Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра «Компьютерные интеллектуальные технологии»

К У Р С О В А Я Р А Б О Т А

Каталогизация медиа-файлов

по дисциплине «Технологии объектно-ориентированного программирования»

Выполнил
студент гр.23536/2 Е.Ю. Шолохова

Руководитель К. А. Туральчук
«___» _____ 201__ г.

Санкт-Петербург

2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Данная работа посвящена изучению этапов разработки программного обеспечения. Основная цель - показать умение применять объектно-ориентированный подход и использовать различный инструментарий в процессе разработки на примере программы-каталогизатора медиа-файлов. Исходя из поставленной цели, можно сформулировать следующие задачи:

- проанализировать предметную область, выбрать средства разработки
- декомпозировать задачу
- реализовать основной функционал программы и протестировать его

Отчет содержит в себе как теоретические сведения, отобранные и систематизированные в процессе работы, так и подробное описание реализации программы, начиная с декомпозиции и заканчивая замечаниями по итогам работы.

1. ОРГАНИЗАЦИЯ ХРАНЕНИЯ ДАННЫХ. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1. Постановка задачи

Задача, которая ставится перед нами в данной работе, - создание приложения для каталогизации медиа-файлов. Под медиа-файлами в данном случае будем понимать аудио-, видео-файлы – именно с этими типами файлов должно работать наше приложение. Поиск файлов должен осуществляться в выбранной пользователем директории и ее поддиректориях. В нем также должна быть предусмотрена возможность копировании файлов в указанную директорию в каталогизированном в виде, т.е. рассортированными по тем каталогам, которые для них определил пользователь.

1.2. Анализ предметной области

Программа должна производить поиск и отбор файлов по некоторому критерию. Обратимся к теории хранения данных, и определим для себя понятия файл и файловая система.

Вся информация в компьютере хранится в долговременной памяти в виде файлов. Таким образом, файл - это именованная область памяти на каком-либо носителе информации. Имя файла состоит из двух частей, разделенных точкой: собственно имя файла и расширение, определяющее его тип (программа, данные и т.д.).

Порядок хранения файлов на диске определяется установленной файловой системой. Файловая система - это система хранения файлов и организации каталогов. Каталог (папка, директория) - это, с одной стороны, набор файлов, сгруппированных пользователем по некоторому принципу, с другой же, он сам по себе является файлом и содержит системную информацию о группе файлов, его составляющих. Каталоги могут образовывать иерархическую структуру за счет того, что каталог более низкого уровня может входить в каталог более высокого уровня. Если на диске хранятся сотни и тысячи файлов, то для удобства поиска файлы организуются в многоуровневую иерархическую файловую систему, которая имеет «древовидную» структуру («дерево каталогов»).

Для того чтобы найти файл в иерархической файловой структуре необходимо указать путь к файлу. Путь к файлу - это последовательность из имен вложенных друг в друга каталогов, в последнем из которых находится нужный файл, разделённых символом «\». Корневой каталог - начальный каталог в структуре каталогов. Путь к файлу от корневого каталога называют абсолютным путём, а от текущего — относительным путём. Путь к каталогу файла и имя файла, разделённые «\», перед которыми указано имя диска, представляет собой полное имя файла.

1.3. Описание работы программы

Программа представляет из себя оконное приложение с удобным и функциональным пользовательским интерфейсом.

Главное окно программы поделено на 2 части. Справа - таблица, предназначенная для отображения информации о найденных файлах. Слева – список уж созданных пользователем каталогов. Над этими элементами расположено поле для ввода с надписью Path to the directory. В начале работы пользователь должен ввести в это поле путь до папки, в которой будет производиться поиск файлов его аудио-/видео- коллекции. Рядом расположены кнопки Browse и Search. Первая позволяет задать путь через файловый обозреватель, а вторая - провести поиск в указанной папке. Поиск можно ограничить, выбрав файлы только с определенными расширениями. Для этого достаточно напечатать эти расширения через разделитель в строке Extensions. Слева от нее кнопка Filters, открывающая доступ к функциям фильтрации найденных файлов. В нижней части главного окна расположены 2 кнопки – AddToDir и Generate Catalog.

Для добавления файлов в каталог, их нужно сначала пометить галочками, а потом нажать AddToDir. Во всплывающем окне можно будет указать название каталога. После того как пользователь закончил сортировку своих файлов, он может либо сгенерировать свой библиотеку медиа-файлов, либо просто сохранить информацию о сортировке, чтобы продолжить позднее. Кнопка Generate Catalog отвечает за создание уже каталогизированной библиотеки медиа-файлов. При нажатии на нее программа сначала запрашивает у пользователя путь для генерации каталога и имя каталога. Далее происходит проверка наличия свободного места на указанном диске и копирование файлов.

1.4. Выбор средств и методов реализации (ООП+CPP+MVC+QT)

1.4.1. Применение объектно-ориентированного подхода

В разработке программы мы прежде всего будем полагаться на объектно-ориентированный подход Именно его используют большинство современных программ. Суть данного подхода заключается в представлении программного кода, как системы взаимодействующих друг с другом объектов. Каждый объект является экземпляром некоторого класса, где класс - самостоятельная сущность, выделяемая программистом при декомпозиции задачи. Классы могут как состоять в иерархии, так и быть независимы друг от друга или дружественными друг другу. Иерархи подразумевает, что есть более общие по смыслу классы (они стоят вверху иерархии), а есть более конкретизированные (они наследуют свойства от более общих и добавляют свои свойства, как бы уточняя характеристики описываемой сущности). Связь между экземплярами классов осуществляется с помощью методов (действий, доступных каждому классу). На деле, метода мало чем отличаются от функций, а свойства от переменных. Но такая организация программного кода

позволяет сосредоточиться на общей логике работы программы, не отвлекаясь на внутреннее устройство каждого отдельно взятого объекта.

