Dokumentacja projektu:
“Symulator automatu biletowego”

Opracowane przez:

Michał Raźny, Piotr Pindel

ver. 0.2

Spis treści:

Wstępny opis działania symulatora 3

Wymagania funkcjonalne 3

Wymagania niefunkcjonalne 5

Diagram przypadków użycia 5

Etap 2 - zrealizowane funkcjonalności 6

- **Struktura projektu** 7
- **Implementacja wyglądu** 8
- **Implementacja kontrolera** 11
- **Implementacja fabryki biletów** 13

Historia prac 14

1. Wstępny opis działania symulatora

Symulator automatu biletowego będzie miał za zadanie symulować proces zakupu biletów na przejazdy komunikacją miejską. Dostępne będą różne rodzaje biletów (podział na bilety normalne oraz ulgowe, bilety strefowe itp). Płatność za wybrane bilety będzie możliwa przy pomocy monet oraz banknotów różnych nominałów (co najmniej 5 różnych monet oraz banknotów). W przypadku napotkania błędów (np.: brak możliwości wydania reszty), symulator wyświetli odpowiedni komunikat. Każda akcja ze strony użytkownika (wrzucanie monet, dodawanie biletów) będzie wyświetlana na symulatorze. Widoczna będzie informacja o kwocie do zapłaty oraz aktualnie wrzuconej gotówce. Biletomat będzie wydawać resztę w przypadku zapłaty większej ilości gotówki.

2. Wymagania funkcjonalne

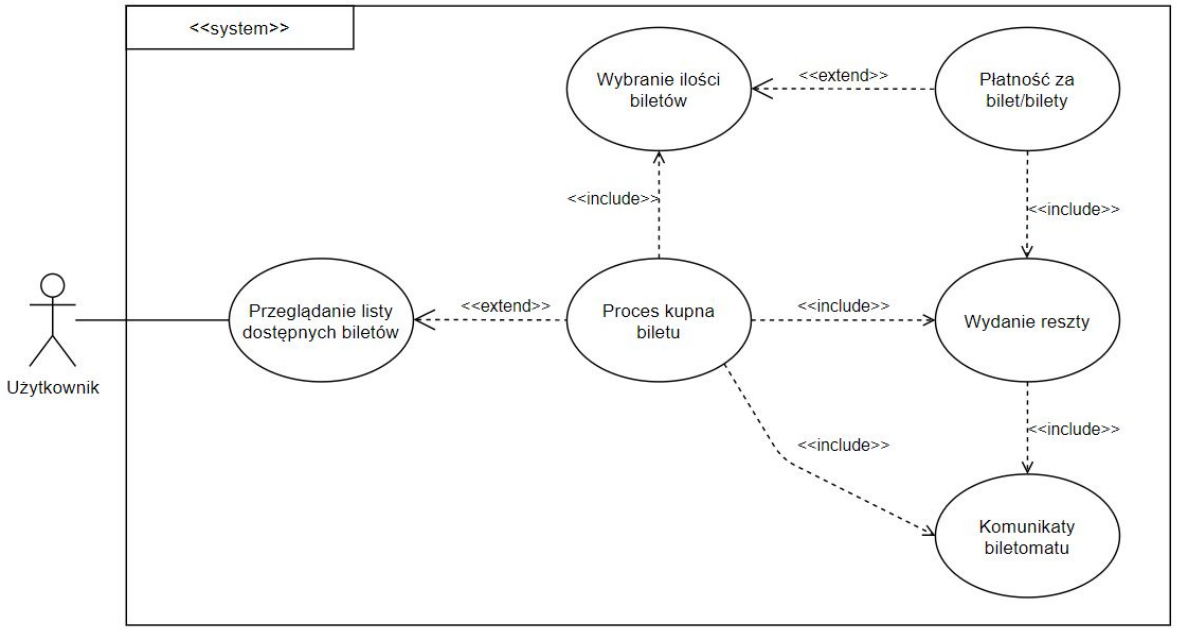
- **Możliwość zakupu biletów:**
  - Opis: Zakup wybranych biletów
  - Wejście: Wybrane bilety oraz gotówka
  - Wyjście: Kupione bilety
  - Wymagania: Odpowiednia ilość gotówki oraz wybrane bilety
- **Możliwość przeglądu dostępnych biletów:**
  - Opis: przegląd dostępnych biletów
  - Wejście: kryteria wyboru
  - Wyjście: pasujące do kryteriów bilety
  - Wymagania: wykonanie akcji dostępnych w biletomacie
- **Udostępnienie wielu rodzajów biletów (normalne, ulgowe, strefowe, czasowe):**
  - Opis: Udostępnienie kupującemu wielu rodzajów biletów
  - Wejście: podgląd biletów
  - Wyjście: bilety podzielone na wiele rodzajów

