Konstrukcija aplikacije u biblioteci Qt5

Čupić, Damir

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:577542

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-24**

Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Damir Čupić

KONSTRUKCIJA APLIKACIJE U BIBLIOTECI QT5

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, 26. kolovoza 2021.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Mojoj obitelji

Sadržaj

Uvod **1**

1 Uvod u Qt **3**
1.1 Stvaranje novog projekta 3
1.2 Grafičko sučelje 4
1.3 Sustav meta objekata 6

2 Organizacija koda **9**
2.1 Podjela na komponente 9
2.2 Projektne datoteke 10
2.3 Poslužiteljska komponenta 11
2.4 Podatkovne klase 12

3 Baza podataka **15**
3.1 Upravitelj baze podataka 15
3.2 Klase za pristup podacima 17
3.3 Uređivanje pokazivača 21

4 Implementacija modela **23**
4.1 Model/View arhitektura 23
4.2 Model za album 24
4.3 Model za slike 28

5 Temelji korisničkog sučelja **31**
5.1 Organizacija sučelja 32

6 Izgradnja korisničkog sučelja **35**
6.1 Popis albuma 35
6.2 Stvaranje minijatura 38
6.3 Prikaz albuma 44

6.4 Dodaci na minijature 55
6.5 Prikaz slika 58

7 Povezivanje korisničkog sučelja **63**
7.1 Prikaz galerije 63
7.2 Glavni prozor 64
7.3 Završne napomene 66

8 Pakiranje projekta **67**
8.1 Pakiranje za Windows 67
8.2 Pakiranje za Linux 69
8.3 Pakiranje za Mac OS X 71

9 Zaključak **73**

Bibliografija **75**

Uvod

Jedan od većih izazova u svijetu izrade aplikacija jest njena distribucija. Glavni je cilj imati što veću mrežu korisnika, no uz veliki broj različitih platformi (poput Windows, Linux, Mac OS, Android, iOS i drugih) koje se koriste, to često nije jednostavno. Stoga je bitno stvoriti dobru kompatibilnost aplikacije što je najlakše postići korištenjem prikladnih razvojnih okruženja, od kojih će u ovom radu biti istaknut Qt.

Qt je razvojno okruženje za izradu GUI i višeplatformskih C++ aplikacija koje je stvorila kompanija **The Qt Company** te ju i dalje unaprjeđuje u takozvanom **Qt Project** projektu zajedno sa drugim samostalnim programerima i kompanijama.

Od dosad postojećih šest glavnih verzija Qt-a, u ovom će se projektu koristiti **Qt5** čija je najveća novost nad prethodom verzijom nova i fleksibilnija sintaksa za komunikaciju Qt-ovih objekata dok u najnovijoj Qt6 verziji koja je izašla u prosincu 2020. godine nije bilo većih promjena na grafičkim komponentama koje će biti korištene za izradu GUI aplikacije. Međutim, Qt6 je također uveo novi pomoćni alat za izgradnju CMake čije mjesto u Qt5 zauzima alata qmake.

Cilj je ovog rada pokazati osnovne funkcionalnosti biblioteke Qt5 te detaljnije o izradi konkretne GUI višeplatformske aplikacije jednostavnog preglednika fotografija koja će se moći pokrenuti na Windows, Linux i Mac sustavima kako bi se pokazale široke mogućnosti Qt platforme. Glavni vodić u izradi rada je knjiga autora Lazar Guillaume i Robin Penea zvana *Mastering Qt 5* koja dodatno obrađuje i izradu aplikacija za mobilne uređaje.

Struktura rada može se podijeliti na četiri veće cjeline. Najprije se trebaju objasniti osnovni Qt koncepti što će činiti uvodno poglavlje, dok će ostatak rada biti posvećen samoj izradi aplikacije. Tijekom izrade će na prikladnim mjestima biti dodatno objašnjene naprednije Qt mogućnosti. Sljedeća je cjelina izgradnja arhitekture za obradu podataka koja uključuje stvaranje jednostavne varijante SQL baze podataka. Nakon toga slijedi izrada korisničkog sučelja odnosno GUI dijela aplikacije. Na koncu je potrebno stvoriti skripte za pakiranje za više platformi.

Poglavlje 1

Uvod u Qt

Za izradu Qt5 aplikacija koristi se IDE[1] **Qt Creator** koji olakšava izradu C++ aplikacija sa grafičkim sučeljem. U ovom je radu korištena verzija *Qt Creator 4.13.3 (Community)*, koja se može besplatno preuzeti na **Qt Company** stranicama [1].

1.1 Stvaranje novog projekta

Najprije je potrebno stvoriti novi projekt:

File → New File or Project → Application → Qt Widgets Application → Choose.

Prije nastanka projekta potrebno je odrediti postavke poput imena projekta i stvorenih klasa, lokacije i izbora *Kit*-a u kojem se definira platforma na kojoj će se izvoditi aplikacija.

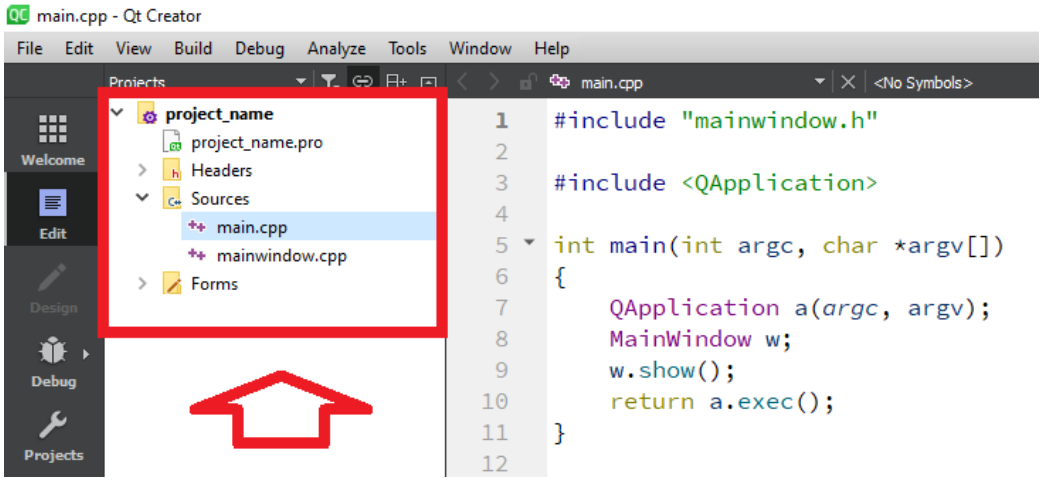
Na slici 1.2 vidi se početno stanje stvorene aplikacije. Ulaz u program je `main.cpp` datoteka. Ona će stvoriti glavni prozor aplikacije (koji je opisan u `main_window.cpp`, `main_window.h` i `main_window.ui` datetekama) te `QApplication` instancu koja će biti u petlji događaja. Posebno je bitna datoteka `project_name.pro`, u kojoj su zapisane postavke projekta te sve upute za izgradnju. Prilikom prve izgradnje te nakon svake izmjene `.pro` datoteke potrebno je izvršiti naredbu `qmake`. Rezultat izgradnje sprema se u novi folder sa prefiksom *build-* te varijabilnim sufiksom ovisno o *Kit* postavkama.

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
```

[1] Integrirano razvojno okruženje (eng. *integrated development environment*)

```
CONFIG += c++11

SOURCES += \
    main.cpp \
    mainwindow.cpp

HEADERS += \
    mainwindow.h

FORMS += \
    mainwindow.ui
```

Slika 1.1: `project_name.pro`



Slika 1.2: Početno stanje stvorenog projekta

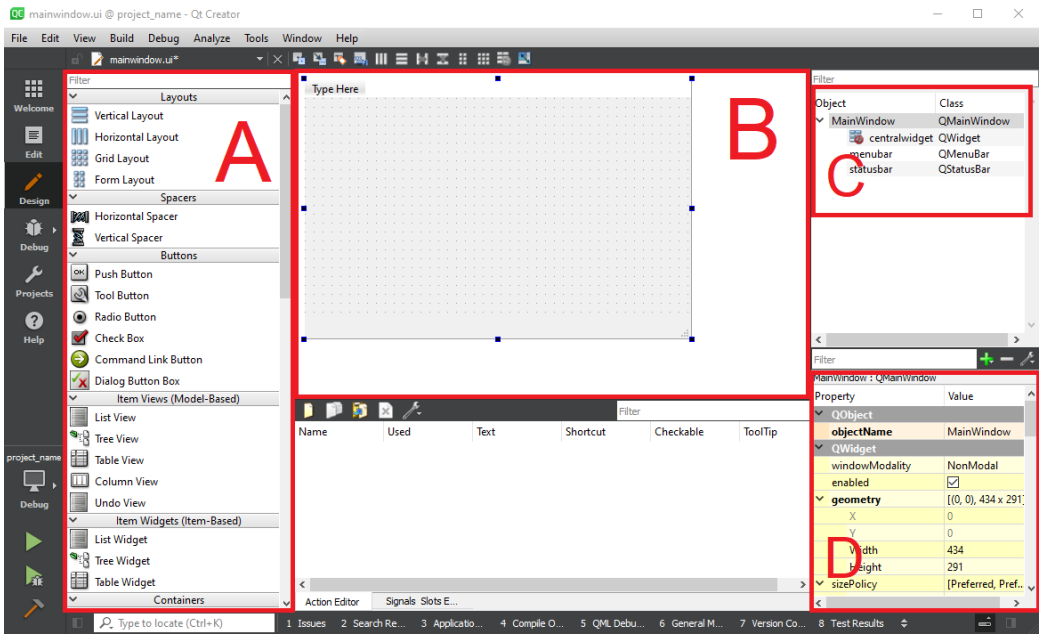
`MainWindow` klasa nasljeđuje `QMainWindow` te je deklarirana kao `Q_OBJECT` čime joj je omogućeno definiranje signala i utora. Ona stvara vezu sa `.ui` datotekom koji sadrži sve komponente grafičkog sučelja. Nadalje je pristup funkcijama i varijablama grafičkog sučelja moguć sa pozivom `ui->[ime varijable/funkcije]`.

1.2 Grafičko sučelje

Prikazivanje i modificiranje grafičkog sučelja koje definira `MainWindow.ui`[2] jednostavno omogućava **Qt Designer** alat. Izgled glavnog prozora nalazi se u sredini (Slika

[2] *.ui* je kratica za korisničko sučelje (*eng. user interface*).

1.3: B). Na njega se zatim *drag-and-drop* tehnikom mogu dodati razni grafički elementi iz izbornika (A). *Layout*-i pomažu sa preglednosti i urednosti prozora. Hijerarhija *Layout*-a te ostalih komponenata vidljiva je na desnoj strani (C). Ispod se nalaze detalji trenutno označenog grafičkog elementa (D).



Slika 1.3: Prikaz Qt Designera

Za precizno i uredno slaganje elemenata u grafičkom sučelju, preporučljivo je koristiti **rasporede** (*layout*-e). Postoje četiri glavne vrste rasporeda, a to su:

- vertikalni raspored — komponente složene jedna ispod druge
- horizontalni raspored — komponente složene jedna pored druge
- mrežni (*grid*) raspored — komponente složene u mrežu ćelija
- obrazac (*form*) — komponente složene kao parovi oznaka i unosa

Još jedna opcija za raspoređivanje su **odstojnici** (*spacer*-i) koji imaju ulogu pozicioniranja elemenata popunjavanjem praznog prostora. Postoji horizontalni i vertikalni odstojnik.

QtWidget

Qt Creator ima mogućnost stvaranja novih grafičkih komponenti, odnosno **QWidget-a**, na sljedeći način:

File → New File or Project → Qt → Qt Designer Form Class.

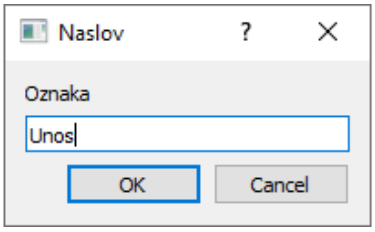
Nakon stvaranja, u projekt se dodaju `[ime QWidget-a].cpp`, `.h` te `.ui` datoteke, a `.pro` datoteka se također prikladno ažurira.

QInputDialog

Jedna od korisnih Qt5 klasa je `QInputDialog`, pomoću koje je na jednostavan način moguće stvoriti dijaloške okvire te provjeriti i iskoristiti pripadni odgovor korisnika. Primjer dijaloškog okvira za dohvat teksta `moj_string`:

```
                                bool ok;
  QString moj_string = QInputDialog::getText(this, tr("Naslov"),
                tr("Oznaka"), QLineEdit::Normal, tr("Unos"), &ok);
```

Izvedbom ove naredbe stvara se dijaloški okvir koji je prikazan na slici 1.4.



Slika 1.4: Prikaz dijaloškog okvira za unos string-a

1.3 Sustav meta objekata

Qt-ov *Meta-Object System* odnosno **sustav meta objekata** bitan je dio Qt sustava koji se koristi za podršku C++ dodataka od kojih je najbitnija komunikacija pomoću signala i utora. On je zasnovan na sljedeće tri točke:

- `QObject` klasa
- `Q_OBJECT` makro

- *Meta-Object Compiler* (moc)

Osnovni objekt u Qt-u je `QObject` i on se sa drugim `QObject`-ima slaže u stablo. Tijekom stvaranja moguće je predati argument `parent` koji će zatim preuzeti vlasništvo nad svojim *child* odnosno `QObject` djetetom. Vlasništvo drugim riječima znači da će se vlasnik nekog objekta pobrinuti za njegovo uništavanje u vlastitom destruktoru. Uništavanje aktivira signal `destroyed()` što je korisno za rješavanje referenci koje više nisu potrebne.

Glavna vrlina `QObject`-a je njegova mogućnost jednostavne komunikacije pomoću koncepta **signala** i **utora**.

Signal je poruka koju šalje objekt. U Qt-u postoje poruke poput `QPushButton::clicked()` koju šalje dugme nakon što je kliknuto. Funkcijom `connect`[3] povezujemo signal sa funkcijom koju zovemo utor. Ulogu utora može imati bilo koja funkcija (ako su argumenti kompatibilni), što ovaj mehanizam čini vrlo fleksibilnim. Funkcija utora bit će pozvana kao posljedica aktivacije signala, pa stoga u ovom kontekstu nema smisla povratna vrijednost funkcije (*return value*).

Funkcija `connect` ima četiri argumenta — pošiljatelj, signal pošiljatelja, primatelj, utor primatelja. Primjer povezivanja klika gumba `Button` i kreirane funkcije `MyFunction()`:

```
connect(ui->Button, &QPushButton::clicked, this, &MyFunction).
```

Zadnja dva argumenta mogu se zamijeniti lambda izrazom. U nastavku se nalaze dva primjera iz gotove aplikacije:

```
connect(ui->sortComboBox, QOverload<int>::of(&QComboBox::activated),
  [=](int sortIndex){
    sortPictures(sortIndex);
  })
```

```
connect(mAlbumModel, &QAbstractItemModel::dataChanged,
  [this] (const QModelIndex &topLeft) {
    if (topLeft == mAlbumSelectionModel->currentIndex()) {
      loadAlbum(topLeft);
    }
  })
```

[3] Postoji i funkcija koja poništava spajanje signala i utora koja se zove `disconnect` no ona neće biti korištena u ovom projektu.

Kada su pojedinoj klasi potrebne meta objekt mogućnosti, onda se koristi makro `Q_OBJECT` koji se postavlja u privatnom dijelu klase kao što se vidi na primjeru sa slike 1.5.

