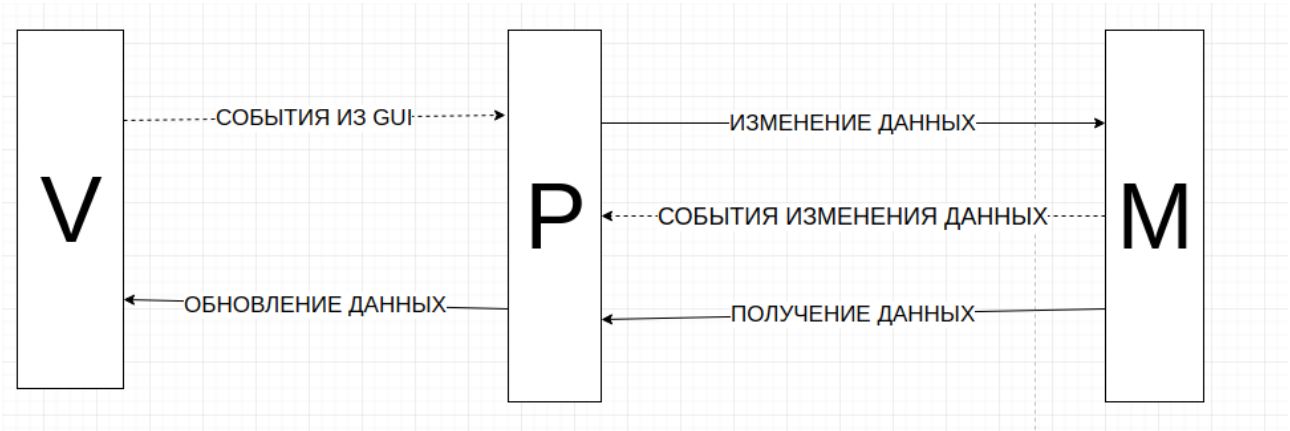
Лекция 3. Архитектура MVP (Model-View-Presenter)
Автор: Набиев Марат

Шаблон проектирования или паттерн в разработке ПО — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.
Шаблон — это не законченный образец, а пример решения задачи, который можно использовать в различных ситуациях.
За счет шаблонов производится унификация деталей решения: модулей, элементов проекта, также облегчает коммуникацию между разработчиками, т. к. каждый шаблон имеет свое имя, и в процессе обсуждения можно сослаться на него и все поймут о чем речь.

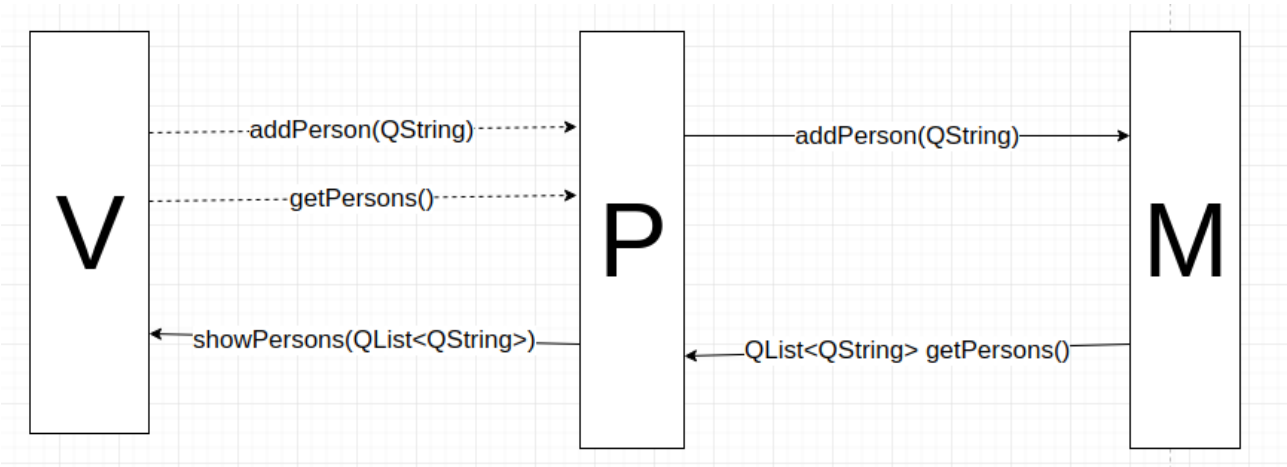
Model-View-Presenter (MVP) — шаблон проектирования, который в основном используется для построения пользовательского интерфейса. Облегчает автоматическое модульное тестирование и улучшает разделение ответственности в логике (отделяется логика и отображение)
Модель (Model) - хранит в себе бизнес-логику, получает данные из хранилища, при необходимости делает запросы на сторонние сервисы.
Вид/Представление (View) — реализует отображение данных и обращается к Презентеру за обновлениями.
Представитель/Презентер (Presenter) — реализует взаимодействие между моделью и представлением.

