Architektura systemów komputerowych - laboratorium 3

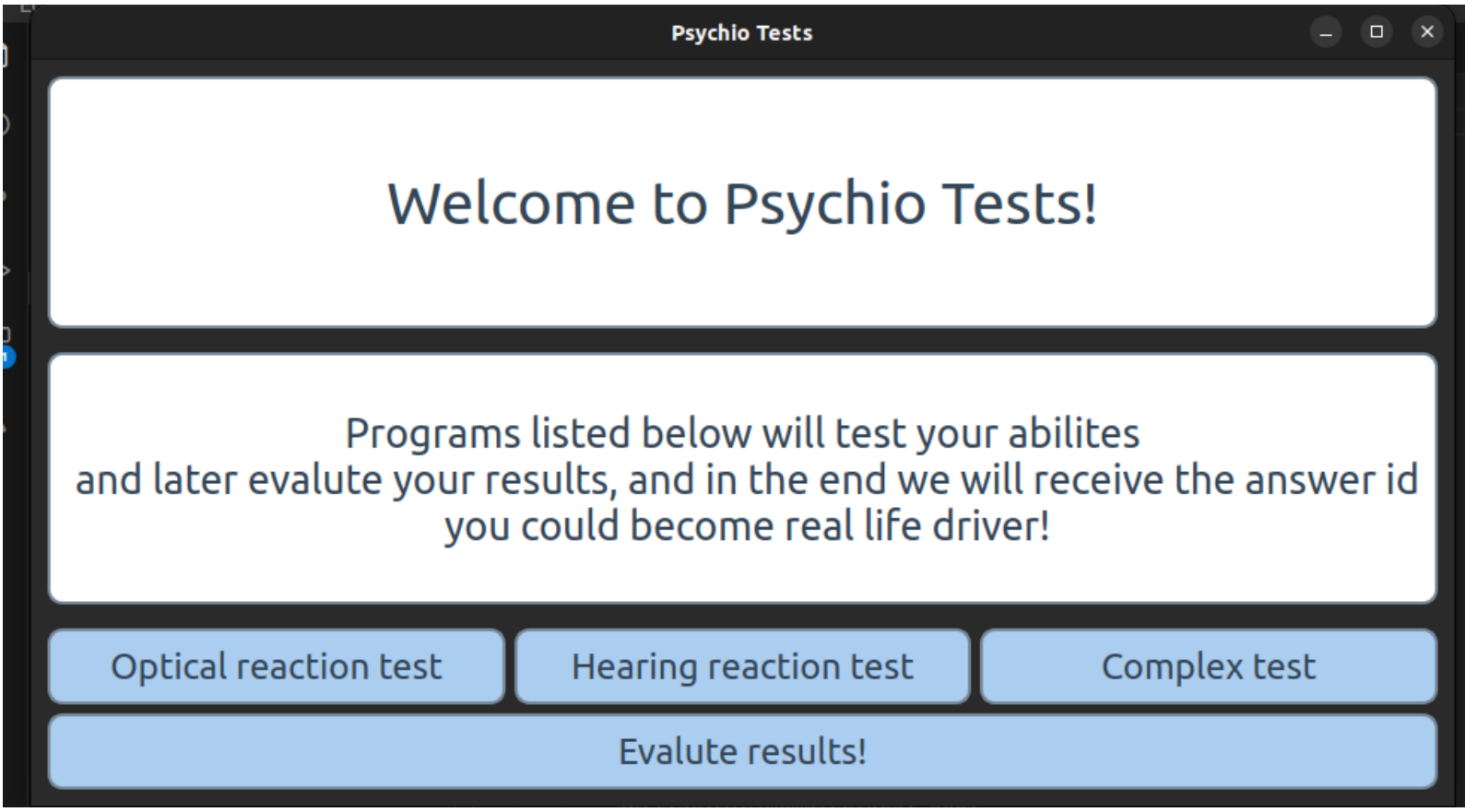
Jakub Kłopotek Główczewski 186067

Tymoteusz Byrwa 184414

Zadanie laboratoryjne polegało na stworzeniu programu który w swojej logice zawiera, odmierzanie czasu. Zatem wykonaliśmy program który testuje sprawność psychomotoryczną kierowcy. Program składa się z trzech testów oraz ewaluacji wyników na wykresie typu Polar. W każdym z testów zadbaliśmy o opcje testowania działania programu, jak było to określone w wymaganiach. Dodatkowo każdy test poprzedza krótkie wprowadzenie.