1.4.2. C++

Теперь для того, чтобы перейти к написанию кода, нам необходимо определиться с языком программирования. Остановим свой выбор на языке C++. На заре появления ООП язык C++ считался языком программирования высокого уровня, но сейчас его причисляют скорее к языкам среднего уровня. Благодаря своей гибкости, высокой производительности, поддержке объектно-ориентированного программирования, наличию библиотеки STL, данный язык находит самое широкое применение. Но главные сферы его использования это, безусловно, системное программирование, высоконагруженные сервера, встраиваемые системы, а также кроссплатформенные приложения (при использовании фреймворков; например, Qt, wxWidgets, GTK) и разработка игр (скрипты для Unreal Engine). C++, как язык, не стоит на месте, его активно развивают. За последние 10 лет вышло сразу несколько новых стандартов языка - C++11, C++14 и C++17. Со всеми нововведениями C++ теперь не уступает самым продвинутым языкам программирования.

1.4.3. MVC

Помимо объектно-ориентированного подхода, в современной теории разработки существуют и другие стандартные методы, которые помогают проектировать ваше приложение – например, различные шаблоны или паттерны.

Один из них – MVC - мы и попытаемся внедрить в нашу программу. MVC — это архитектурный шаблон, который описывает способ построения структуры приложения (расшифровывается как Model-View-Controller или Модель-Представление-Контроллер). Основная суть этого способа заключается в отделении программной логики от пользовательского интерфейса и от способа представления информации пользователю. Все эти 3 компонента в архитектуре MVC независимы, их код не пересекается и чаще всего можно четко выделить, какая часть программы (какие классы) за что отвечает.

MVC один из базовых архитектурных шаблонов и наиболее часто применяется именно в случае оконных приложений. Построение программы на основе MVC-архитектуры непосредственно связано с декомпозицией задачи. Поэтому более подробно мы рассмотрим данный вопрос в соответствующем параграфе.

1.4.4. Фреймворк QT

Для создания приложения с графическим интерфейсом удобно использовать фреймворк или библиотеку, заточенную под такие цели. В качестве такого средства мы будем использовать фреймворк Qt. Qt - это кроссплатформенный инструмент для создания прикладных программ.

Данный фреймворк предоставляет множество возможностей и часто используется именно в связке с C++. Его единственный недостаток – не самая лучшая оптимизация потребляемых программой ресурсов- с лихвой перекрывается множеством достоинств. Библиотека фреймворка содержит огромной количество готовых и классов, а также абстрактных классов, от которых можно наследовать свои, если потребности вашей задачи выходят за рамки стандартных инструментов библиотеки. Qt, что не маловажно, имеет прекрасную документацию с большим количеством примеров. Разработчики поддерживают и активно развивают его. А, благодаря обширному коммьюнити, в сети интернет можно найти решение практически любо проблемы, возникшей в процессе разработки.

У Qt есть также Qt Designer – case-средство для проектирования графического интерфейса. Но для лучшего понимания устройства фреймворка, в частности, того как в нем реализован объектно-ориентированный подход, мы не будем использовать Qt Designer.

2. РАЗРАБОТКА ПРОГРАММЫ ДЛЯ КАТАЛОГИЗАЦИИ

2.1. Декомпозиция задачи

В процессе декомпозиции были выделены подзадачи, которые необходимо выполнить для достижения результата, и определена их последовательность (см.рисунок).

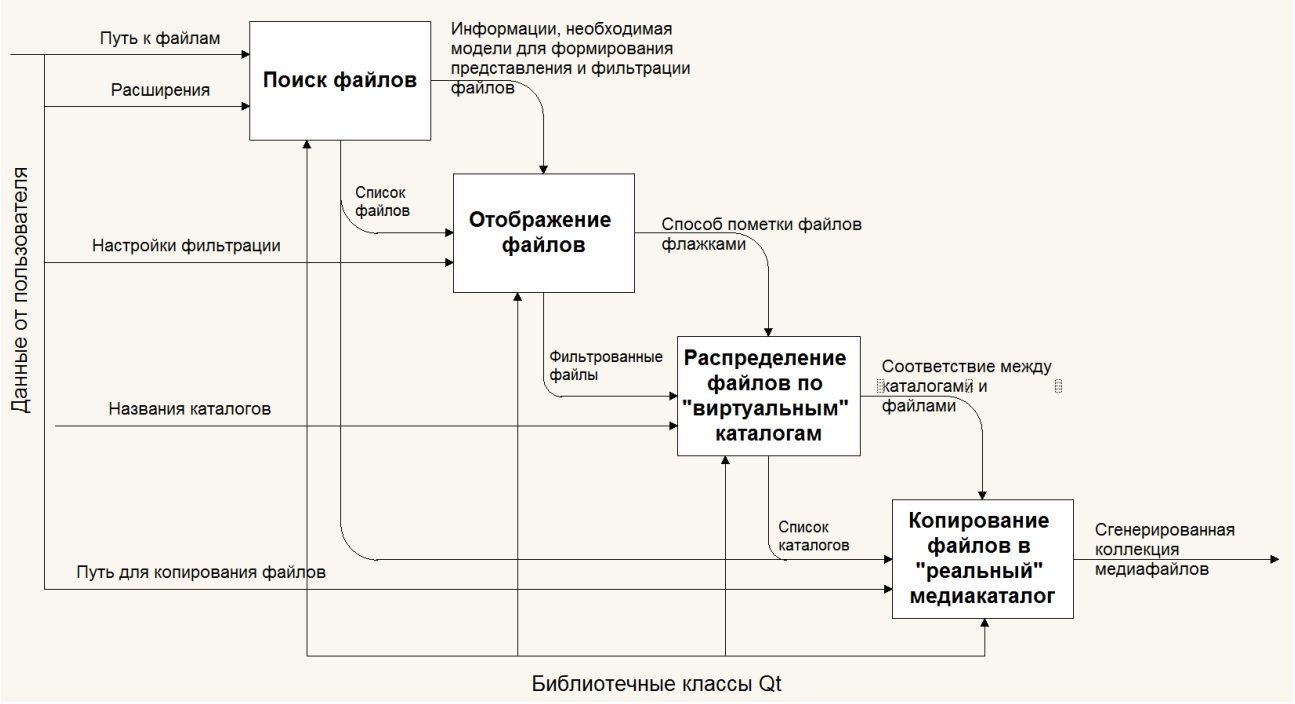


Рисунок 2.1. Диаграмма IDEF0

Была проведена также объектная декомпозиции проектируемой программы, в ходе которой были определены классы, объекты которых нам потребуются при написании программной логики.

