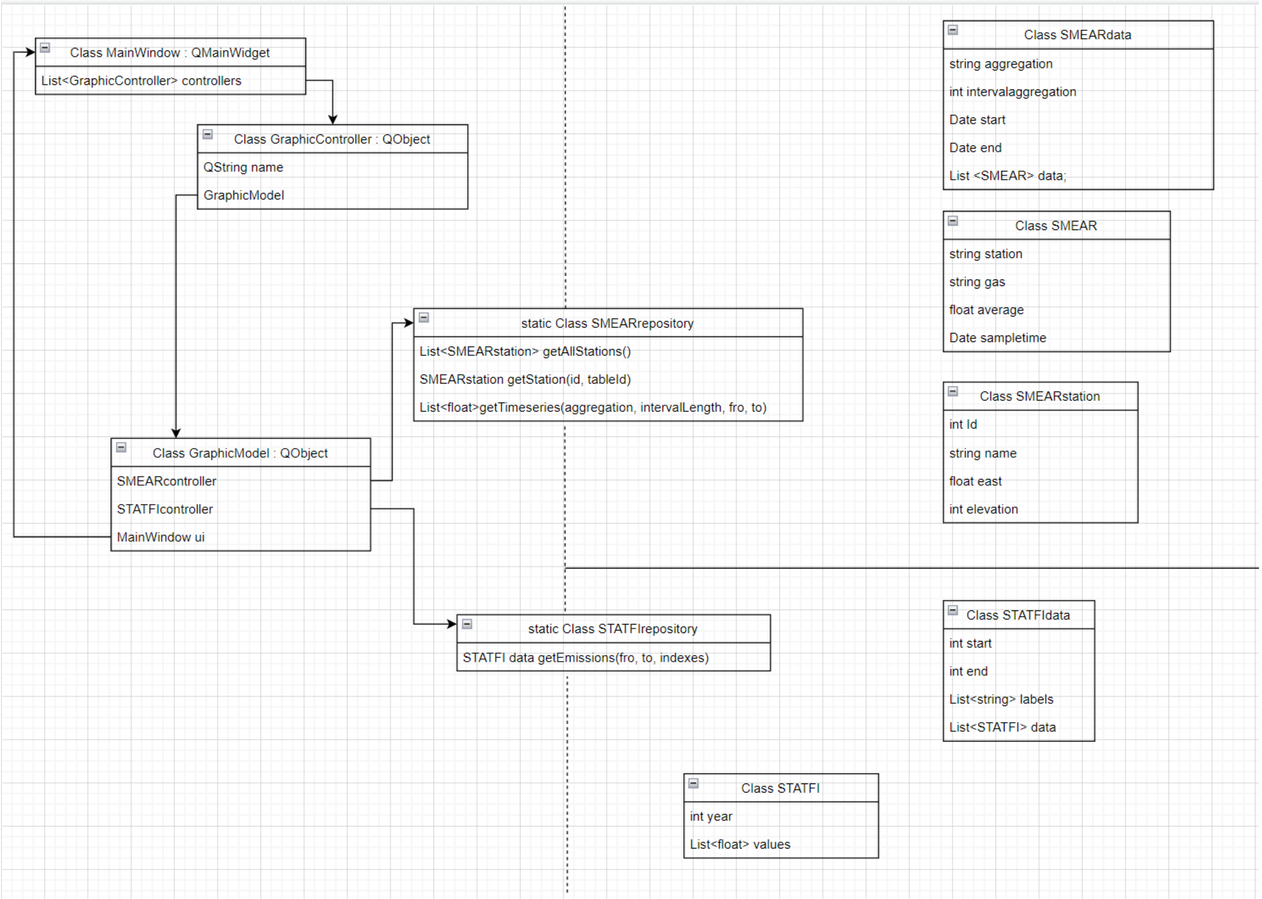
Design document

Midterm submission



Picture 1: UML diagram.

This design was inspired by restful services and MVP model.

Example of how program should function:

1. Changes in UI call controller's functions to change the model's state.
2. When models state is changed. The repository (SMEARrepository) class fetches data from API if it is needed.
3. Function in repository fetches data and stores it into a class (SMEARdata).
4. Function returns data from created object after data is in desired data structure.
5. Model makes changes to UI with the fetched data.

Class MainWindow

- Stores all the controllers
- Uses QtChart library

Class GraphicalController

- Works as a controller for GarphicalModel
- Informs GraphicalModel about changes in UI

Class GraphicalModel

- Updates UI
- Has access to repositories

Class SMEARrepository

- Can fetch data only from SMEARapi
- Returns data structures that are acceptable for GraphicModel

Class STATFIrepository

- Can fetch data only from STATFIapi
- Returns data structures that are acceptable for GraphicModel

Rest of classes are designed to store and process data from API.