Пишут, что обычно View запускает Presenter, но мы не будем так делать.
Примерно следующим образом выглядит взаимодействие этих компонент:

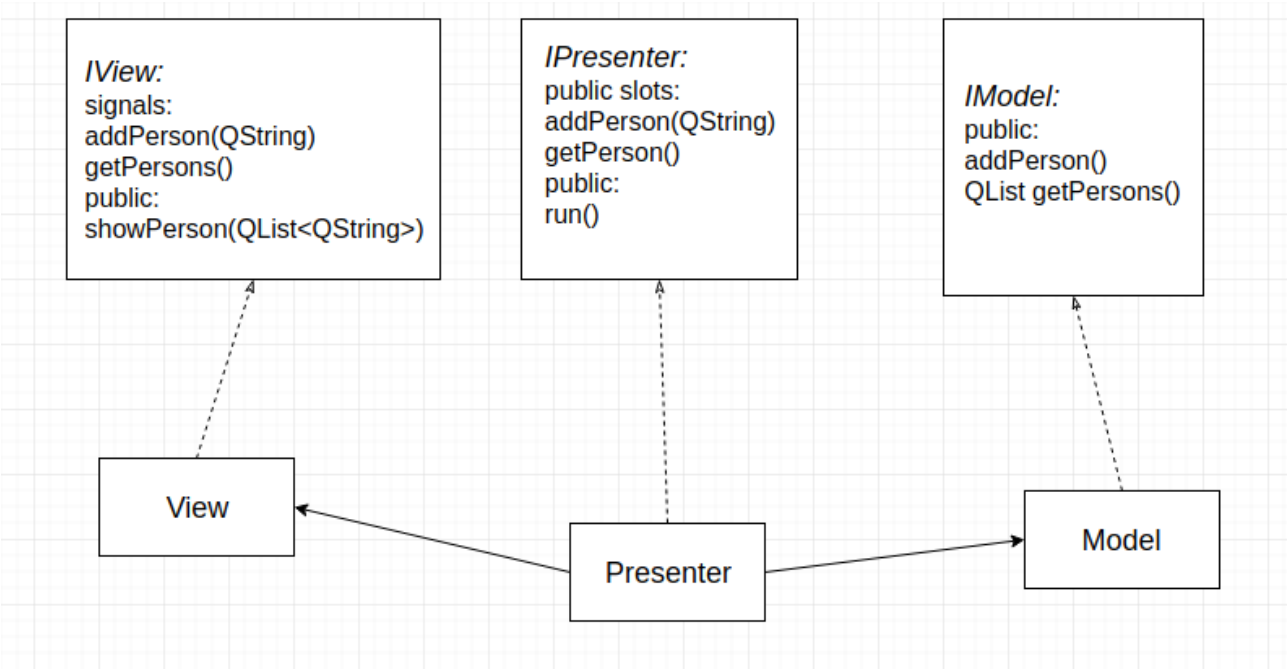


Реализация MVP в Qt от Марата (может есть способ сделать лучше)

Для начала нарисуем примерную нашу архитектуру.

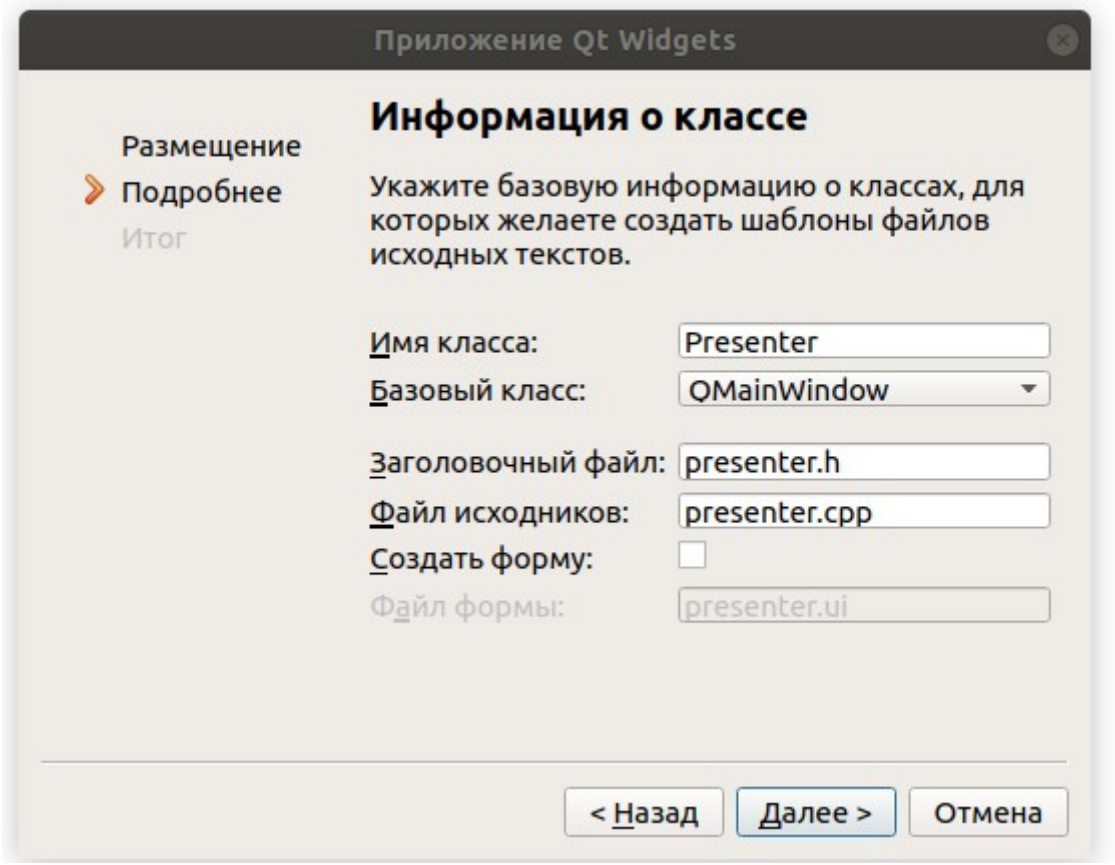


Будет такая взаимосвязь:



