Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

**КУРСОВА РОБОТА**

з дисципліни «Об’єктно-орієнтоване програмування»

на тему

**Шаблони проектування в ООП. Графічний редкатор**

Виконав студент

ІІ курсу групи КП-73

Булаєвський Ігор Олегович

залікова книжка КП-7303

Керівник роботи

доц., к.т.н. Заболотня Т.М.

Оцінка

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(дата, підпис)

Київ-2019

**ЗМІСТ**

**ВСТУП**.........................................................................................................................3

1. **ОПИС СТРУКТУРНО-ЛОГІЧНОЇ ОРГАНІЗАЦІЇ ПРОГРАМИ**.........5
   1. Модульна організація програми............................................................5
   2. Функціональні характеристики.............................................................6
   3. Опис реалізованих класів.......................................................................7
2. **ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ЗА ДОПОМОГОЮ ШАБЛОНІВ ПРОЕКТУВАННЯ**................................................................24
   1. Обґрунтування вибору та опис шаблонів проектування для реалізації програмного забезпечення автомату..................................24
   2. Діаграма класів......................................................................................32
   3. Результати роботи програми................................................................33

**ВИСНОВКИ**..............................................................................................................35

**СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ........................…**.........................36

**ВСТУП**

Дана курсова робота присвячена розробці програмного забезпечення графічного редактора за допомогою використання шаблонів проектування. Задача моделювання програмного забезпечення графічного редактора є досить актуальною, адже графічний редактор – звична річ, яка зазвичай інстальована на всіх комп’ютерах, і для комфортної роботи користувача зі звичним інтерфейсом графічного редактора на різних платформах, необхідно розробити певні рішення, які представлять основні потоки даних зображень, з якими взаємодіють користувачі під час під час редагування улюблених фото. Дана тематика вибрана для курсової роботи тому, що результати абстрагування об’єктів у даній спроектованій системі дозволяють застосувати всі вивчені методи та принципи об’єктно-орієнтованого програмування для створення програмного забезпечення, зокрема мати змогу правильно організувати код за допомогою шаблонів проектування.

**Об’єктом** курсової роботи є *графічний редактор*.

**Метою** роботи є *розроблення програмного забезпечення* *для графічного редактора* з використанням шаблонів проектування.

Для досягнення описаної мети необхідно виконати такі **пункти завдань**:

* Абстрагувати об’єкти предметної галузі;
* Розробити структурну організацію програмного забезпечення за допомогою використання основних принципів об’єктно-орієнтованого програмування та шаблонів проектування;
* Визначити та описати функціональні характеристики програми;
* Обґрунтувати вибір шаблонів проектування;
* Розробити графічний інтерфейс користувача;
* Виконати реалізацію програмного забезпечення відповідно до технічного завдання;
* Виконати тестування розробленої програми;
* Оформити документацію з курсової роботи.

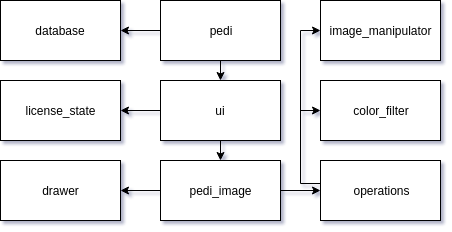
Розроблене програмне забезпечення складається з таких логічних частин: інтерфейсу користувача, накладання на зображення різних фільтрів, модуля, що відповідає за виконання над зображенням маніпуляцій зміни розміру/повороту/відображення.

Використані шаблони проектування: **Adapter, Strategy, Prototype, Proxy (**у вигляді Protection Proxy**), Flyweight,** **Command, State.**

Розроблене програмне забезпечення може бути використане як частина пре-інстальованих на ПК програм, які постачаються сумісно при покупці нового гаджета.

Пояснювальна записка складається зі вступу, двох розділів, загальних висновків та списку використаних джерел (*три* найменування). Робота містить \_\_ рисунків. Загальний обсяг роботи – \_\_ друкованих сторінок.

1. **ОПИС СТРУКТУРНО-ЛОГІЧНОЇ СХЕМИ ПРОГРАМИ**
   1. **Модульна організація програми**



*Рис. 1.1.1. Модульна організація програми*

**pedi** – точка входу в виконання програми. Забезпечує запуск графічного інтерфесу. Зв’язується з базою даних.

**ui** – модуль графічного інтерфейсу. Містить в собі класи, які є сотностями вкладок редагування зображень (застосування фільтрів, зміна параметрів зображення, зміна розміру зображення, поворот/відображення зображення).

