Създаване на потребителски графичен интерфейс с помощта на QT
или
Една идея за програмен модел

Михаил Петров - град Смолян

16.03.2002 г.

Съдържание

1. Малко история или необходимостта от подобни разработки.

- Дълбоко съм убеден, че без създаване на скучни програми, от рода на складови програми, счетоводства и подобни, Linux, като операционна система, не би могъл да заеме полагаемото му се място на пазара на информационни продукти и технологии. Според мен това е пътя за налагане на по - добрата операционна система и популяризирането и. Но от някъде все пак трябва да се започне, е аз започнах от тук.
- Създаването на подобни програмни модели би привлякло на наша страна приложните програмисти, към които и аз се причислявам, което много би помогнало за разработка на програми, ориентирани към конкретния потребител, защото негово величество потребителят не се интересува чак толкова от това дали програмата му е направена под Windows, Linux или QNX, а от това дали дадения продукт върши някаква работа и как я върши.
- Както и да го въртим, не можем да си затворим очите за дългогодишното присъствие на Windows на пазара, което доведе до навика, потребителя да изпълнява определени действия, като щрака по икони, бутони и други атрибути на "привлекателния графичен интерфейс"и той търси тези неща, въпреки, че много често графичният интерфейс няма нищо общо с ефективността на даденото приложение, а по - скоро пречи на добрата работа на приложението.
- Създаването на потребителски графичен интерфейс е по - скоро тъткава работа, но без това няма да минем. И в тази връзка предлагам този програмен модел, с надеждата, че това донякъде ще подпомогне създаването на приложения, съдържащи такъв графичен интерфейс.
- Признавам си без бой, че идеята не е изцяло моя, а е взета от идеята на MS VC++, за "Архитектурата Документ/Изглед в което според мен има достатъчно много хляб. Аз просто претворих донякъде тази идея, като съм се стремял да избегна от нещата, които не ми харесват. Доколко всичко това е добро или не чак толкова оставам на вас да прецените. Считам, колкото и скандално да прозвучи, в нещото наречено Windows има достатъчно много идеи, които просто плачат за по - добра реализация, което от своя страна би довело до качествено нови идеи и разработки, а до по - благосклонно отношение на потребителите към Linux. Предпоставките в това отношение са достатъчно много и остава да ги използваме.
- Тази статия я раждам в продължение на около три месеца, защото се оказва, че е доста трудно да се преведе написаното на C++, на нормален човешки език и в тази връзка моля да бъда извинен за вероятно не добрия стил на статията, но аз съм все пак приложен програмист а не журналист или писател. Надявам се че написаното е достатъчно разбираемо ако не за обикновените потребители то поне за програмистите.
- Самите source code ще можете да свалите, едва след като си направя WEB страница някъде и обещавам, ако има интерес към тях да ги публикувам.
- Не очаквайте да намерите в тази статия начин и описание на класовете на QT, защото описанието на тази библиотека е достатъчно добро и подробно и нямам намерение да го повтарям. Тук ще опиша принципните постановки на програмния модел, който много ми се иска да нарека архитектура, но не съм толкова велик, като Microsoft, така че ще го наричам програмен модел.

2. Структурата DoraRuntimeClass и класът DoraClassObject.

За да създаваме потребителски графичен интерфейс е необходимо да осигурим, освен изчертаването на прозорците, по начин, необходим за конкретното приложение, също и механизъм за взаимодействие между отделните обекти. Това може да се осъществи с направата на RUNTIME механизъм, позволяващ достъпа от даден обект до други обекти от приложението, както и обратната връзка. За постигането на тази цел е необходимо обектите да се създават динамично по време на изпълнение. В настоящият програмен модел това се постига със съвкупността от една структура `DoraRuntimeClass`, класът `DoraClassObject` макросът `RUNTIME_CLASS`. Това само по себе си не е нито ново, нито необичайно. Освен това тук съм длъжен да спомена, че QT, като графичен интерфейс има подобен механизъм, макар и реализиран по по - различен начин, имам предвид технологията signal/slot. Длъжа да ви успокоя, че настоящият програмен модел по никакъв начин не нарушава вътрешната логика на QT, а по - скоро я използва, макар и неявно.