Где стрелочки означают следующее:

Открываем Qt и создаем проект с подпроектами.
И создаем подпроект «Приложение Qt Widget». Называем основной класс `Presenter` и убираем галочку из Создать форму, т. к. у нас `Presenter` не будет отвечать за Gui.



Теперь почистим `Presenter` от «лишних» вещей, т. к. Марат не знает как сделать проект MVP на Qt по-человечески.
Откроем `presenter.h` и удалим все лишнее, а именно, наследование от `QMainWindow`

```
1  #ifndef PRESENTER_H
2  #define PRESENTER_H
3
4
5  class Presenter
6  {
7      Q_OBJECT
8
9  public:
10     Presenter();
11     ~Presenter();
12 };
13
14 #endif // PRESENTER_H
```

Также «почистим» `main.cpp`. Оставим его в таком виде:

```cpp
#include "presenter.h"
#include <QApplication>

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

    return a.exec();
}
```

Также откроем `presenter.cpp` и оставим его в следующем виде:

```cpp
#include "presenter.h"

Presenter::Presenter()
{
}

Presenter::~Presenter()
{

}
```

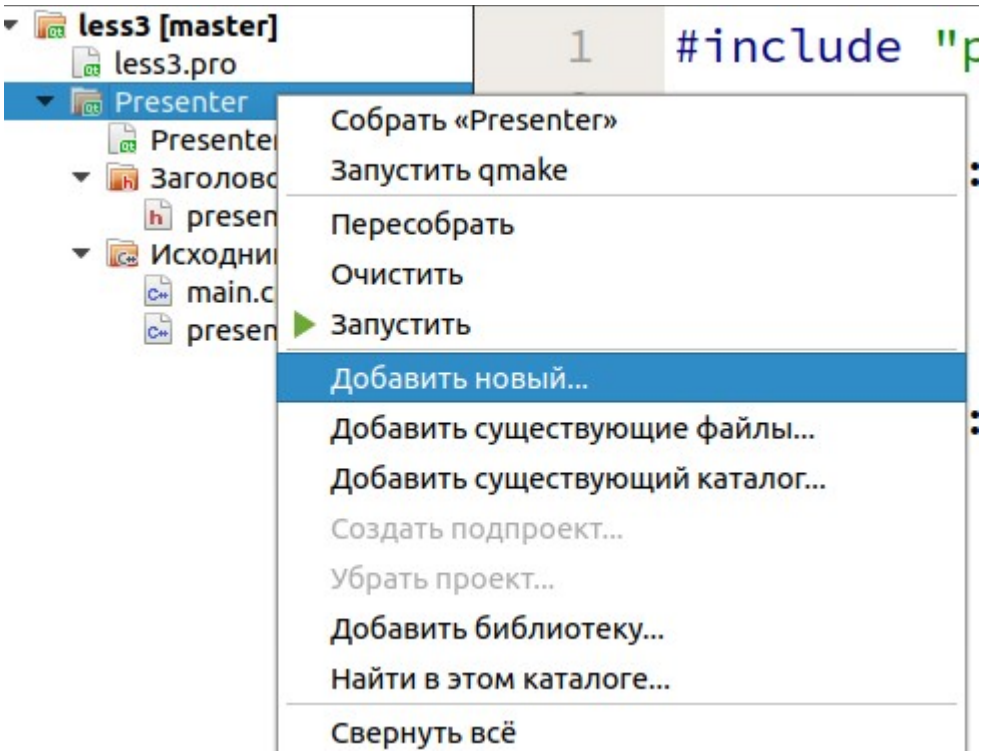
Теперь нам надо описать интерфейс `IPresenter`.

Краткое описание что такое интерфейс и как его реализовать в C++, т. к. в плюсах нет интерфейсов.

