<https://www.youtube.com/watch?v=bCppZf8FKYg>

# Model-View-Presenter Design Pattern

Пользовательские интерфейсы TouchGFX следуют архитектурному шаблону, известному как Model-View-Presenter (MVP), который является производным от шаблона Model-View-Controller (MVC). Оба они широко используются для создания приложений с пользовательским интерфейсом.

В MVP три класса определены следующим образом:

* Model (Модель) – это логический интерфейс, определяющий данные, которые будут отображаться или иным образом обрабатываться в пользовательском интерфейсе.
* View (Представление) - это пассивный интерфейс, который отображает данные (из модели) и направляет пользовательские команды (события) презентатору для выполнения действий с этими данными.
* Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

  Автоматически созданное описаниеPresenter (Ведущий) воздействует на модель и представление. Он извлекает данные из репозиториев (модели) и форматирует их для отображения в представлении.

# Модель (Model)

Класс Model — это одноэлементный класс, который всегда активен и имеет две цели:

1. Сохраняйте информацию о состоянии пользовательского интерфейса. Представления и презентаторы освобождаются при переключении экрана, поэтому их нельзя использовать для хранения информации, которая должна сохраняться при переходах экрана. Вместо этого используйте модель для этого.
2. Действует как интерфейс к серверной системе, передавая события на текущий активный экран и обратно.

Класс модели автоматически настраивается на наличие указателя на текущего активного ведущего (presenter). Когда в модели происходят изменения, текущий активный ведущий получает уведомление об изменении. Это делается с помощью методов в интерфейсе ModelListener в приложении.

Новые приложения, созданные дизайнером, автоматически получат класс модели, готовый к использованию в пользовательском интерфейсе.

Работа программиста по выводу на экран начинается с того, что определяются данные для вывода в модель. Весь обмен пользовательской программы с экраном ведётся исключительно через уровень модели. В модель будет передаваться информация для вывода на экран, из модели будет получена информация и о нажатых кнопках и иных действиях на экране.

# Представление (View)

Класс View (или, более конкретно, класс, производный от класса View TouchGFX) содержит виджеты, которые отображаются в этом представлении как объекты-члены. Он также содержит setupScreen и tearDownScreen функцию, которая автоматически вызывается при входе в этот экран / выходе из него. Обычно вы настраиваете свои виджеты в setupScreen функции.

Ваше представление также будет содержать указатель на связанный с ним презентатор. Этот указатель настраивается фреймворком автоматически. Используя этот указатель, вы можете передавать презентатору события пользовательского интерфейса, такие как нажатия кнопок.

# Ведущий (Presenter)

Ваш класс Presenter (опять же, класс, производный от класса TouchGFX Presenter) отвечает за бизнес-логику текущего активного экрана. Он будет получать "внутренние" события из модели и события пользовательского интерфейса из представления и решать, какое действие предпринять. Например, если от модели получено тревожное событие, ведущий может решить сообщить представлению, что должно быть отображено всплывающее диалоговое окно с предупреждением.

# Сгенерированный код против пользовательского кода

Код, сгенерированный TouchGFX Designer, будет полностью отделен от кода, написанного пользователем. Фактически, сгенерированный код помещается в папку generated/gui\_generated, тогда как написанный от руки код помещается в gui папку.

Изображение выглядит как текст, снимок экрана, число

Автоматически созданное описаниеgui\_generated Код служит базовыми классами для пользовательских классов кода. Базовые классы содержат весь установочный код, настроенный в TouchGFX Designer. Следующая диаграмма классов показывает взаимосвязь классов:

Как показано выше, приложения TouchGFX Designer состоят из 3 разных уровней кода:

* **Движок**: эти классы являются стандартными классами, предоставляемыми TouchGFX. Они действуют как базовые классы для созданных классов.
* **Сгенерированный**: эти классы и соответствующие файлы будут создаваться заново всякий раз, когда TouchGFX Designer генерирует код. Следовательно, эти классы и файлы не следует редактировать вручную, поскольку любые изменения, внесенные вручную, будут перезаписаны при следующем запуске генератора кода. Эти классы являются базовыми классами для пользовательских классов.
* **Пользователь**: эти классы предназначены для рукописного кода. Пользователь может свободно помещать любой код в этот слой. Пользовательские классы будут сгенерированы, если они отсутствуют, но никогда не будут изменены разработчиком TouchGFX. Они *принадлежат пользователю*.

