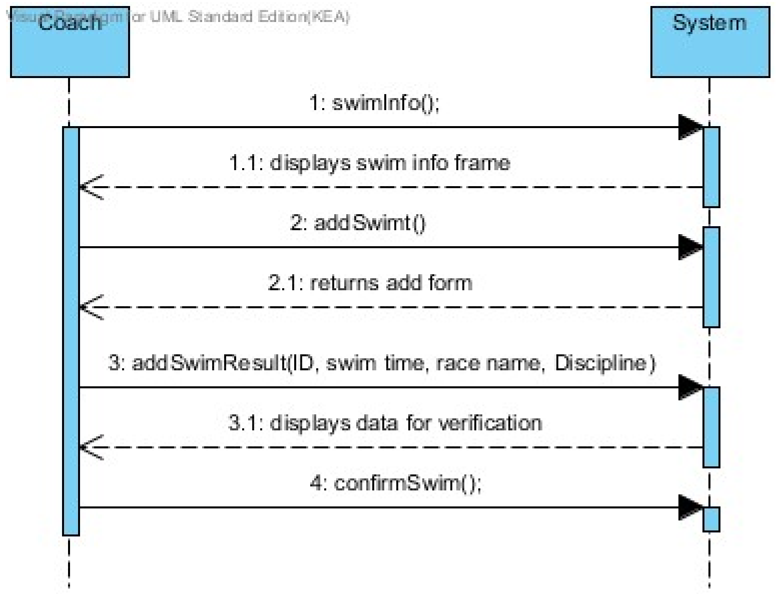
5.b.2 Notify receptionist.

