**TEHNIČKO VELEUČILIŠTE U ZAGREBU**

**STRUČNI STUDIJ INFORMATIKE**

Matej Lelek

0246061169

EDITOR ZA IZRADU RASPOREDA PROSTORIJA

ZAVRŠNI RAD br XXX.

Zagreb, rujan, 2017.

# Orginal Završnog zadatka

# Sažetak

# Sadržaj

Contents

[Orginal Završnog zadatka 2](#_Toc492497187)

[Sažetak 4](#_Toc492497188)

[Sadržaj 5](#_Toc492497189)

[Uvod 7](#_Toc492497190)

[Korištene tehnologije 8](#_Toc492497191)

[C++ 8](#_Toc492497192)

[Qt Creator 9](#_Toc492497193)

[Opis projekta 11](#_Toc492497194)

[Struktura projekta 11](#_Toc492497195)

[Povezanost i pripadnost objekata 14](#_Toc492497196)

[Opis klasa 16](#_Toc492497197)

[AUserWindow klasa 16](#_Toc492497198)

[EditorWindow klasa 19](#_Toc492497199)

[ClientWindow klasa 22](#_Toc492497200)

[Floor klasa 24](#_Toc492497201)

[Room klasa 26](#_Toc492497202)

[Global klasa 27](#_Toc492497203)

[Korištenje programa 28](#_Toc492497204)

[Moguća poboljšanja 29](#_Toc492497205)

[Zaključak 30](#_Toc492497206)

[Popis oznaka i tablica 31](#_Toc492497207)

[Literatura 32](#_Toc492497208)

[Prilozi 33](#_Toc492497209)

# Uvod

Tema ovog završnog rada je izrada editora za izgradnju i rapoređivanje prostorija i katova unutar neke ustanove te korištenje te iste izrađene sheme od strane korisnika kako bi se mogao orijentirati i lakše pronaći prostoriju koju traži. Razlog zbog kojeg je baš ova tema odabrana kao tema završnog rada je jer studenti često imaju problema s orijentacijom unutar neke ustanove ukoliko se još nisu upoznali sa rasporedom prostorija i dvorana, a isto vrijedi i za sve ostale ljude bez obzira u kojoj se ustanovi nalaze. Iako je ovo programsko rješenje u početku bilo namjenjeno za izradu sheme prostorija unutar fakultetskih ustanova može se koristiti i za bilo koji drugi objekat u kojem se potrebna manja ili veća razina orijentacije kako bi se pronašla željena lokacija. Glavni zadatak je bio ostvariti što lakšu izradu tlocrta neke ustanove, osigurati to u što kraćem vremenu, omogućiti lako dodavanje, brisanje ili izmjenu već postojećih prostorija i katova bez poznavanja bilo kakvih naprednih alata ili tehnika za editiranje te omogućiti da se sve informacije o kordinatama, prostorijama, katovima, imenima, itd. o nekoj zgradi koje su zapisane u određenom fajlu mogu lako prenositi, mijenjati i brisati te biti uporabljive bez obzira na promjene samog programa ukoliko se to dogodi u budućnosti.U sljedećim poglavljima ćemo vidjeti koje smo to sve tehnologije koristili, zbog čega baš njih te kako su nam pomogle u ostvarivanju cilja, kakva je bila struktura projekta i zašto je baš takva, kako nam je samo strukturiranje klasa pomoglo u što bržoj izradi programskog riješenja, kako se zapravo koristi editorski, a kako korisnički program te koja su moguća poboljšanja projekta, na koje smo sve probleme nailazili i koji bi to bili bolji načini rješavanja tih problema.

Korištene tehnologije

Prije bilo kakvog planiranja projekta bilo je potrebno odabrati sve metode, alate i tehnologije koje će se koristiti kako bi taj projekt ostvaren. Razlog zbog kojeg je to potrebno odraditi prije je jer neće sve tehnologije i alati pružiti mogućnosti i lakoću razvoja te neće imati sve metode kojima se može što lakše ostvariti cilj te doći do konačnog riješenja. Pod tehnologije i alate se u ovom slučaju misli na korištene programske jezike te razvojno okruženje unutar koje će se razvijati sami projekt.

## C++

Za programski jezik je odabran C++ iz razloga jer je najopširniji programski jezik u smislu funkcija koje se s njim mogu ostvariti te omogućuje veliku razinu kontrole nad svim korištenim podacima. Pod to ulazi čitanje iz datoteka, pisanje u datoteke, njihovo stvaranje, čitanje i zapisivanje slika te svaka ostala vrsta kontrole nad resursima koje se koriste, a tome ponajviše doprinosi velik broj biblioteka s kojima se mogu ostvariti sve naveden funkcije bez previše napora. Korištenje biblioteke za JSON fajlove je znatno doprinjelo brzini razvoja aplikacije jer je potrebno dugo vremena ukoliko bi se pokušao napraviti parser za navedene funkcionalnosti čitanja i pisanja. C++ programski jezik nam također daje puno veću kontrolu nad memorijom nego drugi jezici što može znatno utjecati na kapacitet memorije s kojom raspolažemo i onom koja je trenutno zauzeta, a s obzirom da se memorija na HEAP-u koju zauzimaju svi resursi mora ručno rezervirati i osloboditi, barem kada se govori od varijablama kojima upravljamo, ukoliko je to potrebno može se precizno znati kada će doći do određenih problema zbog preopterećenosti cijelog sustava za razliku od drugih programskih jezika. Ukoliko se zna koristiti brzina koja se ostvaruje je znatno veća od svih ostalih programskih jezika, a osim toga još imamo i veliku razinu skalabilnosti što čini C++ dobrim kandidatom kada se radi na projektima sa velikom količinom alokacije/dealokacije memorije te čitanja/zapisivanja u i iz datoteka. Posljednji razlog zbog kojeg smo odabrali C++ je jer je korišteno razvojno okruženje bazirano baš na tom jeziku te se s njime mogu ostvariti najveće funkcionalnosti i u najvećoj mjeri.

