Sauli Savinainen 22.1.2024

CSV-compatible Personnel Management Application with Qt6

Design & Technical Document

Short Description

The personnel management application fulfils all the requirements set for it in the given instructions by utilizing the Qt Framework and best software development practices to achieve both a usable application and a maintainable software architecture.

Using CSV files allows for easy modification, addition, and deletion of employee data without needing to alter the program's source code. The only limitation is the pre-defined attributes and their specific calculation mechanics. In terms of file types, CSV makes most sense as it is the most file format used in sheet management applications such as Excel as well, and writing and reading to the CSV in the program makes the program actually dynamic and useful.

Overall I concluded that it makes no sense to have the different types of employees as separate classes, maybe for a polymorphism tutorial, but otherwise not especially as I had free reins to design the application logic and data structures. In practical applications, let's say for example video games where we would have a character (comparable to an employee), a singular unified format for handling the attributes it has makes the most sense.

I chose to develop the frontend with QML using QtQuick libraries because it is prime example of UI flexibility, since a new QML UI can just be hooked into the program to utilize the backend logic. It also supports dynamic updating and is a good choice for prototyping.

All the necessary data is loaded at once when the application is started and the calculated salaries of each employee are visible in the employee list. Upon selecting an employee, a detailed profile will open with every attribute shown. The calculation logic, data parsing from the CSV file format and list management is handled in the backend.

Every class and component as well as functionality are documented below.

Resources and UI

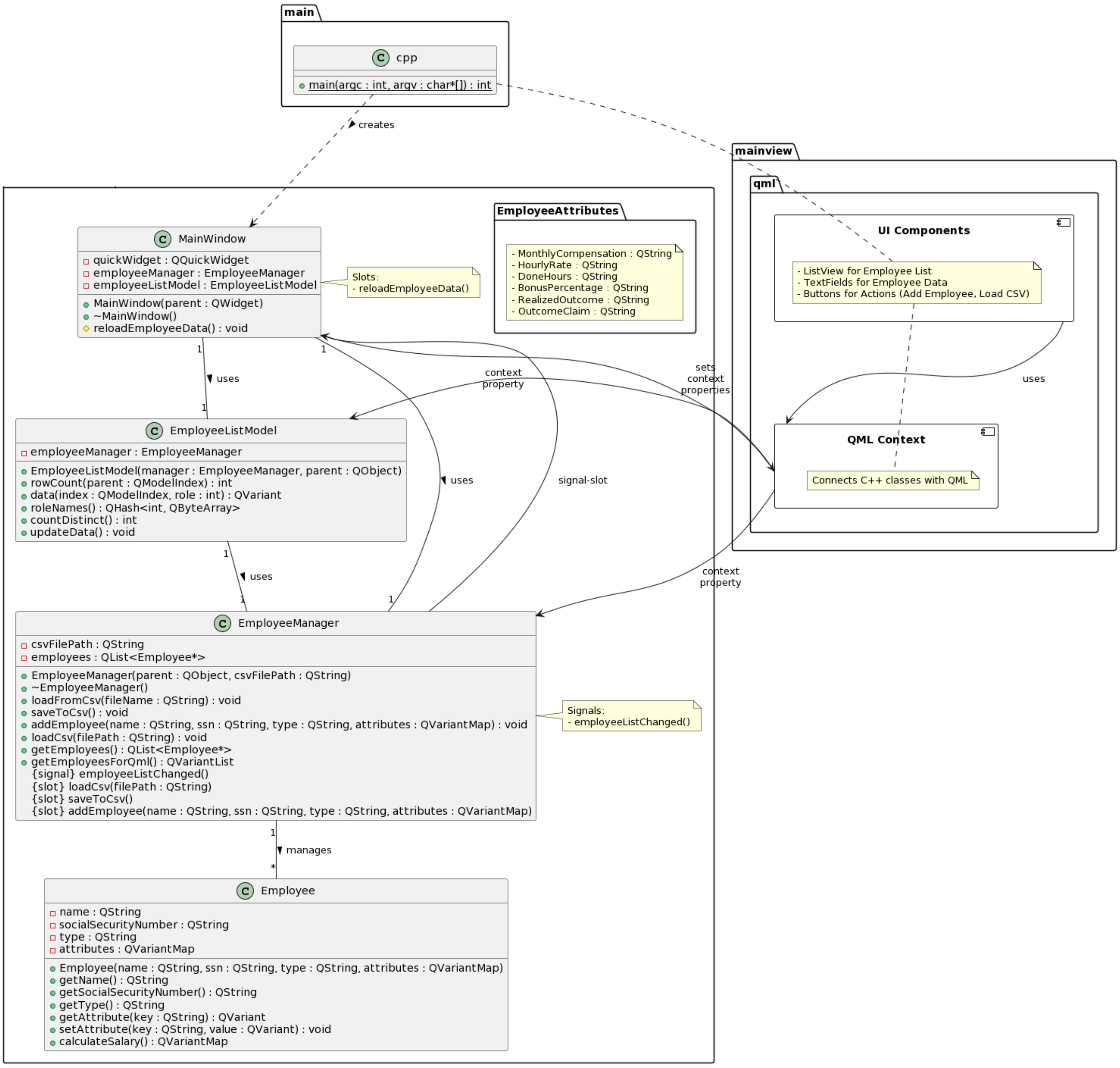
QML & Resources

The use of QML in **mainview.qml** for designing the application's user interface is a conscious choice because of the flexibility the task set out to achieve where the UI needs to be adaptable/replaceable. The clear distinction between the application's logic (handled by the model and controller) and its presentation (managed by QML) exemplifies the MVC separation pattern. The separation allows changes in the UI without affecting the core logic.