**pedi\_image** – модуль, який працює з основною сутністю зображення, відповідає за інтерфейс для створення, копіювання, редувагування та збереженя зображення.

**drawer** – модуль, який відповідає за відображення на екрані зображення.

**operations –** модуль, який укамулює і собі усі операції з інших модулів, які дозволяють будь-яким чином редагувати зображення.

**color\_filter** – модуль, який містить у собі логіку накладання на зображення фільтрів, самостійно редагуючи пікселі зображення.

**image\_manipulator** — модуль, який забезпечеє операції над зображення, для яких використовуються посторонні бібліотеки.

**license\_state** — модуль, який описує логіку поведінки програми в залежності від стану її ліцензійної активації.

**Database** — модуль, який явлає собою сховище існуючих користувачів.

* 1. **Функціональні характеристики**

Робота з програмою розпочинається з вікна авторизації користувача. Користувач має змогу авторизуватися або пропустити цей крок.

З вікна авторизації користувач потрапляє у вікно головного інтерфейсу користувача, де зразу він може за допомогою кнопки Upload обрати зображення з файлової системи для подальшого його редагування.

Як тільки було обране перше зображення на вікні головного інтерфейсу з’являються чотири вкладки, на яких операції редагування посортовані за логічною ідентичністю:

- *FiltersTab —* на цій вкладці можна застосувати до зображення один з трьох фільтрів. Кнопки кожного фільтру містять у собі прев’ю того, як зображення виглядатиме у разу застосування фільтру до нього.

- AdjustingTab — ця вкладка містить повзунки для редагування трьох характеристик зображення: яскравість, контраст так чіткість.

- *ModificationTab —* вкладка, на якій можна змінити розмір зображення.

- *RotationTab —* вкладка, на якій користувач може повернути зображення за/против годиниковою стрілкою, відобразити зображення зліва направо/зверху вниз.

Відредагувавши сображення користувач може відмінити усі зміни натиснувши кнопку *Reset.* При натисканні кнопки *Save* існує два сценарії:

- якщо програма активована, то пропонується вибрати місце, куди зберегти зображення;

- якщо неативована, то відображається повідомлення про неможливість виконати дану операцію.

* 1. **Опис реалізованих класів**

1. **ColorFilter –** клас, який реалізує обробку зображення при накладанні на нього того чи іншого фільтру. Містить класи **SepiaFilter, BlackWhiteFilter, NegativeFilter.**

|  |
| --- |
| color\_filter.py |
| class ColorFilter(ABC):  @abstractmethod  def apply(self, pedi\_image):  pass  class SepiaFilter(ColorFilter):  def apply(self, pedi\_image):  img = pedi\_image.qimage  pix = img.load()  for i in range(img.width):  for j in range(img.height):  s = sum(pix[i, j]) // 3  k = 30  pix[i, j] = (s+k\*2, s+k, s)  class BlackWhiteFilter(ColorFilter):  def apply(self, pedi\_image):  img = pedi\_image.qimage  pix = img.load()  for i in range(img.width):  for j in range(img.height):  s = sum(pix[i, j]) // 3  pix[i, j] = (s, s, s)  class NegativeFilter(ColorFilter):  def apply(self, pedi\_image):  img = pedi\_image.qimage  pix = img.load()  for i in range(img.width):  for j in range(img.height):  pix[i, j] = (255 - pix[i, j][0], 255 -  pix[i, j][1], 255 - pix[i, j][2]) |

**2. Database** –клас для імітації роботи бази даних. Реальна БД була замінена її імітацією тому що головною задачею графічного редактора є саме редагування зображень, а не робота з користувачами.

Містить власне клас **User.**

|  |
| --- |
| database.py |
| class User:  def \_\_init\_\_(self, login):  self.login = login  def \_\_eq\_\_(self, other):  return self.login == other  class DataBase:  \_\_users = [  User('admin'),  User('tmp\_user'),  User('me')  ]  @staticmethod  def get\_user(login):  try:  index = DataBase.\_\_users.index(login)  return DataBase.\_\_users[index]  except:  return None |

**3. Drawer –** клас, який відповідає за відображення на екрані зображення різними способами. Є членом паттерну “Команда”.

|  |
| --- |
| drawer.py |
| class Drawer:  def set\_operation(self, pedi\_image, operation):  pedi\_image.operation = operation  def process(self, pedi\_image, option):  pedi\_image.redraw(option) |

**4. ImageManipulator —** клас, похідні якого утворюють повну множину усіх операцій, які можна виконати над зображеннями. Містить класи **ImageResizer , ImageResizer, ImageRotator, ImageFliper, ImageFilterer, ImageBrightener, ImageContraster, ImageSharpner.**