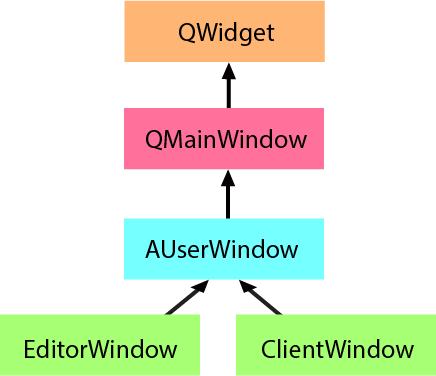
## Qt Creator

Za razvojno okruženje je odabran Qt jer omogućuje izradu programa i aplikacija za različite platforme među kojima su Windows, MAC-OS, Linux, Android, iOS, itd. te se sam program razvija samo jedanput, a zatim se već postojećim baznim kodom razvijenu aplikaciju možemo postaviti i koristiti na različitim operacijskim sustavima bez izvršavanje prevelikih promjena. Qt creator podržava QWidget-e koji su osnovne komponente za izgradnju grafičkog sučelja(GUI) među koje spadaju buttons, labels, itd. Widget se postavlja negdje na sučelje prozora ili se prikazuje u zasebnom prozoru. Svaki widget nasljeđuje QWidget klasu koja nasljeđuje QObject. Često se koristi QWidget kako bi stvorili početni prozor na kojem će biti pozicionirani svi ostali widgeti koji pripadaju tom prozoru. Ukoliko želimo stvoriti novu instancu neke od klasa QWidget-a postoji mogućnost da se toj instaci dodjeli roditeljski widget te ukoliko se on obriše brišu se i sva njegova djeca[1]. Takav način ostvarivanja relacija i ovisnosti između pojedinih elemenata na sučelju omogućuje puno lakše upravljanje i pozicioniranje te se ostvaruje svojstvo pripadnosti gdje se mogu jasno vidjeti zašto je baš neki od tih widgeta roditelj/dijete drugom widget-u na sučelju. S obzirom da QWidget nije abstraktna klasa od nje se na lagan i jednostavan način može dobiti bilo koji novi widget sa funkcijama i karakteristikama koje su potrebne te se time omogućuje stvaranje proizvoljnih widgeta koji mogu imati određena svojstva nekih već postojećih widget klasa. Jedan od najvažnijih razloga zbog kojeg je odabran Qt creator je prvenstveno jer podr\ava funkcionalnost da se po svakom widget-u može crtati pomoću QPainter widget-a. QPainter omogućuje crtanje oblika tj. poligona, povlačenje linija te odabir boja i debljine, ispunjavanje poligona bojom i ostale crtaće funkcije. O QPainter-u i kako se koristi ćemo više čuti u sljedećim poglavljima.

# Opis projekta

## Struktura projekta

Struktura projekta opisuje kako su klase međusobno povezane, kakva sve sučelja postoje, kakva je zavisnost pojedinih klasa i djelova programa te koje klase nasljeđuju druge klase te zbog kojeg razloga.



Slika 1 Prikaz prozora aplikacije

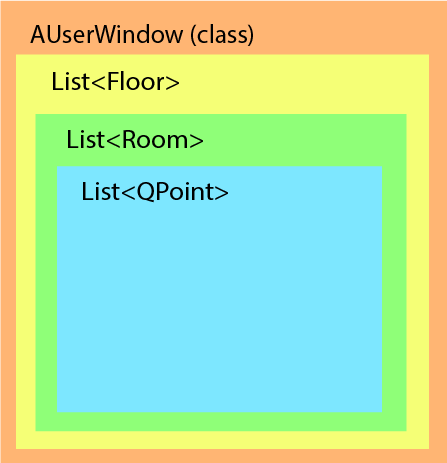
Window klase su one koje definiraju koji je trenutno prozor otvoren tj. naš trenutno otvoreni/aktivni prozor će biti povezan s jednom od window klasa. To su sve zapravo samo proširene QMainWindow klase. Možemo vidjeti da na *slika 1* imamoEditorWindow i ClientWindow klase. One su zapravo Widget klase za naš trenutno otvoreni prozor koji može biti editor(EditorWindow) ako dizajniramo našu shemu prostorija ili korisnik(ClientWindow) ukoliko samo koristimo aplikaciju da nađemo određenu prostoriju. EditorWindow klasa sadržava sve funkcije koje editor mora imati te isto vrijedi i za ClientWindow, dok se u obje klase definira *pokazivač na jednu* od UI formi koje smo prethodno definirali. Forma je zapravo ništa drugo nego dizajnirani prozor u kojem možemo dodavati različite Widget-e te ih raspoređivati i pozicionirati na ekranu. Nakon što pokrenemo jedan od prozora bilo da je to EditorWindow ili

ClientWindow, moramo imati definiran UI pokazivač kako bi znali koju formu tj. koje grafičke elemente(Widget-e) treba prikazati na tom prozoru. Dodanim Widget-ima dajemo neki ID u obliku imena te im pristupamo preko UI pokazivača te je on zapravo referenca na formu preko koje se može doći do reference za bilo koji od Widget-a koji se nalaze unutar te iste forme.