Структурата `DoraRuntimeClass` е дефинирана във файла с дефиниции `DoraClassObject.h` и изглежда по следния начин:

```
class DoraClassObject;

struct DoraRuntimeClass {
  CString class_name;
  DoraClassObject* (*m_pCreateView)(QWidget *parent = 0);
  DoraClassObject* CreateView(QWidget *parent = 0);
  DoraClassObject* (*m_pCreateDocument)();
  DoraClassObject* CreateDocument();
};
```

Както се вижда в дефиницията на структурата няма нищо необичайно. Променливата `class_name` е от тип `CString`, който съм написал за да боравя по - лесно с променливи от типа `char[]`, и той няма нищо общо с познатия до болка CString от MS VC++, като допълнително съм включил в него механизъм за конвертиране на паскалски стрингове в нулево базирани стрингове. Освен това в структурата има и две callback функции, като функцията `CreateView` служи за конструиране на обекти, които са прозорци, а `CreateDocument` е за конструиране на обекти, които не са прозорци.

Класът `DoraClassObject` е дефиниран в същият файл по следния начин:

```
class DoraClassObject {
public:
  virtual DoraRuntimeClass *GetClassObject() const {
    classDoraClassObject.class_name = "";
    return &classDoraClassObject;
  }
  static DoraRuntimeClass classDoraClassObject;
public:
  DoraClassObject() {}
  virtual ~DoraClassObject() {}
};
```

Смятам, че в дефиницията на класа, също няма нищо необичайно. Както се вижда има си конструктор и виртуален деструктор. Функцията `GetClassObject` служи за получаване на името на класа, който се създава, с идеята да се използва по - натък за получаване на указател към съответния обект, чрез който да получа достъп до функциите и променливите на съответния клас. За целта ще е необходима във всички наследници на този клас да се пре-дефинира тази виртуална функция.

Връзката между структурата `DoraRuntimeClass` и класът `DoraClassObject`, се осъществява с един макрос `RUNTIME_CLASS`, дефиниран по следния начин:

```
#define RUNTIME_CLASS(class_name)(&class_name::class##class_name)
```

До тук сме извършили подготовката на необходимото за динамичното създаване на обектите в едно приложение. Структурата `DoraRuntimaClass`, класът `DoraClassObject` и макросът `RUNTIME_CLASS`, е необходимо да се разглеждат заедно, именно тяхната съвкупност осигурява необходимия механизъм за динамичното създаване на обекти и взаимодействието между тях.

Файлът с имплементациите е `DoraClassObject.cpp`, там няма кой знае какво, просто описанието на двете callback функции заедно със задаването на името на класа:

```
DoraRuntimeClass DoraClassObject::classDoraClassObject =
  {"DoraClassObject", NULL, NULL};

DoraClassObject *DoraClassObject::CreateView(QWidget *parent = 0) {
  return (*m_pCreateView)(parent);
}
DoraClassObject *DoraClassObject::CreateDocument() {
  return (*m_pCreateDocument)();
}
```

3. Базов изгледен клас и базов документен клас.

Нека да разделим обектите в едно приложение по следния начин - нека обектите, които показват някакви данни в прозорец на екрана, наречем изгледни класове, а обектите, които съхраняват данните на приложението и/или извършват обработката на данните, наречем документни класове. Тогава създавайки един базов клас `DoraBaseView` служещ за производство на изгледни класове и един базов документен клас `DoraBaseDocument` за документните класове.

Класът `DoraBaseView` е наследник на два класа, `QWidget`, идващ от QT и описаният по - горе клас `DoraClassObject`, той е дефиниран във файла с дефиниции - `DoraBaseView.h`.

```
class DoraBaseView : public QWidget, public DoraClassObject
{
public:
  DoraRuntimeClass *GetClassObject() const {
    return &classDoraBaseView;
  }
```

```
  static DoraRuntimeClass classDoraBaseView;
  static DoraClassObject *CreateView(QWidget *parent = 0);
  static DoraClassObject *CreateDocument();
public:
  DoraBaseView(QWidget *parent = 0, const char *name = 0)
    : QWidget(parent, name), DoraClassObject() {
  }
  ~DoraBaseView() {}

private:
  void paintEvent(QPaintEvent *e);
  void resizeEvent(QResizeEvent *e);
  void mousePressEvent(QMouseEvent *e);
  void showEvent(QShowEvent *e);
  void hideEvent)QHideEvent *e);
  void focusInEvent(QFocusEvent *e);
  void focusOutEvent(QFocusEevent *e);

public:
  virtual void OnDraw(QPainter *p);
  virtual void OnSize(int cx, int cy);
  virtual void OnShowView();
  virtual void OnHideView();
  virtual void OnFocus();
  virtual void OffFocus();
  virtual void PostMouseRightButton();
  virtual void PostMouseLeftButton();
  virtual void PostMouseMidButton();
};
```