Few thoughts:

- Repository and data classes could be inherited from abstract classes
- Service layer could enhance code readability
- Fetching data from API in QT was a new thing so the solution isn't necessarily solid.

Self-evaluation

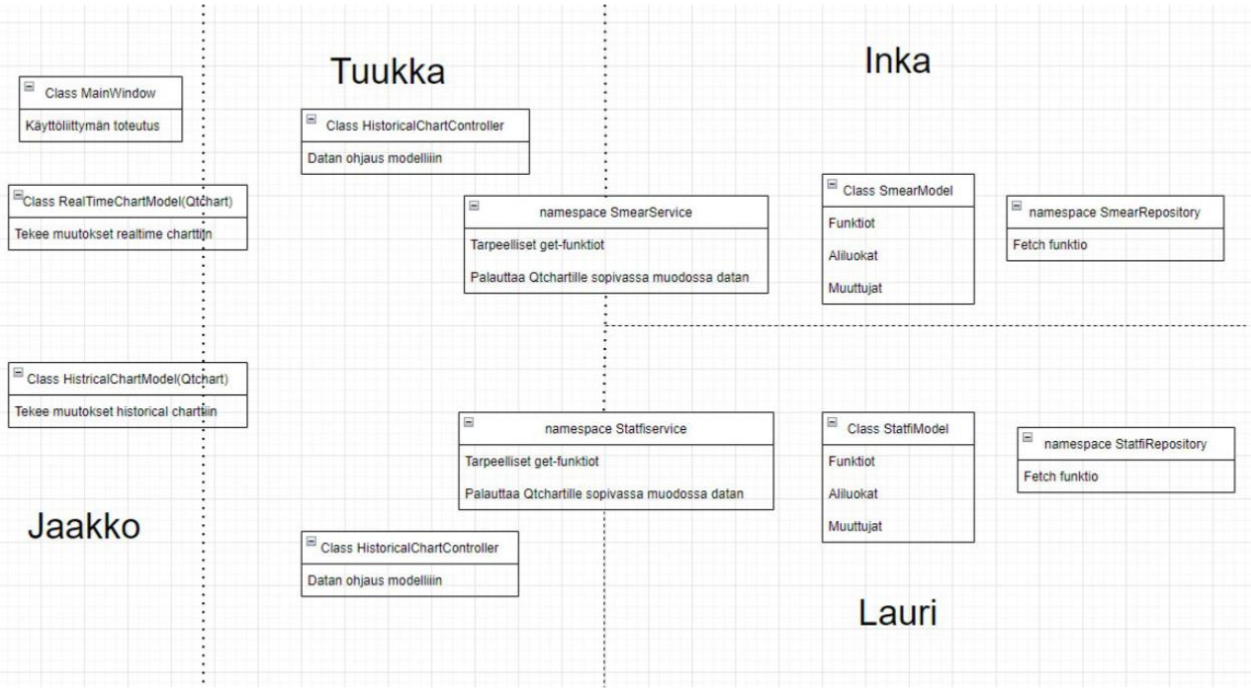
• The original design has greatly supported implementation as it laid a well structured basis for the whole program. It's likely that the original design will give good support for implementing all remaining functionalities too.

• The original design has been followed pretty strictly and all upcoming features will surely be fairly easy to implement using the original plan. No big changes have been made.

In project we have 5 different types of components.

Component	Models(UI)	Controllers	Services	Repositories	ApiModels
Describtion	Stores data for UI components and changes their appearance.	Controls data flows from mainwindow and from services. Also sets values for models	Turns parameters into better formats for repositories. Returns required data using ApiModels.	Makes requests and saves json data into ApiModels	Saves Json data and turns data into formats that can be used in models(UI).