Є членом поттерну “Легковаговик” та “Адаптер”.

|  |
| --- |
| image\_manipulator.py |
| class ImageManipulator(ABC):  @abstractmethod  def execute(self, pedi\_image, option):  pass  class ImageResizer(ImageManipulator):  def execute(self, pedi\_image, option):  pedi\_image.qimage = pedi\_image.qimage.resize(option)  class ImageRotator(ImageManipulator):  def execute(self, pedi\_image, option):  angle = option  pedi\_image.qimage = pedi\_image.qimage.rotate(angle, expand=True)  class ImageFliper(ImageManipulator):  def execute(self, pedi\_image, option):  direction = option  pedi\_image.qimage = pedi\_image.qimage.transpose(direction)  class ImageFilterer(ImageManipulator):  \_\_filters = {  'sepia': SepiaFilter(),  'black\_white': BlackWhiteFilter(),  'negative': NegativeFilter()  }  def execute(self, pedi\_image, option):  filter\_name = option  concrete\_filter = ImageFilterer.\_\_filters[filter\_name]  concrete\_filter.apply(pedi\_image)  class ImageBrightener(ImageManipulator):  def execute(self, pedi\_image, option):  factor = option  if factor > BRIGHTNESS\_FACTOR\_MAX or factor < BRIGHTNESS\_FACTOR\_MIN:  raise ValueError("factor should be [0-2]")  enhancer = ImageEnhance.Brightness(pedi\_image.qimage)  pedi\_image.qimage = enhancer.enhance(factor)  class ImageContraster(ImageManipulator):  def execute(self, pedi\_image, option):  factor = option  if factor > CONTRAST\_FACTOR\_MAX or factor < CONTRAST\_FACTOR\_MIN:  raise ValueError("factor should be [0.5-1.5]")  enhancer = ImageEnhance.Contrast(pedi\_image.qimage)  pedi\_image.qimage = enhancer.enhance(factor)  class ImageSharpner(ImageManipulator):  def execute(self, pedi\_image, option):  factor = option  if factor > SHARPNESS\_FACTOR\_MAX or factor < SHARPNESS\_FACTOR\_MIN:  raise ValueError("factor should be [0.5-1.5]")  enhancer = ImageEnhance.Sharpness(pedi\_image.qimage)  pedi\_image.qimage = enhancer.enhance(factor) |

**5. State**  - клас, який описує поведінку програми на натисненні кнопки *Save*. В залежності від статусу активації використовується один з двох класів **LicensedState/NotLicensedState**. Є членом паттерну “Стан”.

|  |
| --- |
| license\_state.py |
| class State(ABC):  @staticmethod  def run(parent):  pass  class LicensedState(State):  @staticmethod  def run(parent):  logging.debug('on licensed save state')  path, \_ = QFileDialog.getSaveFileName(parent.parent, "QFileDialog.getSaveFileName()",  f"pedi\_{parent.file\_name}",  "Images (\*.png \*.jpg)")  return path  class NotLicensedState(State):  def run(parent):  logging.debug('on not licensed save state')  QMessageBox.warning(parent.parent, 'Error',  "Login to unlock this feature") |

**6. Operations —** клас, який є фабрикою для паттерну “Легковаговик”. Забезпечує доступ до об’єктів конкретних операцій.

|  |
| --- |
| operations.py |
| from image\_manipulator import \*  class Operations:  \_\_all\_operations = {  'resize': [ImageResizer, None],  'rotate': [ImageRotator, None],  'flip': [ImageFliper, None],  'filter': [ImageFilterer, None],  'brightness': [ImageBrightener, None],  'contrast': [ImageContraster, None],  'sharpness': [ImageSharpner, None]  }  def \_\_has\_operation(self, name):  return bool(Operations.\_\_all\_operations[name][1])  def \_\_create\_operation(self, name):  instance = Operations.\_\_all\_operations[name][0]()  Operations.\_\_all\_operations[name][1] = instance  def \_\_get\_instance(self, name):  return Operations.\_\_all\_operations[name][1]  \_\_filters\_name = ['sepia', 'black\_white', 'negative']  def get\_operation(self, name):  if (not self.\_\_has\_operation(name)):  self.\_\_create\_operation(name)  return self.\_\_get\_instance(name) |

**7. PediImage —** клас, який відповідає за основу сутність програми. Описує зображення та надає інтерфейс для роботи з ним. Є учасником паттернів “Сратегія”, “Прототип” та “Команда”. У файлі також описані клас **DisplayStrategy** та похідні від нього **LabelStrategy** та **ThumbStrategy**, які є учансниками паттерну “Стратегія” та відповідаються за способи відображення зображення на екрані.