Както се вижда от дефиницията на класа, е необходимо да се дефинират първо статичната променлива `classDoraBaseView`, двете статични функции за динамичното създаване на обектите - `CreateView` и `CreateDocument` и да се пре-дефинира виртуалната функция `GetClassObject`. Функциите от секция `private` са виртуални функции дефинирани в класа `QWidget` на QT. Виртуалните функции от следващата секция `public` са дефинирани от мен и служат за приемане на съобщенията идващи от средата и обработвани от QT. Те трябва да се пре-дефинират в наследниците на този клас. По този начин предоставям на QT, грижата да установи връзката с графичната среда, която от своя страна има грижата да се занимава с хардуера. Тук му е мястото да спомена, че изборът ми на QT, вероятно не е най - удачният избор на графична библиотека, за разработка на подобен програмен модел, но когато започвах разработката му QT някак много странно ми заприлича на нещо много познато, имам предвид MFC. Въпреки това обаче за мое учудване моделът проработи. Така, че продължавам с имплементацията на класа. Това е направено във файла с имплементации `DoraBaseView.cpp`.

Първо, всеки клас, разработван според този програмен модел, файлът с имплементации, трябва да започва със следният ред:

```
DoraRuntimeClass DoraBaseView::classDoraBaseView =
  {"DoraBaseView", DoraBaseView::CreateView, NULL};
```

за изгледен клас, в случая това е базовият клас DoraBaseView. Също така е необходимо да имплементира двете функции - `CreateView` и `CreateDocument`. За класове, които ще са изгледи функцията `CreateView`, връща указател към клас `DoraClassObject` а за документните класове би трябвало да връща нулев указател. Но аз предпочетох да разделя нещата, така че функцията `CreateView`, да използвам за изгледни класове, а `CreateDocument` за документните класове. За изгледните класове те изглеждат така:

```
DoraClassObject *DoraBaseView::CreateView(QWidget *parent = 0)
{
  return new DoraBaseView(parent);
}

DoraClassObject *DoraBaseView::CreateDocument()
{
  return NULL;
}
```

Тези функции, така написани са предназначени за базовият изгледен клас, така че за производните му класове е необходимо да се пренапишат за конкретния клас. Тук няма до описвам виртуалните функции, защото те не са толкова интересни, и служат само за приемане на съобщенията, които QT изпраща към приложението. Какво точно изпълняват зависи от конкретното приложение. Ще можете да видите детайлно какво представляват от source кодовете, след като ги публикувам.

Нека сега подготвим базовият документен клас - `DoraBaseDocument`. Той не е нещо сложно. В него съм добавил механизъм за управление на изгледни класове, свързани с производния на него документен клас. За целта съм разработил един `template` клас, представляващ динамичен масив и една структура за елементите на масива и понеже нищо друго не ми идваше наум я кръстих `RegistryStruct`, тъй като тя ще се попълва в масива, съм декларирал променлива от тип динамичен масив, по следния начин:

```
typedef DoraArray<RegistryStruct> TypeRegistryItem;
```

и сега самата променлива:

```
TypeRegistryItem localRegistry;
```

Дефиницията `typedef` е хубаво да бъде някъде в друг файл, където да се декларират всички масиви от този тип, за да не се претрупва излишно файлът с дефиниции на базовият документен клас. Е нека сега видим как е дефиниран този базов документен клас:

```
class DoraBaseDocument : public DoraClassObject
{
public:
  DoraRuntimeClass *GetClassObject() const {
    return &classDoraBaseDocument;
  }
  static DoraRuntimeClass classDoraBaseDocument;
  static DoraClassObject *CreateView(QWidget *parent = 0);
  static DoraClassObject *CreateDocument();
```

```
public:
  DoraBaseDocument() : DoraClassObject() {}
  ~DoraBaseDocument() {}
public:
```

В тази секция следват няколко функции за управление на регистъра на свързаните към производните документни класове изгледи. Те извършват следното - попълват масива със елементите на структурата RegistryStruct, премахват ненужните елементи при премахване на изгледния обект, обновяват или по - точно подават сигнал за обновяване на изгледа и т.н.

```
private:
  TypeRegistryItem localRegistry;
};
```

Дефиницията на класа е във файла `DoraBaseDocument.h`.