Division of responsibilities



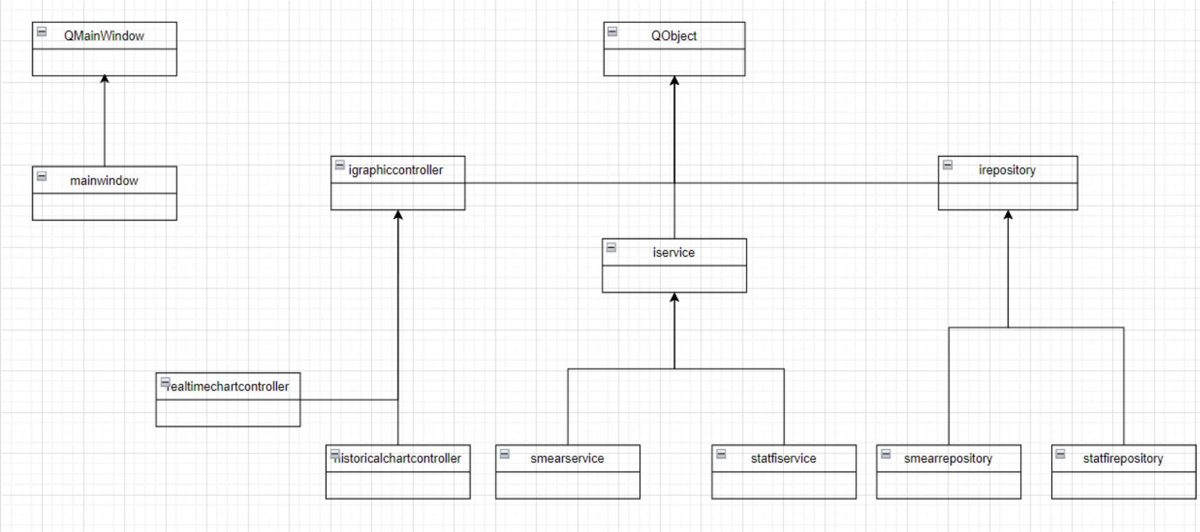
Controllers were changed into classes after creating this picture.

Final Submission 22.4

General descriptions and responsibilities in the class structure

Class	**Description**
ISevice	Abstract base class for service classes
IRepository	Abstract base class for repository classes
IController	Abstract base class for controller classes

Smear/StatfiRepository	Class for handling API requests, creating required queries and saving JSON data.
Smear/StatfiService	Classes handle the JSON data fetched in Repositories. Also responsible for returning the data for the UI by calling the correct ApiModel.
ApiModels	Contains classes that are used to model the raw data into a form that can be used in the UI.
UIComponents	Classes set and render ApiModels that are shown to the user. Controllers pass given commands to Service class.



Internal structure of components

StatfiRepository	
fetchData(url)	*Handles the GET request with given url*
postRequest(url, data)	*Handles the POST request with given url. Data attribute is the query data.*
createQuery(values, timerange)	*Assembles the query for the post request with given values and timerange*
StatfiService	
getYearData()	*Makes a GET request to the API. Returns a vector that has the smallest and largest year.*
createTimerange(stardDate, endDate)	*Creates a timerange to fit the limits of the API data and user specified timerange. Returns a QJsonArray that includes years fitting the criteria.*
handlePostRequest(titles, timerange)	*Calls functions from the StatfiRepository to create a POST request. Returns an apimodel vector that has the needed data stored from the request.*
getHistoricalData(titles, timerange)	*Calls previous StatfiService functions to get the data from the API. Data is then saved to a ApiModel that stores it as QLineSeries. Returns the ApiModel that can be used by the UI*
HistoricalModel	

cleanData(rawData)	*Removes elements that don't have any data in the API.*
toLineSeries(data)	*Creates a map containing the data fit into QLineSeries as the value and returns it.*
SmearRepository	
QDateTime **getStartDate**(QString stationId, QString tableId, QString variableId)	*Returns the date when data is retrievable from API.*
shared_ptr<TimeSeriesTable> **getTimeSeriesData(**QString startDate, QString endDate, std::vector<QString> stations, QString gasType, QString intervalLenght, QString aggregationType**)**	*Returns the data for the given parameters. Fetches data from the API.*
SmearService	
shared_ptr<TimeSeriesTable> **getTimeseries(**QDateTime startDate, QDateTime endDate, std::vector<QString> stations, QString gasType, QString intervalLenght, QString aggregationType**)**	*Makes a request for the SmearRepository to get the data requested.*
QString **getStartDate()**	*Makes a request for the SmearRepository to get the start date for the data.*
TimeSeriesTable	
std::map<QString, std::shared_ptr<QLineSeries>> **toLineSeries**()	Returns line seriesses from classes data which is gathered from SMEAR-APi. Data in class is stored in TimeSeriesRow classes.
database	
bool **saveSmearSettings**(QStringList stations, QDateTime* start, QDateTime* end, QString gas**)**	Saves smear api related settings to database. Return true if saving succeeds.
QSqlTableModel* **retrieveSmearSettings**()	Retrieves all saved smear related settings from the database and returns a pointer to a QSqlTableModel that includes all the data.
bool **saveStatfiSettings**(QDateTime* start, QDateTime* end, QStringList datasets)	Saves statfi api related settings to database. Return true if saving succeeds.
QSqlTableModel* **retrieveStatfiSettings**()	Retrieves all saved statfi related settings from the database and returns a pointer to a QSqlTableModel that includes all the data.
bool **saveCompareSettings**(QStringList stations, QDateTime* SMEARstart, QDateTime* SMEARend, QString gas,QDateTime* start, QDateTime* end, QStringList datasets)	Saves both smear and statfi api related settings from the compare tab to database. Return true if saving succeeds.
QSqlTableModel * **retrieveCompareSetttings**()	Retrieves all saved smear and statfi related settings from the database and returns a pointer to a QSqlTableModel that includes all the data.
SettingsDialog	