|  |
| --- |
| pedi\_image.py |
| class DisplayStrategy(ABC):  def \_\_init\_\_(self, parent):  self.parent = parent  @abstractmethod  def display(self, pedi\_image):  pass  class LabelStrategy(DisplayStrategy):  def display(self, pedi\_image):  pixmap = ImageQt.toqpixmap(pedi\_image.qimage)  self.parent.image\_label.setPixmap(pixmap)  logging.debug('image label updated')  class ThumbStrategy(DisplayStrategy):  def \_\_init\_\_(self, parent):  super().\_\_init\_\_(parent)  self.\_\_filterer = ImageFilterer()  def \_\_display\_one(self, thumb, pedi\_image):  image\_filtered = pedi\_image.clone()  self.\_\_filterer.execute(image\_filtered, thumb.name)  preview\_pixmap = ImageQt.toqpixmap(image\_filtered.qimage)  thumb.setPixmap(preview\_pixmap)  logging.debug('%s thumb updated' % thumb.name)  def display(self, pedi\_image):  for thumb in self.parent:  thread = Thread(target=self.\_\_display\_one,  args=(thumb, pedi\_image))  thread.start()  class ICloneable():  @abstractmethod  def clone(self):  pass  class PediImage(ICloneable):  def \_\_init\_\_(self, img\_path):  pixmap = QPixmap(img\_path)  self.qimage = ImageQt.fromqpixmap(pixmap)  self.strategy = None  @property  def width(self):  return self.qimage.width  @property  def height(self):  return self.qimage.height  def set\_display\_strategy(self, strategy):  self.strategy = strategy  def redraw(self, option):  if (not self.operation):  return  self.operation.execute(self, option)  def show(self):  if (self.strategy):  self.strategy.display(self)  def clone(self):  old\_qimage = self.qimage  qimage\_copy = self.qimage.copy()  self.qimage = None  self\_copy = copy(self)  self.qimage = old\_qimage  self\_copy.qimage = qimage\_copy  return self\_copy  def save(self, new\_img\_path):  self.qimage.save(new\_img\_path) |

**8. ActionTabs –** клас, який відповідає за частину графічного інтерфуйсу, яка містить у собі вкладки з категоріями операцій над зображеннями.

|  |
| --- |
| ui.py |
| class ActionTabs(QTabWidget):  def \_\_init\_\_(self, parent):  super().\_\_init\_\_()  self.parent = parent  self.filters\_tab = FiltersTab(self)  self.adjustment\_tab = AdjustingTab(self)  self.modification\_tab = ModificationTab(self)  self.rotation\_tab = RotationTab(self)  self.addTab(self.filters\_tab, "Filters")  self.addTab(self.adjustment\_tab, "Adjusting")  self.addTab(self.modification\_tab, "Modification")  self.addTab(self.rotation\_tab, "Rotation")  self.setMaximumHeight(190) |

**9. FiltersTab** – вкладка, яка відповідає за частину графічного інтерфейсу, де відображається прев’ю зображень у разу застосування до них кольорного фільтру.

|  |
| --- |
| ui.py |
| class FiltersTab(QWidget):  def \_\_init\_\_(self, parent):  super().\_\_init\_\_()  self.parent = parent  self.main\_layout = QHBoxLayout()  self.main\_layout.setAlignment(Qt.AlignCenter)  self.thumbs = [  self.create\_filter\_thumb('sepia'),  self.create\_filter\_thumb('black\_white'),  self.create\_filter\_thumb('negative')  ]  global strategies  strategies['thumb'] = ThumbStrategy(self.thumbs)  self.setLayout(self.main\_layout)  def create\_filter\_thumb(self, name):  thumb\_label = QLabel()  thumb\_label.name = name  thumb\_label.setStyleSheet("border:2px solid #ccc;")  thumb\_label.setToolTip(f"Apply <b>{name}</b> filter")  thumb\_label.setCursor(Qt.PointingHandCursor)  thumb\_label.setFixedSize(THUMB\_SIZE, THUMB\_SIZE)  thumb\_label.mousePressEvent = partial(self.on\_filter\_select, name)  self.main\_layout.addWidget(thumb\_label)  return thumb\_label  def on\_filter\_select(self, name, e):  logging.debug('%s filter selected' % name)  global image\_preview, drawer, operations  operation = operations.get\_operation('filter')  drawer.set\_operation(image\_preview, operation)  drawer.process(image\_preview, name)  self.parent.refresh\_image() |