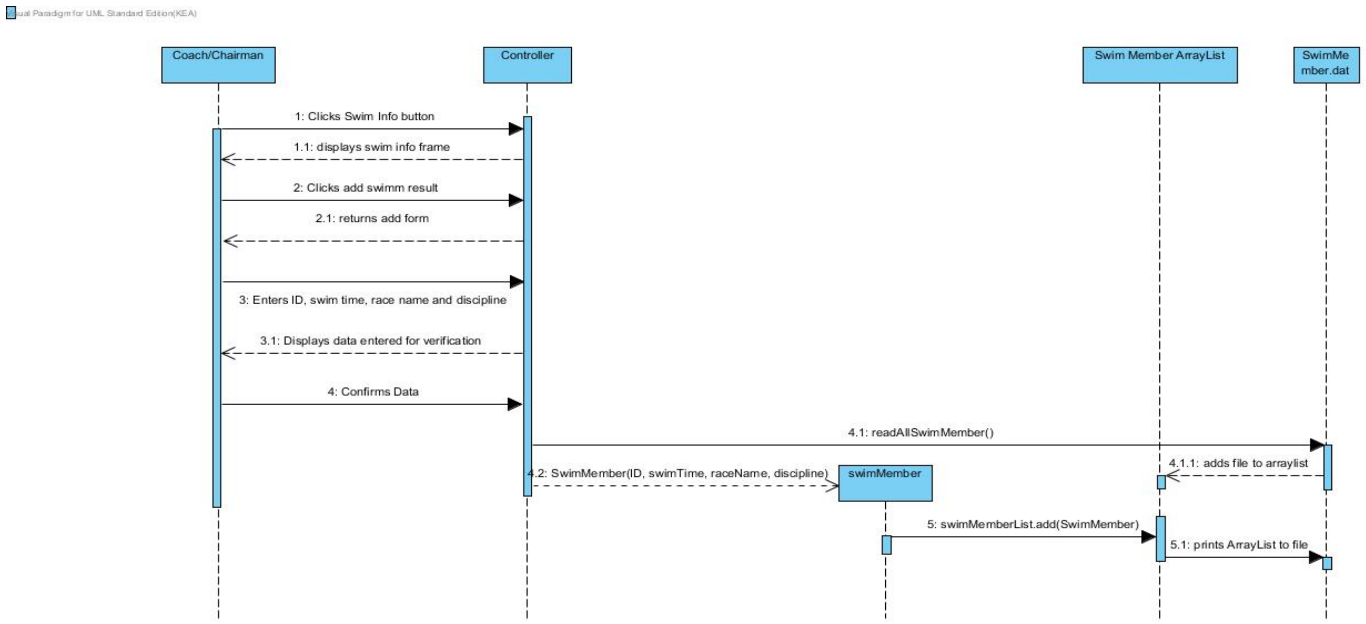
6. System does not prompt verification

    1. re-enter swim data

    2. restart program

    3. notify chairman.


Visual Paradigm for UML Standard Edition(KEA)
Coach
System
1: swimInfo();
1.1: displays swim info frame
2: addSwimt()
2.1: returns add form
3: addSwimResult(ID, swim time, race name, Discipline)
3.1: displays data for verification
4: confirmSwim();


Visual Paradigm for UML Standard Edition(KEA)
Coach/Chairman
Controller
Swim Member ArrayList
SwimMember.dat
1: Clicks Swim Info button
1.1: displays swim info frame
2: Clicks add swimm result
2.1: returns add form
3: Enters ID, swim time, race name and discipline
3.1: Displays data entered for verification
4: Confirms Data
4.1: readAllSwimMember()
4.1.1: adds file to arraylist
4.2: SwimMember(ID, swimTime, raceName, discipline)
swimMember
5: swimMemberList.add(SwimMember)
5.1: prints ArrayList to file

Much like with the make Payment use case if we want to add persistence to the program we need to load the Swim info to an arrayList before we can print the new swim time. For every member added we reprint the arrayList with all the data to a file. In addition to race times we also include the name of the particular race the swimmer has participated in order to be able to distinguish their performance over a variety of different races.

Grasp Responsibilities

Controller - In this use case Creator: The Controller class creates the swim Member instance passing ID, swimtime, raceName and discipline from the GUI textFields.

Information Expert -The Creator can also be said to be the information expert here as it contains all the relevant information needed for adding the swim time(ID, swimTime, raceName, discipline). Other Information experts include the Swim Member Array list and the Swim Member.dat file.

Low Coupling - This use case has relatively low coupling as there are minimal different classes involved.

High Cohesion - It also has high cohesion because these classes requires minimal number of methods to accomplish the work desired. In other words the classes don't constantly call on each other but do so a minimal amount of times.

The Controller is the Swim Member class.

<u>Brief Use Cases</u>

Update Member:
In the event a members info is not up to date (ie has moved, has a new email address or phone number they want to register) the member can request the data to be updated and the Receptionist or Chairman will make the change.

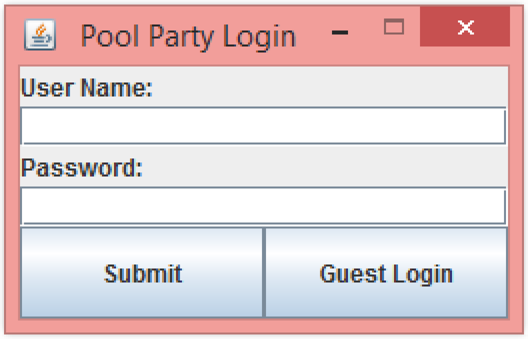
Delete Member:
Because of yearly payment fees we want to have the ability to delete members from the database. Receptionist or Chairman has the ability to delete members upon request by member. All the swimming times are stored in a separate database so as not to lose swim time data when a member is deleted.

View Swimming results:
Elite Swimmers, Swim Coaches, Receptionist and Chairman can look up swimming results, for rally's and training sessions. Results are divided into two categories.
Training and Race which are further subdivided into Junior and Senior
For the Competitions the date, rally and time are visible.
The times are ordered from best to worst

Search Member:
Receptionist, Chairman, Treasurer can look up member info such as name, birth date, address, email, phone number, payment status and whether or not member is active or passive. Additional information will be displayed if the person is an Elite swimmer such as the name of the Coach and Swim Team.