Диаграмма классов UML

Диаграмма последовательностей UML

2.2. Программирование графического интерфейса

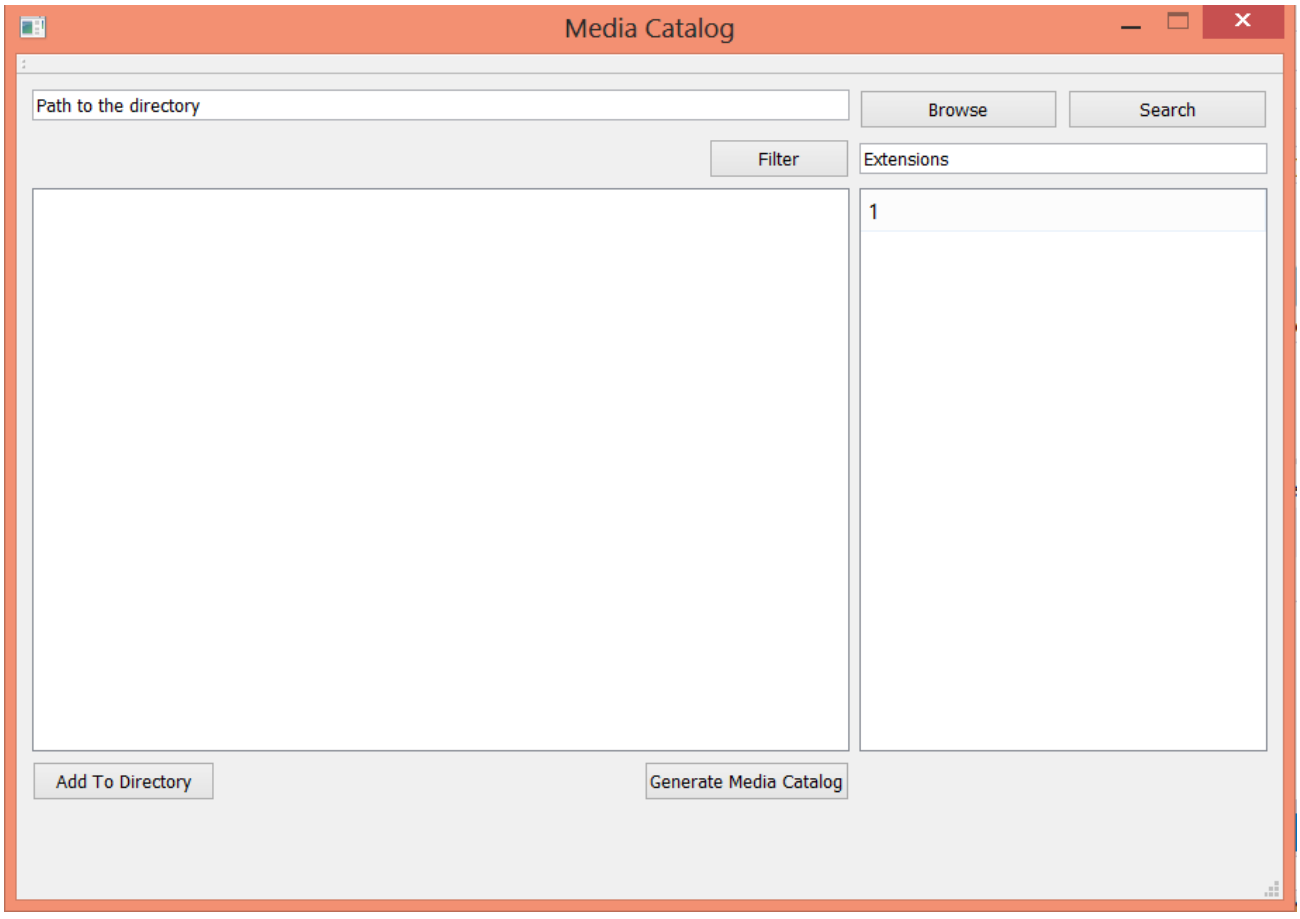


Рисунок 2.2. Интерфейс главного окна программы

Все элементы графического интерфейса были запрограммированы с помощью специальных классов Qt (т.н. виджетов). Все эти классы являются наследниками класса QWidget. Для их правильного расположения на форме главного окна использовались классы группировки виджетов (т.н. слои или layouts) - QGridLayout, QHBoxLayout и QVBoxLayout.

В интерфейс программы были добавлены следующие элементы:

Кнопки (класс QPushButton) butBrowse; butSearch; butAddToDir; butGenerateCat; butFilter;

Поля ввода (класс QLineEdit) editPath и editExt;

Табличное представление (класс QTableView) fileTable;

Древовидный список (класс QTreeWidget) treeDirs;

Следующие классы были унаследованы от стандартных классов Qt. Для этих классов необходимо было написать собственную реализацию, чтобы учесть потребности нашего приложения.

класс FileDataModel model;

класс CheckBoxDelegate chbDelegate;

Классы и структуры для хранения данных:

класс MediaRecord

класс Catalog catalog

```
struct fileTodir
{       MediaRecord rec;
        QString dir;
};
```

Catalog содержит массив QVector типа < fileTodir >.