**10. AdjsutingTab** – вкладка, яка відповідає за частину графічного інтерфейсу, де можна змінювати яскравість, контраст та чіткість зображення.

|  |
| --- |
| ui.py |
| class AdjustingTab(QWidget):  def \_\_init\_\_(self, parent):  super().\_\_init\_\_()  self.parent = parent  contrast\_label = QLabel("Contrast")  contrast\_label.setAlignment(Qt.AlignCenter)  brightness\_label = QLabel("Brightness")  brightness\_label.setAlignment(Qt.AlignCenter)  sharpness\_label = QLabel("Sharpness")  sharpness\_label.setAlignment(Qt.AlignCenter)  contrast\_slider = QSlider(Qt.Horizontal, self)  contrast\_slider.setMinimum(SLIDER\_MIN\_VAL)  contrast\_slider.setMaximum(SLIDER\_MAX\_VAL)  contrast\_slider.sliderReleased.connect(  self.on\_contrast\_slider\_released)  contrast\_slider.setToolTip(str(SLIDER\_MAX\_VAL))  brightness\_slider = QSlider(Qt.Horizontal, self)  brightness\_slider.setMinimum(SLIDER\_MIN\_VAL)  brightness\_slider.setMaximum(SLIDER\_MAX\_VAL)  brightness\_slider.sliderReleased.connect(  self.on\_brightness\_slider\_released)  brightness\_slider.setToolTip(str(SLIDER\_MAX\_VAL))  sharpness\_slider = QSlider(Qt.Horizontal, self)  sharpness\_slider.setMinimum(SLIDER\_MIN\_VAL)  sharpness\_slider.setMaximum(SLIDER\_MAX\_VAL)  sharpness\_slider.sliderReleased.connect(  self.on\_sharpness\_slider\_released)  sharpness\_slider.setToolTip(str(SLIDER\_MAX\_VAL))  self.sliders = {  'brightness': brightness\_slider,  'contrast': contrast\_slider,  'sharpness': sharpness\_slider  }  main\_layout = QVBoxLayout()  main\_layout.setAlignment(Qt.AlignCenter)  main\_layout.addWidget(contrast\_label)  main\_layout.addWidget(contrast\_slider)  main\_layout.addWidget(brightness\_label)  main\_layout.addWidget(brightness\_slider)  main\_layout.addWidget(sharpness\_label)  main\_layout.addWidget(sharpness\_slider)  self.reset\_sliders()  self.setLayout(main\_layout)  def reset\_sliders(self):  self.sliders['brightness'].setValue(SLIDER\_DEF\_VAL)  self.sliders['sharpness'].setValue(SLIDER\_DEF\_VAL)  self.sliders['contrast'].setValue(SLIDER\_DEF\_VAL)  @staticmethod  def \_get\_converted\_point(user\_p1, user\_p2, p1, p2, x):  r = (x - user\_p1) / (user\_p2 - user\_p1)  return p1 + r \* (p2 - p1)  def \_\_adjust(self, prop):  logging.debug('on %s slider released' % prop)  slider = self.sliders[prop]  slider.setToolTip(str(slider.value()))  factor = AdjustingTab.\_get\_converted\_point(SLIDER\_MIN\_VAL, SLIDER\_MAX\_VAL, BRIGHTNESS\_FACTOR\_MIN,  BRIGHTNESS\_FACTOR\_MAX, slider.value())  logging.debug("%s factor: %f" % (prop, factor))  global image\_preview, drawer, operations  operation = operations.get\_operation(prop)  drawer.set\_operation(image\_preview, operation)  drawer.process(image\_preview, factor)  self.parent.parent.refresh\_image()  def on\_brightness\_slider\_released(self):  self.\_\_adjust('brightness')  def on\_contrast\_slider\_released(self):  self.\_\_adjust('contrast')  def on\_sharpness\_slider\_released(self):  self.\_\_adjust('sharpness') |

**11. ModificationTab** – вкладка, яка відповідає за частину графічного інтерфейсу, де можна змінювати розмір зображення у пікселях.