SIGNAL void **settingsFinished**(struct SMEARSettings,struct STATFISettings);	Sends settings that user has selected by emitting a signal.
SaveImageDialog	
SIGNAL void **saveResult**(QString message);	Emits signal whether saving picture succeeded.
RealTimeChartController	
void **getTimeSeriesData**(QDateTime startDate, QDateTime endDate, std::vector<QString> stations, QString gasType, QString intervalLenght, QString aggregationType);	Fethes data from SmearService with parameters from UI and stores data into RealTimeChartModel.
void **addStartDate**(QString station, QString gas);	Adds the earliest date with available data for given station and gas to RealTimeChartModel.
void **deleteOldStartDates**();	Deletes old earliest start dates from RealTimeChartModel.
RealTimeChartModel	
void **setStationValues**(*const* std::map<QString, std::shared_ptr<QLineSeries>> &newStationValues);	Sets the data that model holds into a map with station names as keys for each dataset.
QLineSeries ***getStationValues**(*const* QString &station, *const* QString &gasType);	Returns a QlineSeries pointer to a dataset associated with given station and gas type. Scaled to x-axis according to time range.
void **setStartDateTime**(*const* QDateTime &newStartDateTime);	Sets model's attribute holding starting date a new value.
void **setEndDateTime**(*const* QDateTime &newEndDateTime);	Sets model's attribute holding ending date a new value.
void **setEarliestStartDate**(*const* QDateTime &newStartDateRealTime);	Sets model's attribute holding earliest date with available data a new value.
void **deleteAllStartDates**();	Deletes the contents of a container holding all the earliest start dates of available data of different gases and stations.
const QDateTime **getEarlisestStartDate**();	Returns a QdateTime date with earliest possible date with available data.
HistoricalChartController	
std::map<QString, QLineSeries*> **getHistoricalData**(std::vector<QString> titles, QString startDate, QString endDate);	Returns a map containing historical values fetched from StatfiService with keys according to each dataset.
HistoricalChartModel	
std::map<QString, QLineSeries*> **getGasValues**(std::vector<QString> titles, *const* QString &startDate, *const* QString &endDate) *const*;	Returns a map containing historical values fetched from StatfiService with keys according to each dataset. Scaled to x-axis according to time range.
void **setGasValues**(*const* std::map<QString, std::shared_ptr<QLineSeries> > &newGasValues);	Sets data held in model.
MainWindow	
void **updateSettings**(*const* std::vector<QString> stations, *const* QString gas, *const* QString agg, QDateTime realTimeStart, QDateTime realTimeEnd, QDateTime historicalStart, QDateTime historicalEnd, std::vector<QString> historicalTitles = {});	Set's UI's controls according to given parameters.

Design solutions

Controller Service Repository Architecture:

Our solution implements controller service repository architecture. This architecture solution enables developers to develop their solutions on their own layers and this reduces risks of conflicts. This architecture solution also makes the code reusable. This means that our solutions can used in other projects and APIs can be easily connected to new components. Controller service repository architecture also makes it easy to add new data sources. IController, IServices and IRepository work as a guide at implementing new data sources.

Controller Service Repository Architectures implementation works in a following way. Controllers passes UIs data to service layer. Service layer has the business data. Repository layer is for fetching data from APIs. Repository layer also saves data to apimodels which are objects that store the data. Apimodels are passed to back to controller layer after creation.

Model View Controller:

This project uses the MVC design pattern roughly in the following way.

Model: Repositories, Services and ApiModels that handle requesting and modifying the data.
Controller: UI components responsible for passing commands.
View: UI components rendering the model data.

The UI controller components receive the input that is given by the user. For example, the user can request data about CO2 emissions from specific time range.

API Service component handles the request by calling functions from the API Repositories. These Repositories take care of creating the actual request (GET, POST) to the API. Service component gets the returned data and turns it into an API model data structure where it will be stored.

The created model is then read by the UI components. The controllers store the returned data into charts and render the requested data to the user.

