



МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Балтийский государственный технический университет «ВОЕНМЕХ» им. Д.Ф. Устинова»
(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова)

БГТУ.СМК-Ф-4.2-К5-01

Факультет

И

Информационные и управляющие системы

шифр

наименование

Кафедра

И8

Системы приводов, мехатроника и робототехника

шифр

наименование

Дисциплина

Правоведение

РЕФЕРАТ

на тему

Муниципальное право

Выполнил студент группы И-882

Коваленко Е.М.

Фамилия И.О.

РУКОВОДИТЕЛЬ

Попова Н.П.

Фамилия И.О.

Подпись

Оценка

« ____ » _____ 2019 г.

САНКТ-ПЕТЕРБУРГ

2019 г.

1 Введение

Целью данной работы стало создание функциональной и простой в использовании GUI-библиотеки на базе низкоуровневой графической библиотеке SDL второй версии. Так же, вместе с основной библиотекой, были использованы библиотеки `SDL_ttf`, `SDL_image` и `SDL_gfx`.

В основу данной работы были положены некоторые принципы фреймворка `React` для создания клиентской части веб-сайтов.

2 Техническое задание

Разработать GUI-библиотеку (GUI — графический пользовательский интерфейс) на базе библиотеки SDL2, а так же, дополнительных к ней, библиотеках SDL_*. Библиотека должна предоставлять простой путь создания окон с возможностью наследования для создания собственных классов окон. Библиотека должна предоставлять базовый набор GUI компонентов (кнопка, флажок). Библиотека должна быть построена в объектно-ориентированной парадигме.

3 Описание библиотеки

Библиотека написана на языке C++ с использованием графической библиотеки SDL2. В качестве IDE была использована Visual Studio 2019. В реализации библиотеки не были использованы специфичные возможности операционной системы Windows, что означает, что данная реализация является кроссплатформенной и может быть запущена на любой платформе поддерживаемой библиотекой SDL.

Для удобства разработки вся библиотека была распределена по отдельным папкам. Так вся библиотека расположена в папке `kit` со следующей иерархией:

```
---
- component
  - components
    - components.h
    - components.cpp
  - navigator
    - navigator.h
    - navigator.cpp
  - scroll
    - scroll.h
    - scroll.cpp
  - component.h
  - component.cpp
  - component-header.h
- event
  - event.h
- tools
  - css
    - color
      - color.h
      - color.cpp
    - utils
      - css_utils.cpp
      - utils.h
    - css.h
    - css.cpp
    - css_block.h
    - css_block.cpp
    - css_block_state.h
    - css_block_state.cpp
    - css-attributes.h
    - css-attributes.cpp
    - css-parse.h
    - css-parse.cpp
  - font
    - font.h
    - font.cpp
  - image
    - image.h
    - image.cpp
  - point
```