|  |
| --- |
| ui.py |
| class ModificationTab(QWidget):  """Modification tab widget"""  def \_\_init\_\_(self, parent):  super().\_\_init\_\_()  self.parent = parent  self.width\_label = QLabel('width:', self)  self.width\_label.setFixedWidth(100)  self.height\_label = QLabel('height:', self)  self.height\_label.setFixedWidth(100)  self.width\_box = QLineEdit(self)  self.width\_box.textEdited.connect(self.on\_width\_change)  self.width\_box.setMaximumWidth(100)  self.height\_box = QLineEdit(self)  self.height\_box.textEdited.connect(self.on\_height\_change)  self.height\_box.setMaximumWidth(100)  self.unit\_label = QLabel("px")  self.unit\_label.setMaximumWidth(50)  self.apply\_buttton = QPushButton("Apply")  self.apply\_buttton.setFixedWidth(90)  self.apply\_buttton.clicked.connect(self.on\_apply)  width\_layout = QHBoxLayout()  width\_layout.addWidget(self.width\_box)  width\_layout.addWidget(self.height\_box)  width\_layout.addWidget(self.unit\_label)  apply\_layout = QHBoxLayout()  apply\_layout.addWidget(self.apply\_buttton)  apply\_layout.setAlignment(Qt.AlignRight)  label\_layout = QHBoxLayout()  label\_layout.setAlignment(Qt.AlignLeft)  label\_layout.addWidget(self.width\_label)  label\_layout.addWidget(self.height\_label)  main\_layout = QVBoxLayout()  main\_layout.setAlignment(Qt.AlignCenter)  main\_layout.addLayout(label\_layout)  main\_layout.addLayout(width\_layout)  main\_layout.addLayout(apply\_layout)  self.setLayout(main\_layout)  def set\_boxes(self):  global image\_preview  self.width\_box.setText(str(image\_preview.width))  def on\_height\_change(self):  logging.debug('on height change')  try:  h = int(self.height\_box.text())  self.apply\_buttton.setEnabled(True)  if(h < 0 or h > 1500):  raise Exception()  logging.debug('new height value: %d' % h)  except:  self.apply\_buttton.setEnabled(False)  logging.debug('invalid height value')  def on\_width\_change(self):  try:  w = int(self.width\_box.text())  self.apply\_buttton.setEnabled(True)  if(w < 0 or w > 1500):  raise Exception()  logging.debug('new width value: %d' % w)  except:  self.apply\_buttton.setEnabled(False)  logging.debug('invalid width value')  def on\_apply(self):  logging.debug('on apply')  w = int(self.width\_box.text())  h = int(self.height\_box.text())  global image\_preview, drawer, operations  operation = operations.get\_operation('resize')  drawer.set\_operation(image\_preview, operation)  drawer.process(image\_preview, (w, h))  self.parent.parent.refresh\_image() |

**12. RotationTab** – вкладка, яка відповідає за частину графічного інтерфейсу, де можна обертати зображення та відображати його.

|  |
| --- |
| ui.py |
| class RotationTab(QWidget):  def \_\_init\_\_(self, parent):  super().\_\_init\_\_()  self.parent = parent  rotate\_left\_buttton = QPushButton("↺ 90°")  rotate\_left\_buttton.clicked.connect(self.on\_rotate\_left)  rotate\_right\_buttton = QPushButton("↻ 90°")  rotate\_right\_buttton.clicked.connect(self.on\_rotate\_right)  flip\_left\_buttton = QPushButton("⇆")  flip\_left\_buttton.clicked.connect(self.on\_flip\_left)  flip\_top\_buttton = QPushButton("↑↓")  flip\_top\_buttton.clicked.connect(self.on\_flip\_top)  rotate\_label = QLabel("Rotate")  rotate\_label.setAlignment(Qt.AlignCenter)  rotate\_label.setFixedWidth(140)  flip\_label = QLabel("Flip")  flip\_label.setAlignment(Qt.AlignCenter)  flip\_label.setFixedWidth(140)  label\_layout = QHBoxLayout()  label\_layout.setAlignment(Qt.AlignCenter)  label\_layout.addWidget(rotate\_label)  label\_layout.addWidget(flip\_label)  buttton\_layout = QHBoxLayout()  buttton\_layout.setAlignment(Qt.AlignCenter)  buttton\_layout.addWidget(rotate\_left\_buttton)  buttton\_layout.addWidget(rotate\_right\_buttton)  buttton\_layout.addWidget(flip\_left\_buttton)  buttton\_layout.addWidget(flip\_top\_buttton)  main\_layout = QVBoxLayout()  main\_layout.setAlignment(Qt.AlignCenter)  main\_layout.addLayout(label\_layout)  main\_layout.addLayout(buttton\_layout)  self.setLayout(main\_layout)  def on\_rotate\_left(self):  logging.debug('on rotate left')  global image\_preview, drawer, operations  operation = operations.get\_operation('rotate')  drawer.set\_operation(image\_preview, operation)  drawer.process(image\_preview, 90)  self.parent.parent.refresh\_image()  self.parent.modification\_tab.set\_boxes()  def on\_rotate\_right(self):  logging.debug('on rotate right')  global image\_preview, drawer, operations  operation = operations.get\_operation('rotate')  drawer.set\_operation(image\_preview, operation)  drawer.process(image\_preview, 270)  self.parent.parent.refresh\_image()  self.parent.modification\_tab.set\_boxes()  def on\_flip\_left(self):  logging.debug('on flip left')  global image\_preview, drawer, operations  operation = operations.get\_operation('flip')  drawer.set\_operation(image\_preview, operation)  drawer.process(image\_preview, Image.FLIP\_LEFT\_RIGHT)  self.parent.parent.refresh\_image()  def on\_flip\_top(self):  logging.debug('on flip top')  global image\_preview, drawer, operations  operation = operations.get\_operation('flip')  drawer.set\_operation(image\_preview, operation)  drawer.process(image\_preview, Image.FLIP\_TOP\_BOTTOM)  image\_preview.set\_display\_strategy(strategies['label'])  self.parent.parent.refresh\_image() |