# Файлы, созданные TouchGFX Designer

Как правило, TouchGFX Designer только повторно создает файлы в generated папке проекта TouchGFX, и поэтому важно, чтобы вы не редактировали их вручную, поскольку они будут перезаписаны. Однако TouchGFX Designer также может генерировать недостающие файлы, необходимые для компиляции, например, application.config, simulator/main.cpp изображения обложек, расположенные в assets/images/\_\_designer, и примеры видеороликов, расположенные в assets/videos/\_\_designer. На самом деле TouchGFX Designer требуется только следующее, чтобы иметь возможность генерировать, компилировать и запускать проект:

* .touchgfx Файл, описывающий проект
* Пользовательский код (если есть), расположенный в gui папке
* Пользовательские изображения (если таковые имеются), расположенные в assets/images папке
* Тексты (если таковые имеются), расположенные в assets/texts/texts.xml файле
* Шрифты (если таковые имеются), расположенные в assets/fonts папке
* Видеоролики (если таковые имеются), расположенные в assets/videos папке

# Принцип программирования

Информация для вывода на экран должна быть сохранена в глобальной переменной, к которой имеют доступ пользовательские программы, а затем нужно уведомить Модель о том, что есть готовое сообщение для вывода на экран.

Модель в момент, определённый графическим синхросигналом (tick), проверит наличие готового сообщения. О наличии готового сообщения может сообщить двоичный семафор. Тогда Модель запустит функцию обработки готового сообщения.

Функция обработки сообщения передаст управление слушателю модели (model listener), который передаст управление Ведущему (presenter).

Далее ведущий уже вызовет Вид (view), который и выведет сообщение на экран. Вид вызовет функцию invalidate, чтобы сообщить функции регенерации экрана о необходимости обновления картинки указанного фрейма (области экрана).

Такой порядок вывода сообщения на экран обеспечит логическую целостность и корректность функции вывода изображения при изменении как графики элементов, так и логики графического интерфейса в виде количества и порядка вызовов экранов и кнопок.

# Пример программы

Пример программы вывода на экран данных, поступающих по последовательному каналу. <https://www.youtube.com/watch?v=bCppZf8FKYg>

## TouchGFX

Для вывода данных используется текстовое поле, в котором включается wildcard – подстановочный символ (шаблон), в который, собственно, и будет выводиться информация из пользовательской программы. Нужно использовать буфер для вывода строки символов, при этом размер буфера должен быть хотя бы на один байт больше максимальной длины выводимой строки, поскольку последний символ вывода строки обязательно должен быть нулём.

В параметрах текста для шаблона будет использоваться тип default. В типе default можно установить фильтр Wildcard Ranges для ограничения выводимых символов теми, что попадут в установленный диапазон. Обычно используется диапазон от 0х20 до 0х7Е? Который включает в себя все полезные символы таблицы ASCII.

Генерим код, запускаем CubeIDE.

## CubeMX – файл \*.ioc

Входим в Connectivity и включаем нужный нам USART в асинхронном режиме и устанавливаем для него галочку для глобального прерывания.

Переходим в Middleware, в настройки FREERTOS, чтобы:

* Добавить задачу taskUartMsg с точкой входа StartTaskUartMsg.
* Добавить двоичный семафор binarySemUartMsg

Генерим код из CubeMX.

## CubeIDE

Для приёма данных по последовательному каналу создадим программу polling routine отдельным файлом PollingRoutings.c и PollingRoutings.h.

В PollingRoutings.c создадим две подпрограммы, которые будут запускаться из из main.c:

* Void PollingInit() {}
* Void PollingRoutings() {}

Обе функции объявим в хидере PollingRoutings.h. Поскольку эти функции будут вызываться из main.c, который создаётся исключительно на языке С, в хидере надо сообщить компилятору, что обе функции нужно компилировать С-компилятором. Для этого обозначим функции как extern “C”. Здесь, же внутри блока extern “C”, нужно определить размер буфера: #define UART\_BUF\_SIZE 16.

Возвращаемся в PollingRoutings.c, копируем в неё функцию обработки прерывания USART из драйвера (..drivers/STM32F4xx\_HAL\_Driver/Src/stm32f4xx\_hal\_uart.c). Функцию обработчика прерывания HAL\_UART\_RxCpltCallback копируем без ключа weak.