Нека сега да напишем имплементацията на класа. Това съм го направил във файла `DoraBaseDocument.cpp`:

```
DoraRuntimeClass DoraBaseDocument::classDoraBaseDocument =
  {"DoraBaseDocument", NULL, DoraBaseDocument::CreateDocument};

DoraClassObject *DoraBaseDocument::CreateView(QWidget *parent = 0)
{
  return NULL;
}

DoraClassObject *DoraBaseDocument::CreateDocument()
{
  return new DoraBaseDocument();
}
```

Сега вече след като имаме базов изгледен клас и базов документен клас, можем спокойно да пристъпим към създаване на конкретно приложение според този програмен модел.

4. Създаване на приложения по този програмен модел.

За да създаваме приложения с този програмен модел е необходимо да започнем от класа на приложението - `DoraMainApp`, който да формира класа на главния прозорец - `DoraMainFrame`, първият от всички изгледи на приложението - `DoraCentralView` и централния документен клас - `DoraCentralDocument`. Знаем по дефиниция, че всяка програма на C/C++, започва да се изпълнява от функция `main`. При мен тази функция има за задача да създаде указател към класа `DoraMainApp` и да извика неговата функция `RunApp`. Ето как изглежда това на практика във файла `main.cpp`.

```
int main(int argc, char **argv)
{
```

```
  DoraRuntimeClass *doraClassApp      = RUNTIME_CLASS(DoraClassApp);
  DoraClassObject   *doraClassObject = doraClassApp->CreateView();
  mainApp = dynamic_cast<DoraClassApp *>(doraClassObject);
  int result = mainApp->RunApp(argc, argv);
  delete mainApp;
  return result;
}
```

Дефинирайки глобалната променлива `mainApp`, ние получаваме глобален указател към главния клас на приложението. Където е необходимо да получим достъп до функциите и променливите на главния клас на приложението трябва да дефинираме тази променлива като `extern DoraMainApp *mainApp;` Но нека сега да видим какво представлява главния клас на приложението `DoraMainApp`, и е дефиниран във файла с дефинициите - `DoraMainApp.h` по следния начин:

```
class DoraMainApp : public DoraClassObject
{
public:
  DoraRuntimeClass *GetRuntimeClass() const {
    return &classDoraMainApp;
  }
  static DoraRuntimeClass classDoraMainApp;
  static DoraClassObject *CreateView(QWidget *parent = 0);
  static DoraClassObject *CreateDocument();
public:
  int RunApp(int argc, char **argv);
public:
  DoraMainFrame *mainFrame;
};
```

Както се вижда от дефиницията на класа, няма нищо необичайно в този клас, освен може би това, че този клас не е изгледен, въпреки което създаването му се извършва с функция `CreateView`, но той не показва никакъв прозорец. Първият прозорец на приложението е класът `DoraMainFrame`. Конструирането му се извършва във функция `RunApp`, от файла с имплементации - `DoraMainApp`:

```
DoraRuntimeClassDoraMainApp::classDoraMainApp =
  {"DoraMainApp",DoraMainApp:CreateView, NULL};

DoraClassObject *DoraMainApp::CreateView(QWidget *parent = 0)
{
  return new DoraMainApp();
}
DoraClassObject *DoraMainApp::CreateDocument()
{
  return NULL;
}

int DoraMainApp::RunApp(int argc, char **argv)
{
```

```
  QApplication a(argc, argv);
  QWidget *wid = QApplication::desktop();

  a.setFont("times", 12, QFont::Normal, FALSE, QFont::AnyCharSet);
  int wFrame = wid->width();
  int hFrame = wid->height();
  int w = (int)((wFrame * 2) / 3);
  int h = (int)((hFrame * 2) / 3);
  int x = (int)((wFrame - w) / 2);
  int y = (int)((hFrame - h) / 2);

  DoraRuntimeClass *runtimeMainFrame = RUNTIME_CLASS(DoraMainFrame);
  DoraClassObject *objectMainFrame = runtimeMainFrame->CreateView(NULL);
  mainFrame = dynamic_cast<DoraMainFrame *>(objectMainFrame);
  mainFrame->OnCreate();

  a.setMainFrame(mainFrame);
  mainFrame->setGeometry(x, y, w, h);
  mainFrame->show();
  int result = a.exec();

  return result;
}
```

Ще се спра по - обстойно на функцията `RunApp`, тъй като има доста неща които идват от библиотеката QT. Редът `QApplication a(argc, argv)`, еднозначно определя главния клас на приложението от гледна точка на QT. Променливата `*wid` е указател към desktopa на графичната сред, в случая KDE. По нататък задавам общо валиден шрифт за приложението с израза `a.setFont("times 12, QFont::Normal, False, QFont::AnyCharSet)`. За повече информация за променливите и класовете на QT, прочетете описанието на библиотеката, то е достатъчно добро и разбираемо. Следващите редове определят позицията на екрана, ширината и височината на прозореца, който се каним да покажем. В случая прозорецът има такива размери, че да покрие две трети от видимият екран.