Best practice was followed by attaching the .qml file to the resource system. The **.qrc system** allows budling of QML files, images, and other assets. Potentially leads to easier executable compilation and loading times. No graphical resources were used to keep the UI simple.

Sauli Savinainen 22.1.2024

Class Architecture Diagram


main
cpp
main(argc : int, argv : char*[]) : int
creates
MainWindow
quickWidget : QQuickWidget
employeeManager : EmployeeManager
employeeListModel : EmployeeListModel
MainWindow(parent : QWidget)
~MainWindow()
reloadEmployeeData() : void
Slots:
- reloadEmployeeData()
EmployeeAttributes
- MonthlyCompensation : QString
- HourlyRate : QString
- DoneHours : QString
- BonusPercentage : QString
- RealizedOutcome : QString
- OutcomeClaim : QString
uses
context property
sets context properties
uses
signal-slot
EmployeeListModel
employeeManager : EmployeeManager
EmployeeListModel(manager : EmployeeManager, parent : QObject)
rowCount(parent : QModelIndex) : int
data(index : QModelIndex, role : int) : QVariant
roleNames() : QHash<int, QByteArray>
countDistinct() : int
updateData() : void
uses
EmployeeManager
csvFilePath : QString
employees : QList<Employee*>
EmployeeManager(parent : QObject, csvFilePath : QString)
~EmployeeManager()
loadFromCsv(fileName : QString) : void
saveToCsv() : void
addEmployee(name : QString, ssn : QString, type : QString, attributes : QVariantMap) : void
loadCsv(filePath : QString) : void
getEmployees() : QList<Employee*>
getEmployeesForQml() : QVariantList
{signal} employeeListChanged()
{slot} loadCsv(filePath : QString)
{slot} saveToCsv()
{slot} addEmployee(name : QString, ssn : QString, type : QString, attributes : QVariantMap)
Signals:
- employeeListChanged()
manages
Employee
name : QString
socialSecurityNumber : QString
type : QString
attributes : QVariantMap
Employee(name : QString, ssn : QString, type : QString, attributes : QVariantMap)
getName() : QString
getSocialSecurityNumber() : QString
getType() : QString
getAttribute(key : QString) : QVariant
setAttribute(key : QString, value : QVariant) : void
calculateSalary() : QVariantMap
mainview
qml
UI Components
- ListView for Employee List
- TextFields for Employee Data
- Buttons for Actions (Add Employee, Load CSV)
uses
QML Context
Connects C++ classes with QML
context property
1
1
1
1
1
1
1
*

C++ Classes Explained

Application Backend and logic

- **main.cpp** is the entry point of the application. It **initializes the QApplication object and the MainWindow instance**. The main function here sets up the primary application window and enters the Qt event loop, which is essential for handling events and user interactions in the application. This file is minimal yet crucial for starting and running the Qt application.

- **The MainWindow class** serves as the central controlling component of the application, **bridging the C++ backend and the QML frontend.** It initializes *EmployeeManager* and *EmployeeListModel*, sets them as context properties in the QML environment, and handles the main UI setup.

  The class *facilitates the interaction between backend data operations and frontend* UI representation, responding to updates in employee data and refreshing the UI as needed.

Important part of it is the use of the signals and slots defined in the application. It connects the *employeeListChanged* signal from *EmployeeManager* to the *reloadEmployeeData* slot in *MainWindow*. This means that when employee data changes the UI should update too. In terms of the UI, The *QQuickWidget* sets up the primary application window and sets the context for it which links the UI designed in QML with the rest of the C++ application. QML is fetched from the resources. In short this class manages the main UI and ensures that any changes in the employee data are promptly reflected in the UI.

- **The EmployeeListModel** class manages the data and business logic, integrating with the QML UI. By extending QAbstractListModel, the EmployeeListModel ensures dynamic interactions and updates, making it a pivotal interface for accessing and presenting employee data. This approach ensures a clean separation of concerns, where the model manages the data, and the view handles the presentation, enhancing maintainability and scalability – in other words the principles of MVC architecture are correctly met.

- **The EmployeeManager class** acts as a factory and a data handler for all employee objects. It is capable of loading, saving and managing different types of employee data without resorting to separate classes for each employee type. It adeptly handles the loading, saving and importantly parsing of CSV files and creates the employee objects with also assigning them the relevant attributes. The CSV fields parsing functions might look complicated but they really are not.

- **The Employee class** serves as the fundamental building block of the application where a single class is used to manage all types of employees. It functions as a data model that encapsulates the attributes and functionalities needed for each employee.

  By leveraging a *QVariantMap* for storing employee attributes, the class is enabled to handle a variety of employee types without the unnecessary subclassing of simple types or attributes assigned to employees. Instead of having separate classes for each type of employee, this class can handle any employee, whether they are paid monthly, hourly, or are a “salesman”.

  The *calculateSalary* method in this class is a key feature of the application. It can figure out the salary in different ways depending on the type of employee. For a monthly paid employee, it calculates salary one way, for an hourly worker another way, and for a salesman yet another way. These attributes and types need to of course be pre-defined if we do not in the future add a way for users to add their own employee types and calculation methods to the program memory. Overall the employee class flexibility allows it to adapt to different employee types without needing separate classes for each, making the program easier to manage and more straightforward in logic as well.

- In **attributes.h** the *EmployeeAttributes namespace* is a straightforward component that defines key attributes like monthly compensation, hourly rate, and bonus percentage as constants, which are used across the application to handle the employee data. This approach standardizes attribute names, ensuring consistency and reducing errors in data handling.

End of document