Добавляем в файл PollingRoutings.c #include с нужными хидерами и библиотеками:

* Main.h
* PollingRoutings.h
* String.h
* Cmsis\_os.h

А также добавляем ссылки на внешние идентификаторы порта и семафора:

* Extern UART\_HandleTypeDef
* Extern osSemaphoreID

Нужно определить переменные в формате uint8\_t:

* Массив uartMsgBuf[UART\_BUF\_SIZE]
* Массив uartMsgData[2]
* msgIndex
* msgRdyFlag

## Задача taskUartMsg

В файле main.c нужно при запуске задачи taskUartMsg добавить PollingInit(), но только перед запуском основного цикла. А вот внутри основного цикла задачи поставить вызов PollingRoutings().

PollingInit() должна содержать разрешение глобального прерывания порта после приёма одного байта информации.

В тело HAL\_UART\_RxCpltCallback () вставляем текст побайтной обработки прерывания. В конце обработки каждый раз устанавливаем разрешение прерывания. В конце приёма байта устанавливаем флаг готовности сообщения msgRdyFlag.

В тело PollingRoutings() устанавливаем ожидание освобождения семафора xSemaphoreGive. Пока владелец семафора занимает общий ресурс, семафор занят, задача будет стоять в ожидании, не потребляя ресурсы процессора. Как только семафор освободится, задача отдаст семафор и продолжит выполнение. Основная работа задачи – это ожидание семафора и сброс флага msgRdyFlag. Подпрограмма обеспечивает передачу информации в Модель.

## Модель (Model)

Откроем программу модели – TouchGFX/gui/src/model/Model.cpp.

В Model.cpp нужно включить библиотеку cmsis\_os.h и переменную extern osSemaphoreID.

1. Класс Model имеет tick() функцию, которая автоматически вызывается в каждом кадре и может быть реализована для поиска событий из других подмодулей и реагирования на них.
2. Класс Model содержит указатель на вашего текущего активного ведущего, чтобы иметь возможность уведомлять пользовательский интерфейс о входящих событиях.

Основной цикл работы модели обеспечивает функция Model::tick(). В тело подпрограммы включена обработка ожидания семафора xSemaphoreTake, как только семафор будет передан, подпрограмма заберёт его и начнёт выполнение. Тогда подпрограмма вызовет функцию uartMsgRdyModel().

Функцию Model::uartMsgRdyModel() нужно создать здесь же, в Model.cpp. Префикс Model:: означает определение функции, принадлежащей классу Model, снаружи объявления класса. Но объявить прототип функции uartMsgRdyModel() нужно, сделать это нужно в разделе public класса Model в хидере Model.hpp.

Теперь из Модели нам нужно вызвать слушателя Модели.

## Слушатель Модели (Model Listener)

Из слушателя Модели нужно передать данные ведущему (presenter), который является ведущим экрана, куда мы передаем сообщение. ModelListener является указателем на активного ведущего и используется для отправки данных ведущему. Т.е. ModelListener является переменной, содержащей ссылку - адрес начала кода активного ведущего. Таким образом будет запущена функция, описанная в нужном ведущем. Тем ведущим, которым не нужна данная функция, нужно либо иметь пустое тело функции с таким же названием, либо вообще не иметь этой функции.

Откроем хидер слушателя модели – TouchGFX/gui/include/model/ModelListener.hpp. Добавим функцию uartMsgRdyPresenter() в разделе public класса ModelListener. Функция должна быть описана как виртуальная, потому что она будет переопределена в хидере ведущего, поэтому и имя функции такое же, как имя функции в хидере ведущего.

Возвращаемся в Model.cpp и в теле внешнего определения функции класса Model::uartMsgRdyModel() пишем вызов функции из слушателя Модели:

ModelListener->uartMsgRdyPresenter().

Это действительно хитрый ход, поскольку код вызова функции из Модели всегда один и тот же, а вот ссылка на активный экран подставляется в ModelListener ведущим (presenter), и определяется логикой графического интерфейса. Таким образом, вызываемая функция запустится в ведущем именно того экрана, который определен логикой интерфейса.

Для справки.