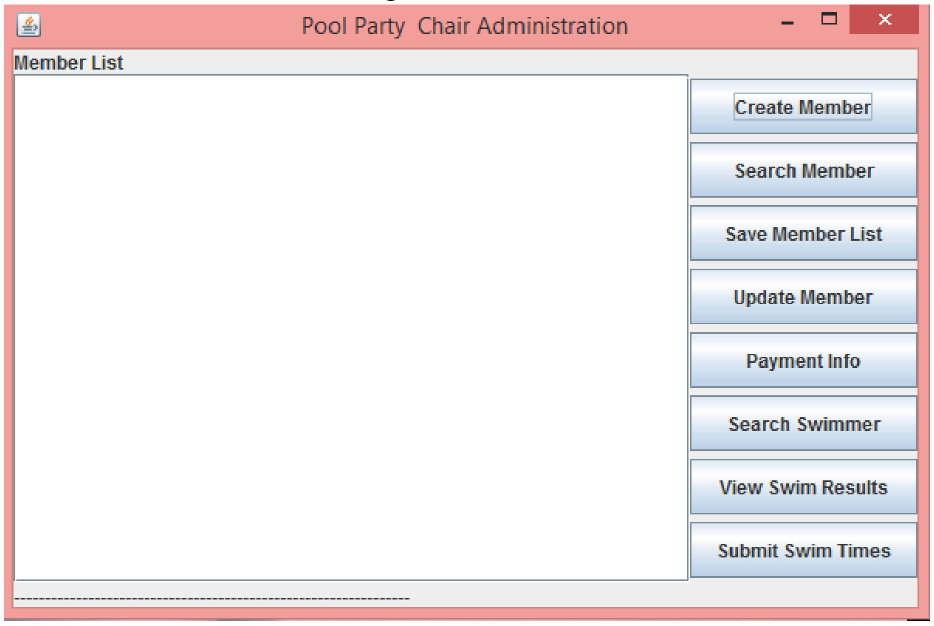
Description of GUI construction

The GUI was designed with simplicity in mind. when the program is run the user can either choose to login with their username and password (Corresponds to their position. In this case only the chairman is available.) or they can use a guest login. The guest login is currently unavailable but only contains search Swimmer and view swim result functionality. In future iterations functionality for treasurers and receptionists will be added that include functions only relevant to their work. This is to prevent the coach from deleting members from the system if he is frustrated with how poorly they swam.



After logging in the frame is divided into two main parts. The bigger area, a textArea will display a list of members and general information. If the person logging in is a coach or treasurer the information displayed will be different and may include swim times or payment information.

On the right Hand side of the frame we use a gridlayout with all of main functionality of the project. In the future We shall group the Member functions and swimmer functions into separate frames as we have done with Payment to prevent the frame from looking too cluttered. This layout however is more convenient for the sake of testing.



With the exception of update Member all buttons include some functionality. Our three use cases we have programmed, Create Member, Search Member and Make Payment(inside Payment info) Represent the only fully functioning aspects of the program. (Subject to further modification in later iterations)

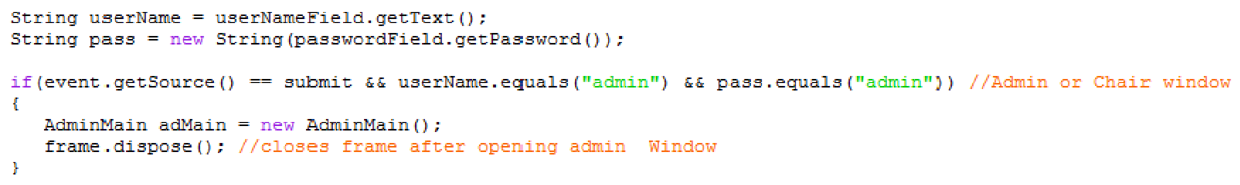
Create Member and Search Member are very similar in design. The search Member use case will prompt a nearly identical frame with the addition of ID. Currently you can only search through the list of members using name. After searching for a member, if the member can be found his or her information will be set in the fields.