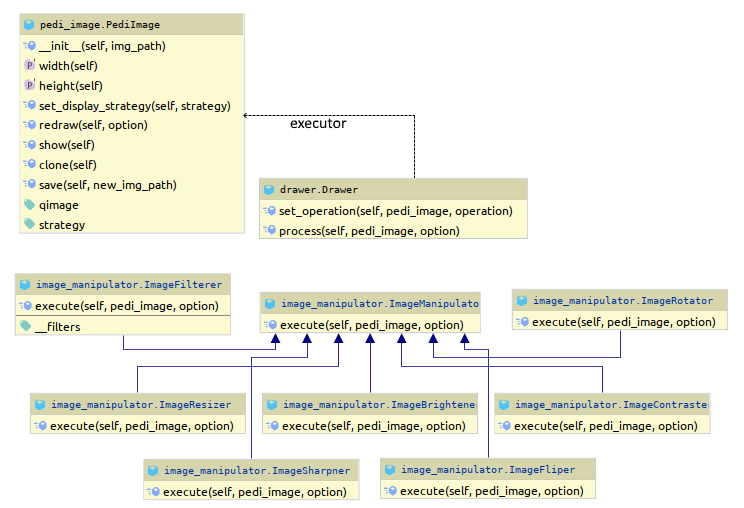
**12. MainLaoyut** – головна частина графічного інтерфейсу, яка акумулює у собі всі його складові. Чатина паттеру ”Стан” та “Заступник”, виступає у ролі об’єкту, доступ до якого треба обмежити.

|  |
| --- |
| ui.py |
| class MainLayout(QVBoxLayout):  def \_\_init\_\_(self, parent):  super().\_\_init\_\_()  self.parent = parent  self.file\_name = None  self.\_\_licensed = None  self.upload\_buttton = QPushButton("Upload")  self.upload\_buttton.clicked.connect(self.on\_upload)  self.reset\_buttton = QPushButton("Reset")  self.reset\_buttton.setEnabled(False)  self.reset\_buttton.clicked.connect(self.on\_reset)  self.save\_buttton = QPushButton("Save")  self.save\_buttton.setEnabled(False)  self.save\_buttton.clicked.connect(self.on\_save)  buttton\_layout = QHBoxLayout()  buttton\_layout.setAlignment(Qt.AlignTop)  buttton\_layout.addWidget(self.upload\_buttton)  buttton\_layout.addWidget(self.reset\_buttton)  buttton\_layout.addWidget(self.save\_buttton)  self.addLayout(buttton\_layout)  self.image\_label = QLabel()  self.image\_label.setAlignment(Qt.AlignCenter)  self.addWidget(self.image\_label)  global strategies  strategies['label'] = LabelStrategy(self)  self.action\_tabs = ActionTabs(self)  self.addWidget(self.action\_tabs)  self.action\_tabs.setVisible(False)  self.user\_label = QPushButton()  self.user\_label.setStyleSheet(  "QPushButton{font-size: 10px; color: grey;}")  self.user\_label.setFlat(True)  self.user\_label.clicked.connect(self.parent.user\_authorize)  self.addWidget(self.user\_label)  def set\_license(self, merchant, state):  if (type(merchant) is not PediUI):  raise Exception  self.\_\_licensed = state  logging.debug('license set to %s', self.\_\_licensed)  def on\_upload(self):  logging.debug('on upload buttton clicked')  img\_path, \_ = QFileDialog.getOpenFileName(self.parent, "Open image",  "/home/traumgedanken/Pictures",  "Images (\*.png \*jpg)")  self.file\_name = ntpath.basename(img\_path)  global image\_original, image\_preview  image\_original = PediImage(img\_path)  image\_preview = image\_original.clone()  self.refresh\_image()  self.reset\_buttton.setEnabled(True)  self.save\_buttton.setEnabled(True)  if (not self.\_\_licensed):  self.save\_buttton.setToolTip('Sign in to unlock this')  self.action\_tabs.setVisible(True)  self.action\_tabs.adjustment\_tab.reset\_sliders()  self.action\_tabs.modification\_tab.set\_boxes()  def on\_reset(self):  logging.debug('on reset buttton clicked')  global image\_original, image\_preview  image\_preview = image\_original.clone()  self.refresh\_image()  def on\_save(self):  logging.debug('on save buttton clicked')  new\_img\_path = LicensedState.run(  self) if self.\_\_licensed else NotLicensedState.run(self)  if new\_img\_path:  logging.debug("save output image to %s" % new\_img\_path)  global image\_preview  image\_preview.save(new\_img\_path)  def refresh\_image(self):  global image\_preview, strategies  image\_preview.set\_display\_strategy(strategies['label'])  image\_preview.show()  image\_preview.set\_display\_strategy(strategies['thumb'])  image\_preview.show()  self.action\_tabs.modification\_tab.set\_boxes() |