* Указатель – переменная, значением которой является адрес ячейки памяти. То есть указатель ссылается на блок данных из области памяти, причём на самое его начало. Принцип объявления указателей такой же, как и принцип объявления переменных. Отличие заключается только в том, что перед именем ставится символ звёздочки \*.
* Доступ к члену класса member объекта Class описывается точкой:  
  … Class.member …;
* Доступ к члену класса member через указатель ptrPointer на объект Class описывается стрелкой:   
  \*ptrPointer = &Class;  
  … ptrPointer->member …;

## Ведущий (Presenter)

Для каждого экрана имеется свой ведущий. Сколько экранов, столько и ведущих.

В нужном ведущем нужно прописать функцию uartMsgRdyPresenter().

Откроем программу ведущего – TouchGFX/gui/src/screen1\_screen/Screen1Presenter.cpp.

Теперь уже в теле программы ведущего пишем (определяем) функцию Screen1Presenter::uartMsgRdyPresenter(). Прототип этой функции размещаем в заголовочном файле Screen1Presenter.hpp в теле объявления класса Screen1Presenter в разделе public. Функция объявляется виртуальной, поскольку она переопределяет функцию, описанную в ModelListener. То есть функция uartMsgRdyPresenter() вызывается через ссылку в ModelListener   
ModelListener->uartMsgRdyPresenter().

Чтобы вызвать функцию активного представления из активного ведущего, используется ссылочный тип данных view. В классе активного ведущего определяется переменная ссылочного типа, которая определяет ссылку view на класс активного представления. В дальнейшей работе программы для обращения к членам класса представления будет использоваться инициированная переменная ссылочного типа.

Далее ведущий вызывает функцию в представлении и через неё передает данные, полученные от модели, в представление для вывода на экран.

view.uartMsgRdyView();

Таким образом из ведущего переходим к представлению.

## Представление (View)

Для каждого экрана имеется своё представление. Сколько экранов, столько и представлений.

Программа представления расположена в одной директории с ведущим: TouchGFX/gui/src/screen1\_screen/Screen1View.cpp.

Включаем в него #include

* string.h
* PollingRoutings.h

Описываем ссылку на внешний массив uint8\_t uartMsgBuf[UART\_BUF\_SIZE].

В теле программы представления пишем (определяем) функцию Screen1View::uartMsgRdyView(), а прототип этой функции размещаем в заголовочном файле Screen1View.hpp в теле объявления класса Screen1View в разделе public.

Внутри функции Screen1View::uartMsgRdyView() реализуем:

* Проверку первого символа массива uartMsgBuf на ноль. Если ноль – сообщения нет, массив пустой, выполнение прерывается.
* Очистку буфера шаблона текстового поля textArea1 на экране - заполнение нулями массива textArea1Buffer.
* копирование содержимого массива uartMsgBuf в textArea1Buffer, который принадлежит текстовому полю textArea1, с помощью функции unicod::strncpy().
* запись в последний байт массива textArea1Buffer[16] значения ноль 0х00.
* Исполнение функции textArea1.invalidate() для обновления фрейма на экране.

Собственно, в представлении view и заключается вывод на экран. На экране отображается буфер текстовой переменной. В настройках текстового поля в TouchGFX Designer нужно включить шаблон WILDCARD и буфер для него. Имя буфера формируется автоматически из имени текстового поля с добавлением окончания Buffer, например, обращение к буферу текстового поля textArea1 производится по имени textArea1Buffer.

В настройках текстового поля в TouchGFX Designer нужно назначить в поле Wildcard Ranges разрешённые для вывода на экран диапазоны символов и можно добавить к ним отдельные символы в поле Wildcard Characters. Нераспознанные символы будут на экране обозначаться символом, установленным в поле Fallback Characters.

## Цепочка передачи информации на экран

Модель либо выбирает данные из серверного модуля, либо получает эти данные с помощью другой задачи.

1. Модель отправляет данные ведущему.
2. Ведущий далее отправляет эти данные в представление.
3. Представление обновляет данные в пользовательском интерфейсе.

Примерно так это выглядит в программе:

* Model::tick() запускает Model::uartMsgRdyModel()
* Model::uartMsgRdyModel() запускает функцию, размещённую в активном ведущем ModelListener->uartMsgRdyPresenter()
* Предыдущая строка означает вызов функции в активном ведущем Screen1Presenter, поскольку ModelListener будет содержать ссылку на активный Screen1Presenter.
* Активный ведущий определяет ссылочный тип данных view, который содержит ссылку на активное представление.
* Screen1Presenter::uartMsgRdyPresenter() запускает view.uartMsgRdyView()
* Screen1View::uartMsgRdyView() реализует вывод на экран.