    - Wymagania: odpowiednie kryteria
- **Biletomat przyjmuje różne rodzaje monet oraz banknotów przeliczając sumę wrzuconej gotówki:**
    - Opis: Przyjmowanie gotówki
    - Wejście: Gotówka
    - Wyjście: Przyjęta określona ilość gotówki
    - Wymagania: Odpowiednie kryteria
- **Biletomat wyświetla komunikaty o błędach takich jak: brak możliwości wydania reszty lub nieodpowiedni banknot/moneta:**
    - Opis: Wyświetlanie komunikatów o błędach
    - Wejście: akcja kupującego
    - Wyjście: komunikat o napotkanym błędzie
    - Wymagania: wykonanie błędnej akcji
    - Efekty uboczne: brak możliwości kupna biletów
- **Biletomat wyświetla informacje o aktualnym stanie kupna biletów:**
    - Opis: Wyświetlenie informacji o aktualnym stanie kupna biletów
    - Wejście: akcja kupującego
    - Wyjście: informacja o aktualnym stanie kupna biletów
    - Wymagania: wykonanie odpowiedniej akcji
- **Biletomat wydaje resztę**
    - Opis: Wydanie reszty kupującemu
    - Wejście: zakup biletów
    - Wyjście: wydanie reszty
    - Wymagania: poprawny zakup biletów
- **Biletomat komunikuje pomyślne zakończenie kupna biletów:**
    - Opis: Komunikat o pomyślnym przebiegu kupna biletów
    - Wejście: Akcja kupującego (wybór biletów, wpłata gotówki)
    - Wyjście: komunikat o zakończeniu transakcji
    - Wymagania: zapłata za bilety

3. Wymagania niefunkcjonalne

- Rozszerzalność - system powinien być rozszerzalny, nastawiony na możliwość rozwijania w późniejszym czasie.
- Odporność na błędy - system powinien być odporny na błędy.
- Intuicyjność - Interfejs systemu powinien być przyjazny i intuicyjny.
- Wydajność - biletomat powinien płynnie reagować na akcje użytkownika.
- Ciągłość działania - aplikacja powinna działać poprawnie przez 95% czasu.
- Zgodność z prawem.