```
class MetaObjectClass
{
    Q_OBJECT

public:
  MetaObjectClass();
  ~MetaObjectClass();

public slots:
    void setValue(int value);

signals:
    void valueChanged(int newValue);

private:
  int mValue;
}
```

Slika 1.5: Klasa sa aktiviranim meta objekt mogućnostima

Qt-ov *Meta-Object Compiler* je program koji sastavlja klase koje sadrže `Q_OBJECT` makro, a izvršava se tijekom izgradnje projekta. Rezultat izgradnje su `.cpp` i `.h` datoteke sa prefiksom `moc_`.

Poglavlje 2

Organizacija koda

U ovom će odjeljku biti opisan plan izrade konkretne Qt5 aplikacije. Preciznije, riječ je o izradi `gallery` aplikacije, odnosno **preglednika fotografija za Windows, Linux i Mac OS X**, koji će imati sljedeće funkcionalnosti:

- Stvaranje, izmjena i brisanje albuma fotografija
- Prikaz fotografija u *grid* rasporedu
- Prikaz fotografije preko cijelog zaslona

Planiranje i implementacija ove aplikacije usko prate rad autora Lazar i Penea (2016).

2.1 Podjela na komponente

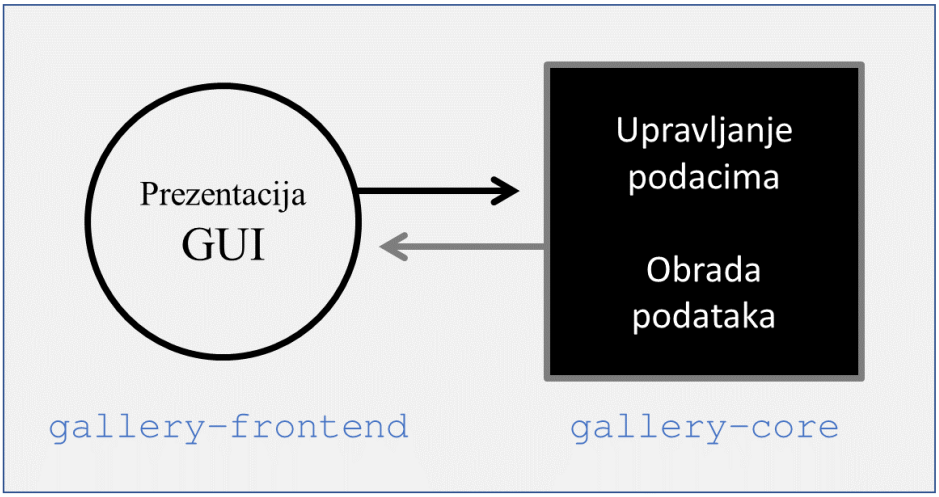
Cijelu je aplikaciju moguće podijeliti na dvije veće komponente — **logička** i **grafička**. Logička komponenta imati će ulogu **upravljanja** i **obrade** podataka, a grafička ulogu **prezentacije**. Zapravo se radi o tzv. *"mršavom" klijentu*, varijanti dvoslojne arhitekture klijent-poslužitelj [4, Poglavlje 2] koji je ilustriran na slici 2.1.

Za stvaranje projekta sa više komponenti u Qt Creator-u, potrebno je koristiti alternativnu proceduru. Najprije se mora stvoriti glavni okvir aplikacije:

$$\text{File} \rightarrow \text{New File or Project} \rightarrow \text{Other Project} \rightarrow \text{Subdirs Project},$$

koji se u ovom slučaju zove `gallery`. Zatim u izborniku koji Qt Creator dalje predlaže stvaramo novu biblioteku:

$$\text{Library} \rightarrow \text{C++ Library},$$



Slika 2.1: Model `gallery` komponenti

kako bi se kreirala poslužiteljska komponenta `gallery-core` koja je smještena u `gallery` okviru. Pripadnoj klasi biblioteke dano je ime `Album`. C++ biblioteke su komponente spremne za integraciju u veći projekt.

2.2 Projektne datoteke

Generirane `.pro` datoteke razlikuju se od one u jednostavnom Qt5 projektu. Prva je razlika što ovdje postoje dvije (za sada) `.pro` datoteke. U `gallery.pro` datoteci glavnog okvira samo je napisana informacija o poddirektoriju `gallery-core` te vrijednost `TEMPLATE = subdirs`, što znači da se radi o datoteci koja bilježi pod-direktorije.

`gallery-core.pro` datoteka, osim informacija opisanih u poglavlju 1, također sadrži detalje konteksta koji se vidi na slici 2.2. U prvoj je liniji koda dodan SQL modul koji je potreban za SQL bazu pomoću koje će biti realizirano upravljanje podataka. Zatim je određena vrsta `.pro` datoteke sa `TEMPLATE = lib`. U zadnjim linijama koda opisan je direktorij instalacije biblioteke na Linux-u, no taj dio trenutno nije potreban pa ga je stoga bolje izbrisati.

```
QT += sql
QT -= gui
```

```
TEMPLATE = lib
DEFINES += GALLERYCORE_LIBRARY

//...

# Default rules for deployment.
unix {
    target.path = /usr/lib
}
!isEmpty(target.path): INSTALLS += target
```

Slika 2.2: Dio koda iz gallery-core.pro

2.3 Poslužiteljska komponenta

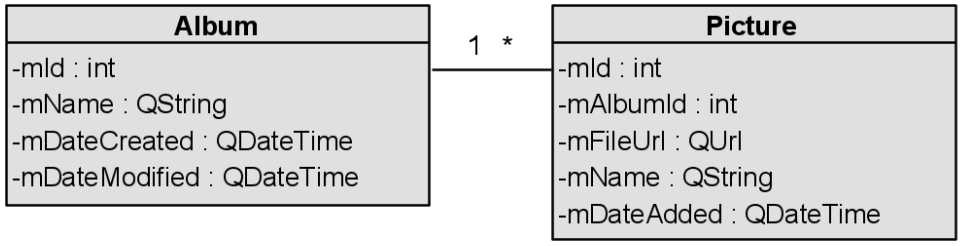
U gallery-core podprojektu nalazi se Album klasa te dodatno zaglavlje gallery-core_global.h koje je prikazano na slici 2.3.GALLERYCORE_LIBRARY je varijabla definirana u gallery-core.pro. Ako je biblioteka korištena kao dio projekta, onda će GALLERYCORE_EXPORT imati vrijednost Q_DECL_EXPORT, a inače (ako je pokrenuta samostalno) Q_DECL_IMPORT. Klasu je za dijeljenje kroz projekt dovoljno označiti sa GALLERYCORE_EXPORT što je u ovom slučaju multi-platform ekvivalent oznake public.

```
#ifndef GALLERYCORE_GLOBAL_H
#define GALLERYCORE_GLOBAL_H

#include <QtCore/qglobal.h>

#if defined(GALLERYCORE_LIBRARY)
#  define GALLERYCORE_EXPORT Q_DECL_EXPORT
#else
#  define GALLERYCORE_EXPORT Q_DECL_IMPORT
#endif

#endif // GALLERYCORE_GLOBAL_H
```

Slika 2.3: Dio koda iz gallery-core_global.h

2.4 Podatkovne klase

Dvije podatkovne klase `gallery` aplikacije su `Album` i `Picture`. Na slici 2.4 nalazi se prikaz njihovih atributa u obliku *class* dijagrama. Instanca `Album` klase predstavlja jedan album koji se sastoji od identifikatora za bazu `mId` te imena sa pripradnim `getter`-ima i `setter`-ima koji ovdje nisu prikazani no imaju klasičnu implementaciju. Konstruktoru sa slike 2.5 proslijeđuje se ime albuma te se identifikator postavlja na nekorištenu vrijednost koja je u našem slučaju -1. Svrha je albuma da sadrži slike, no u ovom su projektu one radi veće fleksibilnosti odvojene u zasebnu klasu `Picture` te se sa albumom povezuju u bazi podataka pomoću atributa `mAlbumId`. Na slici 2.6 prikazani su konstruktori klase `Picture`. Atributi `DateCreated`, `DateModified` klase `Album` te `DateAdded` klase `Picture` nije potrebno inicijalizirati jer će oni biti određeni tek nakon stvaranja pripadnih objekata u bazi podataka. Njihova je svrha prikaz vremena i datuma stvaranja odnosno izmjene albuma i slika.



Slika 2.4: *Class* dijagram za `Album` i `Picture` klase.

```
Album::Album(const QString& name) :
mId(-1),
mName(name)
{
}
```

Slika 2.5: Konstruktor klase `Album`

```
Picture::Picture(const QString& filePath) :
```

```
    Picture(QUrl::fromLocalFile(filePath))
    {
    }

    Picture::Picture(const QUrl& fileUrl) :
    mId(-1),
    mAlbumId(-1),
    mFileUrl(fileUrl),
    mName(fileUrl.fileName())
    {
    }
```

Slika 2.6: Konstruktori klase `Picture`

Poglavlje 3

Baza podataka

Sljedeći je korak implementacija baze podataka. Svojstva koja su potrebna za bazu ove aplikacije su:

- mogućnost jednostavnih upita
- jednostavne transakcije
- ima jednu svrhu.

Dakle potrebna je vrlo jednostavna SQL baza. U Qt-u postoji *driver* `SQLITE3` [1] koji upravo zadovoljava ove kriterije. On je također uključen u `sql` modul koji je prethodno dodan u `gallery-core.pro` datoteci.

3.1 Upravitelj baze podataka

U svrhu lakšeg pozivanja iz baze, podprojektu `gallery-core` dodaje se nova C++ klasa `DatabaseManager` koja će obrađivati sve potrebne zahtjeve. Na slici 3.1 prikazano je cijelo sučelje `DatabaseManager` klase. Funkcijom `instance` pretvaramo `DatabaseManager` u *Singleton* što znači da će uvijek postojati jedinstvena instanca njene klase. Klasa `QSqlDatabase` unaprijed je definirana kako bi se izbjeglo uključivanje potrebnih biblioteka u daljnje datoteke koje će uključiti `DatabaseManager`. Time je očuvana neovisnost o implementaciji iz vanjske perspektive. Funkcija `debugQuery` je korisna za samog programera jer daje bolji uvid u događaje u bazi (slika 3.2). U `DatabaseManager.cpp` sa slike 3.3 koriste se funkcije iz Qt-ove klase `QSqlDatabase`. Argument funkcije `addDatabase(...)`, `"QSQLITE"`, određuje traženi SQL *driver*.

```
const QString DATABASE_FILENAME = "gallery.db";
```

```
class DatabaseManager
{
public:

      static void debugQuery(const QSqlQuery& query);

  static DatabaseManager& instance();
  ~DatabaseManager();

protected:
  DatabaseManager(const QString& path = DATABASE_FILENAME);
  DatabaseManager& operator=(const DatabaseManager& rhs);

private:
  QSqlDatabase* mDatabase;
};
```

Slika 3.1: `DatabaseManager.h`

```
void DatabaseManager::debugQuery(const QSqlQuery& query)
{
    if (query.lastError().type() == QSqlError::ErrorType::NoError) {
        qDebug() << "Query OK:"  << query.lastQuery();
    } else {
       qWarning() << "Query KO:" << query.lastError().text();
       qWarning() << "Query text:" << query.lastQuery();
    }
}
```

Slika 3.2: Funkcija `debugQuery`

```
DatabaseManager& DatabaseManager::instance()
{
  static DatabaseManager singleton;
  return singleton;
}

DatabaseManager::DatabaseManager(const QString& path) :
  mDatabase(new QSqlDatabase(QSqlDatabase::addDatabase("QSQLITE"))
```

```
)
{
  mDatabase->setDatabaseName(path);
  mDatabase->open();
}

DatabaseManager::~DatabaseManager()
{
  mDatabase->close();
  delete mDatabase;
}
```

Slika 3.3: `DatabaseManager.cpp`

3.2 Klase za pristup podacima

Pomoću `mDatabase` moguće je izvršavati SQL upite, no kako bi oni bili bolje organizirani, potrebno ih je grupirati u više klasa koje se nazivaju *data access objects* (DAO). U tu se svrhu stvaraju dvije nove klase — `AlbumDao` i `PictureDao`. Njihova se zaglavlja vide na slikama 3.4 i 3.5. U njima se nalazi konstruktor koji prima bazu, funkcija `init()` koja inicijalizira svoju tablicu u bazu (primjer na slici 3.6) te po jedna funkcija za svaki potrebni upit prema bazi.

Klasa `AlbumDao` ima upite za dodavanje, preimenovanje, brisanje te pregled svih albuma, dok klasa `PictureDao` ima upite za dodavanje, brisanje, brisanje iz albuma te dohvat svih slika iz pojedinog albuma. Ove su funkcije međusobno vrlo slične i izvršavaju se preko objekta Qt5 klase `QSqlQuery`, a jedna takva izdvojena je na slici 3.7.

```
class AlbumDao
{
public:
  explicit AlbumDao(QSqlDatabase& database);
  void init() const;

  void addAlbum(Album& album) const;
  void updateAlbum(const Album& album) const;
  void removeAlbum(int id) const;
  QVector<Album*> albums() const;
     bool albumExists(QString name) const;

private:
  QSqlDatabase& mDatabase;
```

```
};
```

Slika 3.4: AlbumDao.h

```
class PictureDao
{
public:
  explicit PictureDao(QSqlDatabase& database);
  void init() const;

  void addPictureInAlbum(int albumId, Picture& picture) const;
  void removePicture(int id) const;
  void removePicturesForAlbum(int albumId) const;
  QVector<Picture*> picturesForAlbum(int albumId, int sortIndex) const
      ;
  QVector<Picture*> copyPicturesFromAlbum(int albumId, int albumIdNew)
       const;
  bool pictureExists(int album_id, QUrl url) const;

private:
  QSqlDatabase& mDatabase;
};
```

Slika 3.5: PictureDao.h

Forma upita na bazu može se razdvojiti na pet osnovih koraka:

- deklaracija QSqlQuery varijable
- priprema upita - prepare
- vezanje varijabli za upit - bind
- izvršavanje upita - exec
- dohvaćanje rezultata upita - value("key")

Funkcija bind kao drugi parametar treba vrstu QVariant koja može poprimiti bilo koji od popularnih tipova poput QString, int, char itd.

```
void AlbumDao::init() const
{
```

```
    if (!mDatabase.tables().contains("albums")) {
      QSqlQuery query(mDatabase);
      query.exec("CREATE TABLE albums ("
        "id INTEGER PRIMARY KEY AUTOINCREMENT, "
        "name TEXT, "
        "date_created DATE DEFAULT (datetime('now','localtime')), "
        "date_modified DATE DEFAULT (datetime('now','localtime')))");
      DatabaseManager::debugQuery(query);
    }
  }
```

Slika 3.6: `init()` funkcija klase `AlbumDao`

```
void AlbumDao::addAlbum(Album& album) const
{
  QSqlQuery query(mDatabase);
  query.prepare("INSERT INTO albums (name) VALUES (:name)");
  query.bindValue(":name", album.name());
  query.exec();
  DatabaseManager::debugQuery(query);
  album.setId(query.lastInsertId().toInt());
  auto now = QDateTime::currentDateTime();
  album.setDateCreated(now);
  album.setDateModified(now);
}
```

Slika 3.7: Upit za dodavanje albuma u `AlbumDao` klasi

SQLITE 3.35.0 uveo je novu ključnu riječ - RETURNING - koja bi bila korisna za neposredno vraćanje vremena stvaranja albuma uz dodatak na upit u funkciji `addAlbum`:

```
    "RETURNING date_created"
```

Međutim, Qt-ov *plugin* za SQLITE podržava samo verziju 3.0, u kojoj bi se to moglo postići jedino sa novim SELECT upitom. S obzirom da nije potrebna savršena preciznost, izabrano je rješenje pozivanja Qt-ovog lokalnog vremena nakon stvaranja objekta u bazi, čime ne bi trebala doći razlika u vremenima veća od jedne sekunde.

Zanimljiva je i funkcija `picturesForAlbum` koja je prikazana na slici 3.8 koja dohvaća sve slike iz pojedine baze - sortirane ovisno o predanoj `sortIndex` varijabli. Kraj-

nja aplikacija imat će opciju sortiranja prikaza prema imenu ili datumu stvaranja što je najlakše obaviti u samoj bazi. Ključne riječi COLLATE NOCASE znače zanemarivanje velikog i malog slova pri usporedbi imena.

```
QVector<Picture &> PictureDao::picturesForAlbum(int albumId, int
    sortIndex) const
{
  QSqlQuery query(mDatabase);
  QString queryText = "SELECT * FROM pictures WHERE album_id = (:
      album_id)";
  switch(sortIndex){
  case 0: queryText += " ORDER BY date_added DESC";
    break;
  case 1: queryText += " ORDER BY date_added ASC";
    break;
  case 2: queryText += " ORDER BY name COLLATE NOCASE ASC";
    break;
  case 3: queryText += " ORDER BY name COLLATE NOCASE DESC";
    break;
  }

  query.prepare(queryText);
  query.bindValue(":album_id", albumId);
  query.exec();
  DatabaseManager::debugQuery(query);
  QVector<Picture &> list;
  while(query.next()) {
    Picture picture = new Picture();
    picture->setId(query.value("id").toInt());
    picture->setAlbumId(query.value("album_id").toInt());
    picture->setFileUrl(query.value("url").toString());
    picture->setName(query.value("name").toString());
    picture->setDateAdded(query.value("date_added").toDateTime());
    list->push_back(picture);
  }
  return list;
}
```

Slika 3.8: Upit za dohvaćanje slika iz albuma

Sljedeći je korak povezati novostvorene klase u `DatabaseManager`. Potrebno je u `DatabaseManager.h` dodati `AlbumDao` i `PictureDao` objekte tako da su `const` i `public` radi zaštite i dostupnosti. Nadalje potrebno je ažurirati konstruktor odnosno inicijalizirati dodane klase te pokrenuti pripadne `init()` funkcije. Promjene se vide na slici

3.9. Kod inicijalizacije bitno je pripaziti da se `albumDao` i `pictureDao` inicijaliziraju **nakon** `mDatabase`. To se čini tako da se njihova deklaracija u `DatabaseManager.h` postavi **nakon** deklaracije `mDatabase`.

Iz procesa stvaranja ove dvije DAO klase može se primijetiti da dolazi do ponavljanja istog koda. U nešto većem projektu bilo bi povoljno napraviti baznu DAO klasu koju bi zatim nasljeđivale sve ostale. Ona bi također mogla pratiti i bilježiti sve stvorene klase te zatim pozvati pripadne `init()` funkcije u konstruktoru `DatabaseManager()`. Međutim u ovom bi slučaju zbog jednostavnosti strukture projekta to bila nepotrebna komplikacija.

```
DatabaseManager::DatabaseManager(const QString& path) :
    mDatabase(new QSqlDatabase(QSqlDatabase::addDatabase("QSQLITE"))),
    albumDao(*mDatabase),
    pictureDao(*mDatabase)
{
    mDatabase->setDatabaseName(path);
    mDatabase->open();

    albumDao.init();
    pictureDao.init();
}
```

Slika 3.9: Nadopunjeni konstruktor klase `DatabaseManager`

3.3 Uređivanje pokazivača

Korištenje običnih C++ pokazivača (eng. *pointer*-a) često je riskantno. Razlog tome je što nije jasno tko je vlasnik pokazivača, odnosno tko prati dealokaciju memorije na koju se pokazuje nakon što pokazivač više nije potreban. Bolja je opcija koristiti `std::unique_ptr<T>` iz C++ zaglavlja `memory` uz koji je uvijek poznato tko je vlasnik, a moguće je i proslijediti vlasništvo pomoću funkcije `std::move(...)`.

Implementacija ovih promjena za klasu `AlbumDao` vidi se na slici 3.10, a sličnim postupkom mijenja se i `PictureDao` klasa. Popis albuma prije je bio tip `QVector<T>` no promijenjen je u `std::unique_ptr<std::vector<T>>` zato što `QVector` preopterećuje konstruktor kopiranja koji se ovime upravo pokušava izbjeći. Stoga on ne može raditi sa `unique_ptr` objektima. Ovo vrijedi za sve `QtContainer`-e.