3.1. Выборка и отображение медиа-файлов

Первый этап создания программы – поиск медиа-файлов и их отображение в таблице.

a) Способ доступа к файловой системе, директориям и файлам
b) Выбор алгоритма поиска
c) Способ представление информации о файлах

Разберем подробно каждый пункт.

a) Доступ к файловой системе, директориям и файлам

Здесь удобно воспользоваться классами QFileDialog, QDir и QFileInfo.

QFileDialog позволит нам получить доступ к файловой системе. QDir – класс, с помощью которого можно узнать содержимое директории - содержащиеся в ней файлы и поддиректории.

QFileInfo – класс, дающий доступ к информации о файле.

Сигнал clicked() кнопки Browse соединяем со слотом главного окна BrowseClicked(). В нем вызываем статический метод класса QFileDialog для открытия файлового браузера. Получаем на выходе путь к файлам. Так мы гарантируем то, что папка на самом деле существует. Дополнительная проверка теперь не требуется. Путь отображается в поле editPath. Сигнал clicked() кнопки Search соединяем со слотом SearchClicked(). Берем значение из поля editPath и посылаем его через emit pathEntered() в catalog, в слот FillCatalog().FillCatalog() содержит алгоритм обхода директорий. Получая сигнал pathEntered() вместе с путем то файлов, он открывает соотвествующую директорию и начинает обход. Когда каталог заполнен посылается сигнал catalogIsReady().

b) Выбор алгоритма поиска

Для обхода директории и всех поддиректорий есть всего 2 способа:

- Рекурсивный обход директорий и

- Итерационный обход (с запоминанием точки входа)

Мы будем использовать первый вариант, т.к. итерационный подход в данном случае не дает никаких преимуществ, ни по времени, ни по потребляемой памяти, а реализовать его существенно сложнее.

В методе FillCatalog(), получающем на вход dirPath, мы создаем объект класса QDir. Передаем в него dirPath. Теперь в этом объекте содержится вся информация о директории. В том числе, EntryInfoList, из которого можно получить список всех файлов и поддиректорий в этой директории. Проходим в цикле по этому списку, добавляя файлы в массив files. А для тех элементов списка, которые сами являются диретоориями, вызываем метод FillCatalog(). Когда обход закончен посылаем сигнал в модель FileDataModel.

c) Предоставление информации о файлах пользователю

Qt имеет несколько вариантов реализации таблицы – это классы QTableWidget и QTableView. Решено было использовать QTableView с QStandardTableItem, т.к. такое сочетание хоть и не сильно отличается от стандартного QTableWidget, но имеет гораздо больше возможностей для масштабирования. В последствии эта связка была заменена на полноценный QTableView с собственной моделью, наследованной от QAbstarctTableModel. При наличии собственной модели можно полностью отделаить представление от данных. В дальнейшем, когда понадобиться добавить представлению новые функции (типа сортировки по разным полям, выборки, поиска и т.п.), достаточно будет описать в MyModel. QTableView + модель так же являются наглядным примером реализации MVC-архитектуры в Qt.

Далее необходимо решить каким методом модель будет извлекать данные из Catalog. Мы будем использовать указатель на экземпляр класса Catalog. При завершении поиска Catalog посылает сигнал IsReady() и указатель на самого себя в слот модели dataSource(). Так как хранящиеся в классе Catalog данные инкапсулированы, мы не можем получить их напрямую. Поэтому добавляем ему метод GetNextFileData(), который будет получать индекс, а в ответ выдавать информацию о файле в виде QStringList. В модели также необходимо реимплементировать метод data(), который и будет вызывать GetNextFileData() у Catalog, а потом передавать информацию в таблицу.

3.2. Сортировка файлов по директориям

Следующий этап

Разобьём его на такие под этапы:

a) Фильтрация

b) Выбор файлов для сортировки

c) Создание каталогов

d) Обновление данных в табличном представлении

a) Фильтрация

Для отбора и сортировки данных в таблице используем еще один встроенный класс фреймворка – QSortFilterProxyModel. С помощью метода setSourceModel() устанавливаем нашу модель FileDataModel в качестве источника, данные которого будут фильтроваться. И переключаем свойство setSortingEnabled() нашего QTableView. Класс QSortFilterProxyModel содержит в себе множество способов фильтрации элементов в таблице: setFilterWildcard, setFilterFixedString, setFilterRegularExpression, и др.

b) Выбор файлов для сортировки

Для того, чтобы пользователь мог пометить файлы, которые он хочет положить в каталог, напротив каждого файла в таблице добавим элемент QCheckBox. При установке флажка в чэкбокс соответствующий ему файл будет добавлен в список файлов на каталогизацию. Этот список мы будем хранить в модели, и при нажатии кнопки AddToDir передавать в Catalog.

Чтобы наше табличное представление реагировало на действия пользователя с чэкбоксом, для него нужно написать так называемый делегат, т.е. класс, наследованный от QItemDelegate.

Делегаты в Qt – это специальные классы, которые обеспечивают взаимодействие между моделью и представлением. Они являются ключевым элементом в управлении режимами отображения или редактирования ячеек таблицы QTableView. Для корректной работы делегата в модели дополнительно перегрузим метод setData().

c) Создание каталогов

По кнопке AddToDir нам нужно вызывать диалоговое окно, в котором вводится имя каталога.

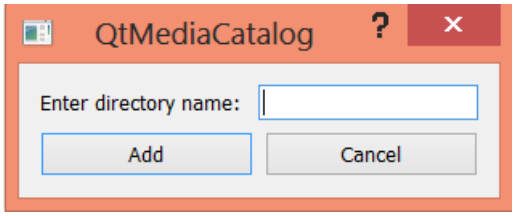


Рисунок 2.3. Диалоговое окно для ввода названия каталога

Для этой цели в Qt есть специальный класс – QDialog. В нашем случае используем наследованный от него QInputDialog.