Получаването на валиден указател на първия прозорец на приложението се извършва на три стъпки. Първо получавам указател към структурата `DoraRuntimeClass` с помощта на макроса `RUNTIME_CLASS`, Този указател се използва за конструиране на променлива от тип `DoraClassObject`, след извикване на функцията `CreateView` с нулев указател, което задава на QT, че това е първият прозорец. Накрая преобразувам върнатия указател към указател към необходимия ми обект. Тук и навсякъде в програмния модел за преобразуване на указатели към определени обекти използвам операцията `dynamic_cast<T>(v)`. Така аз имам RTTI (RunTime Type Information), за всеки обект сформиран по този начин. В скоби казано такова преобразуване при Microsoft изглежда така: `mainFrame = (DoraMainFrame *)GetRuntimeClass()`, но това си е в крайна сметка техен проблем. Метода който използвам за преобразуване на указатели е много по - безопасен и надежден.

Но да продължим нататък. С израза `a.setMainFrame(mainFrame)`, задавам на QT, главният прозорец на приложението - `DoraMainFrame`. След това задавам началната позиция `x`, `y`, ширината `w` и височината `h`, на прозореца, след което го показам. Променливата `rezult` съдържа резултата от изпълнението на приложението, която променлива се вър-

ща във функцията `main`.

Сега нека се върнем към обекта на главния прозорец на приложението - `DoraMainFrame`. Него съм го дефинирал във файла с дефиниции `DoraMainFrame.h`:

```
class DoraMainFrame : public QWidget, public DoraClassObject
{
public:
  DoraRuntimeClass *GetClassObject() const {
    return &classDoraMainFrame;
  }
  static DoraRuntimeClass classDoraMainFrame;
  static DoraClassObject *CreateView(QWidget *parent = 0);
  static DoraClassObject *CreateDocument();
public:
  DoraMainFrame(QWidget *parent = 0, const char *name = 0) :
    QWidget(parent, name), DoraClassObject() {}
  ~DoraMainFrame() {}
public:
  void OnDraw(QPainter *p);
  void OnSize(int cx, int cy);
private:
  void paintEvent(QPaintEvent *e);
  void resizeEvent(QResizeEvent *e);
public:
  DoraCentralDocument *GetDocument();
  void OnCreate();
public:
  DoraCentralView     *m_pCentralView;
  DoraBaseDocument *m_pBaseDocument;
  QRect                         clientRect;
};
```

Така дефиниран този клас не се отличава кой знае колко от базовият клас `DoraBaseView`, с изключение на функцията `GetDocument` и двете променливи `m_pCentralView` и `m_pBaseDocument`. Функцията `GetDocument` връща указател към класа на главния документ, чрез променливата `m_pBaseDocument`, а променливата `m_pCentralView`, съхранява указател към първият изгледен клас на приложението. Сега нека видим какво представлява самият клас. Функциите са описани във файла с имплементации - `DoraMainFrame.cpp`.

```
DoraRuntimeClass DoraMainFrame::classDoraMainFrame =
  {"DoraMainFrame", DoraMainFrame::CreateView, NULL};

DoraClassObject *DoraMainFrame::CreateView(QWidget *parent = 0)
{
  return new DoraMainFrame(parent);
}
DoraClassObject *DoraMainFrame::CreateDocument()
{
  return NULL;
}
```

Функциите `OnDraw` и `OnSize`, не са чак толкова интересни, тяхната задача е да определят размерите на показвания прозорец и да го изрисуват. Така, че аз няма да се спирам на тях, а ще опиша функцията `OnCreate`, защото тя заслужава по - голямо внимание. Тази функция е входната точка на класа, определяща и конструираща обектите, които се пораждат от съответния клас, разбира се бих могъл да я направя виртуална в някакъв базов клас, обаче така бих лишил класовете, които ще създавам от мобилност. По - добре е такава функция да е уникална за всеки клас. Какво представлява функция `OnCreate`, в класа `DoraMainFrame`.

```
void DoraMainFrame::OnCreate()
{
  DoraRuntimeClass *runtimeDocument = RUNTIME_CLASS(DoraCentralDocument);
  DoraClassObject   *objectDocument = runtimeDocument->CreateDocument();
  DoraCentralDocument
   *centtalDocument=dynamic_cast<DoraCentralDocument*>(objectDocument();
  m_pBaseDocument = dynamic_cast<DoraCentralDocument *>(centralDocument);

  DoraRuntimeClass *runtimeView = RUNTIME_CLASS(DoraCentralView);
  DoraClassObject *objectView = runtimeView->CreateView(this);
  m_pCentralView = dynamic_cast<DoraCentralView*>(objectView);
  m_pCentralView->OnCreate(this);
}
```