```
// AlbumDao.h
std::unique_ptr<std::vector<std::unique_ptr<Album>>> albums() const;
```

```
// AlbumDao.cpp
std::unique_ptr<std::vector<std::unique_ptr<Album>>> AlbumDao::albums
    () const
{
    QSqlQuery query("SELECT * FROM albums", mDatabase);
    query.exec();

    std::unique_ptr<std::vector<std::unique_ptr<Album>>> list(new std
        ::vector<std::unique_ptr<Album>>());
    while(query.next()) {
        std::unique_ptr<Album> album(new Album());
        album->setId(query.value("id").toInt());
        album->setName(query.value("name").toString());
        list->push_back(std::move(album));
    }

    return list;
}
```

Slika 3.10: `AlbumDao` klasa nakon implementacije `unique_ptr`-a

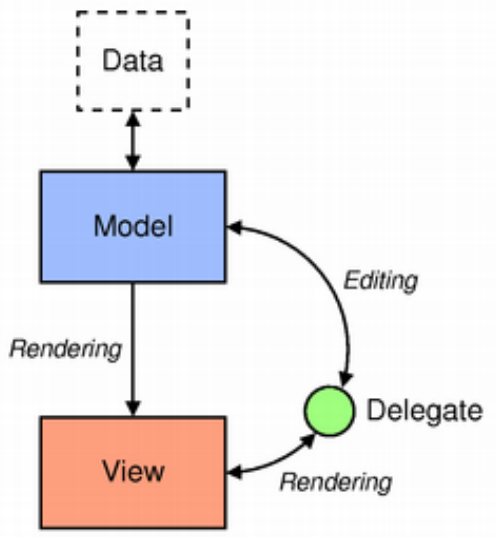
Poglavlje 4

Implementacija modela

4.1 Model/View arhitektura

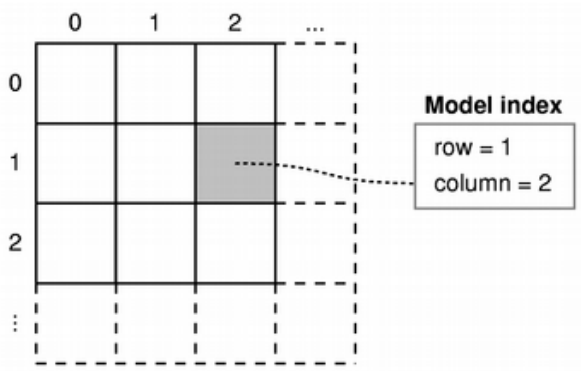
U ovom je poglavlju glavni cilj stvoriti vezu između baze podataka i klijenta. To je moguće korištenjem koncepta sličnog MVC (eng. *Model-View-Controller*) sustavu koji je zapravo klasičan izbor kod izrade grafičkih sučelja. Model se bavi obradom, view prikazom podataka a controller interakcijom korisničkog sučelja i korisnika. Ovakvom podjelom poslova dobiva se na fleksibilnosti i prostoru za izmjenu i dodavanje u budućnosti.

Arhitektura koju Qt koristi u tu svrhu zove se **Model/View**. Sa skice koja se nalazi na slici 4.1 vidimo podjelu na tri komponente - *model*, *view* i *delegate*.



Slika 4.1: Model/View arhitektura [1]

Model je veza između podataka i korisničkog sučelja. Realizira se uz pomoć `QAbstractItemModel` klase. Ovo je veoma opširna bazna klasa koju nasljeđuju mnogi Qt5 predlošci. Početni raspored podataka bazne klase vidi se na slici 4.2. Zapravo se radi o tablici redova i stupaca gdje **Model index** (u Qt-u klasa `QModelIndex`) predstavlja jedan objekt. Spomenuti predlošci zatim oblikuju ovaj sustav pomoću `override` funkcija. Preporučljivo je izabrati jedan od predložaka koji najbolje pristaje traženom modelu jer se inače stvara prevelika ili pak nedovoljna kompleksnost.



Slika 4.2: Prikaz podataka u `QAbstractItemModel` [1]

Lazar i Penea (2016) promatraju nekoliko predložaka, no na koncu odlučuju da je `QAbstractListModel` prigodan za ovaj projekt. Kao što se može naslutiti iz imena, on se koristi kada podaci trebaju biti složeni u listu.

View je skup klasa koje čine korisničko sučelje. Ono komunicira sa model i delegate komponentama pomoću `QModelIndex` objekata. Bazna klasa je `QAbstractItemView`, no preporučljivo je fokusirati se na konkretniju implementaciju koja je u slučaju ove aplikacije `QListView`.

Delegate ili delegat je pomoćna komponenta koja nakon dohvata iz model-a i prije predaje istih u view vrši razne stilske transformacije. Bazna klasa za delegat komponente u Qt-u je `QAbstractItemDelegate`, a `QStyledItemDelegate` je implementirana klasa koja je povoljna za nadogradnje i stvaranje vlastitih delegata

4.2 Model za album

Sljedeći je korak stvaranje klasa koji će implementirati izabrani model. U `gallery-core` projekt dodaju se klase `AlbumModel` i `PictureModel`. Na slici 4.3 prikazana je prva klasa.

```
class GALLERYCORESHARED_EXPORT AlbumModel : public QAbstractListModel
{
  Q_OBJECT
public:

  enum Roles {
    IdRole = Qt::UserRole + 1,
    NameRole,
    DateCreatedRole,
    DateModifiedRole
  };

  AlbumModel(QObject* parent = 0);

  QModelIndex addAlbum(const Album& album);
  QModelIndex copyAlbum(const Album& album, const QString name);
  bool albumExists(QString name) const;

  int rowCount(const QModelIndex& parent = QModelIndex()) const
     override;
  QVariant data(const QModelIndex& index, int role = Qt::DisplayRole)
     const override;
  bool setData(const QModelIndex& index, const QVariant& value, int
     role) override;
  bool removeRows(int row, int count, const QModelIndex& parent)
     override;
  QHash<int, QByteArray> roleNames() const override;

private:
  bool isIndexValid(const QModelIndex& index) const;

private:
  DatabaseManager& mDb;
  std::unique_ptr<std::vector<std::unique_ptr<Album>>> mAlbums;
};
```

Slika 4.3: `AlbumModel.h`

Neke od funkcija klase `AlbumModel` prerađuju funkcije bazne klase `QAbstractListModel` dok druge pokrivaju ostatak potrebnih funkcionalnosti. Slijedi kratko objašnjenje pojedinih članova i metoda:

- `mDb` – baza podataka
- `mAlbums` – DAO pomoću kojeg se izbjegava redundantna komunikacija sa bazom

- `albumExists`, `rowCount`, `data` – `const` funkcije dohvata podataka
- `addAlbum`, `copyAlbum`, `setData`, `removeRows` – funkcije izmjene podataka
- `isIndexValid` – pomoćna funkcija
- `AlbumRole` – enumeracija uloga; funkcije koje traže ovaj argument vrše različite operacije ovisno o njegovoj vrijednosti

Konstruktor klase `AlbumModel` nalazi se na slici 4.4, dok su na slikama 4.5 i 4.7 izabrani primjeri funkcija pripadno dohvata i izmjene podataka. U konstruktoru se dohvaća *Singleton* instanca baze te se odmah zatim `mAlbums` puni sa eventualnim albumima koji se nalaze u bazi.

```
AlbumModel::AlbumModel(QObject* parent) :
    QAbstractListModel(parent),
    mDb(DatabaseManager::instance()),
    mAlbums(mDb.albumDao.albums())
{
}
```

Slika 4.4: Konstruktor klase `AlbumModel`

U funkciji `data` dohvaćaju se podaci albuma ovisno o `role`. Ovdje se vidi korištenje enumeracije `AlbumRole` te funkcije `isIndexValid` čija se implementacija vidi na slici 4.6. Iz nje vidimo da se provjerava je li indeks izvan konteksta te je li model valjan. Iako se provjeravaju i redovi i stupci, u *List* modelu jedino ima smisla promatrati redove. Naime naš je cijeli model jedan stupac, a njegovi su redovi objekti modela.

```
QVariant AlbumModel::data(const QModelIndex& index, int role) const
{
    if (!isIndexValid(index)) {
        return QVariant();
    }
    const Album& album = *mAlbums->at(index.row());

    switch (role) {
        case Roles::IdRole:
            return album.id();

        case Roles::NameRole:
        case Qt::DisplayRole:
            return album.name();
```

```
            break;

        case Roles::DateCreatedRole:
            return album.dateCreated();
            break;

        case Roles::DateModifiedRole:
            return album.dateModified();
            break;

        default:
            return QVariant();
    }
}
```

Slika 4.5: Primjer funkcije dohvata podataka u klasi `AlbumModel`

```
bool AlbumModel::isIndexValid(const QModelIndex& index) const
{
    if (index.row() < 0 || index.row() >= rowCount() || !index.isValid
        ()) {
        return false;
    }
    return true;
}

//QModelIndex, c=collumn, r=row, m=model
Q_DECL_CONSTEXPR inline bool isValid() const noexcept { return (r >=
    0) && (c >= 0) && (m != nullptr); }
```

Slika 4.6: Funkcija koja provjerava valjanost danog indeksa

Funkcija `addAlbum` predstavlja tipičnu funkciju izmjene podataka u ovoj vrsti modela. Glavni dio funkcije potrebno je smjestiti između signalnih funkcija, u ovom slučaju `beginInsertRows` i `endInsertRows`, koje će poslati signal sučelju da se trenutno vrši ubacivanje slika. Ovo će biti korisno u sljedećim poglavljima pri prikazu i ažuriranju sučelja. Glavni dio funkcije vrši promjene na bazi pomoću `mDb`, a odmah zatim ažurira podatke lokalno u `mAlbums`.

```
QModelIndex AlbumModel::addAlbum(const Album& album)
{
```

```
    int rowIndex = rowCount();

    beginInsertRows(QModelIndex(), rowIndex, rowIndex);
    std::unique_ptr<Album> newAlbum(new Album(album));
    mDb.albumDao.addAlbum(*newAlbum);
    mAlbums->push_back(move(newAlbum));
    endInsertRows();

    return index(rowIndex, 0);
}
```

Slika 4.7: Primjer funkcije izmjene podataka u klasi `AlbumModel`

U nekoliko se metoda pojavljuje argument `parent`. Razlog tome je ugrađena hijerarhija klase `QAbstractItemModel` koja se bolje vidi na slici 4.10. Naime, uvijek postoji *root* objekt čija su djeca svi ostali objekti. Ovo je posebno korisno za model stabla, no on neće biti korišten ovdje.

4.3 Model za slike

Sučelje klase `PictureModel` nalazi se na slici 4.8 i ono je nešto kompleksnije od `AlbumModel` klase, no u suštini veoma slično. `PictureModel` predstavlja jedan album odnosno skup slika, dok `AlbumModel` predstavlja galeriju odnosno skup albuma. Kao i u odnosu `Picture` i `Album` klasa, ona je uvijek povezana sa svojom `AlbumModel` instancom. Stoga `PictureModel` ima dodatni argument u konstruktoru (slika 4.9) imena `albumModel` pomoću kojeg se sinkronizira brisanje albuma u dvije klase.

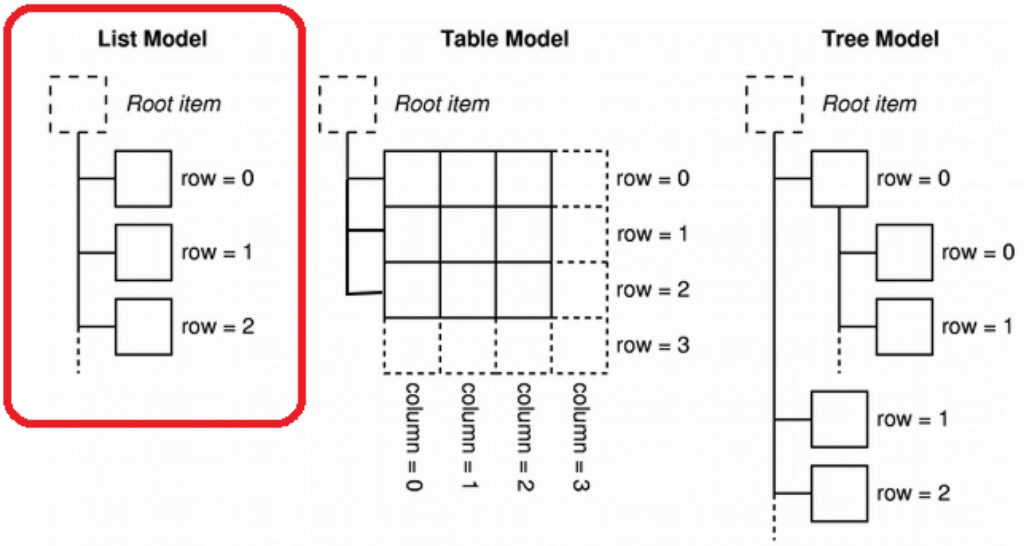
```
class GALLERYCORESHARED_EXPORT PictureModel : public
    QAbstractListModel
{
    Q_OBJECT
public:

    enum Roles {
        UrlRole = Qt::UserRole + 1,
        FilePathRole,
        DateAddedRole
    };
    PictureModel(const AlbumModel& albumModel, QObject* parent = 0);

    QModelIndex addPicture(const Picture& picture);
    void copyPicturesFromAlbum(int albumId);
```

```
    bool pictureExists(int albumId, QUrl url) const;

    int rowCount(const QModelIndex& parent = QModelIndex()) const
        override;
    QVariant data(const QModelIndex& index, int role) const override;
    bool removeRows(int row, int count, const QModelIndex& parent)
        override;
    QHash<int, QByteArray> roleNames() const override;

    void setAlbumId(int albumId, int sortIndex);
    void clearAlbum();
    QVector<QString> getPictureUrls();

public slots:
    void deletePicturesForAlbum();

private:
    void loadPictures(int albumId, int sortIndex);
    bool isIndexValid(const QModelIndex& index) const;

private:
    DatabaseManager& mDb;
    int mAlbumId;
    std::unique_ptr<std::vector<std::unique_ptr<Picture>>> mPictures;
};
```

Slika 4.8: `PictureModel.h`

```
PictureModel::PictureModel(const AlbumModel& albumModel, QObject*
    parent) :
    QAbstractListModel(parent),
    mDb(DatabaseManager::instance()),
    mAlbumId(-1),
    mPicture(new std::vector<std::unique_ptr<Picture>>())
{
    connect(&albumModel, &AlbumModel::rowsRemoved, this, &PictureModel
        ::deletePicturesForAlbum);
}
```

Slika 4.9: Konstruktor klase `PictureModel`



Slika 4.10: Nekoliko implementacija `QAbstractItemModel` klase [1]

Poglavlje 5

Temelji korisničkog sučelja

U prošlom je poglavlju dovršena biblioteka `gallery-core`. Preostalo je napraviti grafičko sučelje te povezati ga sa dosad napravljenim funkcionalnostima. U tu se svrhu dodaje novi pod-projekt koji će se zvati `gallery-frontend`[1]. Postupak je sličan kao i prije – desnim klikom na glavni projekt:

gallery → New subproject → Application → Qt Widgets Application → Choose.

Sljedeći je korak povezivanje sa `gallery-core` bibliotekom, na sljedeći način:

gallery → Add library → Internal library → gallery-core → Next → Finish.

Rezultati stvaranja pod-projekta i dodavanja biblioteke vide se u `gallery-frontend .pro` datoteci na slici 5.1.

Dodavanje biblioteke vrši se naredbom:

```
LIBS += -L<put-do-biblioteke> -l<ime-biblioteke>
```

Za razliku od linux sustava, na windows sustavu se stvaraju `debug` i `release` direktoriji pa se taj dio koda mora pokriti petljom. Zadnje dvije linije koda određuju lokaciju zaglavlja biblioteke. Pritom se koriste qmake varijable `$$OUT_PWD` i `$$PWD` koje označavaju apsolutnu adresu do izlaznog direktorija i trenutne `.pro` datoteke pripadno.

```
QT       += core gui widgets

TARGET = gallery
TEMPLATE = app
```

[1] *Frontend* (eng. *front end* = prednji dio) je termin koji se češće koristi za web aplikacije. On predstavlja grafičko sučelje dok nasuprot njega – *backend* – predstavlja podatke i njihovu obradu. `gallery-core` se stoga može smatrati *backend*-om gallery aplikacije.