4. Diagram przypadków użycia

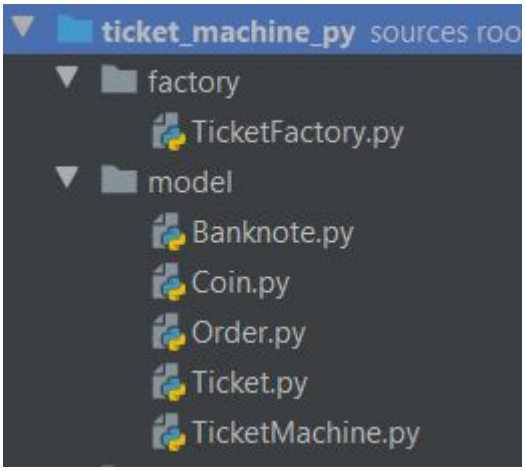


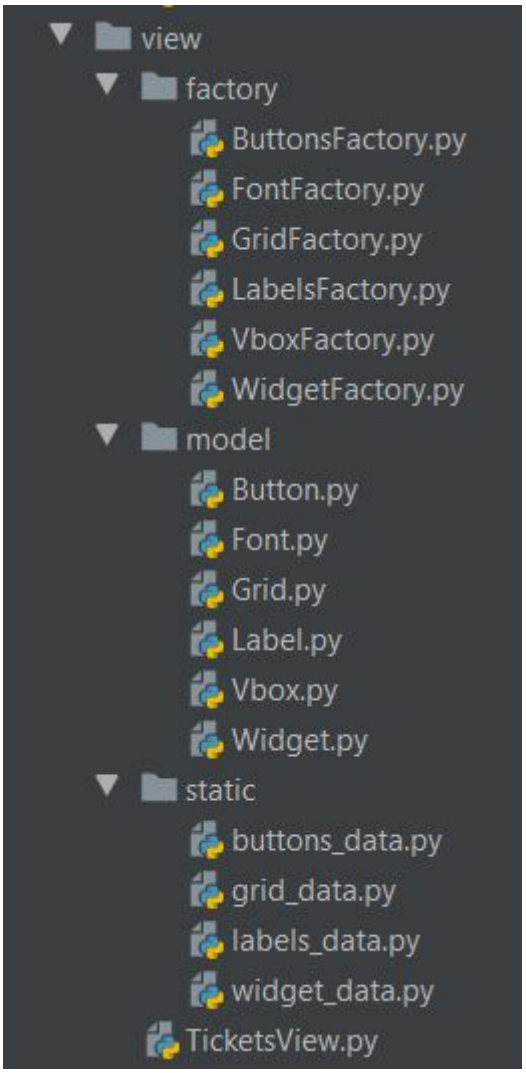
5. Etap 2 - zrealizowane funkcjonalności

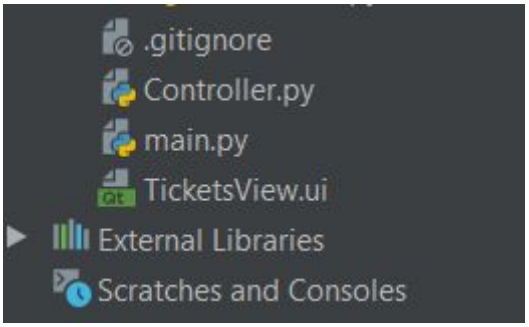
Na potrzeby etapu drugiego udało się zrealizować następujące funkcjonalności:

- Widok głównego ekranu zawierający:
  - listę wszystkich dostępnych biletów wraz z nazwami i cenami. Bilety podzielone zostały na dwie strefy (strefa I, strefa I+II) oraz dwa typy biletu (ulgowy, normalny),
  - aktualną liczbę wybranych biletów, aktualną cenę sumaryczną wszystkich wybranych biletów
  - aktualną ilość wybranych biletów danego typu
  - przycisk obsługujący przejście do płatności (implementacja w następnym etapie)
- Generowanie widoku głównego ekranu dzięki zaimplementowaniu klas dziedziczących po elementach PyQt5
- Stworzenie modeli obiektów, zarówno potrzebnych do stworzenia Biletomatu, na przykład Order lub TicketMachine, jak i obiektów graficznych, takich jak Button czy Grid.
- Zdefiniowanie statycznych elementów obiektów graficznych wraz z kluczami, dzięki którym można się do nich odnieść.
- Zrealizowanie prostych fabryk generujących elementy graficzne, dzięki odniesieniu się do wyżej wspomnianych statycznych cech.
- Zrealizowanie fabryki biletów, generującej typ biletu zależny od przekazanych parametrów.
- Kontroler obsługujący akcje wykonywane w biletomacie:
  - zliczanie sumarycznej ceny,
  - zliczanie ilości poszczególnych biletów danego typu,
  - zliczanie ilości wszystkich biletów,
  - wyświetlanie informacji o aktualnym stanie zamówienia

a. Struktura projektu







b. Implementacja wyglądu

Interfejs graficzny został zaprojektowany dzięki zaimplementowaniu klas dziedziczących po elementach PyQt5. Elementy widoku są podzielone między trzy foldery: model, factory oraz static. Cały widok jest generowany w pliku TicketsView.py.