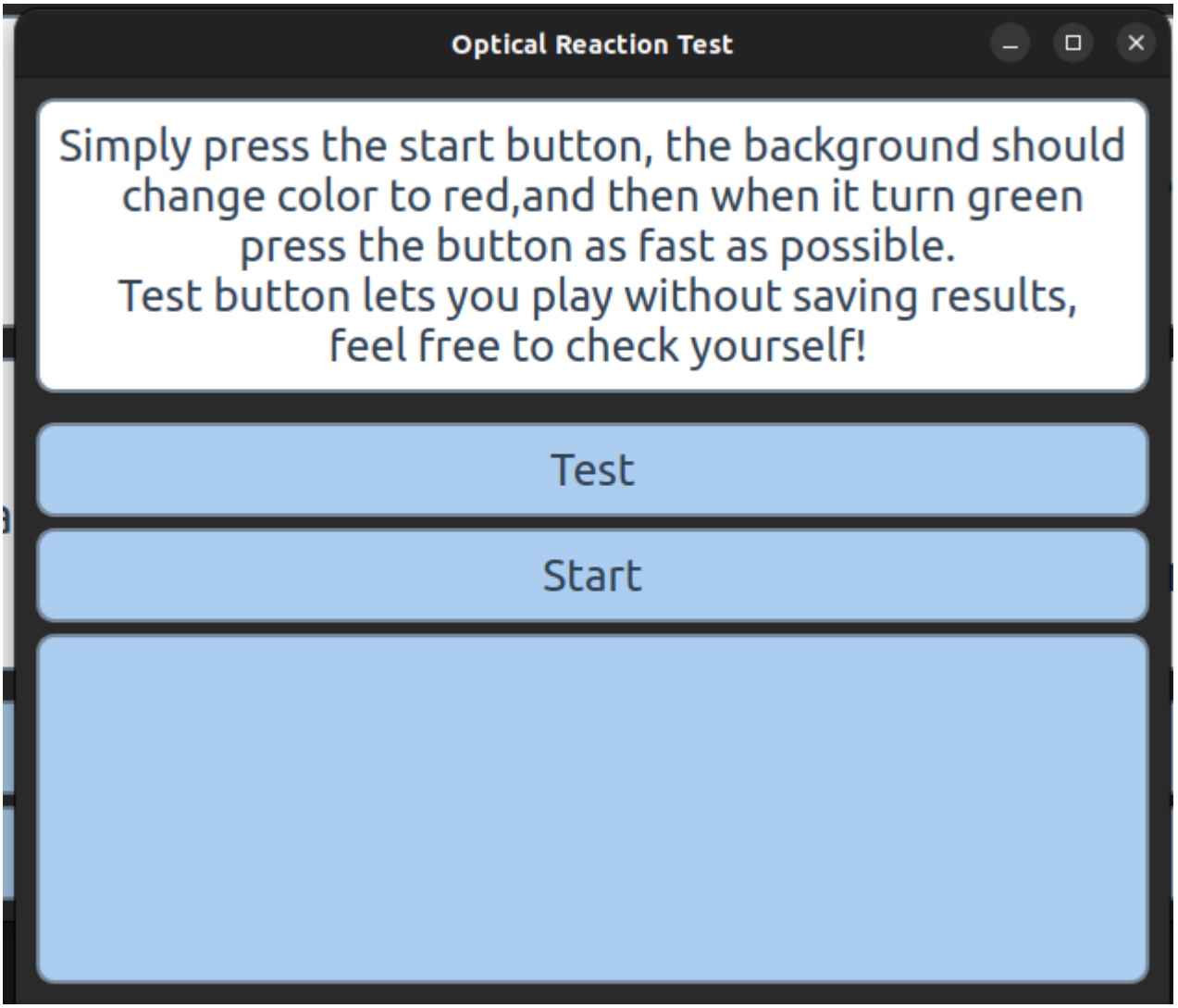
Prezentacja działania programów oraz testy:

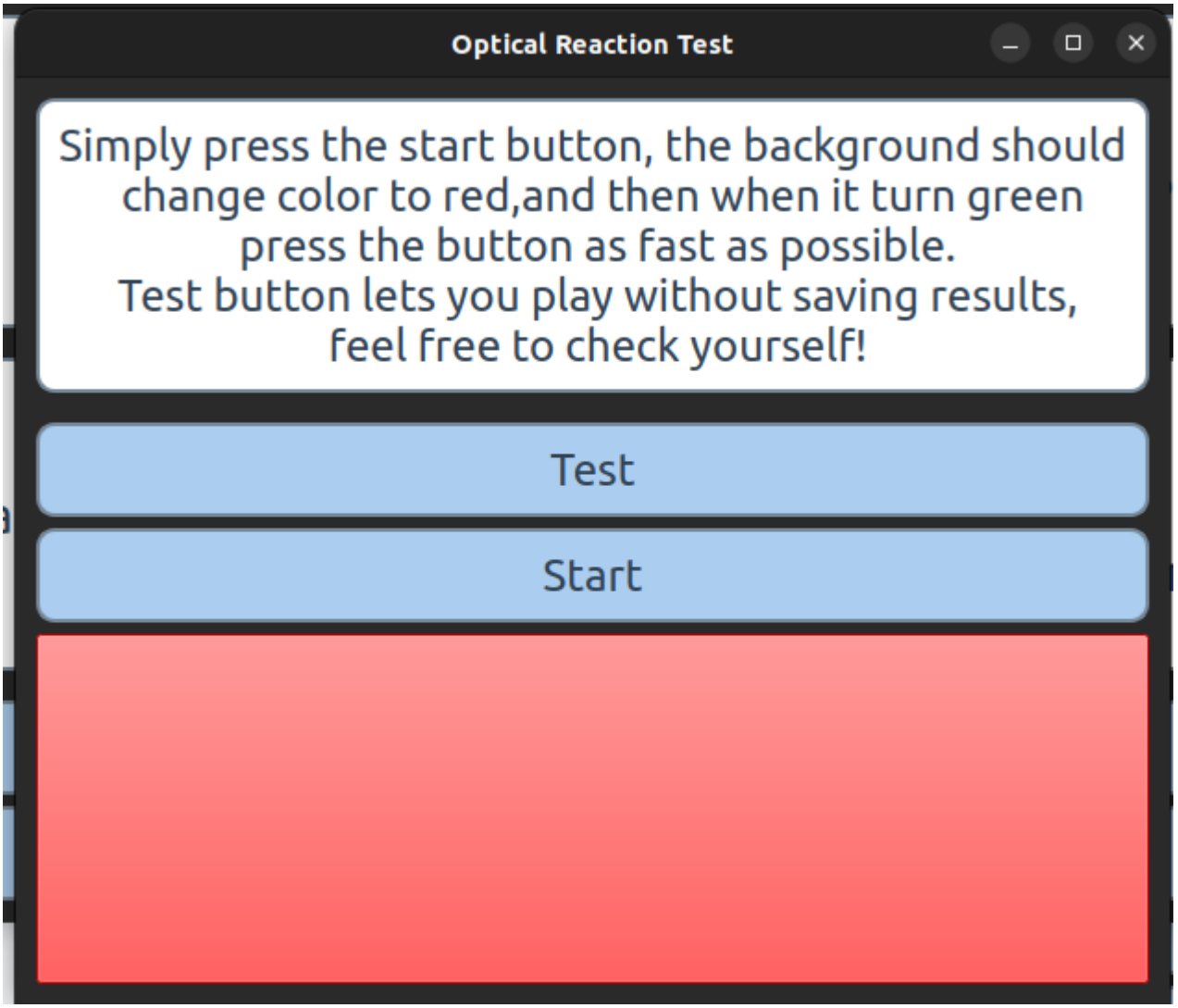
Na początku wita nas takie okno, z którego możemy przejść na dowolny inny test (w dowolnej kolejności). Zadbaliśmy również o to, że można zobaczyć wyniki w dowolnym momencie, chociaż z oczywistych przyczyn wykres będzie niepełny:



Optical reaction test:

Test opiera się na prostej zasadzie, gdy kolor zmieni się na zielony trzeba jak najszybciej nacisnąć przycisk:


Optical Reaction Test
Simply press the start button, the background should change color to red,and then when it turn green press the button as fast as possible.
Test button lets you play without saving results, feel free to check yourself!
Test
Start


Optical Reaction Test
Simply press the start button, the background should change color to red,and then when it turn green press the button as fast as possible.
Test button lets you play without saving results, feel free to check yourself!
Test
Start