```
SOURCES += main.cpp\
        MainWindow.cpp \

HEADERS  += MainWindow.h \

FORMS    += MainWindow.ui \

win32:CONFIG(release, debug|release): LIBS += -L$$OUT_PWD/../gallery-
    core/release/ -lgallery-core
else:win32:CONFIG(debug, debug|release): LIBS += -L$$OUT_PWD/../
    gallery-core/debug/ -lgallery-core
else:unix: LIBS += -L$$OUT_PWD/../gallery-core/ -lgallery-core

INCLUDEPATH += $$PWD/../gallery-core
DEPENDPATH += $$PWD/../gallery-core
```

Slika 5.1: gallery-frontend.pro

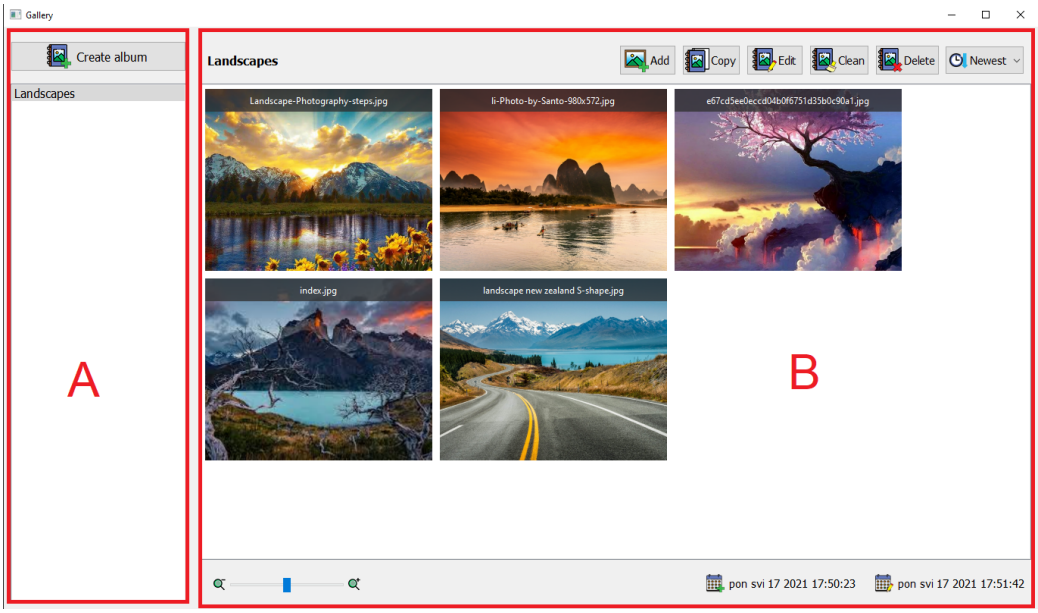
Ove se promjene moraju nadopuniti i u .pro datoteci glavnog projekta korištenjem ključne riječi depends kako je prikazano na slici 5.2. Time je osigurano da gallery-core ima prioritet nad gallery-frontend pri izgradnji.

```
TEMPLATE = subdirs

SUBDIRS += \
    gallery-core \
    gallery-frontend

gallery-frontend.depends = gallery-core
```

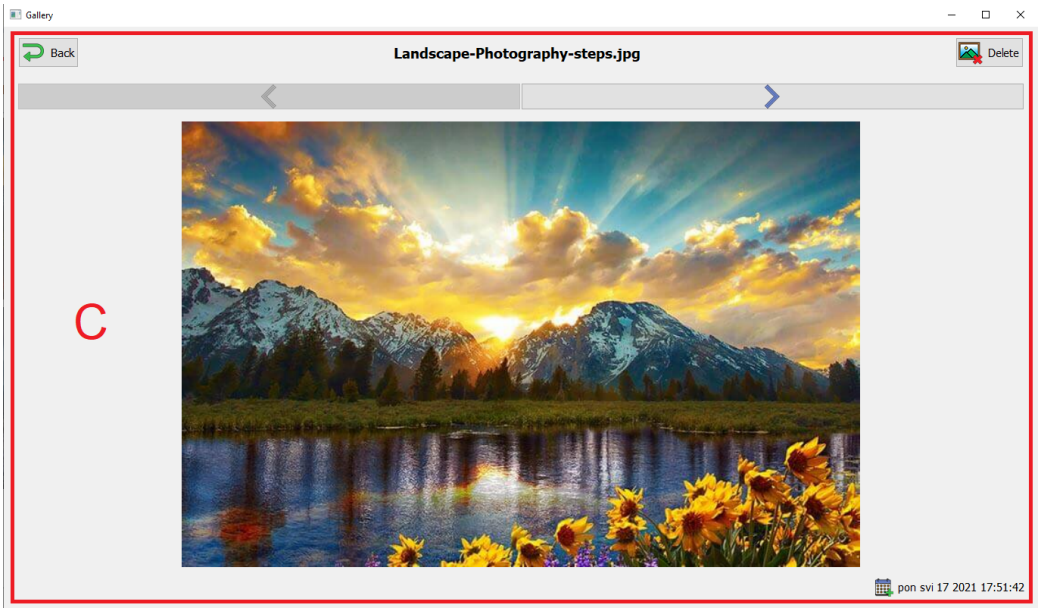
Slika 5.2: gallery.pro

5.1 Organizacija sučelja

Prije implementacije funkcija iz gallery-core u grafičko sučelje, korisno je organizirati projekt u više manjih cjelina. Krajnja aplikacija prikazana je na slikama 5.3 i 5.4.



Slika 5.3: Prikaz popisa album i jednog otvorenog albuma u gotovoj aplikaciji (`GalleryWidget` klasa)



Slika 5.4: Prikaz pojedine slike u gotovoj aplikaciji (`PictureWidget` klasa)

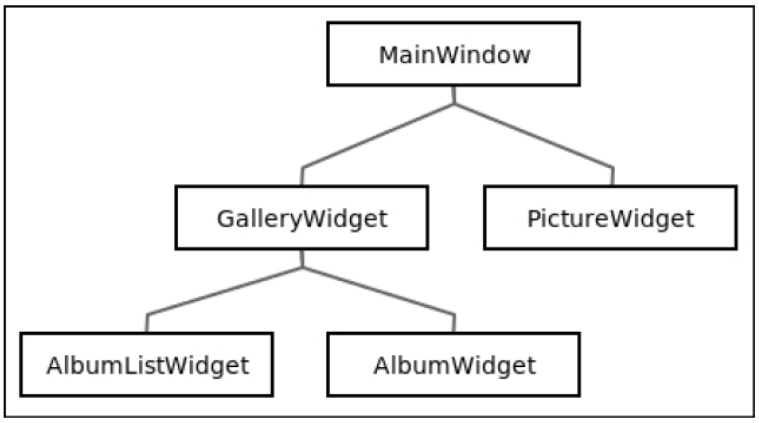
Sa slika se vidi da su izdvojene tri glavne komponente:

A. `AlbumListWidget` - popis albuma

B. `AlbumWidget` - prikaz albuma izabranog sa popisa

C. `PictureWidget` - prikaz slike izabrane iz albuma

Glavna forma aplikacije zove se `MainWindow` i ona je još jedna bitna komponenta, zadužena za izmjenu prikaza između galerije i slike. `AlbumListWidget` i `AlbumWidget` dolaze u paru te zajedno čine `GalleryWidget`. Opisana hijerarhija vidi se na slici 5.5.



Slika 5.5: Hijerarhija formi u `gallery-frontend` [3]

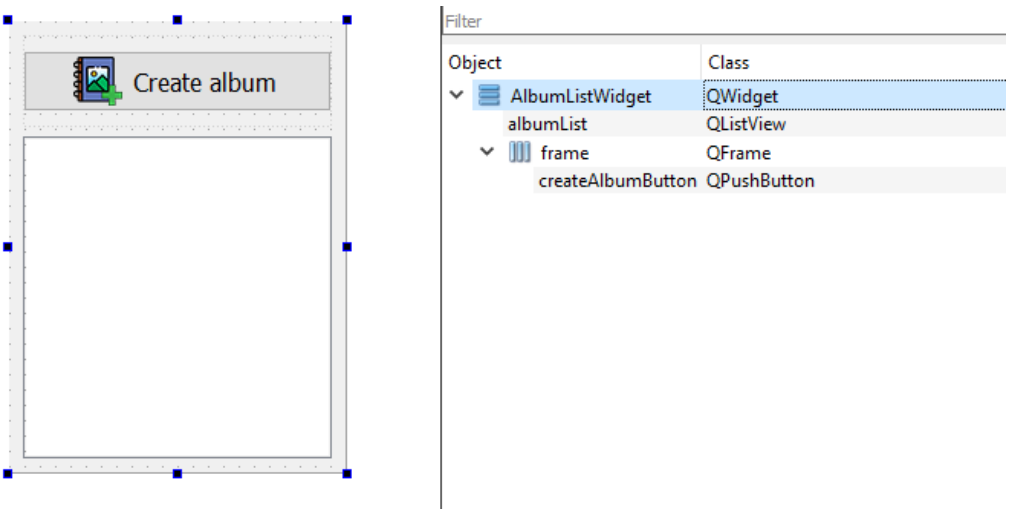
U poglavlju 3 uveden je `std::unique_ptr<std::vector<...>>` umjesto `QVector`, no običaj kod stvaranja grafičkih sučelja u Qt-u je koristiti isključivo Q-objekte pa će stoga u daljnoj implementaciji biti korišten `QVector` i ostali Q spremnici.

Poglavlje 6

Izgradnja korisničkog sučelja

6.1 Popis albuma

Prva klasa na redu je `AlbumListWidget` pa stoga se kreira pripadna klasa i forma. Nakon stvaranja forme potrebno je u nju dodati sve potrebne *QWidget*-e. U ovom se slučaju radi o vrlo jednostavnoj konstrukciji. Na slici 6.1 prikazana je gotova forma. `AlbumListWidget` sastoji se od `QListView` (izabran u prethodnom poglavlju) koji će biti popunjen `Album` objektima te od jednog gumba za stvaranje albuma.



Slika 6.1: Prikaz forme `AlbumListWidget`

Ciljane mogućnosti ove klase su:

- **Create album** - Stvaranje novog albuma

- Prikaz svih albuma

Sa slike se vidi da gumb za dodavanje novog albuma ima posebnu ikonu. Ona se može odrediti u Qt Designer-u dohvatom slike iz resursa. **Resurs** je datoteka koja sadrži druge datoteke koje se koriste u projektu (najčešće slike, videozapisi i sl.). Njega je najprije potrebno stvoriti na sljedeći način:

gallery-frontend → Add New → Qt → Qt Resource File.

Ovom će se naredbom stvoriti nova datoteka `resource.qrc` koja će automatski biti dodana u `gallery-frontend.pro` linijom koda `RESOURCES += resource.qrc`. Sada je moguće dodati sliku u stvorenoj resurs datoteci te ju zatim gumb može dohvatiti kao ikonu u svojim postavkama (vidi slika 1.3, D).

Time je gotova izrada forme i može se preći na sučelje `AlbumWidgetList.h` koje se nalazi na slici 6.2. Najbitnije funkcije su `setModel` i `setSelectionModel` i one prilagođavaju `QListView` popis albuma. Prva definira model koji se koristi (u ovom slučaju `AlbumModel` iz `gallery-core`) dok druga određuje `QItemSelectionModel` koji se koristi za izbor pojedinog albuma u listi. Potonji se model može proslijediti daljnjim *widget*-ima kako bi se ostvarila dobra sinkronizacija. Implementacija ovih funkcija, konstruktora i destruktora nalaze se na slici 6.3.

```
class AlbumListWidget : public QWidget
{
    Q_OBJECT

public:
    explicit AlbumListWidget(QWidget *parent = 0);
    ~AlbumListWidget();

    void setModel(AlbumModel* model);
    void setSelectionModel(QItemSelectionModel* selectionModel);

private slots:
    void createAlbum();

private:
    Ui::AlbumListWidget* ui;
    AlbumModel* mAlbumModel;
};
```

Slika 6.2: `AlbumWidgetList.h`

```
AlbumListWidget::AlbumListWidget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::AlbumListWidget),
    mAlbumModel(nullptr)
{
    ui->setupUi(this);
    ui->albumList->setFont(QFont(ui->albumList->fontInfo().family(),
       12));

    connect(ui->createAlbumButton, &QPushButton::clicked,
            this, &AlbumListWidget::createAlbum);
}

AlbumListWidget::~AlbumListWidget()
{
    delete ui;
}

void AlbumListWidget::setModel(AlbumModel* model)
{
    mAlbumModel = model;
    ui->albumList->setModel(mAlbumModel);
}

void AlbumListWidget::setSelectionModel(QItemSelectionModel*
   selectionModel)
{
    ui->albumList->setSelectionModel(selectionModel);
}
```

Slika 6.3: `AlbumWidgetList.cpp`

Postavkom modela u funkciji `setModel` postiže se da `QListView` u pozadini automatski poziva funkciju `rowCount` pripadnog modela te dohvaća pojedino ime albuma pomoću `Qt::DisplayRole`. U ovom se dijelu koda najviše oslanja na gotove i naslijeđene Qt klase.

U konstruktoru se, između ostaloga, povezuje prethodno stvoreni gumb za funkcijom za stvaranje novog albuma `createAlbum` čija se implementacija nalazi na slici 6.4.

```
AlbumListWidget::createAlbum()
{
  if(!mAlbumModel) {
    return;
```

```
    }
    bool ok;
    while(true){
      QString albumName = QInputDialog::getText(this,
                                    "Create a new Album",
                                    "Choose an name",
                                    QLineEdit::Normal,
                                    "New album",
                                    &ok).trimmed();
      if(!ok) return;
      if(!albumName.isEmpty()) {
        Album album(albumName);
        if(mAlbumModel->albumExists(album.name())){
          QMessageBox::warning(this, "Album already exists",
                          "Album with given name already exists.",
                          QMessageBox::Ok);
        }
        else {
          QModelIndex createdIndex = mAlbumModel->addAlbum(album);
          ui->albumList->setCurrentIndex(createdIndex);
          return;
        }
      }
    }
  }
```

Slika 6.4: Funkcija za stvaranje novog albuma u `AlbumListWidget`

Korisniku se prikazuje dijalog u kojem unosi ime albuma. Radi bolje preglednosti, u galeriji nije moguće imati dva albuma sa istim imenom. Stoga model nakon unosa najprije provjerava sa bazom postoji li već takav album. Ako postoji, korisnik mora ponovno pokušati unijeti ime albuma. Inače je album stvoren, a model vraća njegov indeks pomoću kojeg ga `albumList` može postaviti kao trenutno označenog.

6.2 Stvaranje minijatura

Komponenta `AlbumWidget` prikazivat će sve slike izabranog albuma u obliku minijatura ili *thumbnail*-ova. `PictureModel` ovaj put neće biti dovoljan za takav prikaz s obzirom da `Picture` sadrži samo `QUrl`, a ne cijelu sliku. Rješenje je dodavanje nove klase `ThumbnailProxyModel` koja će biti filter između `AlbumWidget` i `PictureModel` klasa.

Bazna klasa `QAbstractProxyModel` ima dvije podklase: `QIdentityProxyMo-`

del koja je pogodna kada treba izmijeniti data() funkciju i QSortFilterProxyModel koja ima dodatnu opciju sortiranja. Druga je potrebna u ovom slučaju zbog prije najavljene mogućnost sortiranja slika u albumu.

Nakon stvaranja i popunjavanja zaglavlja klase ThumbnailProxyModel, dobiva se rezultat sa slike 6.5.

```
class ThumbnailProxyModel : public QSortFilterProxyModel
{
public:
    ThumbnailProxyModel(QObject* parent = 0);

    QVariant data(const QModelIndex& index, int role) const override;
    void setSourceModel(QAbstractItemModel* sourceModel) override;
    bool lessThan(const QModelIndex &left, const QModelIndex &right)
       const override;
    PictureModel* pictureModel() const;

    qreal getMultiplier() const;
    void setSize(int size);

    int getSortIndex() const;
    void setSort(int sortIndex);

private:
    void reloadThumbnails();

private:
   QHash<QString, QPixmap*> mThumbnails;
   int mSize;
   int mSort;

};
```

Slika 6.5: ThumbnailProxyModel.h

Slijedi kratko objašnjenje pojedinih funkcija:

- data - dohvaća URL slike
- setSourceModel - sinkronizira se sa izvornim modelom (PictureModel)
- lessThan - binarna relacija po kojoj će se vršiti sortiranje
- pictureModel - pomoćna funkcija za raspoznavanje izvornog modela kao PictureModel klase

- `getMultiplier`, `setSize` - funkcije za određivanje veličine minijatura
- `getSortIndex`, `setSort` - funkcije za određivanja načina sortiranja
- `reloadThumbnails` - briše prethodne minijature i generira aktualne

Najbitnija je funkcija `reloadThumbnails` čija se implementacija nalazi na slici 6.7. Ona najprije čisti svoju listu minijatura. Zatim, prije dohvaćanja novih, poziva funkciju `sort` uz argument 0 što znači da će sortirati prvi (jedini) stupac. Sortiranje će iskoristiti prerađuju verziju funkcije `lessThan` koja se nalazi na slici 6.8.

Sortiranje se može izvršiti na četiri načina: padajuće ili rastuće prema imenu slike ili prema datumu dodavanja. Funkcija uspoređuje `QDateTime` ili `QString` ovisno o članu `mSort` čije su vrijednosti po značenju *hard-coded* na isti način kao u bazi. Korisno je u ovakvoj situaciji uvesti posebnu enumeraciju kako bi se lakše pratili slučajevi te mogli dodati novi. No za ovaj jednostavan projekt to nije bilo potrebno.

Vraćanjem na funkciju `reloadThumbnails` koja sada ima sortirane podatke, ona dalje iterira po svim indeksima te za pojedini stvara novi `QPixmap` prema URL koji je dohvatila funkcija `data`. `QPixmap` je grafička Qt klasa koja, između ostaloga, može sadržavati sliku.

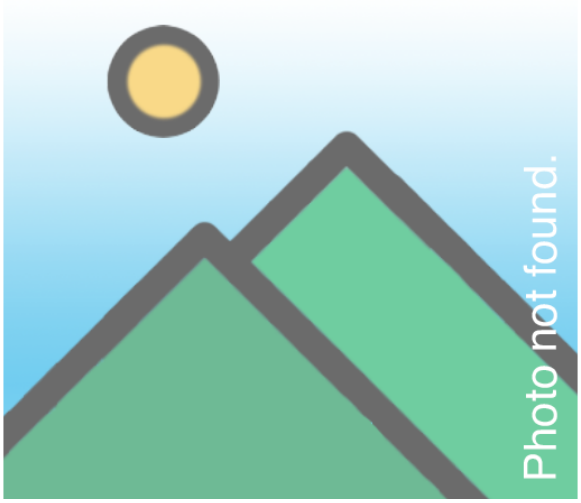
Postoji mogućnost da je URL iz baze postao neispravan, odnosno pripadna je slika pomaknuta ili izbrisana. Tada se u `QPixmap` stavlja generična slika koja je prethodno postavljena u resurs datoteku (slika 6.6). Ovdje se vidi programibilni način dohvaćanja datoteka iz resursa.

Koji god se slučaj dogodio, sljedeći je korak formatiranje slike za prikaz u galeriji. Ono se može podijeliti na sljedeća dva koraka:

1. skaliranje slike na približnu dimenziju određenu konstantnim varijablama `THUMBNAIL_WIDTH` i `THUMBNAIL_HEIGHT` pomnoženih sa koeficijentom koji ovisi o članu `mSize`
2. rezanje slike u okvir tako da dimenzija bude fiksna; pozicija rezanja izračunata je u varijablama `marginTop` i `marginLeft`

Određivanje koeficijenta skaliranja slično je kao određivanje vrste sortiranja. Implementacija funkcije nalazi se na slici 6.9 gdje se vide točno izabrane vrijednosti.