Fragmenty pliku TicketsView.py:

```
class TicketsView(QtWidgets.QMainWindow):
    def __init__(self):
        super(TicketsView, self).__init__()
        self.buttonsFactory = ButtonsFactory()
        self.fontFactory = FontFactory()
        self.widgetFactory = WidgetFactory()
        self.gridFactory = GridFactory()
        self.labelFactory = LabelFactory()
        self.vboxFactory = VboxFactory()
        self.setupUi()
        self.show()
```

```
def generateMainLabels(self, font):
    font.setPointSize(18)
    self.label_title = self.labelFactory.createLabel(self.TicketsWidget, 'label_title', font)
    font.setPointSize(14)
    self.label_tickets_reduced = self.labelFactory.createLabel(self.TicketsWidget, 'label_tickets_reduced', font)
    self.label_tickets_normal = self.labelFactory.createLabel(self.TicketsWidget, 'label_tickets_normal', font)
    font.setPointSize(13)
    self.label_strefa_1 = self.labelFactory.createLabel(self.TicketsWidget, 'label_strefa_1', font)
    self.label_strefa_1.setLayoutDirection(QtCore.Qt.LeftToRight)
    self.label_strefa_1.setTextFormat(QtCore.Qt.PlainText)
    self.label_strefa_1.setWordWrap(True)
    self.label_strefa_2 = self.labelFactory.createLabel(self.TicketsWidget, 'label_strefa_2', font)
    self.label_strefa_2.setLayoutDirection(QtCore.Qt.LeftToRight)
    self.label_strefa_2.setTextFormat(QtCore.Qt.PlainText)
    self.label_strefa_2.setWordWrap(True)
```

Folder ‘model’ zawiera implementację modeli obiektów graficznych - Button, Font, Grid, Label, Vbox oraz Widget. Każda z klas zawiera konstruktor z wymaganymi parametrami.

Przykładowa klasa z folderu ‘model’:

Label.py

```python
from PyQt5 import QtWidgets


class Label(QtWidgets.QLabel):
    def __init__(self, parent, name, geometry, font, alignment, text):
        super(Label, self).__init__()
        self.setParent(parent)
        self.setObjectName(name)
        if geometry != 0:
            self.setGeometry(geometry)
        if font != 0:
            self.setFont(font)
        if alignment != 0:
            self.setAlignment(alignment)
        if text != 0:
            self.setText(text)
```

Folder ‘factory’ zawiera fabryki, które odpowiadają za tworzenie elementów.

Przykładowa klasa z folderu “factory”

LabelsFactory.py

```python
from view.model.Label import Label
from view.static.labels_data import labelsData

class LabelFactory:
    def createLabel(self, parent, code, font):
        return Label(parent, code, labelsData[code]['geometry'], font, labelsData[code]['alignment'], labelsData[code][
            'text'])

```

Folder ‘static’ zawiera statyczne elementów obiektów graficznych wraz z kluczami, dzięki którym można się do nich odnieść.

Przykładowa klasa z folderu ‘static’

labels_data.py

```python
from PyQt5 import QtCore

labelsData = {
    "label_title": {"geometry": QtCore.QRect(0, 0, 861, 111), "text": "--- AUTOMAT BILETOWY RAZPIN ---", "alignment": QtCore.Qt.AlignCenter},
    "label_tickets_reduced": {"geometry": QtCore.QRect(520, 140, 221, 31), "text": "ULGOWE:", "alignment": QtCore.Qt.AlignCenter},
    "label_tickets_normal": {"geometry": QtCore.QRect(80, 130, 221, 28), "text": "NORMALNE:", "alignment": QtCore.Qt.AlignCenter},
    "label_strefa_1": {"geometry": QtCore.QRect(20, 170, 21, 201), "text": "S T R E F A  I", "alignment": QtCore.Qt.AlignCenter},
    "label_strefa_2": {"geometry": QtCore.QRect(20, 390, 21, 241), "text": "S T R E F A I + II", "alignment": QtCore.Qt.AlignCenter},

    "label_total_cost": {"geometry": 0, "text": "Do zapłaty:", "alignment": QtCore.Qt.AlignCenter},
    "label_total_cost_value": {"geometry": 0, "text": "0.00", "alignment": 0},
    "label_tickets_count": {"geometry": 0, "text": "Ilość biletów:", "alignment": QtCore.Qt.AlignCenter},
    "label_tickets_count_value": {"geometry": 0, "text": "0", "alignment": 0},

    "btn_reduced_20_2_count": {"geometry": 0, "text": "0", "alignment": QtCore.Qt.AlignCenter},
    "btn_reduced_40_2_count": {"geometry": 0, "text": "0", "alignment": QtCore.Qt.AlignCenter},
    "btn_reduced_oneway_2_count": {"geometry": 0, "text": "0", "alignment": QtCore.Qt.AlignCenter},
    "btn_reduced_twoway_2_count": {"geometry": 0, "text": "0", "alignment": QtCore.Qt.AlignCenter},

    "btn_normal_20_count": {"geometry": 0, "text": "0", "alignment": QtCore.Qt.AlignCenter},
    "btn_normal_40_count": {"geometry": 0, "text": "0", "alignment": QtCore.Qt.AlignCenter},
    "btn_normal_oneway_count": {"geometry": 0, "text": "0", "alignment": QtCore.Qt.AlignCenter},
    "btn_normal_twoway_count": {"geometry": 0, "text": "0", "alignment": QtCore.Qt.AlignCenter},
```