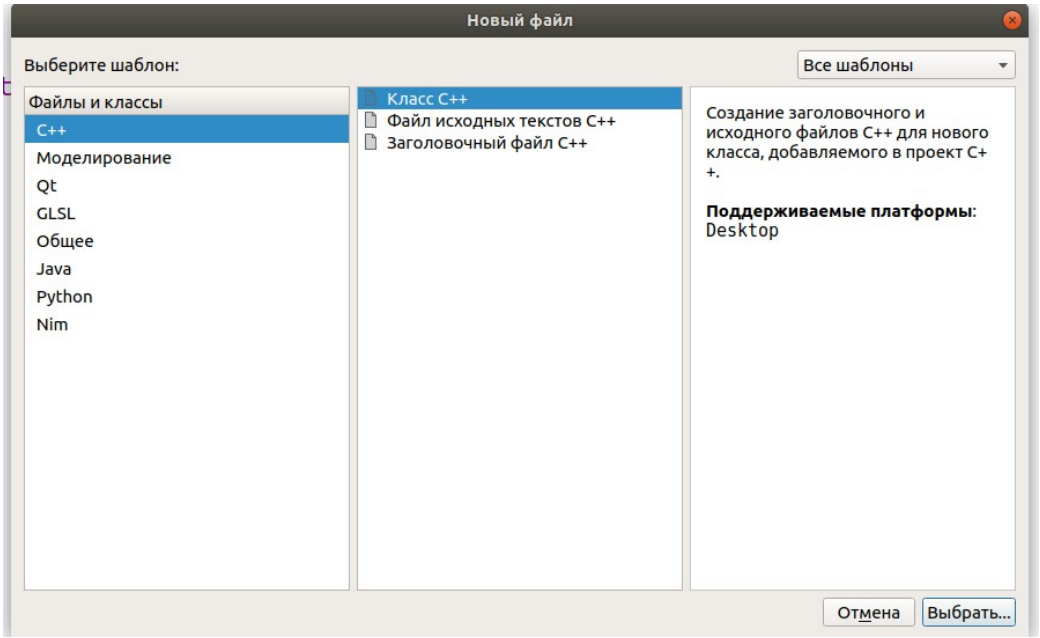
По-простому, **интерфейс** — это такая структура данных, которая только описывает поведение класса, который его реализует, т. е. содержит описание методов(тип результата работы, название метода и входные аргументы), которые должны будут реализованы в классе. Для чего это нужно? Для уменьшения зависимости между классами, т. е. Презентеру важно как можно ему взаимодействовать с представлением, но ему все равно как View будет выводить эти данные.

В C++ нет интерфейсов, что же делать? Все просто, делаем абстрактный класс, в этом классе все методы будут виртуальными и чистыми (после будет стоять =0) или по-простому, абстрактными.

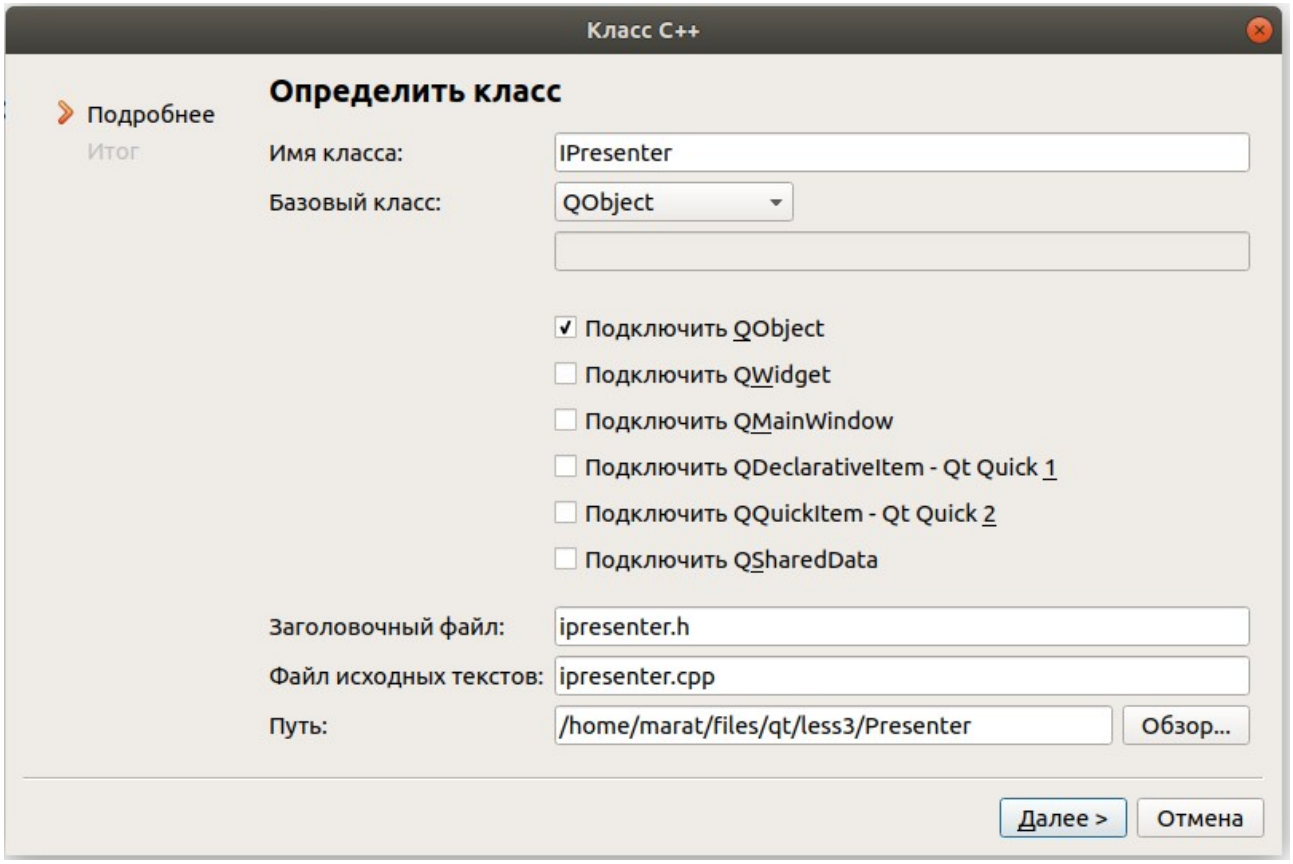
Итак, что должен уметь наш `Presenter`? У него должны быть слоты для обработки сигналов `addPerson` и `getPersons` от `View`. И еще метод который будет все запускать. Добавим новый класс в подпроект `Presenter`.
Для этого нажимаем на `Presenter` ПКМ и выбираем «Добавить новый»