```
void ThumbnailProxyModel::reloadThumbnails()
{
  qDeleteAll(mThumbnails);
  mThumbnails.clear();

  sort(0);
```



Slika 6.6: Generička slika korištena za slike koje nisu pronađene

```
const QAbstractItemModel* model = index(0, 0).model();
for(int row = 0; row < rowCount(); row++) {
  QString filepath = model->data(model->index(row, 0),
          PictureModel::Roles::FilePathRole).toString();
  QPixmap pixmap(filepath);
  if(pixmap.isNull()){
    pixmap = QPixmap(":/images/photo-blank.png");
  }
  pixmap = pixmap.scaled(THUMBNAIL_WIDTH*getMultiplier(),
     THUMBNAIL_HEIGHT*getMultiplier(),
               Qt::KeepAspectRatioByExpanding,
               Qt::SmoothTransformation);
  double marginTop = qMax(pixmap.height()
                - THUMBNAIL_HEIGHT*getMultiplier(), 0.0);
  double marginLeft = qMax(pixmap.width()
                - THUMBNAIL_WIDTH*getMultiplier(), 0.0);
  auto thumbnail = new QPixmap(pixmap.copy(
                     QRect((int)marginLeft/2,
                     (int)marginTop/2,
                     THUMBNAIL_WIDTH*getMultiplier(),
                     THUMBNAIL_HEIGHT*getMultiplier())));

  mThumbnails.insert(filepath, thumbnail);
}
```

```
}
```

Slika 6.7: Implementacija funkcije `reloadThumbnails`

```
bool ThumbnailProxyModel::lessThan(const QModelIndex &left, const
    QModelIndex &right) const
{
    PictureModel* pictureModel = (PictureModel*) sourceModel();

    if(mSort == 0 || mSort == 1){
        QVariant leftDate = pictureModel->data(left, PictureModel::
            Roles::DateAddedRole);
        QVariant rightDate = pictureModel->data(right, PictureModel::
            Roles::DateAddedRole);

        if(mSort == 0) // newest
            return leftDate.toDateTime() > rightDate.toDateTime();
        if(mSort == 1) // oldest
            return leftDate.toDateTime() < rightDate.toDateTime();
    }
    else{
        QString leftString = pictureModel->data(left, Qt::DisplayRole)
            .toString();
        QString rightString = pictureModel->data(right, Qt::
            DisplayRole).toString();

        return !QString::compare(leftString, rightString, Qt::
            CaseInsensitive);
    }
    return 0;
}
```

Slika 6.8: Implementacija funkcije `lessThan`

```
qreal ThumbnailProxyModel::getMultiplier() const
{
    switch(mSize){
    case 0: //small
        return 0.5;

    case 1: //medium
```

```
            return 1;

        default: //large
            return 2;
        }
    }
```

Slika 6.9: Implementacija funkcije `getMultiplier`

Ostatak implementacije klase `ThumbnailProxyModel` nalazi se na slici 6.10 (bez trivijalnih *getter*-a i *setter*-a). Iz konstruktora se vide početne vrijednosti članova `mSize` i `mSort` što znači da će minijature biti neizmjenjene veličine te sortirane silazno po vremenu dodavanja.

Iako izvorni model može biti bilo koja podklasa `QAbstractItemModel`-a, u ovoj se klasi pretpostavlja da će to biti `PictureModel` što se vidi u funkciji `pictureModel`.

```
ThumbnailProxyModel::ThumbnailProxyModel(QObject* parent) :
    QSortFilterProxyModel(parent),
    mThumbnails(),
    mSize(1),
    mSort(0)
{
}

QVariant ThumbnailProxyModel::data(const QModelIndex& index, int role)
    const
{
    if (role != Qt::DecorationRole) {
        return QSortFilterProxyModel::data(index, role);
    }

    QString filepath = sourceModel()->data(index, PictureModel::Roles
        ::FilePathRole).toString();
    return *mThumbnails[filepath];
}

void ThumbnailProxyModel::setSourceModel(QAbstractItemModel*
    sourceModel)
{
    QSortFilterProxyModel::setSourceModel(sourceModel);
    if (!sourceModel) {
        return;
    }
```

```
    connect(sourceModel, &QAbstractItemModel::modelReset, [this] {
        reloadThumbnails();
    });

    connect(sourceModel, &QAbstractItemModel::rowsInserted, [this] {
        reloadThumbnails();
    });
}

PictureModel* ThumbnailProxyModel::pictureModel() const
{
    return static_cast<PictureModel*>(sourceModel());
}
```

Slika 6.10: ThumbnailProxyModel.cpp

6.3 Prikaz albuma

Na redu je do sada najkompleksnije klasa i forma `AlbumWidget`. Postupak stvaranja je sličan kao kod stvaranja liste albuma. Na slici 6.11 se nalazi uređena forma koja se sastoji od tri sloja:

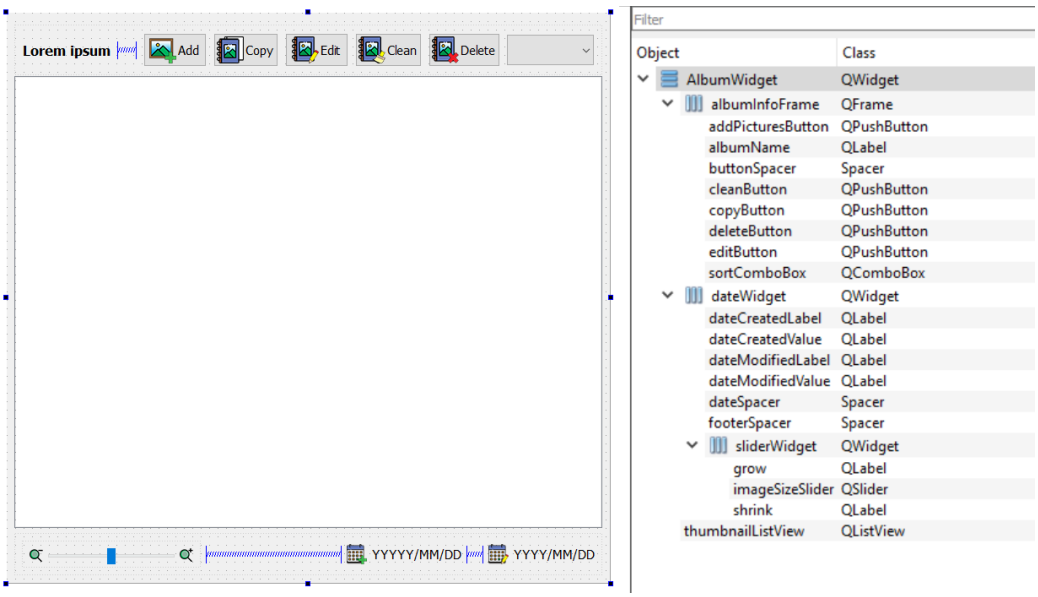
- Alatna traka ili `albumInfoFrame`
- Lista slika albuma ili `thumbnailListView`
- Pomoćna statusna traka ili `dateWidget`

Slojevi su raspoređeni kao redovi, a slojevi su zatim raspoređeni kao nizovi stupaca, čime se postiže elegantno sučelje.

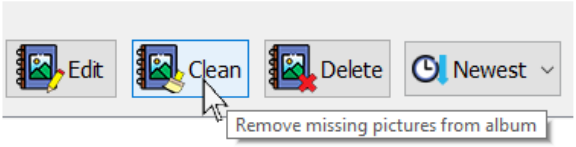
Nakon izrade forme jasno se vide ciljevi klase `AlbumWidget`:

- **Add** - dodavanje slika u album
- **Copy** - kopiranje/dupliciranje albuma
- **Edit** - preimenovanje albuma
- **Clean** - čišćenje neispravnih URL-ova
- **Delete** - brisanje albuma
- sortiranje albuma
- prikaz slika iz albuma (minijature)

- izmjena veličine minijatura
- prikaz podataka o vremenu stvaranja/izmjene albuma



Slika 6.11: Prikaz forme `AlbumWidget`



Slika 6.12: Primjer pomoćne poruke

Ove funkcionalnost možda nisu očite korisniku koji se prvi put susreće sa aplikacijom. Stoga je dobro imati neku vrstu dokumentacije unutar same aplikacije. Jedna je od opcija posebna stranica na ulazu u aplikaciju koja sadrži potrebne upute za korištenje. Također je moguće implementirati dodatni izbornik sa dokumentacijom.

Treća opcija (koja je izabrana u ovoj aplikaciji) jest dodavanje pomoćnih poruka ili *tooltip*-ova na sve bitne *widget*-e u sučelju. One se mogu postaviti u Qt Form Designer-u. *Tooltip* je kratka poruka koja se pojavi ako pokazivač zastane na neko vrijeme nad pojedinim elementom. Primjer ovoga se vidi na slici 6.12.

Slijedi izrada klase, počevši sa zaglavljem koje se nalazi na slici 6.13.

```
class AlbumWidget : public QWidget
{
    Q_OBJECT

public:
    explicit AlbumWidget(QWidget *parent = 0);
    ~AlbumWidget();

    void setAlbumModel(AlbumModel* albumModel);
    void setAlbumSelectionModel(QItemSelectionModel*
       albumSelectionModel);
    void setPictureModel(ThumbnailProxyModel* pictureModel);
    void setPictureSelectionModel(QItemSelectionModel* selectionModel)
       ;

signals:
    void pictureActivated(const QModelIndex& index);

private slots:
    void addPictures();
    void copyAlbum();
    void editAlbum();
    void cleanAlbum();
    void deleteAlbum();
    void sortPictures(int sortIndex);
    void changePictureSize();

private:
    void clearUi();
    void loadAlbum(const QModelIndex& albumIndex);

private:
    Ui::AlbumWidget* ui;
    AlbumModel* mAlbumModel;
    QItemSelectionModel* mAlbumSelectionModel;

    ThumbnailProxyModel* mPictureModel;
    QItemSelectionModel* mPictureSelectionModel;
};
```

Slika 6.13: AlbumWidget.h

Najprije se dodaju *setter*-i za modele. U `AlbumWidget` je potrebno imati `AlbumModel` i `AlbumSelectionModel` za sinkronizaciju sa cijelom aplikacijom te `PictureModel` odnosno `ThumbnailProxyModel` iz prethodnog poglavlja te `Pictu-`

`reSelectionModel` za prikaz slika albuma.

Implementacija *setter*-a nalazi se na slici 6.14. Najzanimljivji su modeli za album. Potrebno je povezati signal izmjene podataka i promjene izbora albuma sa trenutnim prikazom.

```
void AlbumWidget::setAlbumModel(AlbumModel* albumModel)
{
    mAlbumModel = albumModel;

    connect(mAlbumModel, &QAbstractItemModel::dataChanged, [this] (
        const QModelIndex &topLeft) {
        if (topLeft == mAlbumSelectionModel->currentIndex()) {
            loadAlbum(topLeft);
        }
    });
}

void AlbumWidget::setAlbumSelectionModel(QItemSelectionModel*
    albumSelectionModel)
{
    mAlbumSelectionModel = albumSelectionModel;

    connect(mAlbumSelectionModel, &QItemSelectionModel::
        selectionChanged, [this](const QItemSelection &selected) {
        if (selected.isEmpty()) {
            clearUi();
            return;
        }
        loadAlbum(selected.indexes().first());
    });
}

void AlbumWidget::setPictureModel(ThumbnailProxyModel* pictureModel)
{
    mPictureModel = pictureModel;
    ui->thumbnailListView->setModel(pictureModel);
}

void AlbumWidget::setPictureSelectionModel(QItemSelectionModel*
    selectionModel)
{
    ui->thumbnailListView->setSelectionModel(selectionModel);
}
```

Slika 6.14: *setter*-i klase `AlbumWidget`

Natrag u zaglavlju klase vidi se jedan signal `pictureActivated`. On će biti aktiviran pri označavanju slike dvostrukim klikom. Konkretna implementacija vidi se u opsežnom konstruktoru klase na slici 6.15. U njemu se određuju dodatne pojedinosti o prikazu forme te se povezuju funkcije sa pripadnim tipkama u formi.

```
AlbumWidget::AlbumWidget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::AlbumWidget),
    mAlbumModel(nullptr),
    mAlbumSelectionModel(nullptr),
    mPictureModel(nullptr),
    mPictureSelectionModel(nullptr)
{
    ui->setupUi(this);
    clearUi();

    ui->thumbnailListView->setSpacing(5);
    ui->thumbnailListView->setResizeMode(QListView::Adjust);
    ui->thumbnailListView->setFlow(QListView::LeftToRight);
    ui->thumbnailListView->setWrapping(true);

    connect(ui->thumbnailListView, &QListView::doubleClicked,
            this, &AlbumWidget::pictureActivated);

    connect(ui->addPicturesButton, &QPushButton::clicked,
            this, &AlbumWidget::addPictures);

    connect(ui->copyButton, &QPushButton::clicked,
            this, &AlbumWidget::copyAlbum);

    connect(ui->editButton, &QPushButton::clicked,
            this, &AlbumWidget::editAlbum);

    connect(ui->cleanButton, &QPushButton::clicked,
            this, &AlbumWidget::cleanAlbum);

    connect(ui->deleteButton, &QPushButton::clicked,
            this, &AlbumWidget::deleteAlbum);

    connect(ui->sortComboBox, QOverload<int>::of(&QComboBox::activated
        ),[=](int sortIndex){
```

```
            sortPictures(sortIndex);
        });

        connect(ui->imageSizeSlider, &QAbstractSlider::valueChanged,
                this, &AlbumWidget::changePictureSize);
    }
```

Slika 6.15: Konstruktor klase `AlbumWidget`

Sljedeće što treba promotriti su bitnije funkcije klase `AlbumWidget`.

Funkcija `addPictures` nalazi se na slici 6.16. Ona najprije provjerava valjanost trenutno označenog indeksa. Zatim se otvara poseban dijalog za višestruki izbor datoteka zvan `QFileDialog`. Njemu se predaje naslov, početni direktorij pri otvaranju te vrsta datoteka koje će se tražiti.

Nakon izbora slika slijedi njihovo filtriranje. Ako neka od slika već postoji odnosno ako ima URL koji već postoji u označenom albumu onda se ona preskače. Sve ispravno izabrane slike se zatim dodaju u album preko modela za slike `mPictureModel`. Ako postoje neispravno dodane slike, korisnik nakon ubacivanja dobiva prikladnu obavijest.

Na koncu se ažurira podatak o vremenu izmjene albuma pomoću funkcije `setData` koja će u isto vrijeme aktivirati signal `dataChanged` koji će zatim ažurirati prikaz albuma.

```
void AlbumWidget::addPictures()
{
    if (mAlbumSelectionModel->selectedIndexes().isEmpty()) {
        return;
    }
    QModelIndex currentAlbumIndex = mAlbumSelectionModel->
       selectedIndexes().first();
    QStringList filenames = QFileDialog::getOpenFileNames(
                this,
                "Add pictures",
                QDir::homePath(),
                "Picture files (*.jpg *.png)");

    int albumId = mAlbumModel->data(currentAlbumIndex, AlbumModel::
       Roles::IdRole).toInt();
    QVector<QString> invalidPictures;
    if (!filenames.isEmpty()) {
        QModelIndex lastModelIndex;
        for (auto filename : filenames) {
            Picture picture(filename);
            if(mPictureModel->pictureModel()->pictureExists(albumId,
```

```
                picture.fileUrl())){
                 invalidPictures.push_back(filename);
            }
            else{
                lastModelIndex = mPictureModel->pictureModel()->
                    addPicture(picture);
            }
        }
        if(invalidPictures.size()) {
            QString replyText = "Some of the pictures added already
                exist:";
            for(auto pictures : invalidPictures) replyText += "\n" +
                pictures;
            QMessageBox::warning(this, "Pictures already exist",
                                 replyText + ".",
                                 QMessageBox::Ok);
            if(invalidPictures.size() == filenames.size()) return;
        }

        ui->thumbnailListView->setCurrentIndex(lastModelIndex);
        mAlbumModel->setData(currentAlbumIndex, QDateTime::
            currentDateTime(), AlbumModel::Roles::DateModifiedRole);
    }
}
```

Slika 6.16: Funkcija za dodavanje slika u album

Na slici 6.17 je implementacija funkcije `deleteAlbum`. Ona prije samog brisanja traži od korisnika potvrdu. Zatim u slučaju potvrdnog odgovora briše pripadni album te pokušava izabrati sljedeći ako postoji. U protivnom funkcija završava bez promjena na albumu.