Както се вижда от описанието на `OnCreate`, функцията първо създава обект от клас `DoraCentralDocument`, който е и първият документен клас на приложението, след това създава обект от клас `DoraCentalView`, който е и първият изгледен клас на нашето приложение. Не бъркайте класа `DoraMainFrame` с този клас, `DoraMainFrame` е рамката на приложението, а `DoraCentralView` е първият изглед. Понеже засега се ограничавам до рамките на едно приложение, не мисля че е необходимо да запазвам указателите на създадените обекти в `registry`.

Нека видим сега, какво представлява функцията `GetDocument`. Както споменах по горе нейната единствена задача е да върне указател към централния документ на приложението `DoraCentralDocument`. Тя е малка и изглежда така:

```
DoraCentralDocument *DoraMainFrame::GetDocument()
{
  DoraCentralDocument *centralDocument;
  centralDocument = dynamic_cast<DoraCentralDocument*>(m_pBaseDocument);
  return centralDocument;
}
```

За да бъде един прилично изглеждащ прозорец, остава да закачим, към него един клас за меню, един клас за лента с инструменти - ToolBar и евентуално един клас за StatusBar. Така на този етап направихме един прозорец, който може да се минимизира и максимизира, благодарение на QT. Притежаващ меню, лента с инструменти и лента за състояние. Сега можем спокойно да го минимизираме и максимизираме до безкрайност, но това приложение все още не върши никаква полезна работа, от гледна точка на потребителите. В такъв случай нека се опитаме да създадем нещо, което върши някаква работа. Например един клас за показване и определяне на дати и време, нещо като календарче, с включен в него часовник.

Такъв клас се конструира от класа `DoraCentralView` и ще се активира от менюто. Този клас нека се състои от три изгледни класа, един документен клас и два `PushButtons` с наименования "Промени"и "Откажи". Ще направим един изгледен клас, който ще го натоварим със задачата да управлява другите два изгледа, а именно класът `CalendarView` и `ClockView`, освен това той ще има грижата за осъществяване на комуникацията между тях и документния клас и прехвърлянето на данните от локалния документен клас и централния документен клас на приложението - `DoraCentralDocument`. Като начало нека декларираме две променливи в класа `DoraCentralDocument – CString curentDate` и `CString curentTime`.

Сега ще декларираме централния изгледен клас - `DateAndTimeView`. Това ще направим във файла с декларации - `DateAndTimeView.h`:

```
class DateAndTimeView : public DoraBaseView
{
public:
  DoraRuntimeClass *GetClassObject() const {
    return &classDateAndTimeView;
  }
  static DoraRuntimeClass classDateAndTimeView;
  static DoraClassObject *CreateView(QWidget *parent = 0);
  static DoraClassObject *CreateDocument();
public:
  DateAndTimeView(QWidget *parent = 0, const char *name = 0) :
    DoraBaseView(parent, name) { }
  ~DateAndTimeView() { }
public:
  void OnDraw(QPainter *p);
  void OnSize(int cx, int cy);
public:
  DateAndTimeDocument *GetDocument();
  void OnCreat(DoraClassObject *pObject, DoraBaseView *pView);
public:
  DoraClassObject *m_pObject;
  DoraBaseView      *m_pView;
  DoraBaseDocument *m_pDocument;
  CalendarView        *m_pCalendar;
  ClockView             *m_pClock;
public:
  QRect clientRect;
  QRect clockRect;
  QRect calendarRect;
  DoraPushButton *buttonApply;
  DoraPushButton *buttonCancel;
};
```

Дължа да ви предупредя, че това не е целия клас, а само онази част, необходима за илюстрация на идеята, заради която съм сътворил целият този модел. Нека сега да видим и файла с имплементации - `DateAndTimeView.cpp`:

```
extern DoraMainApp *mainApp;
```

```
DoraRuntimeClass DateAndTimeView::classDateAndTimeView =
  {"DateAndTimeView", DateAndTimeView::CreateView, NULL};

DoraClassObject *DateAndTimeView::CreateView(QWidget *parent = 0)
{
  return new DateAndTimeView(parent);
}

DoraClassObject *DateAndTimeView::CreateDocument()
{
  return NULL;
}
```