Соединяем сигнал clicked() кнопки AddToDir со слотом главного окна AddClicked(). В нем создаем переменную dirName и присваиваем ей результат вызова статического метода QInputDialog::GetText(). Передаем имя каталога сигналом dirEntered() в слот organizeFiles() класса

Catalog. Записываем имя каталога в список сформированных каталогов dirs, в массиве files типа fileTodir всем помеченным файлам из списка приписываем каталог и посылаем сигнал organized().

d) Обновление данных в табличном представлении

Теперь из таблицы нужно убрать обработанные файлы. Создаем в модели приватный слот __ и соединяем его с сигналом organized(), идущим от Catalog.

В слоте очищаем список помеченных файлов и вызываем перегруженный метод removeRows(), который в свою очередь, вызовет метод beginRemoveRows(), потом удаление соответствующих строк, и в конце endRemoveRows(). Добавление этих 2 методов является необходимо для перегрузки метода removeRows().

3.3. Генерация медиатеки (медиакаталога)

a) Копирование

Копирование файлов в новые директории будет осуществляться с помощью метода Copy() класса MediaRecord, который, по сути, является всего лишь удобной оберткой для одноименного метода в классе QFile.

b) Сохранение информации о сортировке

В файл для бэкапа будет записываться основная информация из объекта catalog: директория, в которой производился поиск, массив files(), список каталогов dirs. Этих данных достаточно для восстановления сеанса. Но на случай изменений в файловой системе (переносе/переименовании файлов/папок) также, безусловно, потребуется возможность проверки загруженных из бэкапа данных.

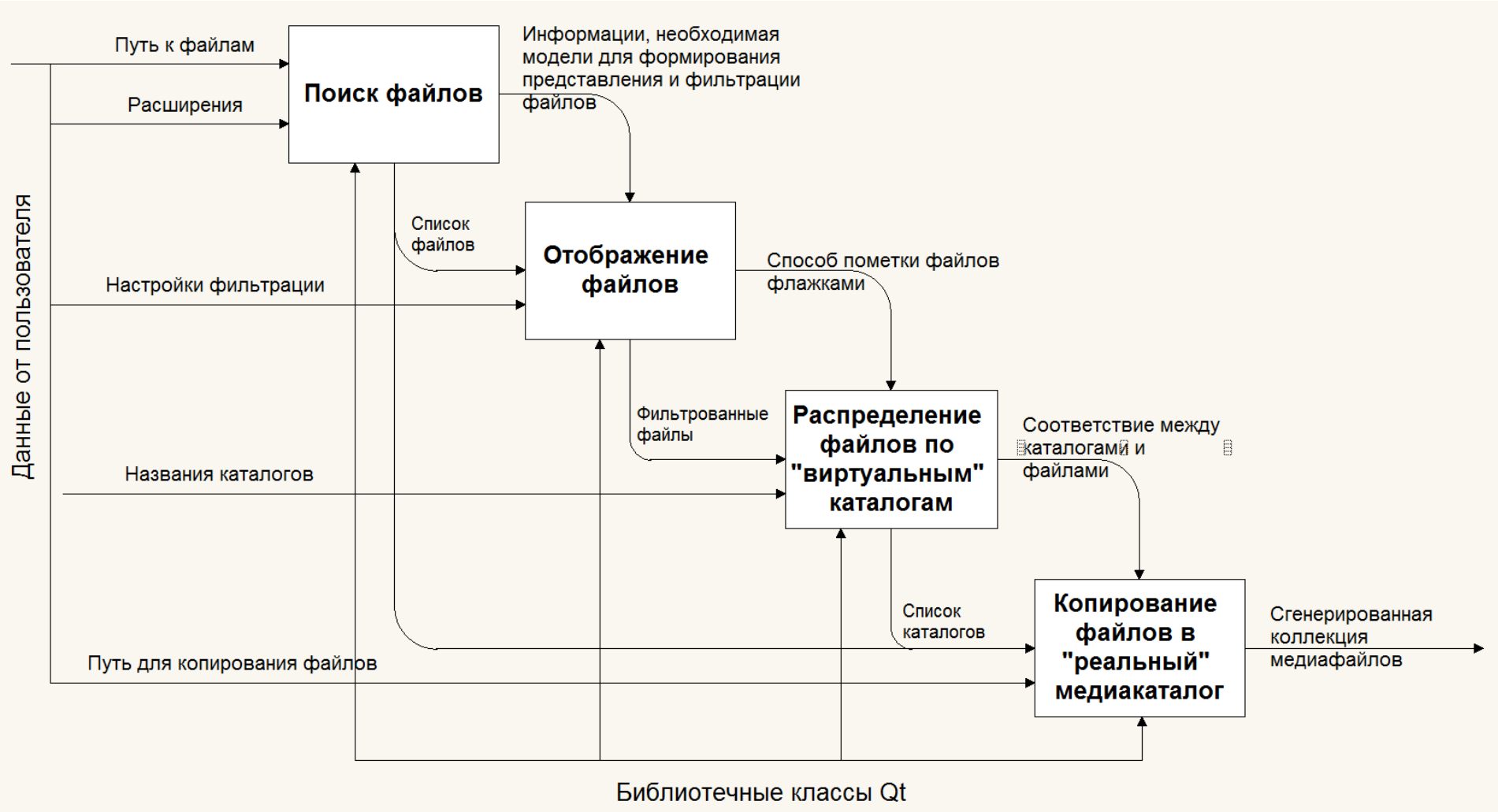
ЗАКЛЮЧЕНИЕ

Результатом данной работы стало создание приложения для каталогизации медиа-файлов. В процессе мы научились декомпозировать задачу и применять различные современные подходы и инструменты для разработки прикладного программного обеспечения.