Przykładowy schemat tworzenia nowego obiektu typu Label (możliwy do prześledzenia poprzez zrzuty ekranu):

- W pliku TicketsView.py w konstruktorze tworzony jest obiekt labelFactory typu LabelFactory (fabryka).
- Na rzecz obiektu labelFactory wywoływana jest metoda createLabel z podaniem jako argumentów rodzica (niezbędnego do wyliczenia pozycji elementu), nazwy oraz czcionki.
- Metoda createLabel tworzy nowy obiekt typu Label, podając jako argumenty rodzica, nazwę, czcionkę oraz trzy elementy statyczne odczytane z pliku labels_data.py za pomocą klucza - geometry (pozycja elementu), alignment (wyrównanie) oraz text.

c. Implementacja kontrolera

Kontroler obsługuje akcje wykonywane w biletomacie.

Konstruktor oraz niezbędne importy:

Controller.py

```python
from PyQt5.QtWidgets import QLabel

from model.Order import *
from model.TicketMachine import *
from view.TicketsView import *
from factory.TicketFactory import TicketFactory

class Controller:
    def __init__(self):
        import sys
        app = QtWidgets.QApplication(sys.argv)
        self.tickets_view = TicketsView()
        self.ticket_machine = TicketMachine()
        self.order = Order()
        self.ticketFactory = TicketFactory()
        self.assign_actions_to_buttons()
        sys.exit(app.exec_())

```

Fragment **metody assign_actions_to_buttons**, która przypisuje odpowiednie akcje do przycisków:

```python
def assign_actions_to_buttons(self):
    # Normalne I Strefa
    self.tickets_view.btn_normal_20_plus.clicked.connect(lambda: self.add_ticket(1, 2.00, 'normal', 'btn_normal_20_count', 'n20'))
    self.tickets_view.btn_normal_40_plus.clicked.connect(lambda: self.add_ticket(1, 3.60, 'normal', 'btn_normal_40_count', 'n40'))
    self.tickets_view.btn_normal_oneway_plus.clicked.connect(lambda: self.add_ticket(1, 4.00, 'normal', 'btn_normal_oneway_count', 'none'))
    self.tickets_view.btn_normal_twoway_plus.clicked.connect(lambda: self.add_ticket(1, 7.00, 'normal', 'btn_normal_twoway_count', 'ntwo'))

    self.tickets_view.btn_normal_20_minus.clicked.connect(lambda: self.remove_ticket('n20', 'btn_normal_20_count'))
    self.tickets_view.btn_normal_40_minus.clicked.connect(lambda: self.remove_ticket('n40', 'btn_normal_40_count'))
    self.tickets_view.btn_normal_oneway_minus.clicked.connect(lambda: self.remove_ticket('none', 'btn_normal_oneway_count'))
    self.tickets_view.btn_normal_twoway_minus.clicked.connect(lambda: self.remove_ticket('ntwo', 'btn_normal_twoway_count'))

    #Normalne II Strefa
    self.tickets_view.btn_normal_20_2_plus.clicked.connect(lambda: self.add_ticket(2, 2.50, 'normal', 'btn_normal_20_2_count', 'n20_2'))
    self.tickets_view.btn_normal_40_2_plus.clicked.connect(lambda: self.add_ticket(2, 4.00, 'normal', 'btn_normal_40_2_count', 'n40_2'))
    self.tickets_view.btn_normal_oneway_2_plus.clicked.connect(lambda: self.add_ticket(2, 4.80, 'normal', 'btn_normal_oneway_2_count', 'none_2
```