EditorWindow i ClientWindow nasljeđuju AUserWindow jer su im neke funkcije i karakteristike zajedničke kao što je bojanje/filanje trenutno aktivno prostorije te pokazivač na tu prostoriju. Trenutno aktivna prostorija ne mora imati isto značenje u obje klase tj. dok u EditorWindow klasi to označava prostoriju koja je trenutno aktivna za uređivanje u ClientWindow klasi će to značiti soba koja je odabrana putem padajućeg izbornika te ju korisnik pokušava pronaći. AUserWindow nasljeđuje QMainWindow klasu koja već ima definirana mjesta gdje će se nalaziti widget-i kao što su widget-i za status i toolbar te baš iz toga razloga naš AUserWindow neće direktno nasljeđivati QWidget klasu. Iako mi zapravo ne koristimo status, toolbar i ostale predefinirane widget-e nepisano pravilo je da kada god postoji neki novo definirani prozor da se ne nasljeđuje QWidget direktno iako nebi bila nikakva greška niti kršenje standarda ukoliko bi se to implementiralo na takav način. QWidget je bazna klasa za sve ostale klase koje nešto iscrtavaju na prozoru te se tu ne misli na crtanje u smislu renderanja soba i katova već na to da tu klasu moraju sasljediti i neke osnovne klase za komponente/widget-e među koje spadaju QPushButton, QCheckBox, QRadioButton, QLabel, QLineEdit, QGroupBox, QHBoxLayout, QVBoxLayout, QGridLayout, QFormLayout, QListView, QGroupBox,QPlainTextEdit, QProgressBar, QCalendarWidget, itd. kako bi se mogli prikazati na ekranu. QWidget također podržava neke osnovne atribute kao što su širina, visina i pozicija po X i Y osi te funkcije za njihovu izmjenu što se može vidjeti jer se i pojedinim QWidget elementima kao i cijelom aktivnom prozoru mogu podešavati dimenzije i njihova pozicija na ekranu ili unutar aktivnog Window prozora. Postoje još brojne druge karakteristike koje se nasljeđuju od QWidget klase no za ovaj projekt to nije bilo previše bitno. Važno je spomenuti da su QWidget i QMainWindow klase koje su dio Qt razvojnog okruženja dok su AUserWindow, EditorWindow i ClientWindow klase napravljene samo u svrhu ovog projekta sa svim dodatnim potrebnim funkcionalnostima.

## Povezanost i pripadnost objekata

Kako bi bila postignuta bolja organiziranost koda unutar projekta određene klase služe kao menađeri za druge klase ovisno o tome kakva je ovisnost među njima.



Slika 2 Kako su osnovne klase unutar projekta povezane

AUserWindow u sebi ima listu svih katova, svaki od tih katova unutar sebe ima listu prostorija koje se nalaze na tom katu, a svaka prostorija ima listu točaka/kordinata od kojih je ta soba izrađena. AUserWindow će unutar sebe imati funkcije za dodavanje, brisanje, organizaciju i izmjenu svih katova, svaki kat će unutar klase Floor imati funkcije za dodavanje, brisanje, organizaciju i izmjenu svih soba/prostroija, svaka soba/prostorija će unutar klase Room imati funkcije za dodavanje, brisanje, organizaciju i izmjenu svih kordinata ili točaka koje čine tu sobu. Svaka klasa višeg stupnja, a u ovom slučaju se to odnosi na svaku klasu koja unutar sebe ima listu nekih objekata kojima upravlja će imati ulogu menađera za listu objekata koju u sebi sadrži. S takvim dizajnom se omogućuje da klasa ne mora upravljati sama sobom već se ta zadaća prebacuje na klasu koja pod sobom ima neku drugu klasu tj. umjesto da imamo neku javnu statičku listu prostorija te unutar klase Room definiramo statičke funkcije za njeno upravljanje, dodavanje ili brisanje to se prebacuje na instancu klase Floor. Ovakav dizajn nije uvjek poželjan te je u nekim lsučajevima potrebno kreirati više klasa gdje svaka ima jednu zadaću no s obzirom da u ovom projektu uvjek postoji jedan aktivan AUserWindow objekt(bilo da je u obliku EditorWindow ili ClientWindow) te se unutar njega nalaze svi podaci o katovima, prostorijama i točkama nema smisla da kreiramo više klasa koji bi služili kao kontroleri.

Drugi način na koji bi ovo moglo biti napravljeno je da klase Floor i Room služe samo da se unutar njih nalaze informacije vezane za njih dok bi neke druge klase služile kao kontroleri koji bi raspolagali s tim informacijama. Razlog zbog kojega to nije potrebno je jer naše klase za prozore(EditorWindow ili ClientWindow) te klase Floor i Room nisu velike te nemaju prevelike funkcionalnosti zbog kojih bi njihovo razlaganje na manje cijeline bilo korisno, a da smo se odlučili za takav dizajn povećali bi opseg projekta bez ikakvog razloga te bi smanjili brzinu njegove izrade. Posebne kontroler klase bi koristili ukoliko bi unutar neke klase imali prevelik broj varijabli i funkcija te bi ih razdvojili u manje funkcionalne cijeline kako bi cijeli projekt bio jednostavniji no s obzirom da se ovdje ne radi o prevelikom i složenom projektu takav dizajn bi potrošio dosta vremena, a ne bi se mogao iskoristiti na najbolji mogući način.

## Opis klasa

U nastavku su opisane klase, njihova uloga i zadaća, od kojih se varijabli i funkcija sastoje te koje su sve biblioteke potrebne kako bi bio omogućen pravilan rad ostalih klasa te cijelog projekta.

### AUserWindow klasa

AUserWindow klasa definira neke osnovne zajedničke funkcionalnosti koje će biti korištene unutar EditorWindow i ClientWindow klasa. Nikada neće postojati instanca ove klase već će se ukoliko koristimo editor kreirati instanca EditorWindow klase, a ukoliko koristimo aplikaciju ya pronalayak sobe instanca ClientWindow klase.