```
void AlbumWidget::deleteAlbum()
{
    QMessageBox::StandardButton reply = QMessageBox::warning(
                this, "Confirm delete album",
                "Are you sure you want to delete this album?",
                QMessageBox::Yes|QMessageBox::No);
    if(reply == QMessageBox::No) return;

    if (mAlbumSelectionModel->selectedIndexes().isEmpty()) {
        return;
    }
    int row = mAlbumSelectionModel->currentIndex().row();
```

```
        mAlbumModel->removeRow(row);

        // Try to select the previous album
        QModelIndex previousModelIndex = mAlbumModel->index(row - 1, 0);
        if(previousModelIndex.isValid()) {
            mAlbumSelectionModel->setCurrentIndex(previousModelIndex,
                QItemSelectionModel::SelectCurrent);
            return;
        }

        // Try to select the next album
        QModelIndex nextModelIndex = mAlbumModel->index(row, 0);
        if(nextModelIndex.isValid()) {
            mAlbumSelectionModel->setCurrentIndex(nextModelIndex,
                QItemSelectionModel::SelectCurrent);
            return;
        }
    }
```

Slika 6.17: Funkcija za brisanje albuma

Funkcije `copyAlbum` i `editAlbum` imaju sličnu formu kao `addPictures` funkcija: provjeri valjanost indeksa, zatraži od korisnika novo ime za album, provjeri postoji li već album sa tim imenom te ukoliko je sve u redu izvrši potrebnu operaciju na bazi. Glavna je razlika u izboru dijaloga, točnije umjesto `QFileDialog` objekta koristi se `QInputDialog`.

Zanimljivija je funkcija `cleanAlbum` čija se implementacija vidi na slici 6.18. Ona prolazi po svim indeksima albuma te za svaki mora otvoriti novi `QPixmap` sa pripadnim URL-om. Ako je rezultat stvaranja objekta `null`, to znači da URL nije ispravan. Stoga se ta slika briše te se dodaje u popis obrisanih. Konačno, korisnik dobiva listu svih na ovaj način izbrisanih slika ako takve postoje, a inače dobiva prikladnu obavijest.

```
AlbumWidget::cleanAlbum()
{
    QString invalidUrls = "";
    QVector<int> indexToRemove;
    if(mPictureModel->pictureModel()->rowCount() != 0){
        auto startIndex = mPictureModel->pictureModel()->index(0,0);
        const QAbstractItemModel* model = startIndex.model();
        for(int row = startIndex.row(); row < mPictureModel->
            pictureModel()->rowCount(); ) {
             QString filepath = model->data(model->index(row, 0),
                 PictureModel::Roles::FilePathRole).toString();
```

```
                QPixmap pixmap(filepath);
                if(pixmap.isNull()){
                    invalidUrls += "\n" + filepath;
                    mPictureModel->pictureModel()->removeRows(row, 1,
                        QModelIndex());
                }
                else row++;
            }
        }
        if(invalidUrls == ""){
            QMessageBox::information(this, "Cleaning pictures",
                                     "Nothing to clean.",
                                     QMessageBox::Ok);
        }
        else{
            QMessageBox::information(this, "Cleaning pictures",
                                     "Missing or deleted pictures were removed
                                         from album:" +
                                     invalidUrls + ".",
                                     QMessageBox::Ok);
        }
    }
```

Slika 6.18: Funkcija za brisanje neispravnih slika

Od preostalih interaktivnih funkcija imaju dvije jednostavne funkcije `changePictureSize` i `sortPictures` koje se nalaze na slici 6.19. One mijenjaju pripadnu varijablu u `ThumbnailListModel` te osvježavaju prikaz.

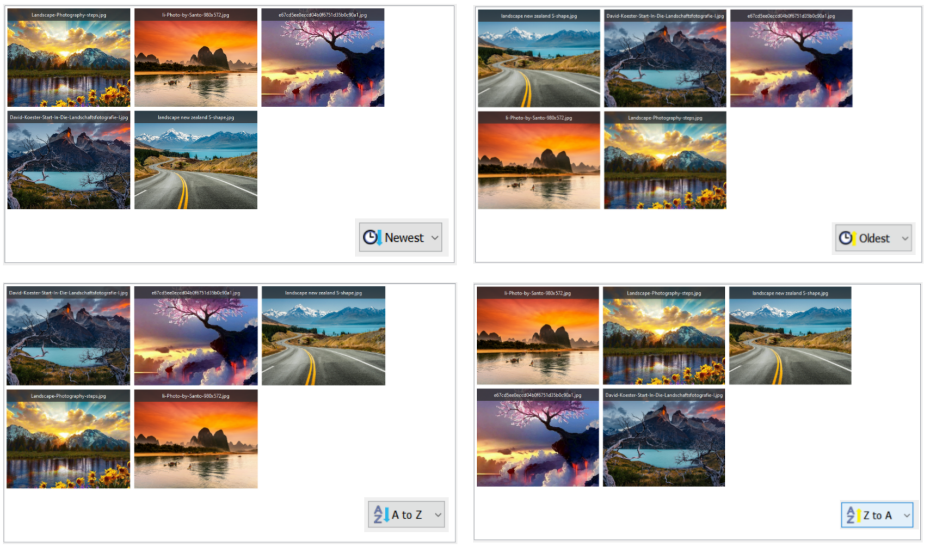
```
void AlbumWidget::changePictureSize()
{
    mPictureModel->setSize(ui->imageSizeSlider->value());
    loadAlbum(mAlbumSelectionModel->selectedIndexes().first());
}

void AlbumWidget::sortPictures(int sortIndex)
{
    mPictureModel->setSort(sortIndex);
    loadAlbum(mAlbumSelectionModel->selectedIndexes().first());
}
```

Slika 6.19: Funkcije za promjenu veličine i načina sortiranja slika

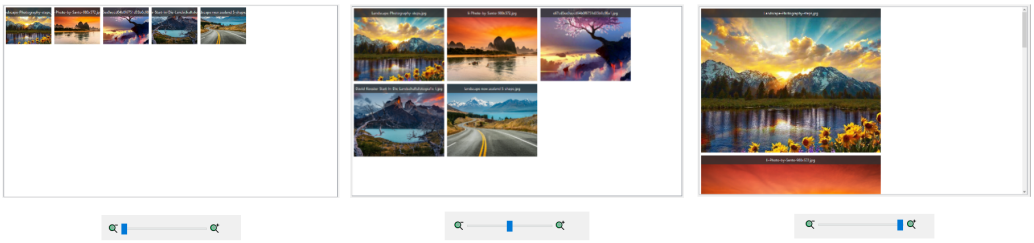
`changePictureSize` funkcija bit će aktivirana promjenom na `QSlider` objektu što je zapravo klizni izbornik sa tri moguća stanja.

Slično se aktivira `sortPictures`, no ovaj put sa `QComboBox` objektom. Radi se o padajućem izborniku sa četiri izbora.

Pojedina stanja ovih izbornika opisana su u prethodnom poglavlju, a na slikama 6.20 i 6.21 dani su primjeri sortiranja i promjena veličine.[1]



Slika 6.20: Primjer više načina sortiranja



Slika 6.21: Primjer više veličina slika

Ostatak implementacije klase `AlbumWidget` nalazi se na slici 6.22. Prva funkcija `clearUi` sakriva veći dio sučelja. Ovakvo stanje bit će vidljivo kada nema nijednog albuma. Druga funkcija učitava album te otkriva sučelje i u njega popunjava podatke o albumu.

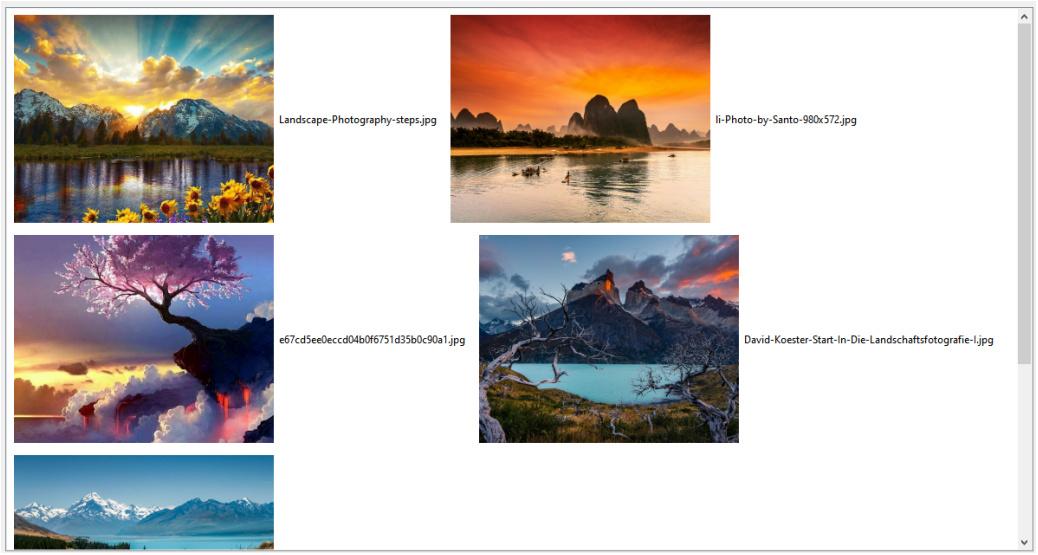
[1] Isječci su uzeti iz gotove aplikacije. Za ovakav izgled minijatura vidi sljedeći odjeljak.

```
void AlbumWidget::clearUi()
{
    ui->albumName->setText("Welcome to GalleryApp");
    ui->dateWidget->setVisible(false);
    ui->deleteButton->setVisible(false);
    ui->cleanButton->setVisible(false);
    ui->editButton->setVisible(false);
    ui->copyButton->setVisible(false);
    ui->addPicturesButton->setVisible(false);
    ui->sliderWidget->setVisible(false);
    ui->sortComboBox->setVisible(false);
}

void AlbumWidget::loadAlbum(const QModelIndex& albumIndex)
{
    mPictureModel->pictureModel()->setAlbumId(mAlbumModel->data(
        albumIndex, AlbumModel::Roles::IdRole).toInt(), mPictureModel->
        getSortIndex());
    ui->albumName->setText(mAlbumModel->data(albumIndex, Qt::
        DisplayRole).toString());
    auto dateCreated = mAlbumModel->data(albumIndex, AlbumModel::Roles
        ::DateCreatedRole).toDateTime();
    ui->dateCreatedValue->setText(dateCreated.date().toString() + " "
        + dateCreated.time().toString());
    auto dateModified = mAlbumModel->data(albumIndex, AlbumModel::
        Roles::DateModifiedRole).toDateTime();
    ui->dateModifiedValue->setText(dateModified.date().toString() + "
        " + dateModified.time().toString());
    ui->dateWidget->setVisible(true);
    ui->deleteButton->setVisible(true);
    ui->cleanButton->setVisible(true);
    ui->editButton->setVisible(true);
    ui->copyButton->setVisible(true);
    ui->addPicturesButton->setVisible(true);
    ui->sliderWidget->setVisible(true);
    ui->sortComboBox->setVisible(true);
}
```

Slika 6.22: AlbumWidget.cpp

6.4 Dodaci na minijature

Trenutni prikaz albuma, što je zapravo `QListView`, iskoristit će `Qt::DisplayRole` i `Qt::DecorationRole` da dohvati ime i način prikaza pojedine slike. Time se dobiva već gotov vizualni model koji se vidi na slici 6.23.



Slika 6.23: Početni prikaz albuma

Ovo je dosta grub način prikazivanja slika. Ime se nalazi zdesna slici što izgleda nespretno. Srećom, postoji način kako to popraviti, a to je klasa `QStyedItemDelegate`. Radi se zapravo o baznoj klasi koju `QListView` koristi bez dodatnih intervencija. Međutim, moguće je stvoriti novu klasu koja nasljeđuje `QStyledItemDelegate` te ju definirati u postavkama liste.

U tu svrhu potrebno je stvoriti novu klasu `PictureDelegate`. Njeno zaglavlje nalazi se na slici 6.24, a implementacija na slici 6.25. Zahvaljujući dobroj strukturi bazne klase, `PictureDelegate` je zapravo vrlo jednostavna klasa kojoj su potrebne samo dvije prerađene funkcije.

```
class PictureDelegate : public QStyledItemDelegate
{
    Q_OBJECT
public:
    PictureDelegate(QObject* parent = 0);
```

```
    void paint(QPainter* painter, const QStyleOptionViewItem& option,
        const QModelIndex& index) const override;
    QSize sizeHint(const QStyleOptionViewItem& option, const
        QModelIndex& index) const override;
};
```

Slika 6.24: `PictureDelegate.h`

```
const unsigned int BANNER_HEIGHT = 30;
const unsigned int BANNER_COLOR = 0x303030;
const unsigned int BANNER_ALPHA = 200;
const unsigned int BANNER_TEXT_COLOR = 0xffffff;
const unsigned int HIGHLIGHT_ALPHA = 100;

PictureDelegate::PictureDelegate(QObject* parent) :
    QStyledItemDelegate(parent)
{
}

void PictureDelegate::paint(QPainter* painter, const
    QStyleOptionViewItem& option, const QModelIndex& index) const
{
    painter->save();

    QPixmap pixmap = index.model()->data(index, Qt::DecorationRole).
        value<QPixmap>();
    painter->drawPixmap(option.rect.x(), option.rect.y(), pixmap);

    QRect bannerRect = QRect(option.rect.x(), option.rect.y(), pixmap.
        width(), BANNER_HEIGHT);
    QColor bannerColor = QColor(BANNER_COLOR);
    bannerColor.setAlpha(BANNER_ALPHA);
    painter->fillRect(bannerRect, bannerColor);

    QString filename = index.model()->data(index, Qt::DisplayRole).
        toString();
    painter->setPen(BANNER_TEXT_COLOR);
    painter->drawText(bannerRect, Qt::AlignCenter, filename);

    if (option.state.testFlag(QStyle::State_Selected)) {
        QColor selectedColor = option.palette.highlight().color();
        selectedColor.setAlpha(HIGHLIGHT_ALPHA);
        painter->fillRect(option.rect, selectedColor);
    }
```

```
    painter->restore();
}

QSize PictureDelegate::sizeHint(const QStyleOptionViewItem& /*option*/
    , const QModelIndex& index) const
{
    const QPixmap& pixmap = index.model()->data(index, Qt::
        DecorationRole).value<QPixmap>();
    return pixmap.size();
}
```

Slika 6.25: PictureDelegate.cpp

Glavna je funkcija paint. Ona će se izvršiti na svakom indeksu pojedinačno i ima sljedeća četiri koraka:

- crtanje slike odnosno QPixmap
- dodavanje crnog zaglavlja
- pisanje imena slike na zaglavlje
- ako je slika označena onda prebojaj platno plavom (uvjetni korak)

Crtanje se realizira Qt klasom crtača odnosno QPainter. Prije crtanja treba spremiti njegovo početno stanje sa painter->save() kako bi se ono moglo vratiti nakon crtanja sa painter->restore(). U protivnom bi se slike taložile u crtaču te bi došlo do nepoželjnog rezultata.

Pojedini objekt se crta sa pripadnom funkcijom, pa tako se QPixmap crta sa QPainter::drawPixmap, zaglavlje koje je zapravo pravokutnik crta se sa QPainter::fillRect() te na kraju ime slike sa QPainter::drawText() funkcijom.

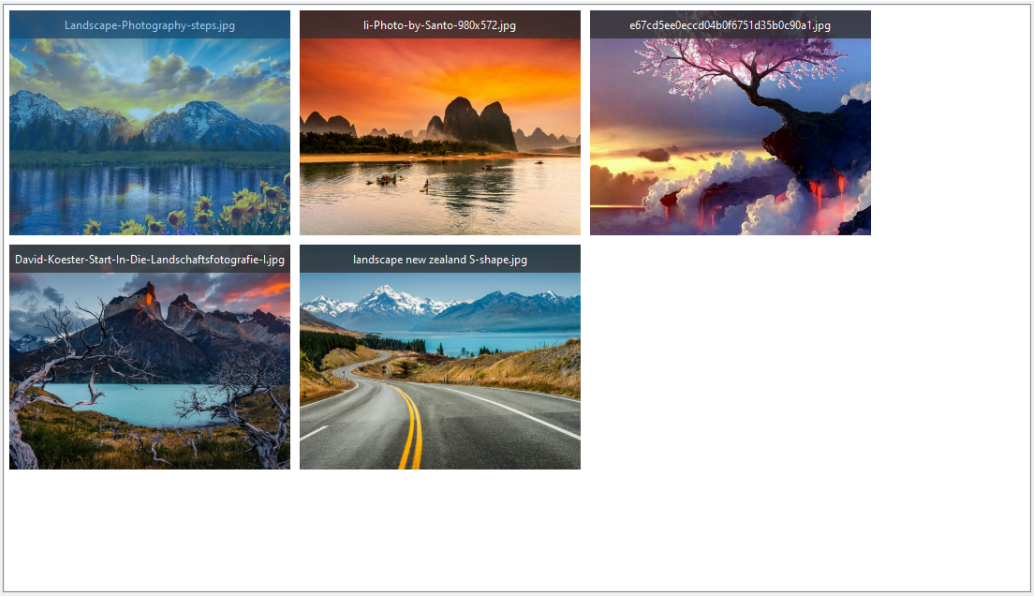
Funkcija sizeHint dohvatit će potrebnu minijaturu te vratiti njene dimenzije. Ovo je bitno kako bi se ostali elementi poboljšane minijature mogli staviti na ispravno mjesto.

Konstante na početku PictureDelegate.cpp datoteke određuju parametre veličina i boje detalja u novoj minijaturi, čime je omogućeno jednostavno variranje izgleda.

Zadnje što treba napraviti je povezati AlbumWidget sa novostvorenim delegatom. To se radi dodavanje sljedeće linije u njen konstruktor:

```
ui->thumbnailListView->setItemDelegate(new PictureDelegate(this));
```

Konačna izmjenjena verzija prikaza albuma vidi se na slici 6.26.



Slika 6.26: Poboljšani prikaz albuma

6.5 Prikaz slika

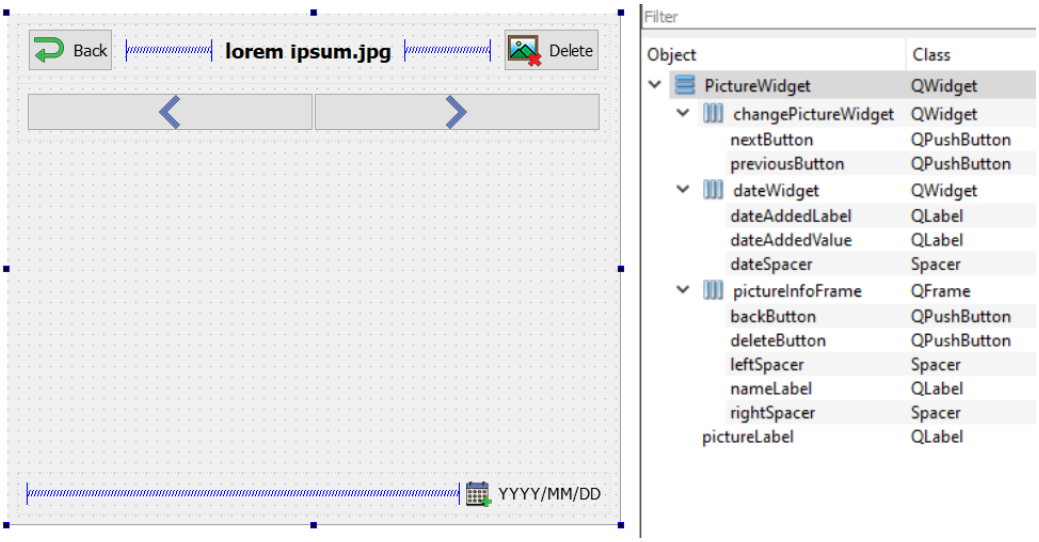
Sljedeće što treba implementirati je prikaz pojedine slike. Za ovo je zadužena klasa `PictureWidget`. Veći dio ovog odjeljka bit će skraćen zbog sličnosti sa prethodno stvorenim klasama (prvenstveno `AlbumWidget`). Najprije se stvara forma kao na slici 6.27.

Slično kao `AlbumWidget` forma, ona ima više slojeva:

- Alatna traka ili `albumInfoFrame`
- Tipke za listanje slika ili `changePictureWidget`
- Uvećani prikaz slike ili `pictureLabel`
- Pomoćna statusna traka ili `dateWidget`

Ciljevi klase `PictureWidget` su sljedeći:

- **Back** - povratak u `AlbumWidget`
- **Delete** - brisanje trenutno otvorene slike
- listanje prethodne/sljedeće slike u albumu
- povećani prikaz slike



Slika 6.27: Forma `PictureWidget`

- prikaz podataka o vremenu dodavanja slike u album

Nadalje promatramo zaglavlje klase `PictureWidget` na slici 6.28. Modeli koji se koriste u ovoj klasi su `ThumbnailProxyModel` te `QItemSelectionModel` što je očekivano s obzirom da su isti korišteni u `AlbumWidget`.

Nekoliko zanimljivosti koje se mogu izdvojiti su mehanizam tipki za prethodnu/sljedeću sliku, funkcija `deletePicture` te funkcija `resizeEvent` pripadno na slikama 6.29, 6.30 i 6.32.

```
class PictureWidget : public QWidget
{
    Q_OBJECT

public:
    explicit PictureWidget(QWidget *parent = 0);
    ~PictureWidget();

    void setModel(ThumbnailProxyModel* model);
    void setSelectionModel(QItemSelectionModel* selectionModel);

signals:
    void backToGallery();

protected:
    void resizeEvent(QResizeEvent* event) override;
```

```
private slots:
    void deletePicture();
    void loadPicture(const QItemSelection& selected);

private:
    void updatePicturePixmap();

private:
    Ui::PictureWidget* ui;
    ThumbnailProxyModel* mModel;
    QItemSelectionModel* mSelectionModel;
    QPixmap mPixmap;
};
```

Slika 6.28: `PictureWidget.h`

Klikom na tipke `previousButton` ili `nextButton` pomiče se izabrani indeks u manji odnosno veći. Bitno je da se ne dogodi prelijev u neispravan indeks, pa se stoga taj uvjet provjerava u funkciji `loadPicture` te ako je nužno tipka se isključuje.

```
PictureWidget::PictureWidget(QWidget *parent) :
  ...
{
  ...

  connect(ui->previousButton, &QPushButton::clicked, [this] () {
    QModelIndex currentModelIndex = mSelectionModel->currentIndex();
    QModelIndex previousModelIndex = mSelectionModel->model()->index(
      currentModelIndex.row() - 1, 0);
    mSelectionModel->setCurrentIndex(previousModelIndex,
        QItemSelectionModel::SelectCurrent);
  });

  connect(ui->nextButton, &QPushButton::clicked, [this] () {
    QModelIndex currentModelIndex = mSelectionModel->currentIndex();
    QModelIndex nextModelIndex = mSelectionModel->model()->index(
      currentModelIndex.row() + 1, 0);
    mSelectionModel->setCurrentIndex(nextModelIndex,
        QItemSelectionModel::SelectCurrent);
  });

  ....
}
```

```
void PictureWidget::loadPicture(const QItemSelection& selected)
{
  ...

    ui->previousButton->setEnabled(current.row() > 0);
    ui->nextButton->setEnabled(current.row() < (mModel->rowCount() -
        1));

  ...
}
```

Slika 6.29: Mehanizam mijenjanja slike

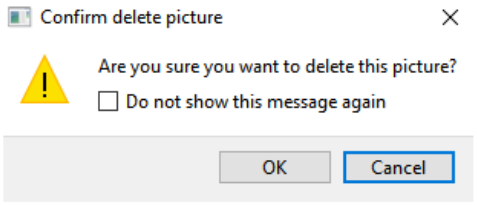
Brisanje slika vrši se slično kao u `AlbumWidget`, no sa malom izmjenom. Uvodi se globalna varijabla `ASK_BEFORE_DELETE` koja je postavljena na `true`. To znači da će korisnik biti upozoren prije brisanja slike te mora potvrditi tu akciju. Međutim korisnik sada ima mogućnost spriječiti daljnja upozorenja s obzirom da možda želi obrisati mnogo slika u kratkom vremenu. Ovo se postiže stvaranjem `QMessageBox`-a te dodavanjem `QCheckBox` objekta pomoću funkcije `setCheckBox`. Ovaj dijalog vidi se na slici 6.31.

Funkcija nadalje briše odnosno ne briše ovisno o odgovoru korisnika, te ukoliko briše mora izabrati sljedeći indeks. Ako je to bila zadnja slika u albumu onda se aktivira signal `backToGallery` koji će biti povezan sa utorom u sljedećem poglavlju.

```
void PictureWidget::deletePicture()
{
    if(ASK_BEFORE_DELETE){
        QCheckBox *cb = new QCheckBox("Do not show this message again"
            );
        QMessageBox msgbox;
        msgbox.setWindowTitle("Confirm delete picture");
        msgbox.setText("Are you sure you want to delete this picture?"
            );
        msgbox.setIcon(QMessageBox::Icon::Warning);
        msgbox.addButton(QMessageBox::Ok);
        msgbox.addButton(QMessageBox::Cancel);
        msgbox.setDefaultButton(QMessageBox::Cancel);
        msgbox.setCheckBox(cb);
        auto reply = msgbox.exec();

        if(msgbox.checkBox()->isChecked())
            ASK_BEFORE_DELETE = false;
        if(reply == QMessageBox::Cancel) return;
    }
```

```
    ...
}
```

Slika 6.30: Funkcija `deletePicture`



Slika 6.31: Upozorenje prije brisanje slike

Funkcija `resizeEvent` prerađena je funkcija klase `QWidget` te se izvršava kada korisnik ili program izmjeni veličinu prozora aplikacije. Implementacija je vrlo jednostavna i dodatno koristi pomoćnu funkciju `updatePicturePixmap` no rezultat je da će slika uvijek zauzimati što veći dio zaslona što je i cilj ove klase.

```
void PictureWidget::resizeEvent(QResizeEvent* event)
{
    QWidget::resizeEvent(event);
    updatePicturePixmap();
}

void PictureWidget::updatePicturePixmap()
{
    if (mPixmap.isNull()) {
        return;
    }
    ui->pictureLabel->setPixmap(mPixmap.scaled(ui->pictureLabel->size
        (), Qt::KeepAspectRatio, Qt::SmoothTransformation));
}
```

Slika 6.32: Funkcija `resizeEvent`

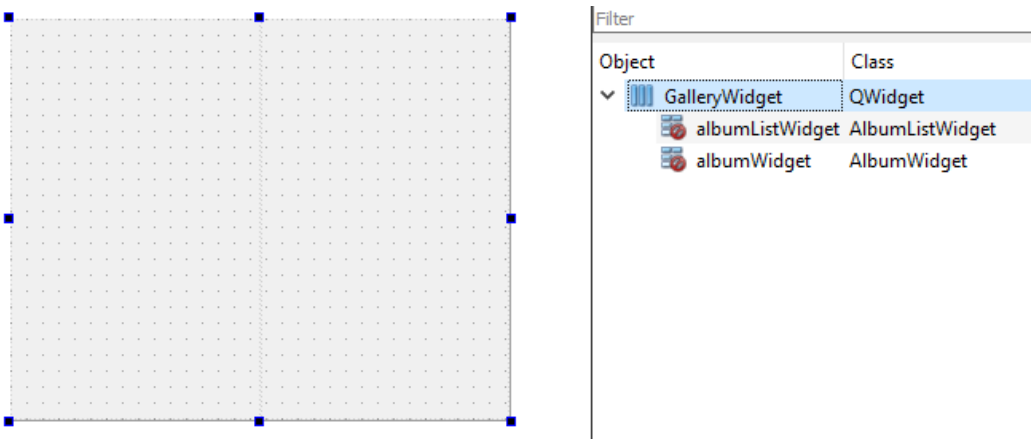
Poglavlje 7

Povezivanje korisničkog sučelja

7.1 Prikaz galerije

Završetkom izrade glavne tri klase `AlbumListWidget`, `AlbumWidget` i `PictureWidget` ostalo je još povezati ih u gotovu aplikaciju. Prvo što treba napraviti je povezati prve dvije u novu najavljenu klasu `GalleryWidget`.

Qt Designer neće imati forme `AlbumListWidget` i `AlbumWidget` među svojim `widget`-ima. Ipak, moguće je staviti ih koristeći postupak koji se zove **promocija**. Najprije je potrebno staviti dva obična `QWidget`-a, a zatim desnim klikom na pojedini i izborom opcije *"Promote widgets..."* otvoriti izbornik sa popisom svih dosad stvorenih klasa koje nasljeđuju `QWidget`. Prikladnim izborom dobiva se rezultat sa slike 7.1.



Slika 7.1: Forma klase `GalleryWidget`

Zaglavlje i implementacija same klase `GalleryWidget` nisu posebno zanimljivi.

Sastoji se od *getter*-a i *setter*-a modela za sliku i album te signala `pictureActivated` pa stoga neće biti analizirani.

7.2 Glavni prozor

Zadnja klasa koju treba pogledati je klasa glavnog prozora odnosno `MainWindow` koja je netaknuta od stvaranja pod-projekta `gallery-frontend`. Ovdje se neće raditi osobite promjene u Qt Designer-u zato što će se većina posla vršiti programibilno unutar zaglavlja i implementacije. Najprije se analizira zaglavlje koje se nalazi na slici 7.2. Ono se sastoji od konstruktora, destruktora i dvije funkcije za izmjenu prikaza.

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

public slots:
    void displayGallery();
    void displayPicture(const QModelIndex& index);

private:
    Ui::MainWindow *ui;
    GalleryWidget* mGalleryWidget;
    PictureWidget* mPictureWidget;
    QStackedWidget* mStackedWidget;
};
```

Slika 7.2: `MainWindow.h`

Konstruktor se nalazi na slici 7.3 i on je najkompleksniji dio klase. U njemu se najprije određuju modeli `AlbumModel`, `AlbumSelectionModel`, `PictureModel` i `PictureSelectionModel` koji su se pojavljivali tijekom prethodnih poglavlja. Za model slika bitno je postaviti `ThumbnailProxyModel` kao glavni model, a `PictureModel` kao njegov izvorni model.

Ostatak konstruktora je uvođenje signala za prijelaz iz jednog prikaza u drugi te dodavanja prikaza u poseban *widget* `mStackedWidget`. Zadnja linija postavlja taj *widget* kao glavni prilikom otvaranja prozora.

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow),
    mGalleryWidget(new GalleryWidget(this)),
    mPictureWidget(new PictureWidget(this)),
    mStackedWidget(new QStackedWidget(this))
{
    ui->setupUi(this);

    AlbumModel* albumModel = new AlbumModel(this);
    QItemSelectionModel* albumSelectionModel = new QItemSelectionModel
        (albumModel, this);
    mGalleryWidget->setAlbumModel(albumModel);
    mGalleryWidget->setAlbumSelectionModel(albumSelectionModel);

    PictureModel* pictureModel = new PictureModel(*albumModel, this);
    ThumbnailProxyModel* thumbnailModel = new ThumbnailProxyModel(this
        );
    thumbnailModel->setSourceModel(pictureModel);

    QItemSelectionModel* pictureSelectionModel = new
       QItemSelectionModel(thumbnailModel, this);
    mGalleryWidget->setPictureModel(thumbnailModel);
    mGalleryWidget->setPictureSelectionModel(pictureSelectionModel);
    mPictureWidget->setModel(thumbnailModel);
    mPictureWidget->setSelectionModel(pictureSelectionModel);

    connect(mGalleryWidget, &GalleryWidget::pictureActivated,
            this, &MainWindow::displayPicture);

    connect(mPictureWidget, &PictureWidget::backToGallery,
            this, &MainWindow::displayGallery);

    mStackedWidget->addWidget(mGalleryWidget);
    mStackedWidget->addWidget(mPictureWidget);
    displayGallery();

    setCentralWidget(mStackedWidget);
}
```

Slika 7.3: Konstruktor klase `MainWindow`

Preostale dvije funkcije nalaze se na slici 7.4 i one su nevjerojatno jednostavne. Sve što je potrebno napraviti je postaviti pripadni prikaz kao trenutni u objekt `mStackedWidget`. Razlog jednostavnosti je Qt-ova klasa `QStackedWidget` koja je dizajnirana za

ovakvu upotrebu. Ona naime može imati nekoliko *widget*-a, ali samo jedan može biti prikazan u isto vrijeme.