Beside Text Fields and Textareas there are a number of other components added to the Gui. In the create Member frame we include checkboxes and in the search swimmer frame there is also a dropdownbox. In addition to that a number of frames include clear buttons which reset the form Fields. In addition to that the login screen includes a Password Field hiding the characters typed from view.

Another note on the GUI is that with the exception of the main screen (Admin) all frames will either dispose on close or will dispose when canceled. This prevents the program from ending and allows for the removal of clutter.

Code - GUI

Starting with the login System

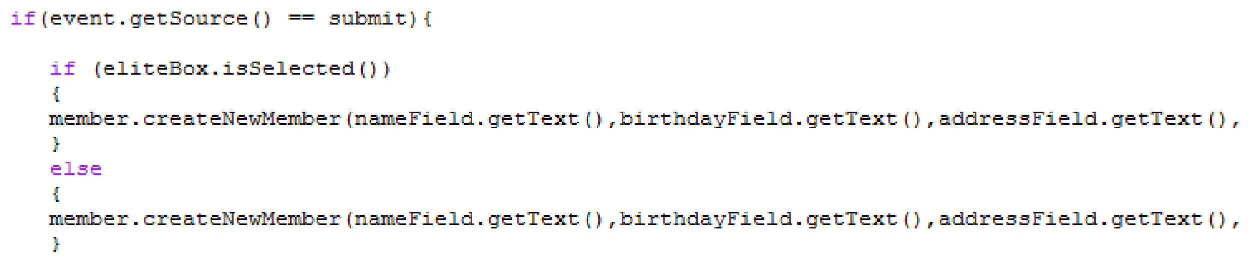


```
String userName = userNameField.getText();
String pass = new String(passwordField.getPassword());

if(event.getSource() == submit && userName.equals("admin") && pass.equals("admin")) //Admin or Chair window
{
   AdminMain adMain = new AdminMain();
   frame.dispose(); //closes frame after opening admin  Window
}
```

This is the most important part of the login system where the if statement checks to make sure what the user entered in the fields corresponds with the appropriate password needed. If the submit button is pressed and the user name and password equals admin then we create an AdminMain object which will open our main window. At this point the login frame is no longer necessary and we dispose of it. frame.setVisible(false) will have done the same job.

The code below is a snippet (end cut off for legibility) of code for the creation of a member using the class constructor and getting the fields from the GUI. The nested if statement ensures that a different member is created if the person is a basic member or is an elite.



```
if(event.getSource() == submit){

   if (eliteBox.isSelected())
   {
   member.createNewMember(nameField.getText(),birthdayField.getText(),addressField.getText(),
   }
   else
   {
   member.createNewMember(nameField.getText(),birthdayField.getText(),addressField.getText(),
   }
```

The Code below shows the search function and how it gets information from the Member class and sets the fields in the Gui to display the information. You can see the creation of the Member object and the use of a number of methods in this class (getMember and searchMember)

```
if(event.getSource() == search)
{
    if (member.searchMember(nameField.getText()) > -1){
        System.out.println("member found");
        Member seMember = new Member();
        seMember = member.getMember(member.searchMember(nameField.getText()));
        idField.setText(Integer.toString(seMember.getMemberID()));
        birthdayField.setText(seMember.getBirthDate());
        addressField.setText(seMember.getAddress());
        emailField.setText(seMember.getEmail());
        phoneNumField.setText(Integer.toString(seMember.getPhoneNo()));
        activeBox.setSelected(seMember.isActive());
        eliteBox.setSelected(false);
```