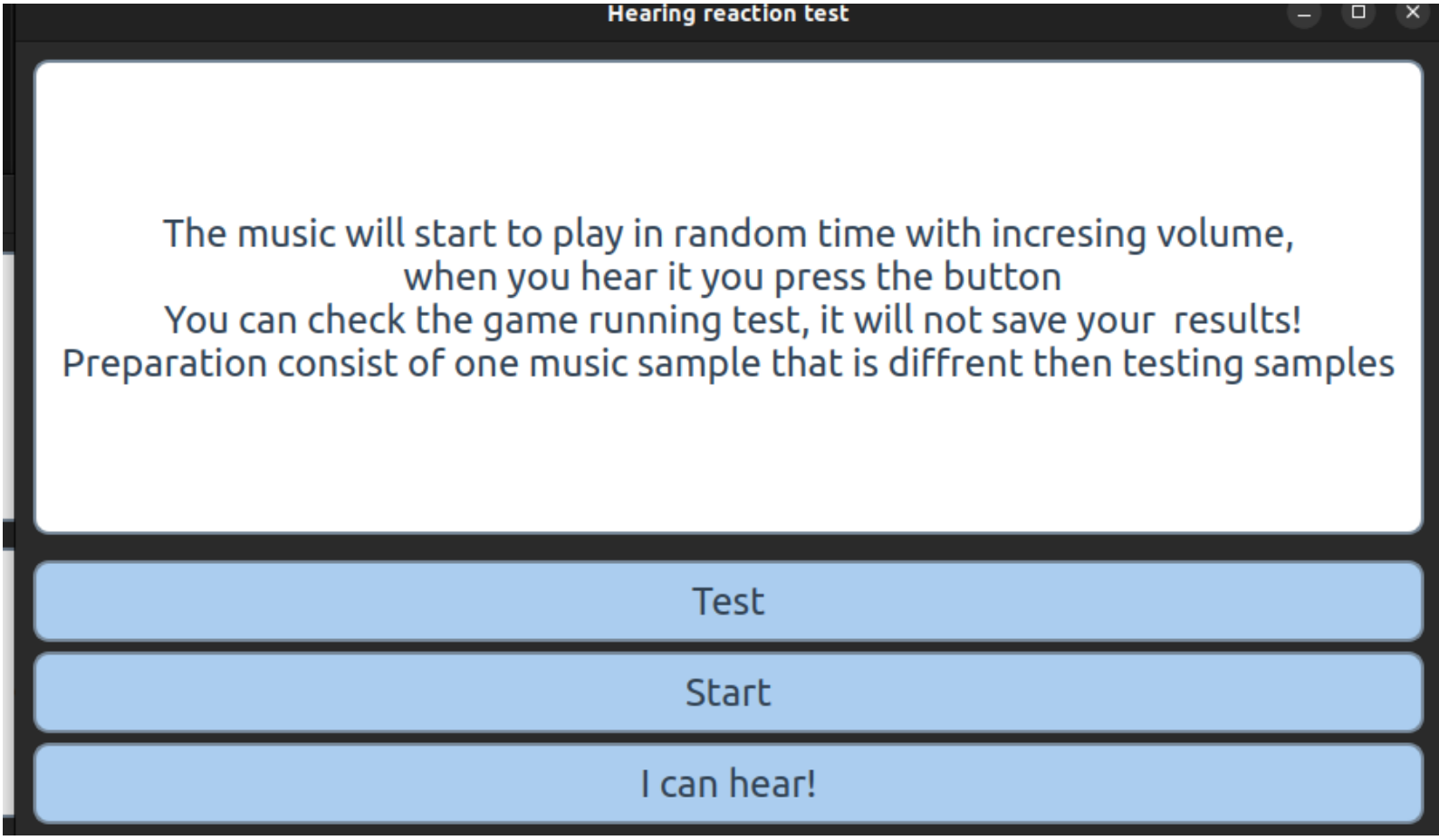
Działanie testu jest bardzo proste:

```
    def start_game(self):

        self.reaction_button.setStyleSheet("background-color: red")
        self.reaction_button.setEnabled(True)
        self.timer.start(random.randint(2000, 5000))  # waits between 2 and 5 seconds

    def change_to_green(self):
        (parameter) self: Self@Green_and_Red
        self.reaction_button.setStyleSheet("background-color: green")

    def stop_game(self):
        if self.reaction_button.styleSheet() == "background-color: green":
            self.end_time = time.time()
            if self.test_mode:
                print("Reaction Time:", self.end_time - self.start_time, "seconds")
                self.reaction_times.append(self.end_time - self.start_time)
            else:
                self.reaction_times.append(self.end_time - self.start_time)
                print("Reaction time saved!")

            if len(self.reaction_times) == 5:
                if self.test_mode:
                    print("Completed preparation, now you can approach real test!")
                    self.reaction_times=[]
                    self.test_mode=False
                else:
                    print("All tests completed please refer to main window")
                    self.test_completed.emit(self.reaction_times)
                self.reaction_button.setEnabled(False)

            elif len(self.reaction_times)!=5 or self.test_mode:
                self.start_game()
        elif self.reaction_button.styleSheet()=="background-color: red":
            print("Failed!,Don't cheat")
```

Nie będziemy szczególnie opisywać tego fragmentu, opiszemy go w Complex teście, ponieważ wszystkie trzy testy są bardzo do siebie podobne pod względem kodu.

Hearing reaction test:

Drugi test opiera się na szybkości z jaką kierowca jest w stanie usłyszeć pojawiający się dźwięk z daleka (np. karetkę). Program puszcza muzykę a następnie krokowo zwiększa jej dźwięk i czeka na reakcję kierowcy.