**Varijable :**

* protected E\_USER userID
* protected Floor \*activeFloor
* protected Room \*activeRoom
* protected QVector<Floor> floors

**Funkcije :**

* public explicit AUserWindow();
* public virtual ~AUserWindow();
* public virtual void CloseWindow();
* protected void paintEvent(QPaintEvent \*e)
* protected void SetActiveFloor(Floor& \_floor)
* protected void SetActiveRoom(Room& \_room)
* protected void static SwitchToWindow(E\_USER \_user, AUserWindow \* \_w)

**Biblioteke i header-i :**

* "User/floor.h"
* "User/room.h"
* "User/jsonconnection.h"
* <QWidget>
* <QMainWindow>
* <QDebug>
* <QPainter>
* <QPolygon>
* <QPen>
* <QtGui>

**Ostalo :**

* protected enum E\_USER{ EDITOR, CLIENT }

Kod AUserWindow klase definirani su eksplicitni konstruktor i virtualni destruktor. Destruktor mora biti virtualan jer postoji mogućnost da ukoliko je potrebno ručno osloboditi memoriju to će biti različito za EditorWindow i ClientWindow klase jer one imaju samo neke zajedničke varijable kao što su activeFloor i activeRoom. Pokazivači activeFloor i activeRoom neče nikada direktno stvarati nove objekte i alocirati memoriju već oni samo služe kao neka vrsta reference na trenutno aktivni kat ili sobu tj. activeFloor će pokazivati na jedan od katova unutar naše liste/vectora katova – floors dok će activeRoom pokazivati na jednu od prostorija koja se nalazi na jednom od katova. Pokazivači za aktivni kat i aktivnu prostoriju će se koristiti različito unutar EditorWindow i ClientWindow klase tj. iako ih obje klase koriste za njih to nema jednako značenje. EditorWindow klasa će koristiti trenutno aktivnu sobu ako ju je potrebno preurediti(obrisati ili izmjeniti točke od kojih se sastoji) dok se kod ClientWindow klase taj pokazivač koristi samo kako bi se obojala unutrašnjost polygona te sobe. Isto tako će EditorWindow klasa koristiti pokazivač na trenutno aktivan kat kako bi se izvršile promjene za taj kat(brisanje, itd.) dok se kod ClientWindow klase pokazivač koristi kako bi se znalo koji je točno kat potrebno prikazati na prozoru.

AUserWindow će imati E\_USER enum koji će biti identifikator da li je trenutno aktivni prozor EditorWindow ili ClientWindow te će na temelju toga ukoliko želimo prijeći iz jednog moda u drugi za vrijeme razvoja znati koji točno prozor treba sljedeće kreirati tako da npr. ako smo trenutno u editoru i želimo se prebaciti na korisnički prozor samo trebamo pozvati statičku funkciju SwitchToWindow kojoj predajemo neki E\_USER vrijednost. Razlog zbog kojeg čuvamo tu informaciju o ID-u prozora je jer ako se u nekom trenutku želimo prebaciti u drugi mod rada(editor ili korisnik) no zbog nekog razloga pozovemo funkciju s ID-em već aktivnog prozora, dodatne provjere koje se događaju u toj funkciji će nam to onemogućiti jer nismo samo otvorili novi prozor već i spremili informaciju o tome koji je trenutno prozor/mod rada aktivan. Parametar AUserWindow koji se predaje funkciji SwitchToWindow je potreban jer je prije otvaranja novog prozora potrebno zatvoriti prozor koji je trenutno aktivan, a taj parametar zapravo predstavlja referencu na aktivan prozor.

Funkcijom SetActiveFloor se mijenja Floor objekt na koji activeFloor pokazuje te se vrše još neke dodatne provjere tako da se aktivni kat nikada ne mijenja direktno već samo preko funkcije SetActiveFloor. Ista stvar vrijedi i za funkciju SetActiveRoom te pokazivač activeRoom.

Funkcija paintEvent je virtualna funkcija koju će AUserWindow klasa nasljediti od QWidgeta. Unutar te funkcije se definira kako će izgledati površina nekog QWidget-a tj. u ovom projektu se u toj funkciji crtaju linije(borderi), popunjavaju poligoni sastavljeni od određenih točaka, definira se kakva će biti vrijednost anti-aliasing-a kako bi izbjegli pikselizaciju i preoštre rubove kod linija koje su nagnute pod nekim kutem te nisu vertikalne ili horizontalne, odabiru se boje za popunjavanje i crtanje linija te se crtaju neke .png slike na poziciju svake točke neke prostorije. Ono što će biti renderrano na Widget-u se ne definira direktno u paintEvent funkciji već se preko pokazivača trenutno aktivnog kata poziva funkcija RenderFloor unutar koje se vrši sva grafička aktivnost. Korake koje je potrebno ostvariti da bi se omoguílo crtanje po nekom Widget-u ćemo vidjeti kasnije u opisu Floor klase.

### EditorWindow klasa

EditorWindow klasa definira sve funkcionalnosti koje će biti korištene unutar editora te će služiti za stvaranje, imenovanje, brisanje i izmjenu katova, prostorija i kordinata soba.

**Varijable :**

* public bool arePointsSticky
* private QPoint \*activePoint
* private bool canDeletePoint
* private canDeleteRoom
* private Ui::EditorWindow \*ui

**Funkcije :**

* public explicit EditorWindow()
* public virtual ~EditorWindow ()
* public virtual void CloseWindow()
* **private slots:**

- void on\_roomAdd\_clicked()

- void on\_pointAdd\_clicked()

- void on\_floorAdd\_clicked()

- void on\_roomDelete\_clicked()

- void on\_pointDelete\_clicked()

- void on\_floorDelete\_clicked()

- void on\_dropbox\_floors\_activated(const QString &\_name)

**-** on\_SaveAll\_Clicked()

**-** on\_switchWindow\_clicked()

- private void mousePressEvent(QMouseEvent \*e) override

- private void mousePressEvent(QMouseEvent \*e) override

- private void mousePressEvent(QMouseEvent \*e) override

- private virtual void OnFloorDropdown()

- public void RefreshDropdownContent()

- public virtual void OnUserSwitchButton()

**Biblioteke i header-i :**

* <QWidget>
* <QDebug>
* <QMouseEvent>
* <QHBoxLayout>
* <QLineEdit>
* <QInputDialog
* "QPoint
* "auserwindow.h"
* "User/global.h
* "User/jsonconnection.h"