Metoda add_ticket, która jest wywoływana po naciśnięciu przycisku ‘+’ przy odpowiednim bilecie. Zwiększa ona licznik wybranych biletów, dodaje go do listy biletów w zamówieniu (obiekt ‘order’) oraz aktualizuje informacje dotyczące ilości biletów w całym zamówieniu oraz cenę.

```
def add_ticket(self, zone_number, price, type, label, code):
    # get label with selected tickets count
    counter = self.tickets_view.findChild(QLabel, label)
    counter_int = int(counter.text())
    counter_int += 1
    counter.setText(str(counter_int))
    # create proper ticket by ticket factory
    t = self.ticketFactory.createTicket(type, zone_number, price, code)
    self.order.add_ticket(t)
    self.update_informations()
```

Metoda remove_ticket, która jest wywoływana po naciśnięciu przycisku ‘-’ przy odpowiednim bilecie. Sprawdza ona, czy konkretny bilet znajduje się już w zamówieniu - jeśli tak, to usuwa go z listy biletów, zmniejsza licznik, oraz aktualizuje informacje dotyczące ilości biletów w całym zamówieniu oraz cenę.

```
def remove_ticket(self, code, label):
    # search for specific ticket in list
    if any(x.code == code for x in self.order.tickets):
        object_to_remove = next((x for x in self.order.tickets if x.code == code))
        self.order.tickets.remove(object_to_remove)
        self.order.cost -= object_to_remove.price
        self.order.cost = float(round(Decimal(self.order.cost), 2))
        self.update_informations()
        # reduce counter
        counter = self.tickets_view.findChild(QLabel, label)
        counter_int = int(counter.text())
        if counter_int>0:
            counter_int -= 1
        counter.setText(str(counter_int))
```

Metoda update_informations odpowiada za aktualizację informacji o zamówieniu - ilość wszystkich biletów oraz cena.

```
def update_informations(self):
    self.tickets_view.label_tickets_count_value.setText(str(len(self.order.tickets)))
    self.tickets_view.label_total_cost_value.setText(str(self.order.cost))
```

d. Implementacja fabryki biletów

Fabryka biletów generuje typ biletu w zależności od przekazanych parametrów. Zawiera jedną statyczną metodę **createTicket**, przyjmującą za argumenty typ biletu, numer strefy, cenę oraz nazwę (code).

TicketFactory.py

```python
from model.Ticket import TicketNormalZone1, TicketNormalZone2, TicketReducedZone1, TicketReducedZone2


class TicketFactory:

    @staticmethod
    def createTicket(ticket_type, zone_number, price, code):
        if ticket_type == 'normal':
            if zone_number == 1:
                return TicketNormalZone1(price, ticket_type, code)
            if zone_number == 2:
                return TicketNormalZone2(price, ticket_type, code)
        if ticket_type == 'reduced':
            if zone_number == 1:
                return TicketReducedZone1(price, ticket_type, code)
            if zone_number == 2:
                return TicketReducedZone2(price, ticket_type, code)

```

W zależności od typu oraz strefy zwraca ona nową instancję obiektu biletu, odwołując się do klasy Ticket z folderu model.

Ticket.py

```python
from abc import abstractmethod, ABCMeta


class ITicket(metaclass=ABCMeta):
    def __init__(self, price, type, code):
        self.price = price
        self.type = type
        self.code = code

    @staticmethod
    @abstractmethod
    def get_info():
        pass


class TicketReducedZone1(ITicket):

    def get_info(self):
        print("Ticket first zone | price: ", self.price, " | type: ", self.type)


class TicketNormalZone1(ITicket):

    def get_info(self):
```

Klasa Ticket zawiera implementację czterech różnych biletów. Każdy z tych klas implementuje interfejs ITicket, która dzięki zastosowaniu metody abstrakcyjnej wymusza zdefiniowanie **get_info** w każdej z nich. Zastosowanie interfejsu daje pewność, że dana metoda zostanie zaimplementowana w każdej klasie, skraca kod i zapobiega jego redundancji.

6. Historia prac

Wersja	**Osoba**	**Wykonane czynności**
0.1	Michał Raźny	● Opracowanie wstępnego opisu działania symulatora. ● Opracowanie diagramu przypadków użycia.
	Piotr Pindel	● Opracowanie wymagań funkcjonalnych i niefunkcjonalnych. ● Opracowanie diagramu przypadków użycia.
0.2	Michał Raźny	● Implementacja kontrolera oraz logiki ● Aktualizacja dokumentu na etap drugi
	Piotr Pindel	● Implementacja wyglądu ● Aktualizacja dokumentu na etap drugi