Technische Universität Berlin

Datenbanksysteme und Informationsmanagement

AIM 1- HDIS (Advanced Information Management I - Heterogeneous and Distributed Information Systems)

Semester Project “NiteOut”

Decision Engine

WS 16/17

Tuesday, 10. January 2017

**Authors:**

# 

|  |
| --- |
| Aqa Mustafa Akhlaqi |
| Gaurav Vashisth |
| Haralampos Gavriilidis  (deputy group representative) |
| Hekmatullah Sajid |
| Venkat Subbarao Chunduri  (group representative) |
| Yousof Sagr Shaladi |
| Zoltan Andras Lux (management) |

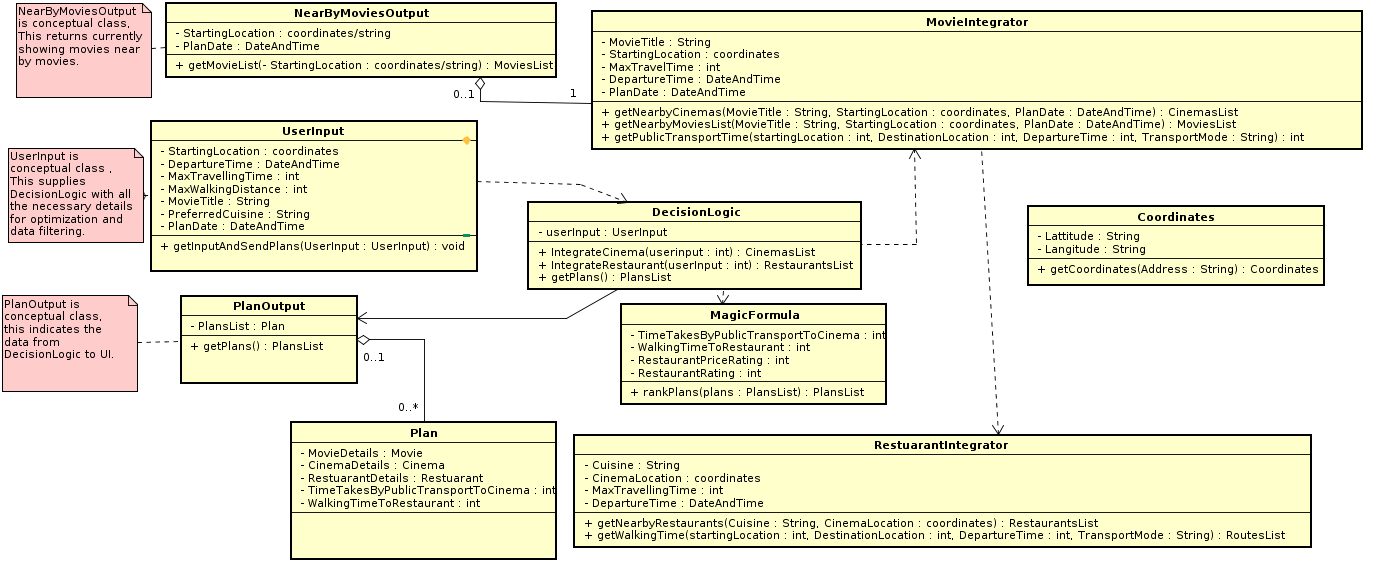
# 1. Introduction

The Decision Engine component is responsible for getting the necessary information about party plans by interacting with the interfaces of the “wrapper” components, that wrap multiple APIs to get the necessary details of public transport, movies and restaurants. The Decision Engine acts as a “Mediator” between the other backend components, and also provides an interface towards the User Interface. As all other backend components, we implemented our component in Java 8. We accesed the other backend components through simple method calls, and we provided a REST interface toward the UI component. The GitLab hosted by the university was also of great help to us, so we could host our project on a private repository with 7 members. In the first stage of the project we created the data requirements to the other backend components and the UI component. In the second stage we consolidated our models. When implementing, we first created a project skeleton, then a prototype, and then we kept adding functionality to satisfy the requirements of the project. During this phase we both provided and received support from other teams on demand. Finally we refactored our code and wrote the documentation.