Kod EditorWindow klase definirani su eksplicitni konstruktor i normalni destruktor. CloseWindow funkcija će se pozivati unutar destruktora ako je to potrebno te će imati ulogu osloboditi memoriju za resurse koji nam više ne trebaju prije nego li prijeđemo na drugi prozor. Bool varijabla arePointsSticky će biti stanje editora da kada mijenjamo kordinate točaka neke sobe da te kordinate poprime vrijednosti okolnih kordinata ukoliko se nalaze u točno određenom radijusu. Na taj način nije potrebno pomaknuti neku točku prostorije na točan piksel već će se točke lijepiti jedna za drugu i time činiti posao lakšim. To stanje se naravno može i isključiti ukoliko postoji potreba za preciznijim dizajnom neke prostorije. Funkcije koje su označene kao slots su zapravo one funkcije koje se okidaju kada nakon interakcije sa grafičkim sučeljem tj. prilikom interakcije između miša/tipkovnice i nekog widget-a koji regira na takve evente. U našem slučaju sve funkcije osim on\_dropBox\_floors\_activated se okidaju kada pritisnemo lijevi klik miša na neki od widget-a, a svi widgeti su u našem slučaju tipa QPushButton. Funkcija on\_dropBox\_floors\_activated se okida kada aktiviramo padajući izbornik te se unutar nje poziva funkcija RefreshDropdownContent koja će osvježiti sadržaj tog izbornika sa najnovijim informacijama tj. sa trenutnim informacijama o katovima, sobama i kordinatama. Funkcija on\_saveAll\_clicked će sve trenutne katove sa svim sobama i svim QPoint točkama parsirati te spremiti u JSON file. Funkcija on\_switchWindow\_clicked se okida kada pritisnemo na gumb(QPushButton) u gornjem desnom kutu te ona samo poziva funkciju SwitchToWindow kojoj je AUserWindow::E\_USER argument tipa E\_USER::CLIENT jer iz editor prozora možemo prijeći samo na korisnički prozor.

Pokazivač ui je tipa Ui::EditorWindow i to je referenca na aktivnu formu koja je u ovom slučaju editorwindow.ui te nam omogućuje pristup svim widget-ima koji su defenirani unutar te forme.

Funkcije mousePressEvent, mouseReleaseEvent i mouseMoveEvent su event funkcije koje se okidaju na određenu akciju mišem. Te virtualne funkcije EditorWindow nasljeđuje od QWidget klase te je potrebno definirati što se događa njihovim okidanjem. Te tri funkcije će mijenjati vrijednosti atributa activePoint, movableRoom, canDeletePoint i canDeleteRoom. Kada se pomakne miš to se detektira na našem prozoru te se poziva funkcija mouseMoveEvent unutar koje se hvataju trenutne kordinate te ako movableRoom nema vrijednost nullptr ta soba na koju pokazivač pokazuje se pomiče zajedno s pomicanjem miša no samo ako je u isto vrijeme okinut i mousePressEvent. Taj mousePressEvent će ukoliko se miš nalazi nad prostorijom promjeniti vrijednost pokazivača movableRoom na tu prostoriju, a ako se miš nalazi nad nekom točkom bilo koje prostorije ona će pokazivač activePoint pokazivati na objekt te točke/kordinate nad kojom se miš nalazi. Atributi canDeletePoint i canDeleteRoom će se mijenjati ovisno o tome da li je neku prostoriju/točku moguće izbrisati te će biti omogućeno ili onemogućeno brisati pojedine elemente sheme.

### ClientWindow klasa

EditorWindow klasa definira sve funkcionalnosti koje će biti korištene unutar klijentske aplikacije te će služiti za pronalazak određene sobe na određenom katu koristeći padajuće izbornike.

**Varijable :**

* private Ui::ClientWindow \*ui

**Funkcije :**

* public explicit ClientWindow()
* public virtual ~ ClientWindow ()
* public virtual void CloseWindow()
* **private slots:**

- void on\_switchWindow\_clicked()

- on\_dropbox\_floors\_activated(const QString& \_name)

- on\_dropbox\_rooms\_activated(const QString& \_name)

- private void RefreshDropdownContent()

- private void RefreshRoomsDropdownContent()

**Biblioteke i header-i :**

* <QWidget>
* "auserwindow.h"