Тук повече внимание ще отделя на функциите `OnCreate` и `GetDocument`, както и на обработката на съобщенията от двата бутона - `buttonApply` и `buttonCancel`. И така, ето как изглежда функцията `OnCreate`, от клас `DateAndTimeView`:

```
void DateAndTimeView::OnCreate(DoraClassObject *pObject, DoraBaseView *pView)
{
  m_pObject = pObject;
  m_pView = pView;

  DoraRuntimeClass *runtimeCalendar = RUNTIME_CLASS(CalendarView);
  DoraClassObject *objectCalendar = runtimeCalendar->CreateView(this);
  calendarView = dynamic_cast<CalendarView *>(objectCalendar);
  calendarView->OnCreate(m_pObject, this);
}
```

Така след конструирането на обекта `CalendarView`, аз му предавам указател към клас `DoraClassObject` и указател към родителския му клас - `DateAndTimeView`. След това повтарям същата манипулация с класа `ClockView`, както и с двата обекта на бутони. След това конструираме клас `DateAndTimeDocument`.

```
DoraRuntimeClass *runtimeDocument = RUNTIME_CLASS(DateAndTimeDocument);
DoraClassObject *objectDocument = runtimeDocument->CreateDocument();
DateAndTimeDocument *dateAndTimeDocument =
  dynbamic_cast<DateAndTimeDocument*>(objectDocument);
m_pDocument = dynamic_cast<DoraBaseDocument *>(dateAndTimeDocument);
```

Така в този момент аз вече имам валидни указатели към двата изгледни класа, към двата бутона и към документния клас. Сега нека направим функцията `GetDocument`.

```
DateAndTimeDocument *DateAndTimeView::GetDocument()
{
  DateAndTimeDocument *dateAndTimeDocument;
  dateAndTimeDocument = dynamic_cast<DateAndTimeDocument *>(m_pDocument);
  return dateAndTimeDocument;
}
```

Сега нека да направим взаимодействието между отделните класове. На първо място нека да опишем функцията, която ще отговаря на бутона Промени. Тя ще има задачата да прочете данните от класовете `CalendarView` и `ClockView`, след което да постави прочетените стойности в главния документен клас на приложението, за да могат те да се използват от всички други класове от приложението. За целта в базовият изгледен клас ще дефинираме и опишем две функции, едната от които е виртуална. Тези функции трябва да реагират на натискане на един от двата бутона - Промени или Откажи. Ето как ще изглеждат те:

Във файла с декларации на класа `DoraBaseView`:

```
void PostMessage(int id_message, int id_control);
virtual void PostPushButton(int id_control);
```

И във файла с имплементации:

```
void DoraBaseView::PostMessage(int id_message, int id_control)
{
  switch(id_message)
  {
  case ID_PUSH_BUTTON:
    PostPushButton(id_control);
    break;
  .
  .
  .
  };
}
void DoraBaseView::PostPushButton(int id_control)
{
}
```

Сега ще трябва да пре-дефинираме функцията `PostPushButton`, в класа `DateAndTimeView`. И във файла с декларации на този клас функцията има следния вид:

```
void PostPushButton(int id_control);
```

И нейното описание във файла с имплементации:

```
void DateAndTimeView::PostPushButton(int id_control)
{
  loadDateAndTime();
  switch(id_control)
  {
  case ID_PUSH_APPLY_BUTTON:
    PostApplyButton();
    break;
  case ID_PUSH_CANCEL_BUTTON:
    PostCancelButton();
    break;
```

```
  }
}
```

Сега трябва да дефинираме и опишем още три функции `loadDateAndTime`, `PostApplyButton` и `PostCancelButton`. Функцията `loadDateAndTime` има задачата да вземе данните от двата изгледните класове `CalendarView` и `ClockView`, след което да ги постави в документния клас `DateAndTimeDocument`. За целта ще дефинираме четири променливи от тип `CString` в този документен клас - `curentDate`, `curentTime`, `oldDate` и `oldTime`. Ето как изглежда функцията `loadDateAndTime`:

```
void DateAndTimeView::loadDateAndTime()
{
  DateAndTimeDocument *pDoc = GetDocument();
  pDoc->oldDate = "";
  pDoc->oldDate = pDoc->curentDate;
  pDoc->oldTime = "";
  pDoc->oldTime = pDoc->curentTime;

  pDoc->curentDate = "";
  pDoc->curentDate = calendarView->getData();
  pDoc->curentTime = "";
  pDoc->curentTime = clockView->getData();
}
```