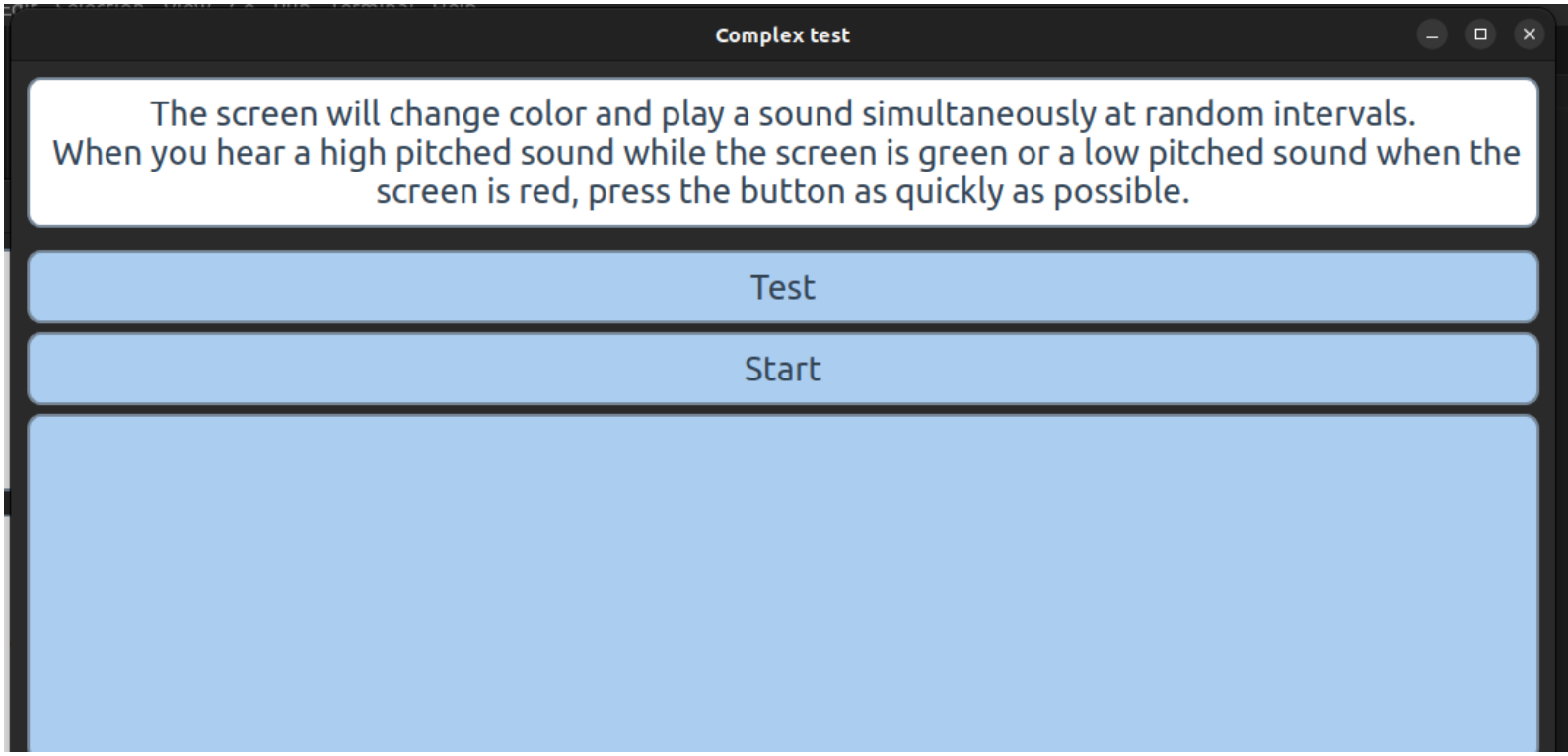


Działanie kodu:

```
def start_game(self):
    self.timer.start(random.randint(2000, 5000))  # waits between 2 and 5 seconds
def play_music(self):
    self.timer.stop()
    self.start_time=time.time()
    self.audioOutput = QAudioOutput()
    self.player.setAudioOutput(self.audioOutput)
    if self.test_mode:
        self.player.setSource(QUrl.fromLocalFile("ambulance.wav"))
    else:

        song_to_play = random.choice(self.songs)
       # print("playing:",song_to_play)
        self.player.setSource(QUrl.fromLocalFile(song_to_play))
        self.songs.remove(song_to_play)
    self.audioOutput.setVolume(0)  # start with volume 0
    self.player.play()
  #  print("starting")
    self.volume_increase_timer.start(100)  # increase volume every 1 second

def increase_volume(self):
    current_volume = self.audioOutput.volume()
    if current_volume < 1:  # 100 is the max volume
      #  print("inncresed volume")
        self.audioOutput.setVolume(current_volume + 0.01)  # increase volume by 1
    else:
        self.volume_increase_timer.stop()  # stop increasing volume once it reaches max
```

Complex test:

Trzeci test skupia się na sprawdzeniu czułości kierowcy. Otóż podczas gry gracz dostaje wartość dźwięku oraz koloru jaka sobie odpowiada, a potem musi jak najszybciej ją wybrać, jeśli kombinacja jest odpowiednia.