Kod ClientWindow klase definirani su eksplicitni konstruktor i normalni destruktor. CloseWindow funkcija će se pozivati unutar destruktora ako je to potrebno te će imati ulogu osloboditi memoriju za resurse koji nam više ne trebaju prije nego li prijeđemo na drugi prozor. Slot funkcije će se okidati pri interakciji sa grafičkim sučeljem, a u slučaju ovog projekta okidanje se događa pritiskom lijevog klika miša. Funkcija on\_switchWindow\_clicked se poziva kada je potrebno iz klijentske aplikacije prijeći u editor te se unutar nje poziva AUserWindow::SwitchToWindow(E\_USER::EDITOR, this) gdje je prvi argument tip prozora/moda na koji želimo prijeći dok za drugi argument samo prosljeđujemo pokazivač na neki AUserWindow objekt koji je u našem slučaju pokazivač na samu klasu tipa ClientWindow unutar koje se nalazimo. Funkcija on\_dropBox\_floors\_activated se okida kada odaberemo neku opciju iz padajućeg izbornika nakon čega se kao argument funkcije prosljeđuje ime te izabrane opcije/item-a te se taj string može koristiti unutar funkcije. U funkciji on\_dropBox\_floors\_activated prvo iteriramo kroz sve katove iz floors vektora te provjeravamo da li je naziv opcije koju smo odabrali iz padajućeg izbornika jednaka imenu kata na trenutnoj iteraciji te ako su imena ista poziva se funkcija SetActiveFloor s kojom postavljamo taj kat za aktivni kat nakon čega pozivamo funkciju RefreshRoomsDropdownContent koja će osvježiti ponuđene opcije unutar dropdown izbornika za odabir soba. Funkcija on\_dropBox\_rooms\_activated ima istu ulogu kao i on\_dropBox\_floors\_activated samo nije vezana za katove već sobe na trenutnom katu tj. nakon što odaberemo određeni kat on će se prikazati na prozoru te pomoći padajućeg izbornika za prostorije možemo odabrati željenu prostoriju. Funkciji će biti prosljeđena opcija koju smo odabrali nakon čega iteriramo kroz sve prostorije te ukoliko je ime prostorije isto kao i ime opcije koju smo odabrali nad tom prostorijom se poziva funkcija SetActiveRoom(Room &\_room) tj. kao argument \_room joj prosljeđujemo prostoriju za koju želimo da bude aktivna. Kod pozivanja SetActiveRoom funkcije prije postavljanja nove aktivne prostorije ukoliko activeRoom pokazivač pokazuje na neku vrijednost biti će potrebno promjeniti stanje aktivnosti te aktivne sobe na false tj. activeRoom->isActive poprima vrijednost false. Varijabla isActive nam služi kako bi znali da li soba treba biti ispunjena bojom ili ne, a razlog tome je jer se funkcionalnost crtanja/render-anja nalazi u klasi Room te svaka prostorija mora imati neku internu varijablu kako bi ona sama znala kako sebe renderati. Ukoliko bi tu funkcionalnost prebacili u klase ClientWindow i EditorWindow nebi imali potrebe čuvati tu dodatnu vrijednost no onda bi sama struktura podataka bila puno kompliciranija. Funkcija RefreshFloorDropdownContent će se pozivati samo pri pokretanju ClientWindow prozora jer korisnik neće moći mijenjati shemu zgrade te neće imati potrebe za ponovnim osvježavanjem katova jer će oni biti pročitani iz JSON-a i neće se mijenjati. Za razliku od toga funkciju za osvježavanje soba RefreshRoomsDropdownContent ćemo zvati skavi puta kada se primijeni trenutno aktivni kat jer korisnik ima mogućnost mijenjanja prikazanog kata, a svaki od njih će imati drugačije prostorije.

### Floor klasa

U Floor klasi se nalaze svi podaci vezani za katove te obuhvaća funkcije vezane za prikaz kata na ekranu i funkcija za manipulaciju podataka o prostorijama.

**Varijable :**

* public QVector<Room> rooms
* public QString nameID

**Funkcije :**

* public Floor(QString \_name)
* public Floor()
* public ~Floor()
* public RenderFloor(QWidget \*\_window)
* public void AddRoom(Room &\_room)
* public void DeleteRoom(Room &\_room)
* public void SetAllTextBoxVisibility(bool \_visibility)

**Biblioteke i header-i :**

* "User/room.h"

Klasa Floor sadržava 2 konstruktora od kojih jedan prima za argument ime kata te je tipa QString dok je drugi default-ni konstruktor koji mora postojati ukoliko se ime kata mijenja nakon njegove inicijalizacije. Funkcija RenderFloor će proći kroz sve prostorije unutar vektora rooms te za svaki pozvati funkciju RenderRoom. Ta funkcija se nalazi unutar svake sobe te joj se za argument predaje objekt tipa QWidget\* tj. pokazivač na prozor/widget na kojem je potrebno nešto nacrtati. AddRoom i DeleteRoom funkcije služe za dodavanje i brisanje prostorija unutar liste prostorija koja postoji za svaki kat te klasa Floor djeluje poput menađera za prostorije od kojih se taj kat sastoji. Obje funkcije primaju za argument referencu na određenu sobu te ju u slučaju AddRoom dodaju u listu prostorija dok se u slučaju funkcije DeleteRoom ta referencirana soba traži i briše iz liste ukoliko se pronađe. S obzirom da će postojati potreba da se sakrije svaki textbox za svaku sobu unutar klase Floor imamo i funkciju SetAllTextBoxVisiblity(bool \_visibility) koja za argument prima neki bool nakon čega iteriramo kroz sve sobe te za svaku od njih pozivamo funkciju SetTextboxVisiblity(bool \_v) koja pripada svakoj od soba. Nakon što se proslijedi argument na temelju kojeg će funkcija prikazati ili sakriti textbox onda moramo provjeriti da li je pokazivač na textbox uopće dostupante ako je potrebno ga je sakriti tj. ne prikazivati na aktivnom prozoru. Vektor rooms koji je tipa QVector<Room> služi kako bi se u njega pohranili svi podaci o prostorijama na pripadnom katu, a funkcije AddRoom, DeleteRoom i SetAllTextBoxVisibility vrše manipulaciju nad članovima tog vektora tj. u tim funkcijama dolazi do promjene nad podacima te liste prostorija ili dodavanjem nove prostorije ili iteracijom i promjenom stanja sobe nad kojom trenutno vršimo iteraciju. Varijabla nameID predstavlja naziv kata te ujedno služi kao i identifikator za svaki kat.

### Room klasa

Room klasa sadrži funkcije za crtanje prostorije na prozoru koristeći točke od kojih se ta soba sastoji te se unutar nje defniraju grafike/slike za srednju i rubne točke prostorije.