Далее выбираем «Класс C++»



Называем его `IPresenter` и делаем наследником `QObject` (чтобы работать с сигналами и слотами)



файл `ipresenter.cpp` сразу удаляем, т. к. у нас не будет реализации `IPresenter`.
И опишем наш интерфейс:

```
 6  class IPresenter : public QObject
 7  {
 8      Q_OBJECT
 9  public:
10      virtual void run() = 0;
11
12  public slots:
13      virtual void addPerson(QString name) = 0;
14      virtual void getPersons() = 0;
15  };
```

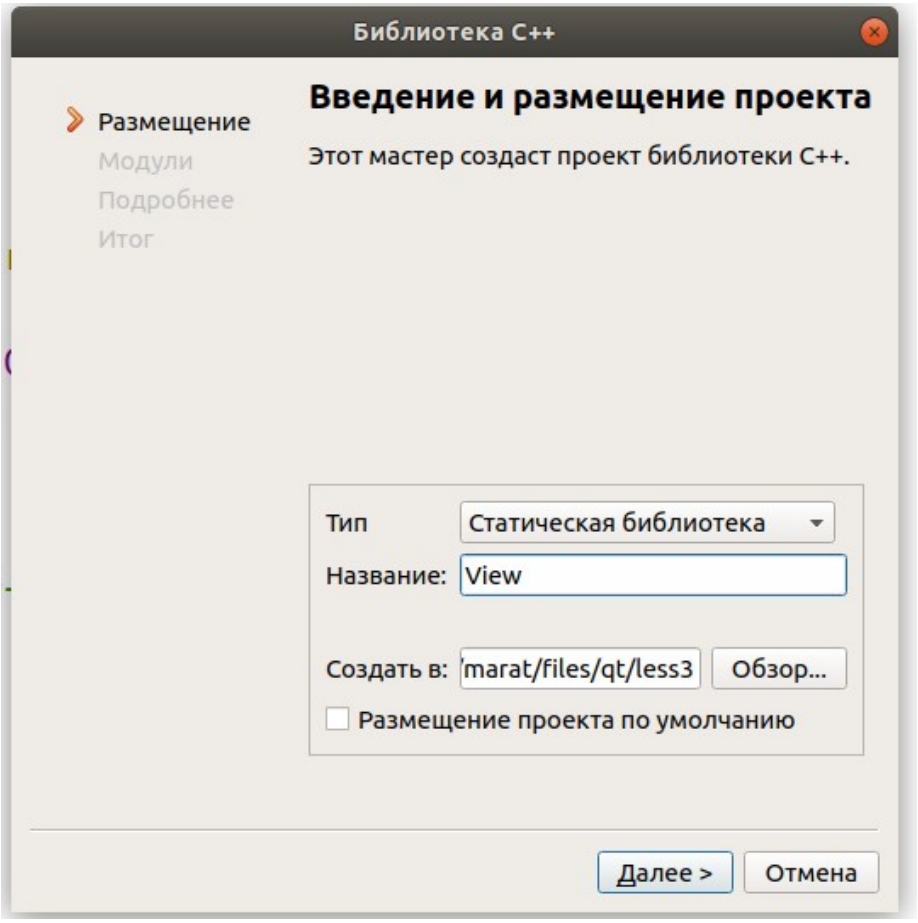
Про сигналы и слоты. Не помню, говорил об этом или нет, но вместе с сигналами можно передавать данные, которые аргументами для слотов. И в слоте `addPerson` мы будет ожидать имя нового человека.

Теперь унаследуем `IPresenter` в `Presenter` и подготовим почву для переопределения.