The code below shows the elite classes constructor. It displays the information that is passed on from the member class and sets elite as being true by default. In addition to that it includes the ability to add coaches and a team.

```
public Elite(int memberID, String name,String birthDate, String address, int phoneNo, String email,
             boolean active, String coach, String team)
{
    super(memberID, name, birthDate,address, phoneNo, email, active);
    super.elite = true;
    this.coach = coach;
    this.team = team;
}
```

Below is the method saveAllMembers. There are a number of interesting aspects about this. In general this code is responsible for writing out the contents of the arraylists to their respective files. This particular method does both members and elites at the same time and uses a simple traversal loop to access every member. One of the more interesting aspects is the try-catch aspect. The action Listener sections of the code seem to be particularly picky when dealing with exceptions. Rather than throwing a FileNotFoundException in the header of the method, we need to use try - catch to catch the error and throw it manually.

```
public void saveAllMembers(){
   System.out.println("saving members");
   try {
      PrintStream outputMember = new PrintStream(new File("Member.dat"));
      for (int i = 0;i< memberList.size();i++){
         Member member = memberList.get(i);
         member.writeMember(outputMember);
      }
      PrintStream outputElite = new PrintStream(new File("Elite.dat"));
      for (int i = 0;i< eliteList.size();i++){
         Elite elite = eliteList.get(i);
         elite.writeElite(outputElite);
      }
   }
   catch (FileNotFoundException e) {
      System.out.println("Error reading file: " + e);
   }
}
```

We have two different methods dealing with calculate age. The current method is shown for its brevity. In this method we have to parse the strings we got as integers so we can do calculations on them. Before we do that however we use the substring method to split the input into day, month and year variable.

```
public int calculateAge(String date){
   int dayDate = Integer.parseInt(date.substring(0,2));
   int monthDate = Integer.parseInt(date.substring(3,5));
   int yearDate = Integer.parseInt(date.substring(6,10));
   int dayBirthDate = Integer.parseInt(this.birthDate.substring(0,2));
   int monthBirthDate = Integer.parseInt(this.birthDate.substring(3,5));
   int yearBirthDate = Integer.parseInt(this.birthDate.substring(6,10));

   if (monthDate > monthBirthDate){
      return yearDate - yearBirthDate;
   }
   if (monthDate < monthBirthDate) {
      return yearDate - yearBirthDate - 1;
   }
   if (monthDate == monthBirthDate){
      if (dayDate > dayBirthDate) {
         return yearDate - yearBirthDate;
      }
      if (dayDate < dayBirthDate) {
         return yearDate - yearBirthDate - 1;
      }
      if (dayDate == dayBirthDate) {
         return yearDate - yearBirthDate;
      }
   }
```

```
//calculates Anual fee by age and activity
public String fee(Member member)
{
    calculateAge(cpr);
    if (age>18 && age<60 && member.isActive())
    {
       Amount = "1600";
    }
    else if(age<=18 && member.isActive())
    {
       Amount = "1000";
    }
    else if(age>=60 && member.isActive())
    {
       Amount = "1200";
    }
    else
    {
       Amount = "500";
    }
    return Amount;
}
```

The above code takes the birthdate (variable is called cpr but it takes the birthdate and calculates fee on this basis. Code is in the payment class.

```
//adds the date of payment
public String pay()
{
   payDate = Integer.toString(dayPay)+"-"+Integer.toString(monthPay)+"-"+Integer.toString(yearPay);

   return payDate;
}

//checks payment status
public String hasPayed()
{
   if(dayPay == date.get(Calendar.DAY_OF_MONTH) && monthPay == date.get(Calendar.MONTH) && yearPay ==
   {
      //System.out.print("needs to pay ");
      this.hasPay = "false";
   }
   else
   {
      //System.out.print("has payed ");
      this.hasPay = "true";
   }
   return hasPay;
}
```

Above code snippets deal with the paying for a member.