**Varijable :**

* public QWidget \*window
* QLineEdit\* editBox
* QVector<QPoint> points
* QPoint centerPoint
* bool isActive
* static float pointRadius
* private int penWidth

**Strukture :**

* **IMGInfo**
  + static float imgSize
  + static QImage edgePointIMG
  + QImage centerPointIMG
* **TextboxInfo**
  + static int w
  + static int h
  + static int xDist
  + static int yDist

**Funkcije :**

* public Room(QWidget\* \_window)
* public Room()
* public ~Room()
* public void SetName(QString \_name)
* public void UpdateTextboxPos()
* public void SetActive(bool \_state)
* public void SetTextboxVisibility(bool \_state)
* public void AddPoint(QPoint &\_point)
* public void DeletePoint(QPoint &\_point)
* public void MoveRoomTo(QPoint &\_p)
* public void RenderRoom(QWidget \*\_window)
* public void UpdateMiddlePointPos()
* void RenderPoint()

**Biblioteke i header-i :**

* #include <string>
* #include <QDebug>
* #include <QPoint>
* #include <QImage>
* #include <QPainter>
* #include <QPolygon>
* #include <QPen>
* #include <QRect>
* #include <QPainterPath>
* #include <QLineEdit>
* #include <QLayout>
* #include <cmath>
* #include "auserwindow.h"
* #include "global.h"

Klasa Room sadržava 2 konstruktora od kojih jedan prima za argument pokazivač na prozor tipa QWidget dok je drugi default-ni konstruktor koji mora postojati ukoliko ne prosljeđujemo objekt u trenutku pozivanja konstruktora te ga želimo inicijalizirati kasnije. Pokazivač window tipa QWidget zapravo pokazuje na objekt koji smo proslijedili putem konstruktora te je to zapravo referenca na prozor na kojeg želimo crtati sobu i može se reinicijalizirati u bilo kojem trenutku no s obzirom da će u nekom trenutku postojati samo jedan aktivni prozor neće postojati potreba za promjenom pokazivača na taj aktivni prozor. U trenutku nastanka objekta prostorije, window će pokazivati na aktivni prozor te se njegovo stanje neće mijenjati sve dok ta prostorija postoji tj. dok ne prekinemo program, prijeđemo na drugi prozor ili izbrišemo prostoriju koja sadrži taj pokazivač. Pokazivač editBox tipa QLineEdit će pokazivati na grafički objekt kojeg ima svaka soba i koji služi kako bi se u njega ako smo u editor mod-u upisalo ime aktivne prostorije, a ako smo u klijentskom mod-u pročitalo ime tražene/aktivne prostorije. Taj objekt inicijalizira u konstruktoru i postoji sve dok postoji i prostorija kojoj pripada tj. instanci klase Room koja ga sadržava. Funkcije SetTextboxVisibility(bool \_state) i SetTextboxEditability(bool \_state) su funkcije koje operiraju nad pokazivačem editBox te obje za argument primaju vairjablu tipa bool te mijenjaju stanje editBox objekta ovisno o toj vrijednosti. SetTextboxVisibility funckija će prvo provjeriti da li editBox pokazuje na nullptr vrijednost te ako ne pokazuje postavlja vidljivost editBox objekta na bool stanje koje smo poslali kao argument, a uz to se i izvršava funkcija editBox->SetEnabled(bool \_state) koja će ga aktivirati/deaktivirati. Unutar liste točaka points tipa QVector<QPoint> se nalaze sve točke od kojih se ta soba sastoji, a od svih svojstava klase QPoint nama su jedino važne varijable x i y koje će nam ujedno i služiti kao i kordinate na kojima ćemo prikazivati neki grafički element na prozoru, a u našem slučaju su to grafike tipa QImage. Funkcija AddPoint(QPoint& \_point) će služiti da točku primljenu kao argument dodamo points vektoru dok će DeletePoint(QPoint& \_p) iterirati kroz points vektor dok ne nađe točku na koju se argument referencira te će ona biti izbrisana/maknuta iz liste točaka. Kada brišemo neku točku iz liste ne tražimo točku sa istim kordinatama kao i predani argument već se iteracijom kroz listu traži baš ta točka na koju se argument tipa QPoint referencira i stoga nije moguće brisanje više točaka unutar jednog poziva funkcije DeletePoint(). Funkcija MoveRoomTo(QPoint &\_p) za argument prima neku točku na koju želimo pomaknuti prostoriju tj. pomaknuti sve točke od kojih se ta prostorija sastoji. Ono što se zapravo događa je da sve točke pomičemo za kordinate centralne točke oduzete za kordinatu nove točke na koju želimo pomaknuti cijelu prostoriju, a unutar funkcije bi formula za to bila moveBy = centerPoint - \_p. Vrijednost kordinata moveBy će biti vrijednosti koje moramo dodati svakoj točki od koje se prostorija sastoji kako bi ona bila pomaknuta na novu poziciju na ekranu tj. dobila nove kordinate. Unutar funkcije Funkcija MoveRoomTo pozivamo i funkciju UpdateTextboxPos kako bi izračunali gdje će biti novo mjesto na kojem će se nalaziti editBox element, a to će uvjek biti negdje oko kordinate na kojima se nalazi centerPoint. Centralna točka će se uvjek nalaziti na sjecištu simetrala kutova pravokutnika čije se 4 točke koje ga definiraju sastoje od najdaljnjih točaka po horizontalnoj i vertikalnoj osi.

[Ubaci sliku] sadasd sadasd

Ukoliko je potrebno ponovno izračunati koordinate centralne točke to se postiže pozivanjem funkcije UpdateMiddlePointPos(), a ona se poziva samo ukoliko su se promijenile koordinate točaka od kojih se prostorija sastoji te je potrebno ponovno osvježiti koordinate središnje točke. Funkcije GetName() i SetName(QString \_name) služe za dohvaćanje i postavljanje imena prostorije te se to ime ne odnosi na neki QString koji predstavlja ID prostorije kao kod katova već se odnosi na string koji se nalazi unutar editBox u trenutku pozivanja funkcija. Funkcija GetName() će zapravo samo vratiti editBox->text() dok će SetName(QString \_name) pozvati funkciju editBox->setText(\_name) te postaviti tekst unutar editBox grafičkog elementa na vrijednost koju smo predali kao argument. Kako bi se zaista prikazala ta prostorija na ekranu potrebno je pozvati funkciju RenderRoom(QWidget \*\_window) svaki puta pri ponovnom osvježavanju aktivnog prozora tj. to je zapravo funkcija koja vrši crtanje sobe na prozoru te se poziva unutar AUserWindow::paintEvent no ne poziva se direktno već preko drugih funkcija. Varijabla penWidth tipa int definira širinu linija kojom će točke prostorije biti povezane, a koristi si prilikom crtanja tj. unutar funkcije RenderRoom.