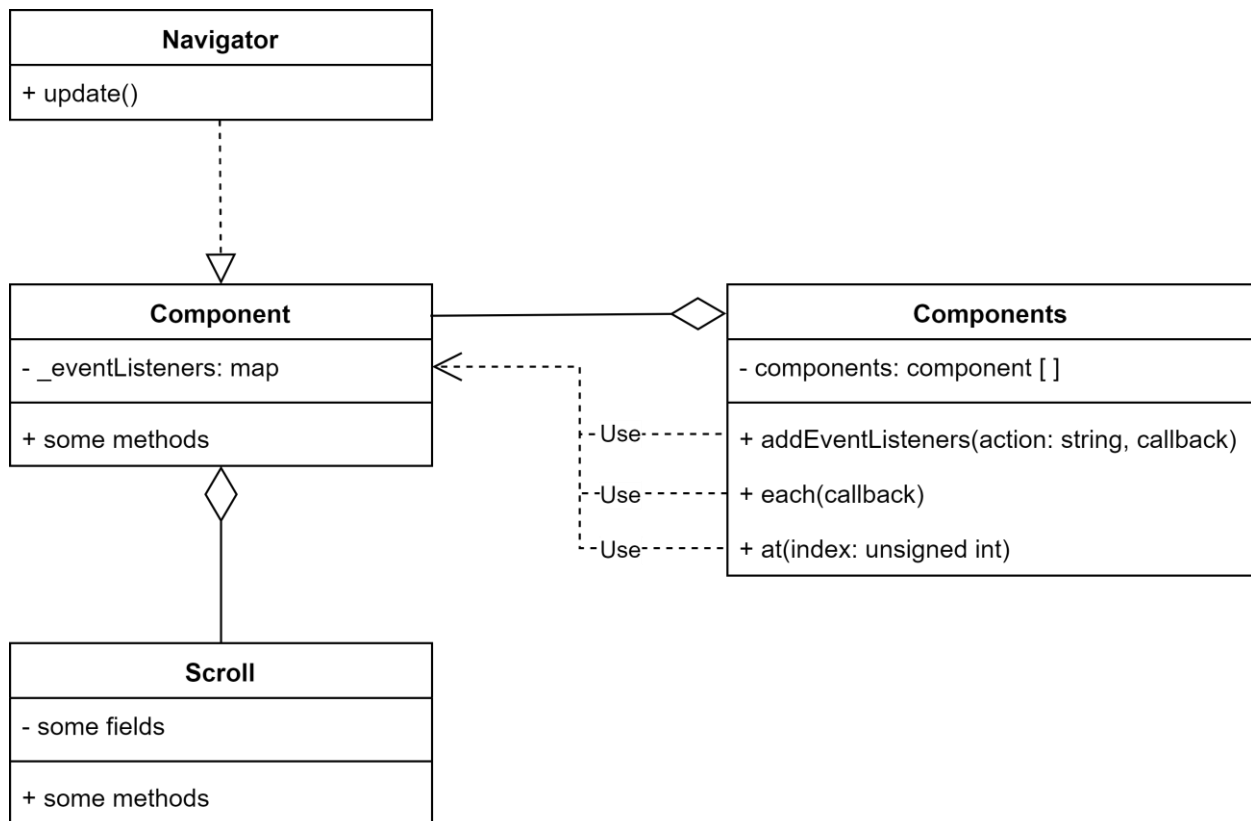
- simple-point
 - simple-point.h
- extended-point
 - extended-point.h
 - extended-point.cpp
- rect
 - simple-rect
 - simple-rect.h
 - extended-rect
 - extended-rect.h
 - extended-rect.cpp
- sdl_gfx
 - SDL2_gfxPromitives.c
 - SDL2_rotozoom.cpp
- size
 - simple-size
 - simple-size.h
 - extended-size.cpp
 - extended-size
 - extended-size.h
 - extended-size.cpp
- text
 - text.h
 - text.cpp
 - text-line.h
 - text-line.cpp
- utils
 - utils.h
 - utils.cpp
- window
 - window.h
 - window.cpp
- kit.h
- kit-main.h
- kit-main.cpp
- kit-enter-point

3.1 Описание некоторых составных блоков

3.1.1 Класс Component и его приложения

Класс Component предоставляет универсальный строительный блок интерфейса. На базе данного класса можно построить любой необходимый элемент интерфейса. Пример создания приведен в главе N. В дополнение к нему, имеются класс Components, который является оберткой над контейнером объектов класса Component для удобной работы с выборкой элементов, класс Navigator представляющий из себя класс-наследник для Component и использующийся в окне, как главный компонент, а также класс Scroll реализующий в себе логику работы скроллинга класса Component.