A teraz trochę o kodzie:

```python
#class complex test
class Complex_test(QWidget):
    #Signal to send to Main Class when test is over
    test_completed=Signal(list,int)
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Complex test")
        self.setMinimumSize(500,500)

        self.layout = QVBoxLayout()

        description_label = QLabel("The screen will change color and play a sound simultaneously at
        description_label.setAlignment(Qt.AlignCenter)
        description_label.setStyleSheet("""
            font-size: 18pt;
            color: #334455;
            border: 2px solid #778899;
            border-radius: 10px;
            padding: 10px;
            margin-bottom: 10px;
            background-color: #ffffff;
        """)
```

(variable) description_label: QLabel

Tworzymy klasę, tworzymy sygnał (w innych testach robimy dokładnie to samo) który wyśle do naszego głównego okna wartości które potem wysyłamy do klasy która przedstawia wyniki. Poza tym tworzymy napisy, okno, oraz ustawiamy styl na nasze napisy.

```
    #counter of wrong answers
    self.wrong_option=0

    self.timer = QTimer()
    self.timer.timeout.connect(self.play_sound_and_change_color)
    self.timer2=QTimer()
    self.timer2.timeout.connect(self.timer_timeout)
    self.player = QMediaPlayer()
    self.audioOutput = QAudioOutput()
    self.player.setAudioOutput(self.audioOutput)
    self.sounds = {"high": "high_pitch.wav", "low": "low_pitch.wav"}

    self.colors = {"high": "green", "low": "red"}
    self.sound_combination = None
    self.start_time = 0
    self.end_time = 0
    self.complex_reaction_times=[]
    self.test_mode=False
```

Tutaj tworzymy dwa Timery (jeden do wybrania losowego czasu kiedy zacznie program działać, a drugi do odliczania czasu kiedy już program działa- by go przerwać jeśli kombinacja sie nie zgadza) Dodatkowo tworzymy odpowiednie elementy do puszczania dźwięku oraz ustalamy kombinacje dźwięków oraz kolorów w słowniku.

```
def timer_timeout(self):
    self.timeout=True
    self.player.stop()
    self.reaction_button.setStyleSheet(f"background-color: white")
    self.timer2.stop()
    self.check_reaction()
    #play music and initialize game after timer runs out
def play_sound_and_change_color(self):
    self.timer.stop()
    #second timer to stop if combinations do not match
    self.timer2.start(5000)
    self.start_time=time.time()
    self.sound_combination = random.choice(list(self.sounds.keys()))
    self.player.setSource(QUrl.fromLocalFile(self.sounds[self.sound_combination]))
    self.color_combination=random.choice(list(self.colors.keys()))
    self.reaction_button.setStyleSheet(f"background-color: {self.colors[self.color_combi
    self.reaction_button.setEnabled(True)
    self.player.play()
    self.start_time=time.time()
#starting game
def start_game(self):
    self.reaction_button.setStyleSheet("background-color: white")
    self.timer.start(random.randint(2000, 5000))
#for test mode
```

Zaczynając od dołu w start_game startujemy timer który wywoła funkcję play_sound_and_change_color(), gdzie startujemy kolejny timer (do odmierzania czasu na ruch gracza), oraz wybieramy kombinacje kolorów oraz dźwięków. Puszczamy wszystko i czekamy na ruch gracza.