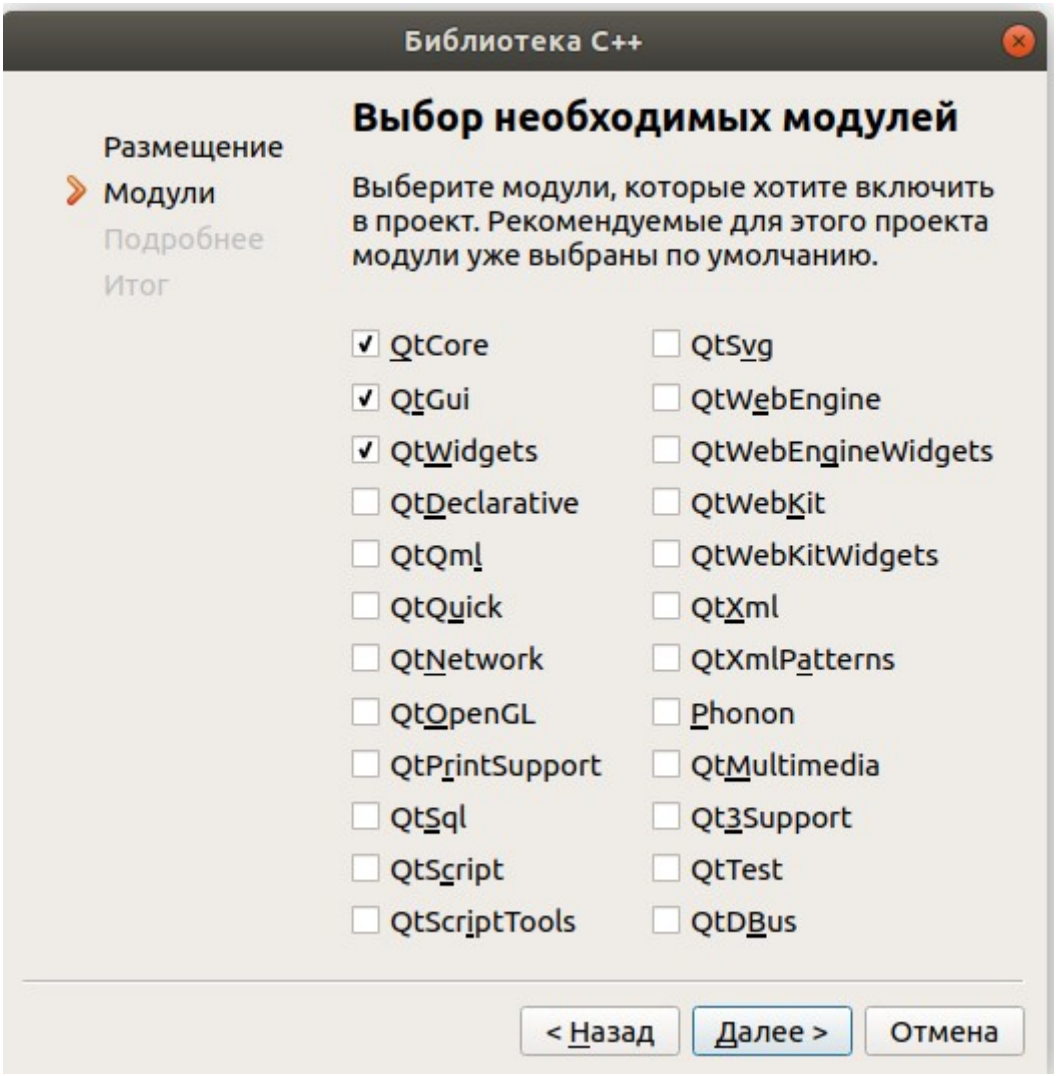
```
#include "ipresenter.h"
class Presenter:public IPresenter
{
    Q_OBJECT

public:
    Presenter();
    ~Presenter();
    void run() override;
public slots:
    void addPerson(QString name) override;
    void getPersons() override;
};
```

Пока на этом с `Presenter` все.
Сейчас реализуем `View`. Для этого добавим статическую библиотеку в наш проект.