СПИСОК ЛИТЕРАТУРЫ

1. C/C++. Процедурное и объектно-ориентированное программирование. Учебник для вузов. Стандарт 3-го поколения.Павловская Т.А. СПБ: Питер, 2015 г. - 495 с.
2. C++ Бархатный путь. Марченко А.Л. - 2-е изд. Издательство: М.: Горячая линия — телеком, 2005 г. - 399 с.
3. Qt 5.3. Профессиональное программирование на C++. Макс Шлее. BHV, 2015 г. - 928 с.
4. Лекции по технологии ООП с дистанционного курса на сайте http://dl.spbstu.ru/
5. Документация QtFramework - http://doc.qt.io/
6. Раздел по программированию на Qt форума http://stackoverflow.com/

ПРИЛОЖЕНИЕ 1


Путь к файлам
Расширения
Поиск файлов
Информации, необходимая модели для формирования представления и фильтрации файлов
Данные от пользователя
Список файлов
Отображение файлов
Способ пометки файлов флажками
Настройки фильтрации
Фильтрованные файлы
Распределение файлов по "виртуальным" каталогам
Соответствие между каталогами и файлами
Названия каталогов
Список каталогов
Копирование файлов в "реальный" медиакаталог
Сгенерированная коллекция медиафайлов
Путь для копирования файлов
Библиотечные классы Qt