```
void MainWindow::displayGallery()
{
    mStackedWidget->setCurrentWidget(mGalleryWidget);
}

void MainWindow::displayPicture(const QModelIndex& /*index*/)
{
    mStackedWidget->setCurrentWidget(mPictureWidget);
}
```

Slika 7.4: Funkcije izmjene prikaza u `MainWindow`

7.3 Završne napomene

- Dimenzije aplikacije mogu se odrediti u dizajnu `MainWindow` forme, no dizajnirana je za početnu veličinu 1366×768.
- Veličina font-a kroz cijelu aplikaciju je 11pt, osim naslova albuma i slika koji ima font veličine 12pt, a vrsta fonta je *"MS Shell Dlg 2"*.
- Ikone su većinom dimenzija 32×32. Izuzetak su ikone na `dateWidget`-ima koje imaju ikone veličine 16×16 ili 24×24.

Sa time je aplikacija službeno gotova i spremna za korištenje na Windows, Linux i Mac-OS sustavima uz pomoć Qt-a. Kako bi aplikacija mogla biti pokrenuta neovisno o Qt-u, potrebno ju je pakirati za pojedini operacijski sustav.

Poglavlje 8

Pakiranje projekta

Cilj pakiranja projekta je dobiti samostalnu funkcionalnu aplikaciju. To se postiže tako da se povežu svi potrebni pod-projekti te biblioteke pomoću *script*-ova odnosno skripta.

Sve će skripte imati sličan slijed događaja:

1. Postavi direktorije ulaza i izlaza
2. Stvori *Makefile* datoteke
3. Izgradi projekt
4. Dohvati sve potrebne datoteke za izlaz
5. Pakiraj projekt ovisno o operacijskom sustavu
6. Spremi pakirani projekt u izlaz

U ovom će poglavlju biti stvorene skripte za Windows, Linux i MacOS. Prije svega stvara se novi direktorij imena `scripts` unutar projektnog direktorija.

8.1 Pakiranje za Windows

Skriptu za Windows bit će moguće pokrenuti pomoću naredbenog retka koristeći MinGW za kompilaciju. Qt-ov alat `windeployqt` za pakiranje pobrinut će se da aplikacija ima sve potrebne biblioteke odnosno `.dll` datoteke. Također je potrebno namjestiti varijable okruženja Windows sustava kao u sljedećem primjeru:

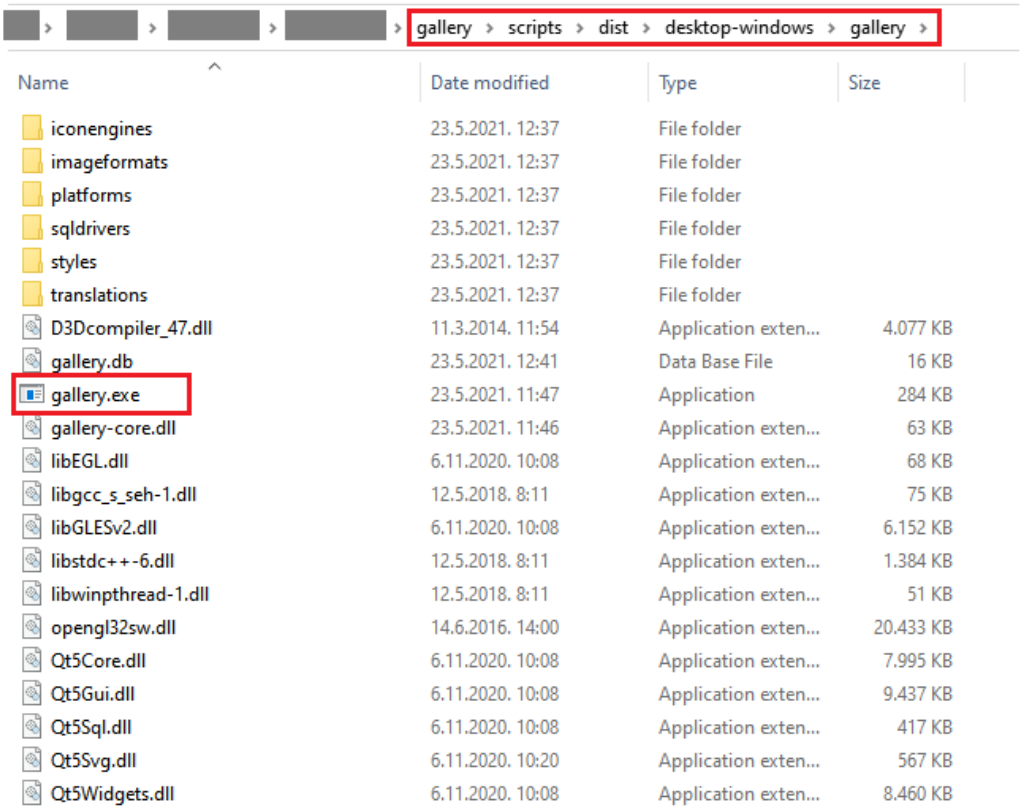
- `QTDIR - C:\Qt\5.15.2\mingw81_64`
- `MINGWROOT - C:\Qt\Tools\mingw810_64`

Nakon obavljenih priprema može se stvoriti `package-windows.bat` datoteka čiji se sadržaj vidi na slici 8.1.

```
@ECHO off

set DIST_DIR=dist\desktop-windows
set BUILD_DIR=build
set OUT_DIR=gallery

mkdir %DIST_DIR%
pushd %DIST_DIR%
mkdir %BUILD_DIR% %OUT_DIR%

pushd %BUILD_DIR%
%QTDIR%\bin\qmake.exe ^
  -spec win32-g++ ^
  "CONFIG += release" ^
  ..\..\..\..\gallery.pro

%MINGWROOT%\bin\mingw32-make.exe qmake_all

pushd gallery-core
%MINGWROOT%\bin\mingw32-make.exe && popd

pushd gallery-frontend
%MINGWROOT%\bin\mingw32-make.exe && popd

popd
copy %BUILD_DIR%\gallery-core\release\gallery-core.dll %OUT_DIR%
copy %BUILD_DIR%\gallery-frontend\release\gallery.exe %OUT_DIR%
%QTDIR%\bin\windeployqt %OUT_DIR%\gallery.exe %OUT_DIR%\gallery-core.
   dll

popd
```

Slika 8.1: Skripta za pakiranje u Windows

Izvođenjem skripte na Windows sustavu dobiva se gotova samostalna aplikacija u direktoriju `gallery\scripts\dist\desktop-windows\gallery` čiji se sadržaj vidi na slici 8.2.



Slika 8.2: Direktorij Windows aplikacije stvoren pakiranjem

8.2 Pakiranje za Linux

Proces pakiranja na Linux sustavima malo je kompliciraniji zato što postoji više formata pakiranja (DEB, ebuild, RPM, pkg.tar.xz,...) koji su pojedinačno podržani u određenim distribucijama dok u ostalima nisu. Stoga bi korištenje postupka kao kod pakiranja za Windows sustav bilo znato teže.

Međutim, postoji i alternativno riješenje koje se zove **AppImage**. Radi se o posebnom formatu koji u sebi sadrži aplikaciju zajedno sa svim njoj potrebnim bibliotekama. Njenim pokretanjem stvara se virtualna slika aplikacije koja se može slobodno koristiti. Ovim će se rješenjem pokriti sve Linux distribucije. Proces pakiranja ima dva koraka:

- Dohvaćanje potrebnih biblioteka
- Pakiranje galerije i njenih biblioteka u AppImage formatu

Alat koji će se koristiti za pakiranje zove se `linuxdeployqt` i on je trenutno službeni

alat za pakiranje Linux aplikacija u AppImage format, međutim za razliku od windowsdeployqt on ne dolazi uz Qt nego ga je potrebno preuzeti sa probonopd/linuxdeployqt Github repozitorija.

Prije pisanje skripte moraju se dodati nužne varijable okruženja sustava:

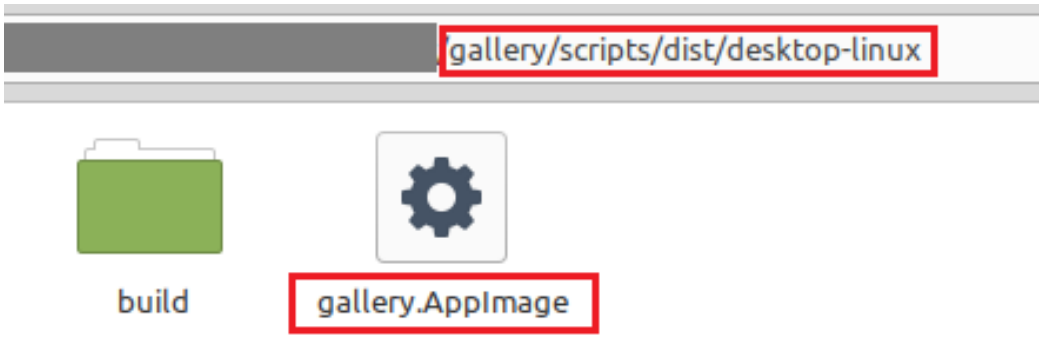
- QTDIR - $HOME:\Qt\5.15.2\gcc_64

Zatim u direktoriju scripts treba stvoriti novu datoteku package-linux.sh čiji je sadržaj prikazan na slici 8.3.

```
DIST_DIR=dist/desktop-linux
BUILD_DIR=build

mkdir -p $DIST_DIR
cd $DIST_DIR
mkdir -p $BUILD_DIR

pushd $BUILD_DIR
$QTDIR/bin/qmake \
    -spec linux-g++ \
    "CONFIG += release" \
    ../../../../gallery.pro

make qmake_all
pushd gallery-core && make ; popd
pushd gallery-frontend && make ; popd
popd

export QT_PLUGIN_PATH=$QTDIR/plugins/
export LD_LIBRARY_PATH=$QTDIR/lib:$(pwd)/build/gallery-core

linuxdeployqt \
    build/gallery-frontent/gallery \
    -appimage

mv build/gallery.AppImage .
```

Slika 8.3: Skripta za pakiranje u Linux-u

Pokretanjem skripte package-linux.sh dobiva se AppImage izvršna datoteka kao što se vidi na slici 8.4.



Slika 8.4: AppImage datoteka Linux aplikacije stvorena pakiranjem

8.3 Pakiranje za Mac OS X

Pakiranjem za Mac OS X sustav cilj je dobiti jednu `.dmg` datoteku koja je ekvivalentna `.iso` formatu na Windows sustavu. Ovo je zapravo vrlo slično pakiranju za Linux što nije sasvim neobično s obzirom da je `linuxdeployqt` stvoren po uzoru na Qt-ov alat `macdeployqt` koji će se ovdje koristiti. Prije pisanja skripte treba postaviti Qt varijablu okruženja sustava:

- `QTDIR - $HOME:\Qt\5.15.2\clang_64`

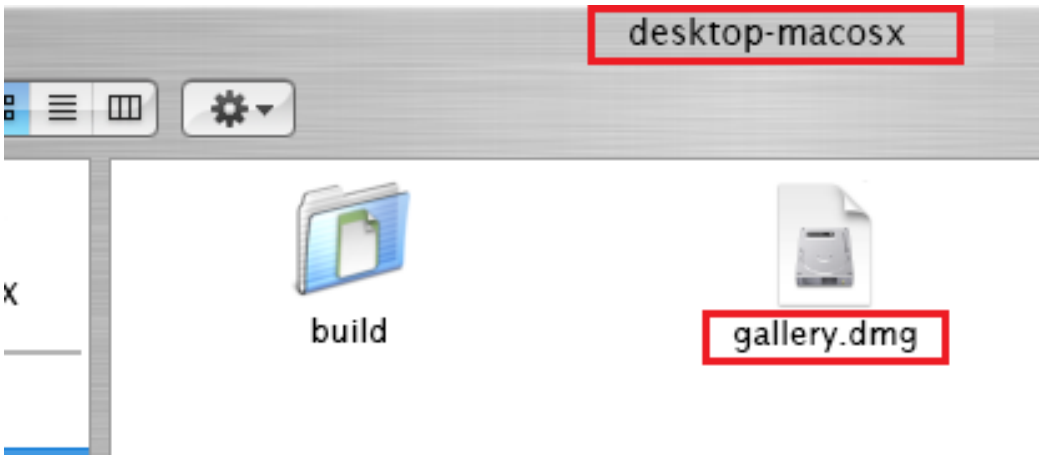
U direktoriju `scripts` ponovno se stvara nova datoteka `package-macosx.sh` koja je prikazana na slici 8.5.

```
DIST_DIR=dist/desktop-macosx
BUILD_DIR=build

mkdir -p $DIST_DIR && cd $DIST_DIR
mkdir -p $BUILD_DIR

pushd $BUILD_DIR
$QTDIR/bin/qmake \
  -spec macx-clang \
  "CONFIG += release x86_64" \
  ../../../../gallery.pro

make qmake_all
pushd gallery-core && make ; popd
pushd gallery-frontent && make ; popd
```

```
cp gallery-core/*.dylib \
    gallery-frontend/gallery.app/Contents/Frameworks/

install_name_tool -change \
  libgallery-core.1.dylib \
  @rpath/libgallery-core.1.dylib \
  gallery-frontend/gallery.app/Contents/MacOS/gallery
popd

$QTDIR/bin/macdeployqt \
    build/gallery-frontend/gallery.app \
    -dmg

mv build/gallery-frontend/gallery.dmg .
```

Slika 8.5: Skripta za pakiranje u Mac OS X-u

Pokretanjem skripte `package-macosx.sh` dobiva se `.dmg` izvršna datoteka kao što se vidi na slici 8.6.



Slika 8.6: .dmg datoteka Mac OS X aplikacije stvorena pakiranjem

Poglavlje 9

Zaključak

U ovom je radu cilj bio pokazati mogućnosti programskog okvira Qt u izradi multiplatformske *desktop* aplikacije sa korisničkim sučeljem u programskom jeziku C++ na konkretnom primjeru galerije slika uz reference na rad autora Lazar i Penea (2016).

Taj je cilj postignut zahvaljujući Qt-ovim opsežnom i fleksibilnom skupom alata.

Mogućnosti qmake-a i .pro datoteka nisu u potpunosti istražene u ovom radu, no većinu osnovnih radnji automatski obavlja Qt što čini osnovno povezivanje pod-projekata vrlo jednostavno. Najveći doprinos u izradi aplikacije bila je Qt-ova Model/View arhitektura u koju su uključene korisne Qt-ove klase koje korisnik Qt-a može preraditi po volji pomoću nasljeđivanja klasa. U obradi podataka pomaže dodatak za korištenje SQLITE baze podataka. Izrada grafičkog sučelja znatno je olakšana u Qt-ovom alatu Qt Designer te Qt-specifičnim klasama za grafičke objekte, a njihova međusobna komunikacija je jednostavna i pregledna korištenjem signala i utora.

U daljnjem radu na aplikaciji korisno je izraditi mobilnu verziju aplikacije koja bi pokrila tržište mobilnih aplikacija koje je trenutno možda i konkurentnije od tržišta *desktop* aplikacija. Za tu bi svrhu i dalje bio iskoristiv modul za obradu podataka, dok bi grafičko sučelje bilo potrebno konstruirati od početka. Ostali dodaci na aplikaciju otvoreni su na potrebe korisnika aplikacije, no postojeći kod nije teško preraditi i nadograditi zahvaljujući dobroj organizaciji koda po uzoru na [3].

Zbog svog velikog raspona Qt specifičnih klasa i alata, Qt može biti zastrašujuć za početnike ukoliko nemaju potrebne smjernice. S druge strane, on omogućava iskusnom korisniku Qt-a stvaranje vrlo kompleksnih i elegantnih rješenja. Iako je prvo izdanje objavljeno davne 1997. godine, Qt je i dalje popularan u C++ svijetu zbog svoje kontinuirane nadogradnje te je koristan za učenje na područjima razvoja aplikacijske arhitekture i objektno orijentiranog programiranja. To ga zato čini vrlo aktualanim i zanimljivim programskim okvirom kojeg se svakako isplati istražiti.

Literatura

[1] *Qt 5 dokumentacija*, The Qt Company, https://doc.qt.io/.

[2] Lazar Guillaume, *Mastering Qt 5 GitHub repository*, https://github.com/PacktPublishing/Mastering-Qt-5.

[3] Lazar Guillaume i Penea Robin, *Mastering Qt 5*, Packt, 2016.

[4] Robert Manger, *Softversko inženjerstvo*, Element, 2016.

Sažetak

Ovaj rad bavi se programskim okvirom Qt i njegovom uporabom u izradi multiplatformske C++ *desktop* aplikacije sa grafičkim sučeljem na konkretnom primjeru galerije slika za Windows, Linux i Mac.

U prvom se poglavlju obrađuju osnove Qt klasa i izrade korisničkog sučelja općenito te korištenje alata Qt Designer. Ostatak je rada sama izrada aplikacije podijeljena u tri glavna dijela - obrada podataka, izrada korisničkog sučelja te pakiranje aplikacije. Ovdje se također daje pregled u Qt-ovu Model/View arhitekturu. Za segment obrade podataka stvara se SQLITE baza podataka koja se zatim povezuje sa podatkovnim klasama za glavna dva objekta u aplikaciji galerije slika - album i slika. Korisničko sučelje je najopsežniji dio rada te pokriva dizajniranje formi za pregled albuma, slika u albumu te pojedine slike, a zatim njihovo povezivanje sa prethodno stvorenom logikom iz obrade podataka. Na koncu se stvaraju skripte kojima se provodi pakiranje aplikacije za platforme Windows, Linux i Mac OS X.

Summary

This paper deals with the Qt framework and how it can be used in the development of a multiplatform C ++ *desktop* application with a graphical interface on the example of an image gallery for Windows, Linux and Mac.

The first chapter discusses the basics of Qt classes and user interface design in general and using the Qt Designer tool. The rest of the work is the development of the application itself whcih is divided into three main parts - data processing, user interface development and application packaging. An overview of Qt's Model/View architecture is also provided here. For the data processing segment, an SQLITE database is created which is then linked to the data classes for the two main objects in the image gallery application - album and image. The user interface is the most comprehensive part of the work and covers the design of forms for viewing albums, images in the album and individual images, and then connecting them with previously created logic from data processing. Finally, scripts are created to package the application for Windows, Linux, and Mac OS X.

Životopis

Rođen sam 19. svibnja 1997. godine u Osijeku. Nakon osnovne škole koju sam pohađao u Poreču, 2016. godine završavam smijer opće gimnazije u Srednjoj Školi Mate Balote Poreč. Preddiplomski studij matematike upisao sam iste godine na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta Sveučilišta u Zagrebu. Tri godine nakon na istom mjestu upisao sam diplomski studij Računarstvo i matematika.

Za vrijeme studiranja radio sam sezonski kao redar na bazenu u hotelu, a od ožujka 2021. godine zaposlen sam se kao *software developer* u Ericsson-u.

Neka od mojih postignuća su četiri prva plasmana na županijskim natjecanjima iz matematike u osnovnoj i srednjoj školi te prvo mjesto na državnom natjecanju iz matematike 2014. godine.