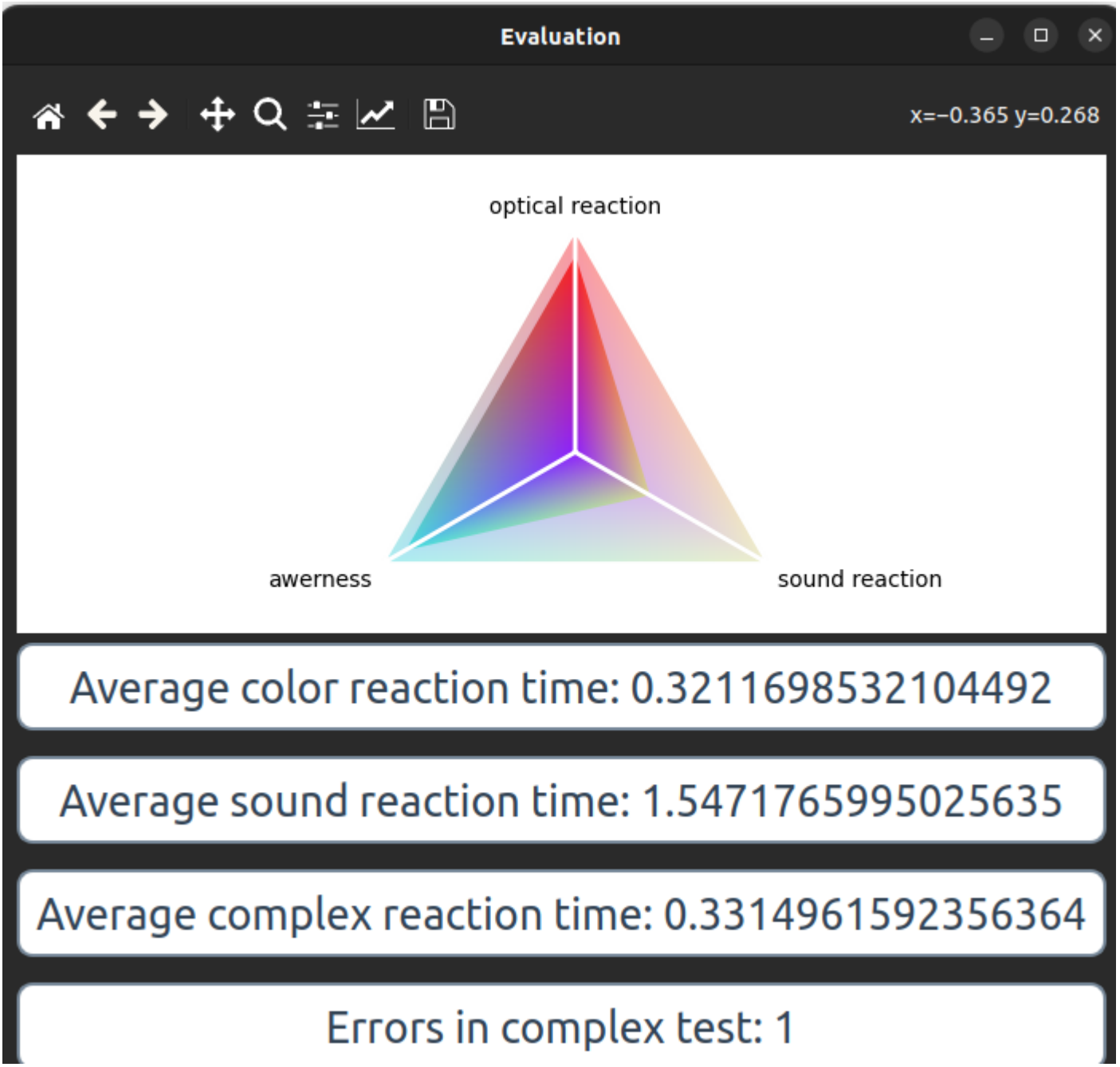
```python
def check_reaction(self):
    if self.sound_combination==self.color_combination and self.timeout==False:
        self.player.stop()
        self.end_time=time.time()
        self.timer2.stop()
        print("you are correct!")
        print("reaction time saved!")
        self.complex_reaction_times.append(self.end_time-self.start_time)
    elif self.sound_combination!=self.color_combination and self.timeout==False:
        self.player.stop()
        self.end_time=time.time()
        self.timer2.stop()
        print("you are incorrect!")
        self.wrong_option+=1
    elif self.timeout==True and self.sound_combination!=self.color_combination:
        print("you are right! Combinations do not match!")
        self.timeout=False
    elif self.timeout==True and self.sound_combination==self.color_combination:
        print("Wrong! Sound and color do match!")
        self.wrong_option+=1
        self.timeout=False
    if len(self.complex_reaction_times)<3:
        self.start_game()
    elif len(self.complex_reaction_times)==3 and self.test_mode==False:   #if len is enoguh then
        print("All tests completed, please refer to main page!")
        self.test_completed.emit(self.complex_reaction_times,self.wrong_option)
        self.reaction_button.setEnabled(False)
        self.reaction_button.setStyleSheet("background-color: black")
    elif len(self.complex_reaction_times)==3 and self.test_mode==True:
        print("Prepartaion completed! Now apporach test!")
        self.test_mode=False
        self.complex_reaction_times=[]
        self.reaction_button.setEnabled(False)
```

(method) def stop() -> None

W tej części kodu sprawdzamy posunięcie gracza oraz przypisujemy mu odpowiedni wynik. Oczywiście zatrzymujemy wszystkie timery oraz playery. Dodatkowo na dole tego bloku kodu sprawdzamy czy mamy wystarczającą ilość danych do ewaluacji. Jeśli tak to kończymy test,

Jak widać w kodzie, tryb testowy jest to po prostu tryb normalnej gry ale z zmienna boolowską test_mode, która wpływa na zachowanie programu na sam koniec gry.
W każdych z trzech testów mechanika testowa, timerowa oraz zmiany kolorów działa praktycznie tak samo więc jej nie będziemy opisywać.