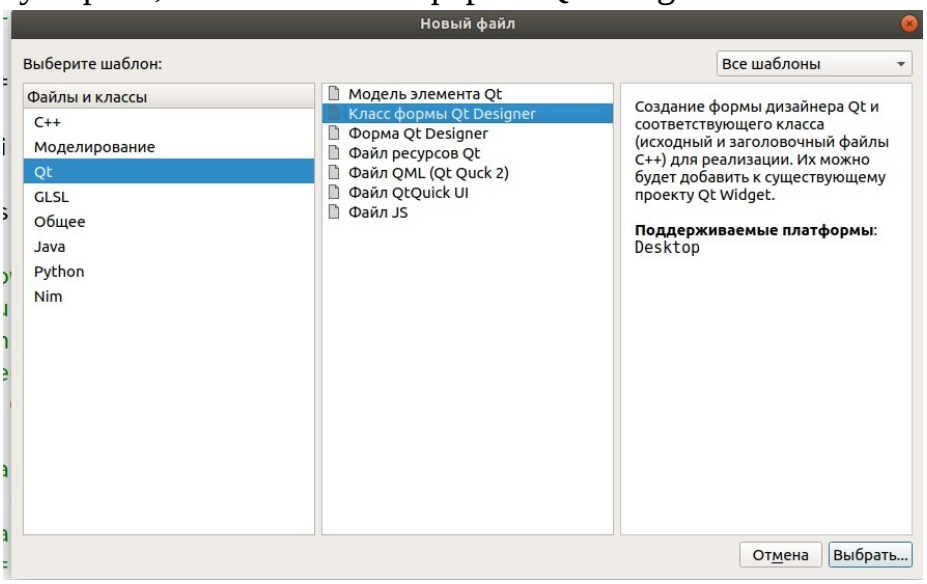


Добавим модули для работы с графическим интерфейсом



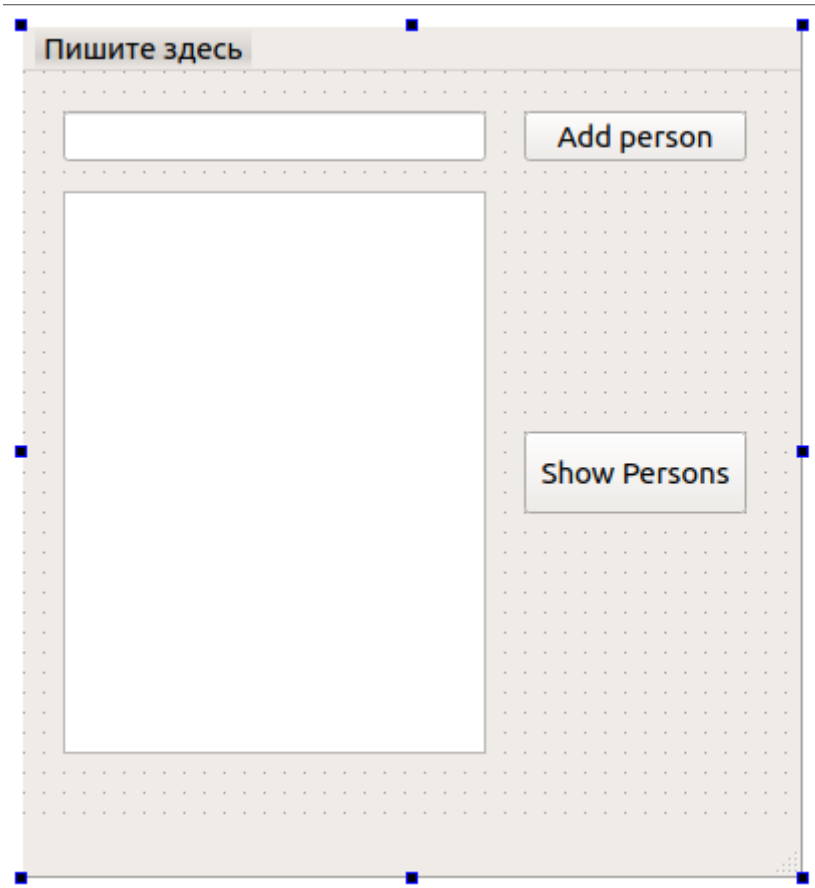
И далее, далее готово.

Снова время костылей от Марата. Удалим файлы `view.h` и `view.cpp` и добавим новую файл, а именно класс формы Qt Designer



и назовем его `View`.
Теперь добавим 4 элемента:
1) `Line Edit` для ввода имени (назову `nameLine`)
2) кнопка для добавления (назову `addButton`)
3) `List Widget` для вывода всех имен (`listWidget`)
4) кнопку для вывода (`showPersonsButton`)

Сделаем следующую форму:



Теперь опишем интерфейс `IView`. Там будут сигналы добавления, запроса всех имен и метод для отображения списка имен. Создадим его как наследника `QMainWindow`. `iview.cpp` можно сразу удалить.
Наш интерфейс будет выглядеть следующим образом:

```
7  class IView : public QMainWindow
8  {
9      Q_OBJECT
10 public:
11     virtual void showPersons(const QList<QString> & names)=0;
12 signals:
13     void addPerson(QString name);
14     void getPersons();
15
16 };
```

Теперь унаследуем его и реализуем методы в классе `View`. Заменим `QMainWindow` на `IView` , т. к. мы специально сделали `IView` наследником `QMainWindow` во избежание такой страшной и тяжелоуправляемой вещи как множественное наследование. Еще не забудем добавить слоты для обработки сигналов кнопок.

```
10  class View : public IView
11  {
12      Q_OBJECT
13
14  public:
15      explicit View(QWidget *parent = 0);
16      ~View();
17  public:
18      void showPersons(const QList<QString> &names) override;
19  private:
20      Ui::View *ui;
21  public slots:
22      void addButtonClick();
23      void showButtonClick();
24  };
```

Сконнектим кнопки и слоты:

```
4   View::View(QWidget *parent) :
5       ui(new Ui::View)
6   {
7       ui->setupUi(this);
8       connect(ui->addButton, &QPushButton::clicked,
9               this, &View::addButtonClick);
10      connect(ui->showPersonsButton, &QPushButton::clicked,
11              this, &View::showButtonClick);
12  }
```

Реализуем все наши слоты и методы. Все они имеют крайне простую реализацию

```
18  void View::addButtonClick()
19  {
20      QString name = ui->nameLine->text();
21      //очистим сразу, чтобы не было мусора
22      ui->nameLine->clear();
23      //эмитим сигнал о добавлении нового имени
24      emit addPerson(name);
25  }
26  void View::showButtonClick()
27  {
28      //просто говорим, что хотим получить
29      emit getPersons();
30  }
31  void View::showPersons(const QList<QString> &names)
32  {
33      //очистим наш списочек
34      ui->listWidget->clear();
35      //и отобразим имена
36      ui->listWidget->addItems(names);
37  }
```

Если пересоберем, то не должно быть серьезных ошибок (только предупреждения максимум)

Далее перейдем к реализации модели, которая будет хранить список имен. Ее интерфейс прост. Там будет 2 метода: для добавления имени и для получения списка имен.
Добавим статическую библиотеку `Model` и сразу добавим туда класс `IModel` . (`imodel.cpp` удалим)

```
6  class IModel
7  {
8  public:
9      virtual void addPerson(QString name)=0;
10     virtual QList<QString> getPersons() const =0;
11 };
12
```