# 2. Modeling

## 2.1 Class Diagram

The decision engine’s class diagram is as follows:



* User Input: This class gets the input data and sends it to *MovieIntegrator* and *RestaurantIntegrator* to get appropriate movies and restaurant location as per user(s) priorities. It has the following fields.
  + **StartingLocation**: The location where the user(s) are currently located or where they want to gather and start the party
  + **DepartureTime**: The time which user(s) leave current location and go for watching movie
  + **MaxTravelingTime**: The maximum time user(s) want to travel in order to get to cinema
  + **MaxWalkingTime**: The maximum time user(s) can walk while going for movie and restaurant
  + **MovieTitle**: The title of preferred movie
  + **PreferredCuisine**: The preferred cuisine which user(s) want to have after movie.
* **MovieIntegrator**: This class gets user(s) input and send it to Movie model and Movie model returns the list of movie based on the following user(s) preferences. These inputs are already explained in User Input class:
  + **MovieID**
  + **StartingLocation**.
  + **MaxTravelingTime**.
  + **DepartureTime**
* **ResturantIntegrator**: This class get the following inputs and send it to Food model, and Food model returns the restaurant based on these criteria or preferences. Input to this class are as follows.
  + **Cuisine**: The preferred cuisine name
  + **CinemaLocaiton**: The location of cinema where user(s) want to watch movie and after end of movie leave for food
  + **MaxTravelingTime**: The maximum time user(s) want to travel for watching movie and eating food. Previously explained
  + **DepartureTime**: It is time when movie ends and user(s) leave cinema for restaurant
* **GeoService:**This class is used to convert addresses with the attributes “street”, “house”, “city” and “postalcode” into a geolocation of latitude and longitude. It simply exposes the functionality of the Public Transport, therefore it is not displayed on our class diagram. To build the address we use the class **Address** of the Public Transport component which provides these four attributes. These attributes are stored in the address, an instance of the **Address** class. The location, i. e. the latitude and longitude, is calculated through the *getLocation* method which is called through an instance of the **Location** class. The **Location** class provides get/set methods to get and set latitude and longitude and through the **GeocodingService** method *getLocation* we determine the exact latitude and longitude of a given address. For this we use the *convertToGeoCode* method of type Response. It produces as output a JSON containing the latitude and longitude of the following four parameters:
  + **Street**: The street name of a given location
  + **House**: The house number of a given location
  + **City**: The city name where the street and house number is located
  + **Postalcode**: The zip code for a given city, street and house number so the location can be uniquely identified

## 2.2 General Sequence Diagram

The general sequence diagram shows, how the decision engine gets user’s input from GUI component, its interaction with other components (Cinema/Movie, Food/Restaurant and public Transport). It also shows which information are passed to each one of these components and which information are requested. In last the final result (plans in Decision Engine terminology) will be passed to GUI for showing the result to the user.



General sequence diagram detailed description:

The general sequence diagram describes the interaction between the individual components of the NiteOut project. The Decision Engine is responsible for the interaction of the individual components, meaning that there are no direct connections between the components.

* First Step: The user makes a request call for currently available movies shown in Berlin cinemas. The Decision Engine component forwards the call to the CinemaMovie component to find all available movies which are shown in cinemas in Berlin cinemas. The CinemaMovie responds to this call with a set of matching movies
* Second Step: After viewing the list of available movies, the user of the NiteOut application has to choose one and also provide some additional options to the Decision Engine component, in order for the plan generation process to start

The options to be provided are:

* + Starting location: The proposed plans will have this physical location as a starting point location (latitude, longitude)
  + Starting date: The decision engine will generate plans which will consider this date as the earliest possible time
  + Maximum travelling time: This parameter reflects the time the user wants to spend during transportation from the starting point to the cinema, using public transport.
  + Maximum walking time: This parameter reflects the time the user wants to spend during the transportation from the cinema to the restaurant (this trip should contain walking routes, as described in the requirement specifications)
  + Preferred Movie: The movie the user has chosen. The Decision Engine generates plans based on this movie
  + Preferred Cuisine: The type of restaurant the user wants to eat after watching the movie
  + Restaurant Expense Factor (optional): Every restaurant has a value from 1 to 5, that describes how expensive it is (1 is cheapest)
* Third Step: In this step the decision engine will pass the starting-location, date-time and chosen movie to the CinemaMovie component. It responds with all cinemas’ screening information, which are located near the starting location and screening the given movie at the given date and time
* Fourth step: In this step the Decision Engine will ask the Public Transport component for the traveling route and time duration between the starting point and each available cinema. The starting-location, cinema-location, starting time of the movie, maximum travelling time and maximum walking time are arguments which are passed to the Public Transport component. The Decision Engine receives the traveling time for each <starting point, cinema> pair
* Fifth step: in this step the decision engine asks the Restaurant component for restaurants nearby each cinema. The cinema-location, a radius around the cinema location, the preferred cuisine and optionally the expense factor are passed as arguments to the call. The response consists of restaurants matching the given criteria
* Sixth step: After finding possible options for restaurants, another call to the Public Transport component is made for every restaurant, this time for finding the walking time needed between the cinema and a given restaurant
* Final step: in this step the decision engine creates the final plans that consist of combinations of cinemas, restaurants, and the according public transport route. After ordering them using the Magic Formula, the plans are forwarded to the user

The Decision Engine consists of two basic activities, the Decision Logic and the Magic Formula. In the following we describe how the sequence diagram was implemented, and at which points the exclusion criteria were applied, in order to optimize the procedure of plan generation.

# 3. Decision Logic:

The Decision Logic implements the actions described in the sequence diagram. Since there were difficulties in parsing all relevant data fields, e.g. time attributes, to a common format across the individual components, it is the Decision Logic’s job to integrate all incoming data on the fly. To achieve that, it first calls the individual components methods in a previously decided order (as depicted in the sequence diagram), using the output of some components as an input for other components. The majority of the data sanitization happens between the individual component calls. After finishing the individual component calls, the Decision Logic is also responsible for building up instances of the Plan Class. These Plans are just feasible recommendations for complete plans, including the description of the route from starting point to the cinema, the information related to the cinema and movie that was chosen, the route from the cinema to a restaurant matching the user defined criteria and finally information related to the proposed restaurant. In order to comply with certain restrictions regarding external API call number and frequency (Google Places, Foursquare, Here etc), the Decision Logic only computes up to five feasible plans, which are then forwarded to the Magic Formula, to meaningfully order them using different weights, as described in the next section. In following the workflow of the Decision Logic activity is described:

1. Creating of Restaurant, CinemaMovie and Public Transport component instances
2. Get all available Movies by calling appropriate method from CinemaMovie component, and forward them to the GUI
3. After a specific movie has been chosen, the construction of the plans starts:
   1. A call to the CinemaMovie component is made, asking for cinemas and showtimes (sessions) relevant to the movie that was chosen
   2. User constraints are being applied to the returned sessions, more specifically. Therefore the relevant method from the Public Transport component is called for each proposed session, in order to check if the travelling time from the starting location until the cinema is matching the user defined criteria. If a given cinema matches, it is added to the feasible (partial) plans list.
   3. The next step is to find a restaurant, which is reachable by walking from the cinema and also matches the user defined criteria. Therefore, given a (constant) radius around the cinema’s location, the possible choices of restaurants are populated using the appropriate Restaurant Component method call. This call returns the all possible restaurants matching the radius, meaning that there still needs to be applied filtering using the **exclusion criteria**: i) the maximum walking time and the ii) opening hours of the restaurant. If a proposed restaurant matches the criteria, then it is added to the plan, alongside the cinema calculated in the previous step
4. After having built a list of possible Plans, consisting of feasible plans matching the mentioned constraints (excluding criteria) and user defined criteria, the Decision Logic forwards this list to the Magic Formula, to apply further optimizations and finally return an ordered (by score) list to the GUI and respectively to the user. At this point the work of the Decision Logic Activity is done

# 4. Magic Formula:

As previously described Magic formula takes all the shortlisted plans from the Decision Engine and calculates a score for each plan based on cumulative weight of the following parameters:

1. Public transport time to cinema
2. Walking time to restaurant
3. Restaurant price
4. Restaurant rating

These parameters are normalized with Min-Max normalization and multiplied with predefined weights. Once the weighting process done, the Magic Formula orders the given plans by their scores, and finally selects the best plans matching the user’s wishes.  
  