[slika funkcije RenderRoom]

### Global klasa

Unutar klase Global nalaze se statičke funkcije te nam nikada neće trebati instanca te klase već se sve funkcije pozivaju nad samom klasom.

**Funkcije :**

* static bool IsPosInsidePointRadius(QPoint \_p1, QPoint \_p2, float \_rad)
* static QPoint GetMiddlePoint(QPoint& \_p1, QPoint& \_p2)
* static QPoint GetShapeCenter(QVector<QPoint> &\_vec)

Funkcija IsPosInsidePointRadius(QPoint \_p1, QPoint \_p2, float \_rad) za prva 2 parametra prihvaća 2 objekta tipa QPoint koji zapravo predstavljaju točke za koje je potrebno provjeriti da li je njihova udaljenost manja od trećeg parametra koji predstavlja radius. Ukoliko se prva točka nalazi unutar radiusa druge točke tj. ako se druga točka nalazi unutar radiusa prve točke tada će funkcija vratiti vrijednost true, a u bilo kojem drugom slučaju povratna vrijednost će biti false. Tu funkciju koristimo kod editora kada provjeravamo da li grafika neke točke dodiruje grafiku neke druge točke te će naš treći parametar \_rad će biti iste vrijednosti kao i polovica širine slike koju koristimo za crtanje točaka na prozoru dok će prva 2 parametra imati iste kordinate kao i dvije točke koje sudjeluju u provjeri. Kako bi pronašli točku koja se nalazi između neke 2 točke koristimo funkciju GetMiddlePoint(QPoint& \_p1, QPoint& \_p2) kojoj za parametre predajemo neke dvije točke, a za povratni tip dobivamo srednju točku. Ovdje je zapravo samo potrebno izračunati srednje koordinate x i y po formuli *midAxis = (p1.axis + p2.axis) / 2* gdje se pod *axis* misli na x ili y kordinatu. Funkcija GetShapeCenter(QVector<QPoint> &\_vec) služi kako bi izračunali srednju točku neke prostorije te se za parametar predaje lista točaka od kojih se ta prostorija sastoji. Ono što se zapravo računa su 4 krajnje točke prostorije tj. potrebno je znati koordinate najgornje, najdonje, najlijevije i najdesnije točke. Nakon toga se računa sredina najgornje i najdonje točke za y koordinatu te sredina najlijevije i najdesnije točke za x koordinatu, a dobivena kombinacija x i y koordinata predstavlja poziciju sredine prostorije.

### JSONConnection

JSONConnection klasa nam omogućuje zapisivanje i čitanje JSON file-ova.

**Funkcije :**

* public JSONConnection()
* public void ReadAll(QVector<Floor>& \_floors, QWidget \*\_window)
* public void WriteAll(QVector<Floor>& \_floors)

**Varijable :**

* public static QString path

Prilikom pozivanja konstruktora JSONConnection() u konzolu se ispisuje i inicijalizira vrijednost path varijable tako da ukoliko dođe do nekih problema kod pronalaska ili čitanja file-a zna se točno koji path se koristi. Varijabla path će se uvijek inicijalizirati na povratnu vrijednost funkcije QDir::currentPath() na koju će još biti dodan string "/json.json" gdje upisujemo željeno ime JSON file-a. QDir::currentPath() će uvjek vratiti absolutni put od trenutnog direktorija u kojem se nalazi pokrenuta aplikacija[2]. Funkcija ReadAll(QVector<Floor> &\_floors, QWidget \*\_window) služi kako bi se iz JSON-a pročitali svi podaci vezani za katove, prostorije, točke prostorije i imena. Za prvi parametar se prosljeđuje lista katova u koje želimo spremiti sve pročitane podatke iz JSON-a te je pri ulazu u funkciju ta lista(vektor) prazna. Za drugi parametar predaje se pokazivač na aktivni prozor \_window jer se unutar funkcije ReadAll moraju kreirati prostorije tako da se pri kreiranju sobe poziva Room konstruktora te se odmah predaje aktivan prozor kako se poslije ne bi moralo iterirati kroz svaku prostoriju i dodavati referencu na aktivan prozor. Qt podržava klasu QFileInfo za čitanje datoteka u string formatu i klase QJsonDocument, QJsonObject i QJsonArray za mapiranje JSON string-a kako bi samo čitanje podataka bilo lakše. Funkcija WriteAll(QVector<Floor> &\_floors) služi kako bi sve podatke od kojih se katovi sastoje zapisali u JSON formatu unutar tekstualne datoteke. Za parametar predajemo listu svih katova iz kojih se izvlače informacije te se konstruiraju po JSON pravilima i zapisuju u string varijablu,a nakon toga taj string zapisuje u tekstualnu datoteku.

# Korištenje programa

-> korištenje editora, korištenje client-a, objasniti stanja(remove Point ON/OFF)

# Moguća poboljšanja

-> navesti probleme s realokacijom vektora, pointer to nothing, nemogučnost inputa u dekonstruktoru 'Room' klase, vrednovanje rješenja problema i kako bi moglo biti bolje, smjernice za daljnji razvoj(na mobitele)

# Zaključak

# Popis oznaka i tablica

# Literatura

1. zibib, What's the difference between QMainWindow, QWidget and QDialog? ,s Interneta, <https://stackoverflow.com/questions/3298792/whats-the-difference-between-qmainwindow-qwidget-and-qdialog>, 01.09.2017
2. Službena Qt dokumentacija, Qt Documentation (QDir QDir::current()), s Interneta, <http://doc.qt.io/qt-4.8/qdir.html>, 09.09.2017

# Prilozi