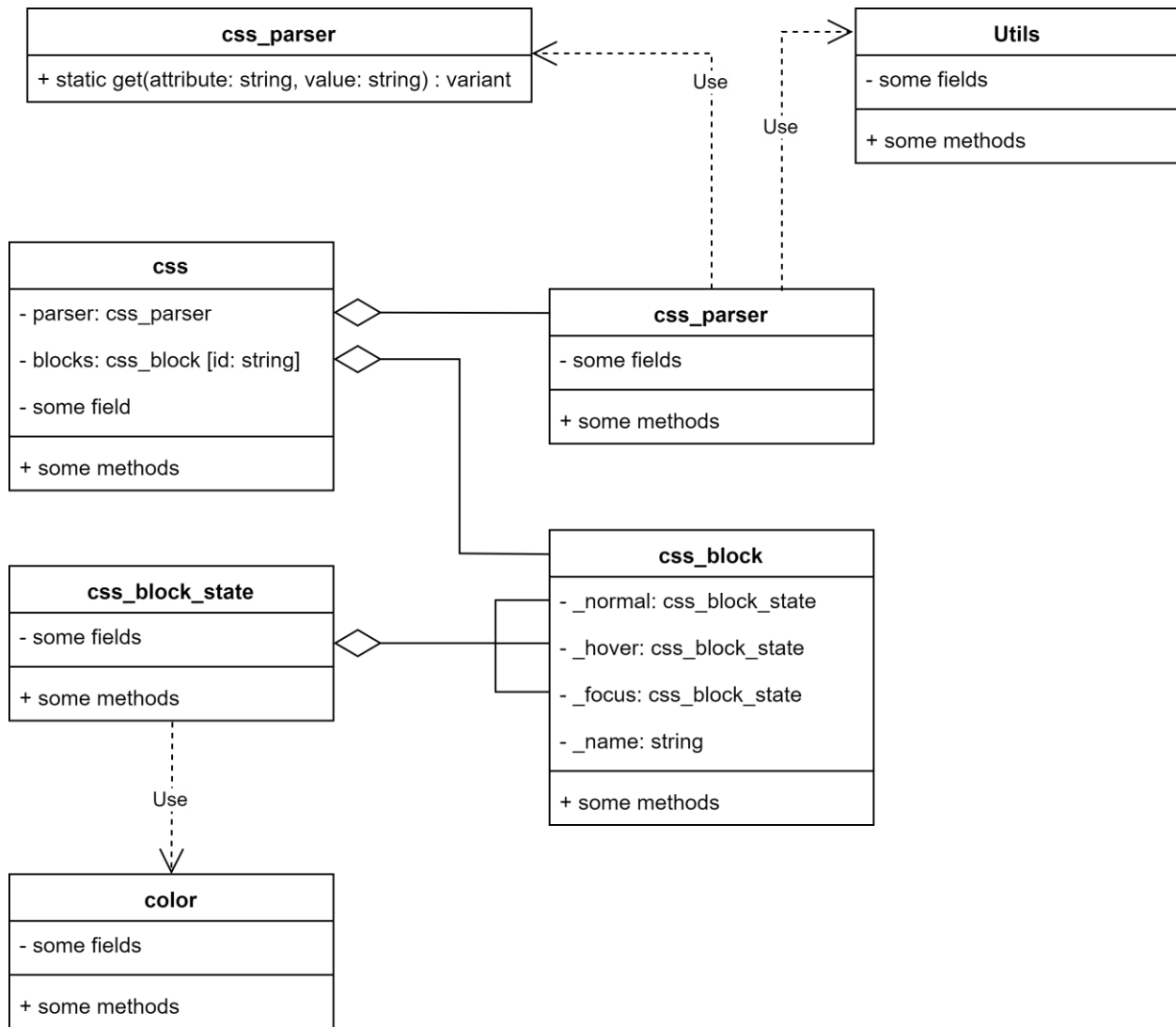
Диаграмму взаимодействий данных классов можно представить следующим образом:



3.1.2 Класс css и его приложения

Класс `css` реализует логику хранения и обработки стилей для каждого из окон. Класс `css` включает в себя класс `css_parser` реализующий логику разбора `css` файлов со стилями. Так же включает в себя ассоциативную коллекцию объектов класса `css_block` реализующий логику хранения стилей для каждого из блоков в окне. Класс `css_block` хранит в себе три возможных состояния, такие как нормальное состояние, состояния при наведении курсора мыши и состояния нажатия на элемент. Эти состояния описываются классом `css_block_state`. В дополнение для этих классов есть еще класс `css_attribute` реализующий определение и возврат нужного типа для каждого значения по его атрибуту.