The scoring procedure works as follows.

The following three attributes decrease the rank of a plan, thus their weight given in points has a negative effect on the rating:

1. Public transport. Weight: 15 points.
2. Walking distance. Weight: 45 points.
3. Restaurant price. Weight: 20 points.

The higher the value for the negative attributes the less relevant the plan becomes and is pushed to the bottom of the ordered list.

The positive attribute is:

1. Restaurant rating weighted at 20 points

The positive attribute increases the likelihood for a specific plan as its value increases. For example, the plan with a restaurant rating 4 is preferred over the one with rating 1, so magic formula tries to push up the plan.

The user can also give an optional preferred price. The attribute is to be understood as the maximum preferred price. If this attribute is supplied then the Magic Formula tries to push the plans ahead giving more priority to the plan with price less than preferred price and close to the preferred price with the assumption that user wants to get plans within the preferred price but within the given price rating range. For example, if the preferred price rating is set to 3 then the Magic Formula looks for the plans with a restaurant matching the price rating 3. If there are no such plans, it tries to get plans with restaurant rating 2 followed by 1. If no restaurants with price range 3, 2, 1 are included in the Plans list, then the Magic Formula orders plans with price rating 4 followed by 5 (Even if preferred price rating is given by user, the plan ordering depends on all other parameters like travelling time, walking time, restaurant rating etc. but price rating priority is visible if all these parameters are almost same for every plan).

# 5. Discussion:

In this chapter we discuss the experienced problems and their respective solutions related to the Decision Engine.

## Complexity of API Queries:

When designing the Decision Engine’s data requirements, we had to address the issue of making our calculations in an efficient way, so we will not have to query for transportation times between all cinema and restaurant pairs in Berlin. The decision making process we agreed on is described in detail in chapter 2.2. General Sequence Diagram. As we proposed the idea of getting nearby restaurants from the Food component and nearby screenings of movies from the CinemaMovie component for a specified location, we could avoid making too many requests to the Public Transport component, because with our current API-s we have to make an individual API request for the travelling time between location pairs. We further optimized this process by not considering every possible party plan, but only search until we find 5 good enough movie screening and restaurant pairs from our starting location.

## Communication:

With a project of almost 30 participants’ communication the Decision Engine team’s communication has to be organized in an effective way. We created a slack channel for our group of 7 students, where we could send each other messages. We also used email to a certain extent. Our documents were held in Google Drive. We occasionally used Skype to discuss urgent issues when we could not meet in person and instant messaging on Slack was not sufficient.

## Dependent Tasks:

As the Decision Engine relies on the CinemaMovie component, the Restaurant component and the Public Transport component, we had to find a way to start developing before all these components reached a mature and stable state. First we created a skeleton and looked for example Java REST Tutorials. We received the Food component first, which we could already use to test Java based RESTful webservices. Then we soon got the CinemaMovie component. At this stage we could already start developing our prototype, which was made available to the UI component, so they could start to use our component sooner. As we needed the Public Transport component to get travelling times for our party plans, we could assume at this stage that all nearby cinemas and restaurants are reachable within the specified maximum walking or travelling time. After we received the Public Transport component we could quickly integrate it, and at this stage we already filtered party plans based on most of the exclusion criteria from the specification. The CinameMovie component’s second release provided a partially different interface compared to the first release, therefore we also had to adjust our code to it. This new CinemaMovie release prevented us from having issues with character encoding issues by enforcing the use of movie IDs instead of movie titles.

Furthermore, the UI component was dependent on us, thus we had to deliver them a prototype of the Decision Engine as soon as possible. This task we accomplished in one week after receiving the first versions of the Food component and the CinemaMovie component.