ПРИЛОЖЕНИЕ 2

```
#include <QObject>
#include <QFileInfo>
#include <QFile>
#include <QDir>
#pragma once

class MediaRecord
{
private:
	QFileInfo fileInfo;

public:
	MediaRecord();
	MediaRecord(QString filePath);
	MediaRecord(QFileInfo fInfo);
	//MediaRecord(const MediaRecord& rec);
	~MediaRecord();
	void CopyFile(QString newPath);
	QStringList ToString();//returns array with file info? char*[]
};
#include "MediaRecord.h"
#include <qDebug>
#pragma once

MediaRecord::MediaRecord() {}
MediaRecord::MediaRecord(QString filePath)
{
	fileInfo.setFile((filePath));
}
MediaRecord::MediaRecord(QFileInfo fInfo)
{
	fileInfo = fInfo;
}
/*MediaRecord::MediaRecord(const MediaRecord & rec)
{
	qDebug() << "In copy constructor";
	fileInfo = rec.fileInfo;
}*/
MediaRecord::~MediaRecord() {}

void MediaRecord::CopyFile(QString newPath)
{
}

QStringList MediaRecord::ToString()
{
	return (QStringList()<<fileInfo.baseName()
		<< fileInfo.completeSuffix())
		<< QString::number(fileInfo.size());
}
```

```
#include <QObject>
#include <QVector>
#include <QList>
#include "MediaRecord.h"
#pragma once

class Catalog : public QObject
{
	Q_OBJECT
private:
	QString path;
	struct fileTodir {
		MediaRecord rec;
		QString dir;
	};
	QVector <fileTodir> files;
	QStringList dirs;
	int n;

	int it;
public:
	Catalog(QObject *parent = Q_NULLPTR);
	~Catalog();
	int GetSize() { return n; };
	QStringList GetNextFileData(int i);
public slots:
	void FillCatalog(QString path);
	void OrganizeFiles(QString dirName, QList<int> chfiles);
	void GenerateCatalog(QString newPath);

signals:
	void catalogIsReady();
	void dataChanged();
	void organized();
};

#include "Catalog.h"
#include <QDir>
#include <qDebug>
#pragma once

Catalog::Catalog(QObject *parent)
	: QObject(parent)
{
	n = 0;
	it = 0;
}

Catalog::~Catalog()
{
```

```
}

void Catalog::FillCatalog(QString path)
{

	QDir dir(path);qDebug() << "FILLING CATALOG"<< dir.absolutePath();
	for (QFileInfo i : dir.entryInfoList()) {

		if (i.absolutePath() == dir.absolutePath())
		{
			if (i.isDir())
			{
				FillCatalog(i.absolutePath()+"/"+i.baseName());
			}
			else
			{
				fileTodir tempstruct{ MediaRecord(i), "" };
				files.push_back(tempstruct);
				qDebug() << "ADDING " << i.fileName();
				MediaRecord temp(files.at(n).rec);

				qDebug() << "ADDED"<<temp.ToString().at(0);
				n += 1;
			}
		};
	};
	qDebug() << n << " FILES ADDED";
	emit catalogIsReady();
}

QStringList Catalog::GetNextFileData(int i)
{
	if ((i<GetSize())&&(i>=0))
	{

		MediaRecord temp(files.at(i).rec);
		return temp.ToString();
	}
	return QStringList();
}

void Catalog::OrganizeFiles(QString dirName, QList<int> chfiles)
{
}

void Catalog::GenerateCatalog(QString newPath)
{
}

#pragma once
```

```
#include <QItemDelegate>
#include <QObject>
#include <QModelIndex>
#include <QSize>
#include <QCheckBox>

class CheckBoxDelegate : public QItemDelegate
{
	Q_OBJECT

public:
	explicit CheckBoxDelegate(QObject *parent);
	~CheckBoxDelegate();

	QWidget *createEditor(QWidget *parent, const QStyleOptionViewItem
&option, const QModelIndex &index) const override;
	void setEditorData(QWidget *editor, const QModelIndex &index) const
override;
	void setModelData(QWidget *editor, QAbstractItemModel *model, const
QModelIndex &index) const override;
	QSize sizeHint(const QStyleOptionViewItem &option, const QModelIndex
&index) const override;
	void updateEditorGeometry(QWidget *editor, const QStyleOptionViewItem
&option, const QModelIndex &index) const override;
};

#include "CheckBoxDelegate.h"

CheckBoxDelegate::CheckBoxDelegate(QObject *parent)
	: QItemDelegate(parent)
{
}

CheckBoxDelegate::~CheckBoxDelegate()
{
}

QWidget * CheckBoxDelegate::createEditor(QWidget * parent, const
QStyleOptionViewItem & option, const QModelIndex & index) const
{
	QCheckBox *checkBox = new QCheckBox(parent);
	checkBox->setChecked(false);
	return checkBox;
}

void CheckBoxDelegate::setEditorData(QWidget * editor, const QModelIndex &
index) const
{
	bool value = index.model()->data(index,
Qt::EditRole).toBool();//Qt::CheckStateRole
	QCheckBox *checkBox = static_cast<QCheckBox*>(editor);
```

```
	checkBox->setChecked(value);
}

void CheckBoxDelegate::setModelData(QWidget * editor, QAbstractItemModel *
model, const QModelIndex & index) const
{
	QCheckBox *checkBox = static_cast<QCheckBox*>(editor);
	model->setData(index, checkBox->isChecked(), Qt::EditRole);
}

QSize CheckBoxDelegate::sizeHint(const QStyleOptionViewItem & option, const
QModelIndex & index) const
{
	return QSize();
}

void CheckBoxDelegate::updateEditorGeometry(QWidget * editor, const
QStyleOptionViewItem & option, const QModelIndex & index) const
{
	editor->setGeometry(option.rect);
}
```

```
#pragma once

#include <QAbstractTableModel>
#include "Catalog.h"

class FileDataModel : public QAbstractTableModel
{
	Q_OBJECT

public:
	FileDataModel(QObject *parent = Q_NULLPTR);
	~FileDataModel();
	int rowCount(const QModelIndex &parent = QModelIndex()) const override
{ return dataSource->GetSize(); };
	int columnCount(const QModelIndex &parent = QModelIndex()) const
override { return 5;};

	QVariant data(const QModelIndex &index, int role) const override;
	bool setData(const QModelIndex &index, const QVariant &value, int role
= Qt::EditRole) override;
	Qt::ItemFlags flags(const QModelIndex & index) const override;
	//QVariant headerData(int section, Qt::Orientation orientation, int
role) const override;
	bool insertRows(int row, int count, const QModelIndex &parent =
QModelIndex()) override;
	void setDataSource(Catalog *source);
	void addData();
private:
	Catalog *dataSource;
```

```
	QList<int> *filesChecked;//
	void updateCatalog();
private slots:
};
#include "FileDataModel.h"
#include <qDebug>
#include <QCheckBox>

FileDataModel::FileDataModel(QObject *parent)
	: QAbstractTableModel(parent)
{

}

FileDataModel::~FileDataModel()//catalog?
{
}

QVariant FileDataModel::data(const QModelIndex & index, int role) const
{
	if (!index.isValid())
		return QVariant();
	if (index.row() >= dataSource->GetSize() || index.row() < 0)
		return QVariant();

	if (role == Qt::DisplayRole) {
		const QStringList curr = dataSource-
>GetNextFileData(index.row());// надо прописать в методе возврат индекса
вектора

		switch (index.column()) {
		case 0:
			return "CheckBox is here";
		case 1:
			return "index";
		default:
			return curr.at(index.column() - 2);
		}}
	if ((role == Qt::CheckStateRole)&& (index.column() == 0)) {