Диаграмму их взаимодействия можно представить следующим образом:



3.1.3 Класс Window

Класс Window реализует логику создания окон, а так же их наполнения компонентами интерфейса. Класс является базовым, от него можно унаследоваться для создания более комплексного класса окна, или для создания собственных окон с настраиваемым макетом. Подробнее о создании собственных классов Окна на базе класса Window в главе N.

3.1.4 Дополнительный инструментарий

Для реализации тех или иных методов были созданы следующие вспомогательные классы:

- 1) Класс Font — реализует логику работы со шрифтами;
- 2) Класс Image — реализует логику работы с картинками;

- 3) Класс `Point` — реализует хранение точки в программе;
- 4) Класс `Size` — реализует хранение размеров в программе;
- 5) Класс `Rect` — реализует хранение прямоугольника в программе;
- 6) Класс `Text` — реализует логику работы с текстом;
- 7) Класс `Utils` — реализует дополнительные функции.

4 Использование библиотеки

Для использования библиотеки необходимо подключить файл `kit.h`

```
#include "kit/kit.h"
```

И для удобства прописать

```
using namespace Lib;
```

так как вся библиотека находится в пространстве имен `Lib`.

Далее, пользователь сразу получает доступ к объекту главного класса через короткое имя `$`.

Для запуска библиотеки необходимо вызвать у `$` функцию-член `run`.

Важно! Функция `main` должна принимать две переменных: `int argc`, `char** argv`

```
#include "kit/kit.h"
using namespace Lib;

int main(int argc, char** argv)
{
    $.run();
    return 0;
}
```

Здесь до вызова `run`, пользователь имеет возможность добавлять в приложение окно с помощью следующей функции-члена:

```
addWindow(Window* window);
```

Пример добавления окна:

```
#include "kit/kit.h"
using namespace Lib;

int main(int argc, char** argv)
{
    $.addWindow(new Window("new window", { 100, 100, 1000, 500 }));
    $.run();
    return 0;
}
```

Данная программа выведет пустое окно размерами 1000 на 500 пикселей с координатами 100, 100. Для добавления компонентов в окно, необходимо создать на базе класса `Window` свой класс окна.

Создадим папку `MyWindow` рядом с папкой `kit`. И создадим `MyWindow.h`.

Для наследования необходимо подключить заголовок с окном:

```
#include "../kit/window/window.h"
```

После подключения создаем пустой класс и наследуем его от Window

```
#pragma once
```

```
#include "../kit/window/window.h"
```

```
using namespace Lib;
```

```
class MyWindow : public Window
```

```
{
```

```
public:
```

```
    MyWindow(string title, SimpleRect size)  
        : Window(title, size)
```

```
{
```

```
    setup();
```

```
};
```

```
public:
```

```
    void setup()
```

```
{
```

```
}
```

```
};
```

Пока класс небольшой, реализацию можно писать в заголовочном файле для краткости.

Все что необходимо, это перегрузить конструктор и добавить функцию setup для настройки, которую надо вызвать в конструкторе. В функции setup добавляются новые компоненты интерфейса.

Теперь подключаем данный класс в main.cpp и создаем экземпляр.

```
#include "kit/kit.h"
```

```
#include "MyWindow/MyWindow.h"
```

```
using namespace Lib;
```

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

```
{
```

```
    $.addWindow(new MyWindow("new window", { 100, 100, 1000, 500 }));
```

```
    $.run();
```

```
    return 0;
```

```
}
```

Теперь перейдем к настройке интерфейса.

4.1 Простейшее применение

Изначально, пользователь может создавать только объекты базового класса `Component`. Рассмотрим класс повнимательнее.

Класс `Component` — это комплексный класс для создания любых элементов интерфейса.

В каждом окне существует специальный компонент `Navigator` который является главным для любого компонента интерфейса, и который также является классом-наследником от `Component`. В классе окна его можно использовать по короткому имени `$$` или по имени `navigator`.