# Пример программы 2

Ниже будет приведен простой пример приложения, имеющего как сгенерированный, так и пользовательский код. Это должно более подробно проиллюстрировать вышеупомянутые аспекты.

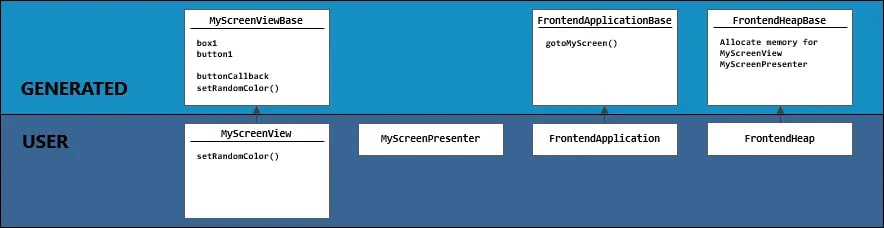
## Представление конструктора TouchGFX

В следующем примере представлен только один экран. Экран MyScreen состоит из поля box1 и кнопки button1. Мы настроили взаимодействие для вызова виртуальной функции setRandomColor() при button1 нажатии.

Всякий раз, когда мы нажимаем кнопку, мы хотели бы изменить цвет фонового поля на новый случайный цвет. Чтобы продемонстрировать, как писать свой собственный пользовательский код, мы внедрим это поведение в пользовательский код.

## Слои

Различные классы, задействованные в этом примере, можно увидеть ниже:



Мы видим, что:

* MyScreenViewBase, FrontendApplicationBase и FrontendHeapBase находятся в сгенерированном пространстве, подразумевая, что:
  + Они будут обновляться всякий раз, когда вносятся изменения в TouchGFX Designer
  + Пользователь не должен вручную редактировать эти классы
* MyScreenViewMyScreenPresenter, FrontendApplication и FrontendHeap создаются в пространстве пользовательского кода, что означает, что:
  + Они не будут восстановлены при внесении изменений в TouchGFX Designer
  + Пользователь может свободно добавлять сюда собственный код
* Вся настройка box1 и button1 выполняется в сгенерированном базовом классе view MyScreenViewBase.
* Все функции для перехода между экранами находятся в сгенерированном базовом классе приложения FrontendApplicationBase.
* Весь бухгалтерский учет, обеспечивающий выделение нужного объема памяти, находится в сгенерированном базовом классе heap FrontendHeapBase.

Пользователь может свободно редактировать классы пользовательского кода. Например, вы могли бы добавить больше виджетов. Сейчас мы просто реализуем setRandomColor функцию для фактического изменения цвета box1.

## Код

Просматривая базовый код представления, мы видим настройки окна и кнопки, созданные TouchGFX Designer. Мы также видим настройку и вызов виртуальной функции setRandomColor в buttonCallbackHandler, но на данный момент эта функция ничего не делает:

##### MyApplication/generated/gui\_generated/src/my\_screen/MyScreenViewBase.cpp

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*  
*/\*\*\*\*\*\*\*\*\*\* THIS FILE IS GENERATED BY TOUCHGFX DESIGNER, DO NOT MODIFY \*\*\*\*\*\*\*\*\*\*\*/*  
*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*  
#*include* <gui\_generated/myscreen\_screen/MyScreenViewBase.hpp>  
#*include* <touchgfx/Color.hpp>  
#*include* "BitmapDatabase.hpp"  
  
MyScreenViewBase::MyScreenViewBase() :  
 buttonCallback(*this*, &MyScreenViewBase::buttonCallbackHandler)  
{  
 box1.setPosition(0, 0, 800, 480);  
 box1.setColor(touchgfx::Color::getColorFromRGB(255, 255, 255));  
  
 button1.setXY(155, 106);  
 button1.setBitmaps(touchgfx::Bitmap(BITMAP\_BLUE\_BUTTONS\_ROUND\_EDGE\_SMALL\_ID), touchgfx::Bitmap(BITMAP\_BLUE\_BUTTONS\_ROUND\_EDGE\_SMALL\_PRESSED\_ID));  
 button1.setAction(buttonCallback);  
  
 add(box1);  
 add(button1);  
}  
  
*void* MyScreenViewBase::setupScreen()  
{  
  
}  
  
*void* MyScreenViewBase::buttonCallbackHandler(*const* touchgfx::AbstractButton& src)  
{  
 *if* (&src == &button1)  
 {  
 *//Interaction1*  
 *//When button1 clicked call virtual function*  
 *//Call setRandomColor*  
 setRandomColor();  
 }  
}