### **Support to UI team:**

We provided personal support to the UI team, to make sure they can use the Decision Engine’s JSON output for development. By giving them access to our Git repository, they could always have the most up-to-date version of our component.

## API Provider Related Risks:

Our project heavily relies on the availability of our API-s. We access these through the respective “Wrapper” components, though we still have to be prepared for temporary unavailability of the API Providers. In this cases we give back empty results for the requests to the Decision Engine.

## Task Distribution:

As there were many Decision Engine members, the equal distribution of tasks had to be managed. We achieved, that everyone wrote both code and documentation, regardless of previous experience with Java.

# 6. Task Distribution:

In our team everyone contributed significantly to both the implementation of the software and the documentation of the project. More details are in the table below:

|  |  |
| --- | --- |
| **Name** | **Tasks** |
| Aqa Mustafa Akhlaqi | Documentation, Movie Input Validation for REST request (MovieService class) |
| Gaurav Vashisth | REST Request for Party Plans, PartyPlan class,Documentation, advise for Magic Formula |
| Haralampos Gavriilidis  (deputy group representative) | Decision Logic class, Magic Formula class, Documentation |
| Hekmatullah Sajid | Partial-Plan and its integration to Decision Logic, Documentation, GitLab Readme |
| Venkat Subbarao Chunduri  (group representative) | Magic Formula class, Decision Logic class, Documentation |
| Yousof Sagr Shaladi | REST request for movies (MovieService class), Documentation |
| Zoltan Andras Lux (management) | Decision Logic,GeoCoding Service, Support to UI team, Documentation, Integrating other backend components |

# 7. Results:

We developed a stable version of the Decision Engine packaged into a JAR file, which also incorporates the CinameMovie, Restaurant and Public Transport components. The software deploys a RESTful web service, that provides JSON output to the User Interface. The source code of the decision enigne is hosted on the university GitLab: <https://gitlab.tubit.tu-berlin.de/luxzoltan/AIM-1_HDIS_DecisionEngine>

## The Decision Engine’s example output for a query for plans:

http://localhost:8080/de/plans/getplan/?Movie=20570&Food=Italian&MaxTravelTime=30&MaxWalk=20&Date=2017-02-12%2014:23&Lat=52.5201848&Longt=13.4099753&MovieDuration=100&PreferredPrice=3



# 8. Glossary of Terms

**Party Plan:** These are the list of plans based upon the weight(importance) given to each attributes involved in the decision making of plan for party. The PlanService class gets a input request from the UI component and returns a list of best 5 plans to the UI layer.

**MovieID** : ID of the movie, that the user has chosen to watch.

**Food** : Name of the food/cuisine that user has decided to eat.

**Movie** **Duration** : Length of the movie that user is willing to watch.

**Lat and Long** : Latitude and Longitude information of a source or destination location.

**MaxTravelTime** : Maximum travel time that user is willing to spend to the movie from his current position to the cineplex.

**MaxWalkingTime** : Maximum walking time that user is willing to spend to the movie from his current position to the cineplex.

**UI** : User interface, it is the layer which is directly accessed by the user and provides several selection of attributes like movie name , food, maximum travel time that user is willing to travel to see the movie, duration of the movie and the maximum walking distance that the user is willing to walk in order to watch the movie.

**GitLab:**  Central repository where all the code base was kept, it is University GitHub like repository and follows same git command.

**MovieIntegrator**: This class integrate the movie component with the decision engine component

**Restaurant** **Integrator**: This class integrate the restaurant component with the decision engine component.

**Public Transport Integrator:** This class integrate the public transport component with the decision engine component.

**Magic Formula:** This is technique is used to determine and obtain best 5 plans out the total plans that decision logic layer decides for.

In this technique certain weightage is given to individual attributes, which then gives positive ratings/score to the best plans and negative to poor plan.

**Decision Logic:** This component is used to integrate all incoming data on the fly. This class calls the individual components methods as depicted in the sequence diagram.

**Sequence Diagram:** Flow of the actions/ data from one component to other is presented in the sequence diagram.