Для добавления нового компонента, необходимо вызвать функцию `append` у навигатора.

Рассмотрим простейшее применение. Учтем что у нас уже есть класс `MyWindow` и будем рассматривать только функцию `setup`.

```
void setup()  
{  
    $$->append(new Component("#comp-id", { 450, 30, 200, 300 }, ".class1  
    .class2"));  
}
```

Функция `append` принимает указатель на объект на основе класса `Component`.

Класс `Component` имеет несколько конструкторов. Рассмотрим самый базовый.

Первым параметром он принимает строку-идентификатор компонента, по которому его в дальнейшем можно будет найти в окне. Важно, в окне не может быть двух компонентов с одинаковым идентификатором.

Вторым параметром идут размеры компонента. Размеры можно указывать, как только числами, так и строками вида `20px` или `20%` при этом размер в процентах будет рассчитываться относительно родительского. Так же поддерживаются записи вида `x + y` или `x - y`, например `100% - 20px` или `100% - 23%`.

Третьим параметром идет строка с набором классовых идентификаторов через пробел. Данные классовые идентификаторы могут повторяться у разных элементов и именно они являются стилевыми идентификаторами. Данная библиоте-

ка предоставляет простой путь для стилизации при помощи небольшой части языка стилей CSS. Для подключения единого для окна стилевого файла нужно в функции `setup` в самом начале функции вызвать функцию `include`

```
void include(string path);
```

и передать первым параметром путь к файлу. После этого вы можете прописывать в классовых идентификаторах необходимые вам, а в `css` файле писать для них некоторые стили.

Предположим мы создали компонент:

```
$$->append(new Component("#button", { 50, 50, 75, 25 }, ".button"));
```

и хотим его стилизовать. Создаем папку `css` в папке `MyWindow` и создаем в ней файл `style.css`.

Подключим данный файл в наше окно.

```
void setup()
{
    include("css/style.css");

    $$->append(new Component("#button", { 50, 50, 75, 25 }, ".button"));
}
```

и пропишем в `style.css` следующие:

```
.button
{
    background-color: #263238;
    border-color: #0F1518;
}
```

теперь наш компонент будет иметь новые цвета фона и обводки. Вы также можете стилизовать и `Navigator`, его классовый идентификатор равен `.navigator`. Добавим ему фон:

```
.navigator
{
    background-color: #263238;
}

.button
{
    background-color: #263238;
    border-color: #0F1518;
}
```

Вот уже у нас есть подобие кнопки. Но без текста это не кнопка. Для добавления текста необходимо вызвать функцию-член `setText` у компонента:

```
Component* setText(string text);
```

Есть два способа сделать это. Первый способ — вызвать функцию у `append`, прописав следующие:

```
$$->append(new Component("#button", { 50, 50, 75, 25 }, ".button")->setText("text");
```

Это возможно, так как функция `append` возвращает указатель на добавленный элемент.

Второй способ, это получить компонент по его идентифкатору и вызвать непосредственно у него функцию `setText`. Для этого у окна есть функция `getElementById`:

```
Component* getElementById(string id);
```

Таким образом:

```
void setup()
{
    include("css/style.css");

    $$->append(new Component("#button", { 50, 50, 75, 25 }, ".button");

    Window::getElementById("#button")->setText("text");
}
```

Первый вариант является предпочтительным, так как не несет дополнительных расходов на поиск элемента и создание компонента происходит в одном месте.

После того, как текст установлен, его нужно стилизовать, так как пока что он вглядит мягко скажем не очень.