Функцията `PostApplyButton`, ще вземе данните от документния клас - `DateAndTimeDocument` и ще ги постави в централния документен клас - `DoraCentralDocument`. Ето как ще направим това:

```
void DateAndTimeView::PostApplyButton()
{
  DoraCentralDocument *centralDocument = mainApp->mainFrame->getDocument();
  DataAndTimeDocument *pDoc = GetDocument();

  centralDocument->curentDate = "";
  centralDocument->curentData = pDoc->curentDate;
  centralDocument->curentTime = "";
  centralDocument->curentTime = pDoc->curentTime;
}
```

Накрая остава да опишем и функцията `PostCancelButton`. Ето как изглежда тя:

```
void DateAndTimeView::PostCancelButton()
{
  DateAndTimeDocument *pDoc = GetDocument();
  calendarView->setData(pDoc->oldDate);
  clockView->setData(pDoc->oldTime);
}
```

Какво всъщност направихме до момента - на първо място осигурихме механизъм за прехвърляне на данни от един клас в друг с помощта на документните класове и освен

това, принудихме нашите класове да си комуникират посредством съобщения. Така например функцията `loadDateAndTime`, се изпълнява при промяна на данните в класовете `CalendarView` и `ClockView`, след като се извика функцията `PostMessage` от базовия клас - `DoraBaseView`, която от своя страна извиква предефинираната функция `PostButtonMessage` от класа `DateAndTimeView`. Това стана възможно благодарение на това, че обектите на приложението се създават динамично, по време на изпълнение на приложението. Освен това даже ако премахна, отново по време на изпълнение, класа `DateAndTimeView`, данните ще се запазят в централния документен клас, от където ще могат да се прочетат. Класовете `CalendarView` и `ClockView`, нямат функция `GetDocument`, което пък ми позволява, да ги използвам в други изгледни класове, без да е необходимо да внасям в тях някакви корекции. С което създадох две блокчета, които мога да използвам във всяко мое друго приложение.

Вървейки по тази логика на разсъждение, неминуемо ще се доберем до идеята за COM, само че в рамките на едно приложение. Така ако погледнем по - широко на нещата, можем да третираме съвкупността от класовете `DoraMainFrame`, `DocraCentralView` и `DoraCentralDocument`, като едно контейнерно приложение, а класовете `DateAndTimeView`, `CalendarView`, `ClockView` и `DateAndTimeDocument`, като вътрешен компонент на приложението. Това според Microsoft е "очевидно фалшиво обстоятелство", но аз не разбирам защо да е фалшиво и при мен си е съвсем нормална ситуация. Тук ще си позволя да изкажа едно мое лично мнение, в смисъл че една от причините, които правят Windows във всичките му модификации е толкова тромав, е именно COM и OLE2, освен това идеята да се изпълняват части или цели приложения в рамките на друго приложение си е отворена задна врата на къща, пълна с хубави неща и където няма никой. Толкова за Microsoft и нещото наречено Windows.

5. Какво се постига с този програмен модел.

- Аз тук говоря за QT, която е една от най - популярните графични библиотеки в Linux, но конкретно за такъв програмен модел според мен не е особено удачна. Този програмен модел е достатъчно мобилен за да може да се използува и с други графични библиотеки.
- След като се създадат "блокчета по този програмен модел аз мога да сглобявам достатъчно бързо приложения, при това ще са достатъчно стабилни, поради факта че са във вид на source code, което би ми позволило да достигна 100% съвместимост между компонентите на дадено приложение.
- Прилагайки подхода на класическото писане на програми, аз печеля относителна независимост от хардуера на конкретната машина и от графичната среда с която работя. Това е така защото съм предоставил грижата за тези важни неща на самата графична библиотека, което не е маловажно.
- По мое скромно мнение QT като графична библиотека, достатъчно много прилича на MFC, така че не виждам защо да няма и механизъм подобен на архитектурата Document/View.

6. Посока на развитие на този модел.

Тук много ми се искаше да кажа, че ще развивам модела в посока на COM или нещо подобно, но за да стане това имам усещането, че ще трябва да се осигури среда, която да се грижи за пренасянето на обектите от едно приложение към друго. Което директно ни отвежда към вече направената графична среда Windows, от която всеки от нас повече или по - малко е имал главоболия. Така че засега ще се въздържа.

Смятам да довърша базовите си класове, изградени според този модел, както и класове като например - диалогови панели, списъчни изгледи и т.н. Също така един такъв програмен модел би бил непълен, без да има класове за многонишково програмиране и обработка на данни, Така че ще работя в тази насока.