		if (filesChecked->contains(index.row()))
		{
			return Qt::Checked;
		}
		else
			return Qt::Unchecked;
	}/**/
	return QVariant();
}

bool FileDataModel::setData(const QModelIndex & index, const QVariant &
value, int role)
```

```
{
	if (role == Qt::EditRole)
	{
		//check value from editor
		if (value.toBool()) {
			filesChecked->append(index.row());
		}
		else{
			int i = filesChecked->indexOf(index.row());//решить как
посылать реальные индексы
			filesChecked->removeAt(i);
			qDebug()<<"LIST LENGTH:  "<<filesChecked->length();
		}
	}
	return true;
}

Qt::ItemFlags FileDataModel::flags(const QModelIndex & index) const
{
	if (index.column() == 0)
		return Qt::ItemIsUserCheckable | QAbstractTableModel::flags(index)
| Qt::ItemIsEnabled | Qt::ItemIsEditable;
	else
		return QAbstractTableModel::flags(index);
}

bool FileDataModel::insertRows(int row, int count, const QModelIndex &
parent)
{
	beginInsertRows(QModelIndex(), row, row + count - 1);
	for (int i = 0; i < count; ++i)
	{

	}
	endInsertRows();
	return true;
}

void FileDataModel::setDataSource(Catalog * source)
{
	dataSource = source;
	filesChecked = new QList<int>;
}

void FileDataModel::addData()
{
}

void FileDataModel::updateCatalog()
{
```

```
	beginResetModel(); resetInternalData(); endResetModel();

	emit layoutChanged();
}
```

```
#pragma once

#include <QtWidgets/QMainWindow>
#include "ui_QtMediaCatalog.h"
#include "FileDataModel.h"
#include "CheckBoxDelegate.h"
#include <QSortFilterProxyModel>
#include <QStandardItemModel>

#include "qlayout.h"
#include "qlabel.h"
#include "qlineedit.h"
#include "qpushbutton.h"
#include "qtreewidget.h"
#include "qtableview.h"
#include "qlist.h"
#include "Catalog.h"

class QtMediaCatalog : public QMainWindow
{
	Q_OBJECT

public:
	QtMediaCatalog(QWidget *parent = Q_NULLPTR);
private:
	Ui::QtMediaCatalogClass ui;
	QPushButton *butBrowse;
	QPushButton *butSearch;
	QPushButton *butAddToDir;
	QPushButton *butGenerateCat;
	QPushButton *butFilter;
	QLineEdit *editPath;
	QLineEdit *editExt;
	QTableView *fileTable;
	QTreeWidget *treeDirs;
	FileDataModel *model;
	CheckBoxDelegate *chbDelegate;
	QSortFilterProxyModel *filterModel;
	Catalog* catalog;
signals:
	//void BrowseClicked();
	void PathEntered(QString dirPath);
	void DirEntered(QString);
	//void GenerateClicked();

public slots:
```

```
	void UpdateTable();
private slots :
	void SearchClicked();
	void OpenFileBrowser();
	void AddClicked();
	/*void AddFileToCat();
	void GenerateCatalog();*/
};

#include "QtMediaCatalog.h"
#include "EnterPathDialog.h"
#include <QHeaderView>
#include <QInputDialog>
#include <QFileDialog>
#include <QDir>
#include <qDebug>
#pragma once

QtMediaCatalog::QtMediaCatalog(QWidget *parent)
	: QMainWindow(parent)
{
	ui.setupUi(this);
	QWidget *mainWdgt = new QWidget(this);
	catalog = new Catalog();

		//Widgets
	butBrowse = new QPushButton(tr("Browse"), mainWdgt);//?tr
	butSearch = new QPushButton("Search", mainWdgt);
	butAddToDir = new QPushButton("Add To Directory", mainWdgt);
	butGenerateCat = new QPushButton("Generate Media Catalog", mainWdgt);
	butFilter = new QPushButton("Filter", mainWdgt);

	editPath = new QLineEdit("Path to the directory", mainWdgt);
	editPath->setReadOnly(true);
	editExt = new QLineEdit("Extensions", mainWdgt);
	treeDirs = new QTreeWidget(mainWdgt);
	fileTable = new QTableView(mainWdgt);//QTableWidget(1,5,mainWdgt)
	model = Q_NULLPTR;
	chbDelegate = new CheckBoxDelegate(this);//parent- fileTable?
	fileTable->setItemDelegateForColumn(0,chbDelegate);
	filterModel = new QSortFilterProxyModel(this);
	fileTable->setSortingEnabled(true);

	fileTable->setFixedHeight(400);
	fileTable->verticalHeader()->hide();
	fileTable->setSelectionBehavior(QAbstractItemView::SelectRows);

	QGridLayout *mainLayout = new QGridLayout;

	mainLayout->addWidget(editPath, 0, 0, 1, 4, Qt::AlignTop);
	mainLayout->addWidget(butBrowse,0,4);
```

```
	mainLayout->addWidget(butSearch,0,5);
	mainLayout->addWidget(butFilter, 2, 3, Qt::AlignRight);
	butFilter->setFixedWidth(100);
	mainLayout->addWidget(editExt, 2, 4, 1, 2);
	mainLayout->addWidget(fileTable, 3, 0, 16, 4);
	mainLayout->addWidget(treeDirs, 3, 4, 16, 2);
	mainLayout->addWidget(butAddToDir,20,0);
	butAddToDir->setFixedWidth(130);
	mainLayout->addWidget(butGenerateCat,20,3);

	mainLayout->setAlignment(Qt::AlignTop);
	mainWdgt->setLayout(mainLayout);
	setCentralWidget(mainWdgt);
	setWindowTitle("Media Catalog");
	setFixedSize(900, 600);
	connect(this, SIGNAL(PathEntered(QString)), catalog,
SLOT(FillCatalog(QString)));
	connect(catalog, SIGNAL(catalogIsReady()), this, SLOT(UpdateTable()));
	connect(butBrowse, SIGNAL(clicked()), this, SLOT(OpenFileBrowser()));
	connect(butSearch, SIGNAL(clicked()), this, SLOT(SearchClicked()));
	connect(butAddToDir, SIGNAL(clicked()), this, SLOT(AddClicked()));
	/*
	connect(butGenerateCat, SIGNAL(clicked()), this,
SLOT(GenerateCatalog()));*/
}

void QtMediaCatalog::SearchClicked() {
	qDebug() << "PATH ENTERED"<< editPath->text();
	emit PathEntered(editPath->text());

}
void QtMediaCatalog::UpdateTable()
{
	if (model != Q_NULLPTR)	{	delete model;	} //model-
>deleteLater(); }
		qDebug() << "INITIALIZING MODEL";
		model = new FileDataModel(this);
		model->setDataSource(catalog);
		fileTable->setModel(model);
}
void QtMediaCatalog::OpenFileBrowser() {
	qDebug() << "OPEN FILE DIALOG";
	/*//Файловый браузер в стиле Qt
	QFileDialog *fileBrowser = new QFileDialog(this, tr("Choose
Directory"));
	fileBrowser->setOptions(QFileDialog::ShowDirsOnly
		| QFileDialog::DontResolveSymlinks);
	fileBrowser->setFileMode(QFileDialog::Directory);
	fileBrowser->show();
	editPath->setText(fileBrowser->directory().absolutePath());//надо ли
делать exec
```

```
	*/
	editPath->setText(QFileDialog::getExistingDirectory(this, tr("Choose Directory"),
		"C:/",
		QFileDialog::ShowDirsOnly
		| QFileDialog::DontResolveSymlinks));//native file dialog
}

void QtMediaCatalog::AddClicked()
{
	bool ok;
	QString dirName = QInputDialog::getText(this, "Input Dialog", "Enter directory name", QLineEdit::Normal, QString(), &ok);

	if (ok && !dirName.isEmpty())
		emit DirEntered(dirName);
	;
}

#include "QtMediaCatalog.h"
#include "Catalog.h"
#include <QtWidgets/QApplication>

#include "qlayout.h"
#include "qlabel.h"
#include "qline.h"
#include "qpushbutton.h"
#include "qtablewidget.h"
#include "qlist.h"
#pragma once

int main(int argc, char *argv[])
{
	QApplication a(argc, argv);
	QtMediaCatalog w;

	w.show();

	return a.exec();
}
```