Для стилизации текста можно использовать следующие свойства:

```
color: цвет текста (HEX обязательна полная запись);
font-size: размер текста (number + px);
line-height: междустрочный интервал (double);
text-align: выравнивание текста по горизонтали (left center right);
vertical-align: выравнивание текста по вертикали (top center bottom);
margin-top: сдвиг текста сверху (number + px);
margin-bottom: сдвиг текста снизу (number + px);
margin-left: сдвиг текста слева (number + px);
margin-right: сдвиг текста справа (number + px);
```

Добавим нашему тексту немного стилей:

```
.button
{
    background-color: #263238;
    border-color: #0F1518;
    color: #ffffff;

    font-size: 12px;
    line-height: 1.2;

    text-align: center;
    vertical-align: center;
}
```

Теперь компонент походит на кнопку еще больше. А если мы хотим чтобы при наведении меняла цвет? Для этого используется псевдокласс `hover`:

```
.button:~hover
{
    background-color: #0D1012;
    border-color: #0F1518;
    color: #ffffff;

    font-size: 12px;
    line-height: 1.2;

    text-align: center;
    vertical-align: center;
}

.button
{
    background-color: #263238;
    border-color: #0F1518;
    color: #ffffff;

    font-size: 12px;
    line-height: 1.2;

    text-align: center;
    vertical-align: center;
}
```

теперь при наведении на кнопку она изменит свой фоновый цвет.

Но пока что эта кнопка ничего не делает. Для того, чтобы добавить действия, по нажатию и не только, используется функция:

```
// callback --- function<void(Component* sender, Event* e)>
Component* addEventListener(string action, callbackEvent callback);
```

Библиотека позволяет отслеживать 7 событий:

- 1) Нажатие кнопки мыши (`onmousedown`);
- 2) Отпускание кнопки мыши (`onmouseup`);

- 3) Движение курсора мыши по элементу (mousemove);
- 4) Попадание курсора мыши на элемент (onmouseover);
- 5) Выход курсора мыши из элемента (onmouseout);
- 6) Наведение на элемент курсора (hover);
- 7) Клик по элементу (click).

Добавим прослушиватель для события `click` с помощью лямбда-функции:

```
void setup()
{
    include("css/style.css");

    $$->append(new Component("#button", { 50, 50, 75, 25 }, ".button")-
>setText("text");

    Window::getElementById("#button")->addEventListener("click",
    [](Component* sender, Event* e)
    {
        std::cout << "Button clicked" << std::endl;
    });
}
```

Для вывода не забудьте подключить библиотеку `iostream`.

Теперь при клике на вашу кнопку, в консоли будут появляться сообщения о том, что кнопка нажата.

4.2 Создание компонентов на базе класса `Component`

Рассмотрим создание собственных компонентов на базе класса `Component`. Создание собственных компонентов очень похоже на создание окон. Для начала нужно подключить в заголовке заголовочный файл базового компонента. Далее нужно унаследовать класс от класса `Component` и перегрузить конструктор. Для настройки, как и в окнах, нужно добавить функцию `setup` и вызвать ее в конструкторе.

Создадим папку `Button` рядом с папкой `MyWindow` и добавим `Button.h` и пропишем следующие:

```
#pragma once
#include "../kit/component/component.h"

using namespace Lib;

class Button : public Component
```

```

{
private:
    string text;

public:
    Button(string id, Rect size, string classes, string text)
        : Component(id, size, classes)
    {
        this->text = text;
        setup();
    }

public:
    void setup()
    {
    }
};

```

сразу же добавим поле `text` и в конструкторе так же добавим его последним параметром. Теперь в методе `setup` мы можем вызывать любые функции-члены класса `Component`. Вызовем функцию `setText` чтобы не вызывать ее при создании.

```

void setup()
{
    setText(this->text);
}

```

Простейший компонент готов. Для его подключения добавьте заголовочный файл в заголовочный файл созданного класса окна и с помощью `append` добавьте экземпляр нового компонента-кнопки.

```

#pragma once

#include "../kit/window/window.h"
#include "../Button/Button.h"
using namespace Lib;

class MyWindow : public Window
{
public:
    MyWindow(string title, SimpleRect size)
        : Window(title, size)
    {
        setup();
    };

public:
    void setup()
    {
        include("css/style.css");

        $$->append(new Button("#button", { 50, 50, 75, 25 }, ".button",
"text"));
    }
};

```



```
}; }
```

Таким образом строятся любые сложные компоненты интерфейса.