Просматривая файл заголовка для базового класса view, мы видим объявление setRandomColor и инструкцию по переопределению и реализации функции в пользовательском коде:

##### Мое приложение/ сгенерированное/gui\_generated/src/my\_screen/MyScreenViewBase.hpp

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*  
*/\*\*\*\*\*\*\*\*\*\* THIS FILE IS GENERATED BY TOUCHGFX DESIGNER, DO NOT MODIFY \*\*\*\*\*\*\*\*\*\*\*/*  
*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*  
#*ifndef* MYSCREENVIEWBASE\_HPP  
#*define* MYSCREENVIEWBASE\_HPP  
  
#*include* <gui/common/FrontendApplication.hpp>  
#*include* <mvp/View.hpp>  
#*include* <gui/myscreen\_screen/MyScreenPresenter.hpp>  
#*include* <touchgfx/widgets/Box.hpp>  
#*include* <touchgfx/widgets/Button.hpp>  
  
*class* MyScreenViewBase : *public* touchgfx::View<MyScreenPresenter>  
{  
*public*:  
 MyScreenViewBase();  
 *virtual* ~MyScreenViewBase() {}  
 *virtual* *void* setupScreen();  
  
 */\**  
 *\* Virtual Action Handlers*  
 *\*/*  
 *virtual* *void* setRandomColor()  
 {  
 *// Override and implement this function in MyScreen*  
 }  
  
*protected*:  
 FrontendApplication& application() {  
 *return* \*static\_cast<FrontendApplication\*>(touchgfx::Application::getInstance());  
 }  
  
 */\**  
 *\* Member Declarations*  
 *\*/*  
 touchgfx::Box box1;  
 touchgfx::Button button1;  
  
*private*:  
  
 */\**  
 *\* Callback Declarations*  
 *\*/*  
 touchgfx::Callback<MyScreenViewBase, *const* touchgfx::AbstractButton&> buttonCallback;  
  
 */\**  
 *\* Callback Handler Declarations*  
 *\*/*  
 *void* buttonCallbackHandler(*const* touchgfx::AbstractButton& src);  
  
};  
  
#*endif* *// MYSCREENVIEWBASE\_HPP*

Итак, давайте сделаем именно это. Перейдите к заголовочному файлу пользовательского кода MyScreenView.hpp и переопределите функцию:

##### MyApplication/сгенерированное/gui\_generated/src/my\_screen/MyScreenView.hpp

#*ifndef* MYSCREENVIEW\_HPP  
#*define* MYSCREENVIEW\_HPP  
  
#*include* <gui\_generated/myscreen\_screen/MyScreenViewBase.hpp>  
#*include* <gui/myscreen\_screen/MyScreenPresenter.hpp>  
  
*class* MyScreenView : *public* MyScreenViewBase  
{  
*public*:  
 MyScreenView();  
 *virtual* ~MyScreenView() {}  
 *virtual* *void* setupScreen();  
 *virtual* *void* tearDownScreen();  
 *virtual* *void* setRandomColor();  
*protected*:  
};  
  
#*endif* *// MYSCREENVIEW\_HPP*

Затем нам нужно реализовать фактическое поведение setRandomColor в cpp-файле для MyScreenView:

##### MyApplication/gui/src/my\_screen/MyScreenView.cpp

#*include* <gui/myscreen\_screen/MyScreenView.hpp>  
#*include* <touchgfx/Color.hpp>  
#*include* <stdlib.h>  
  
MyScreenView::MyScreenView()  
{  
  
}  
  
*void* MyScreenView::setupScreen()  
{  
 MyScreenViewBase::setupScreen();  
}  
  
*void* MyScreenView::tearDownScreen()  
{  
 MyScreenViewBase::tearDownScreen();  
}  
  
*void* MyScreenView::setRandomColor()  
{  
 box1.setColor(touchgfx::Color::getColorFromRGB(rand()&0xff, rand()&0xff, rand()&0xff));  
 box1.invalidate();  
}