Database Class:

Database class is a layer between application and database. Its purpose is to save data. Database class formats data that will be saved to database.

Data flow

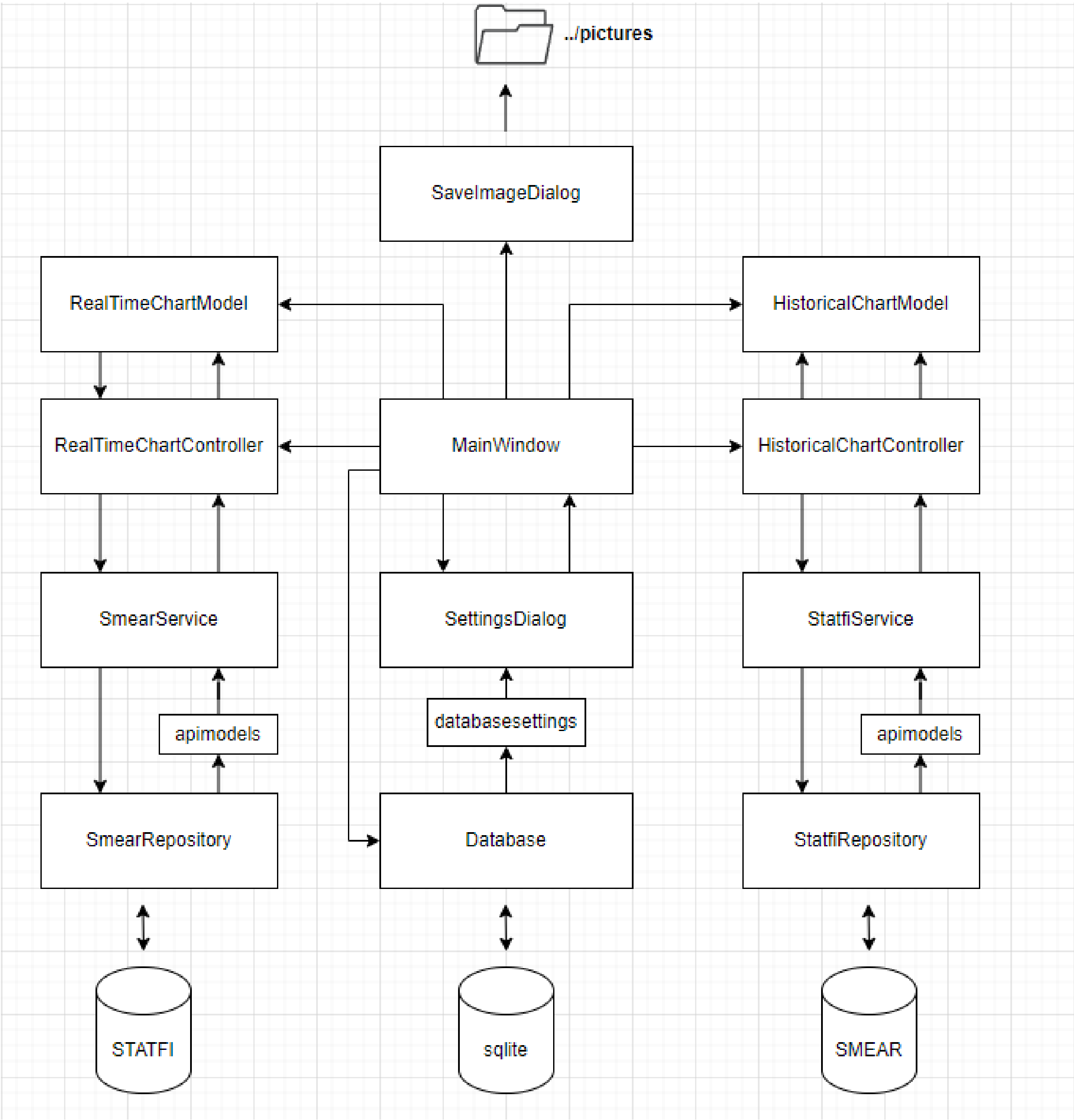


Figure 1: Data flow

Data flow is described in picture above.

Self-evaluation

We were able to stick on our original plan well. We were able to implement our solutions with little as possible communication. Biggest problems in our project were the bottle necks.

There were bottle necks, because UI and code that connected to APIs to it were developed at a different pace. Also, we failed at division of labour for example only one person was responsible for UI and its components. This gave too much pressure for only one person. Especially when project was reaching its end. Also, the lack of conversation about how active models are in implementation of MVC architecture caused problems. For example, our own database(SQLite) was connected to view because of that.