Опишем `Model`, которая реализует этот интерфейс.

```
#include "imodel.h"
class Model: public IModel
{
private:
    //наше хранилище имен
    QList<QString> names;
public:
    Model();
    void addPerson(QString name) override;
    QList<QString> getPersons() const;
};
```

Реализуем эти методы!

```
void Model::addPerson(QString name)
{
    //добавляем в конец имя
    names.append(name);
}
QList<QString> Model::getPersons() const
{
    //просто вернем список
    return names;
}
```

И если пересобрать этот подпроект, то все должно быть хорошо!
Теперь вернемся к `Presenter`. К этому подпроекту добавим наши библиотеки `View` и `Model` (как это сделать было на прошлом занятии) и добавим указатели на `IView` и `IModel`

```
#include "iview.h"
#include "imodel.h"
class Presenter:public IPresenter
{
    Q_OBJECT
private:
    IView  *view;
    IModel *model;
```

теперь реализуем все методы `Presenter`, которые мы оставили на десерт.

1) в классе создадим представление и модель и сконнектимся с сигналами преставления

```
4  Presenter::Presenter()
5  {
6      view = new View;
7      model = new Model;
8      connect(view, &IView::addPerson,
9              this, &Presenter::addPerson);
10     connect(view, &IView::getPersons,
11             this, &Presenter::getPersons);
12 }
```

2) в деструкторе очистим память

```
14 Presenter::~Presenter()
15 {
16     delete view;
17     delete model;
18 }
```

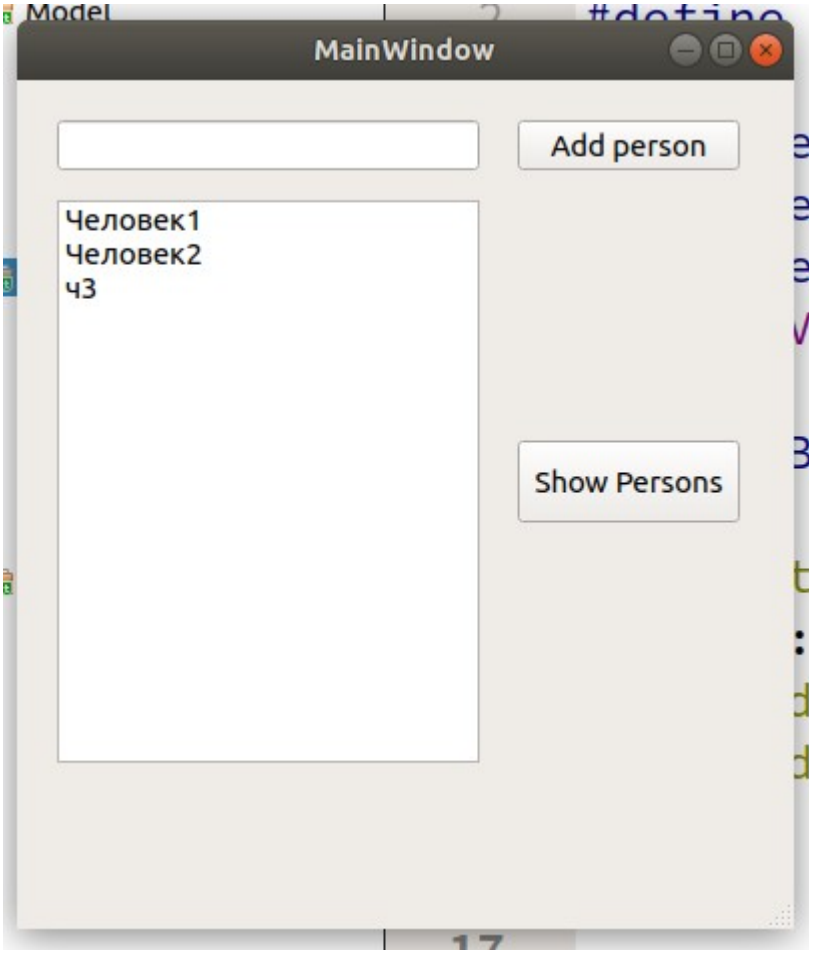
Реализуем методы интерфейса:

```
19 void Presenter::addPerson(QString name)
20 {
21     //добавляем в модель имя, не знаем как оно там дальше будет храниться
22     model->addPerson(name);
23 }
24 void Presenter::getPersons()
25 {
26     //получаем список
27     QList<QString> names = model->getPersons();
28     //не знаем, что с ним дальше
29     view->showPersons(names);
30 }
31 void Presenter::run()
32 {
33     //запускаем
34     view->show();
35 }
```

Теперь в `main.cpp` запусим наш `Presenter`!

```
4  int main(int argc, char *argv[])
5  {
6      QApplication a(argc, argv);
7      IPresenter *presenter = new Presenter;
8      presenter->run();
9      //не очищаем память, иначе даже не запустится :D
10     return a.exec();
11 }
```

Попробуем собрать, и если все хорошо, то соберется без ошибок
И все это будет работать!



Ссылка на исходники: https://github.com/nma2207/qt-lessions/tree/master/less3
В следующем занятии попробуем затронуть SQL (скорее всего sqlite) и возможно, напишем приложении на базе вот этого. Будем сохранять в БД, а не в список.