Ewaluacja wyników:
Wyniki przedstawiamy jak pisaliśmy wyżej na Polarnym wykresie oraz numerycznie. Poniżej przykładowe wartości testu jakie udało nam się uzyskać:



Tutaj troszkę o kodzie:

```
proportions,avg_clr,avg_snd,avg_cplx = self.calculate_proportions(process_color_results,sound_times,errors,complex_times)
print(proportions)
labels = ['optical reaction', 'sound reaction', 'awerness']
N = len(proportions)
proportions = np.append(proportions, 1)
theta = np.linspace(0, 2 * np.pi, N, endpoint=False)
x = np.append(np.sin(theta), 0)
y = np.append(np.cos(theta), 0)
triangles = [[N, i, (i + 1) % N] for i in range(N)]
triang_backgr = tri.Triangulation(x, y, triangles)
triang_foregr = tri.Triangulation(x * proportions, y * proportions, triangles)
cmap = plt.cm.rainbow_r
colors = np.linspace(0, 1, N + 1)

self._static_ax = static_canvas.figure.subplots()
self._static_ax.tripcolor(triang_backgr, colors, cmap=cmap, shading='gouraud', alpha=0.4)
self._static_ax.tripcolor(triang_foregr, colors, cmap=cmap, shading='gouraud', alpha=0.8)
self._static_ax.triplot(triang_backgr, color='white', lw=2)
for label, color, xi, yi in zip(labels, colors, x, y):
    self._static_ax.text(xi * 1.05, yi * 1.05, label,
                         ha='left' if xi > 0.1 else 'right' if xi < -0.1 else 'center',
                         va='bottom' if yi > 0.1 else 'top' if yi < -0.1 else 'center')
self._static_ax.axis('off')
self._static_ax.set_aspect('equal')
```

Ten kod od górnej linijki liczy wartości znormalizowane (punkty za nasze odpowiedzi) które następnie są rysowane na Polarnym grafie którego kod jest od razu pod nimi.

Natomiast numeryczne przedstawienie jest napisane tutaj (od razu pod kodem na wykres):

```
color_label = QLabel(f"Average color reaction time: {avg_clr}")
color_label.setAlignment(Qt.AlignCenter)
color_label.setStyleSheet(label_style)
layout.addWidget(color_label)

sound_label = QLabel(f"Average sound reaction time: {avg_snd}")
sound_label.setAlignment(Qt.AlignCenter)
sound_label.setStyleSheet(label_style)
layout.addWidget(sound_label)

complex_label = QLabel(f"Average complex reaction time: {avg_cplx}")
complex_label.setAlignment(Qt.AlignCenter)
complex_label.setStyleSheet(label_style)
layout.addWidget(complex_label)

error_label = QLabel(f"Err (variable) label_style: Literal['\n            font-size: 20pt;\n            color: #…']
error_label.setAlignment(Q
error_label.setStyleSheet(label_style)
layout.addWidget(error_label)
```

Kod na obliczanie punktów:

```
def calculate_proportions(self, process_color_results, sound_times, errors, complex_times):
    # Define maximum possible values
    max_time = 2.5  # Assuming 2.5 sec as the maximum potential time

    # Check if the user didn't play the game (None case), then set the normalized values to 0
    if process_color_results is None:
        norm_clr = 0
        avg_clr = 0  # Define default average in case of None
    else:
        avg_clr = sum(process_color_results) / len(process_color_results)
        norm_clr = 1 - (avg_clr / max_time)

    if sound_times is None:
        norm_snd = 0
        avg_snd = 0  # Define default average in case of None
    else:
        avg_snd = sum(sound_times) / len(sound_times)
        norm_snd = 1 - (avg_snd / max_time)

    if complex_times is None or errors == 0: # Also handle case when errors is 0 to prevent division by zero
        norm_cplx = 0
        avg_cplx = 0  # Define default average in case of None or errors being 0
    else:
        avg_cplx = (sum(complex_times) / len(complex_times)) / (errors+1)
        norm_cplx = 1 - (avg_cplx / max_time)

    return [norm_clr, norm_snd, norm_cplx], avg_clr, avg_snd, avg_cplx
```

Jest to zwykłe odjęcie od wartości sufitowej oraz podzielenie przez nią.

Odnośnie naszego projektu, ciężko w nim znaleźć jakieś większe problemy, natomiast można zadać sobie pytanie czy takie testy są wystarczające by określić sprawność kierowcy? Kolejnym problemem jest fakt normalizacji wyników na punkty, ponieważ nie posiadamy jakiś większych danych naszych wyników więc nasze punkty przyznawane są, tak jak uważamy i nie są poparte żadna merytoryczną metryką.