**12. PediUI** – клас, який служить точкою входу в виконання програми. Член паттерну “Заступник”. В своєму конструкторі дає змогу користувачу авторизуватися. Член паттернів “Стан” та “Заступник”.

|  |
| --- |
| ui.py |
| class PediUI(QWidget):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.main\_layout = MainLayout(self)  self.setLayout(self.main\_layout)  self.setMinimumSize(600, 500)  self.setMaximumSize(900, 900)  self.setGeometry(600, 600, 600, 600)  self.setWindowTitle('pedi OOP2')  self.center()  self.show()  self.user\_authorize()  def user\_authorize(self):  user = None  while(not user):  login, ok\_pressed = QInputDialog.getText(  self, "SIGN IN", "Enter your login:", QLineEdit.Normal, "")  if (ok\_pressed):  user = DataBase.get\_user(login)  else:  break  self.main\_layout.user\_label.setText(  f'licenced to: {user.login}' if user else 'not licensed')  self.main\_layout.set\_license(self, bool(user))  def center(self):  qr = self.frameGeometry()  cp = QDesktopWidget().availableGeometry().center()  qr.moveCenter(cp)  self.move(qr.topLeft()) |

1. **ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ЗА ДОПОМОГОЮ ШАБЛОНІВ ПРОЕКТУВАННЯ**
   1. **Обґрунтування вибору та опис шаблонів проектування для реалізації програмного забезпечення автомату**
2. **Command**

****

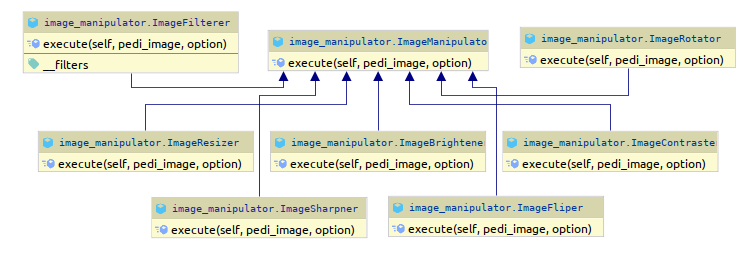
*Рис. 2.1.1. UML-діаграма шаблону* ***Command***

*Поведінковий шаблон*. Перетворює операцію в об'єкт. Це дозволяє передавати операції як аргументи при викликах методів, ставити операції в чергу, логувати їх, а також підтримувати можливість скасування операцій.

*Структура*.Реалізований для класу **PediImage**. Об’єкти цього класу мають у середині об’єкт **Operation**, і цей об’єкт слугує логічною частиною способу обробки зображення. Клас **Drawer** може застосувати операцію до забраження.

*Обґрунтування використання даного шаблону*. При написанні коду завжди слід прагнути розділення інтерфейсу та логіки. За допомогою формування такого класу і можна досягти інкапсуляції логіки в одному об’єкті.

1. **Adapter**

****

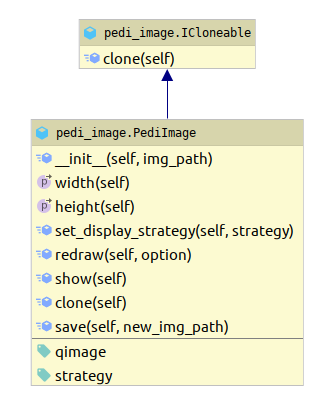
*Рис. 2.1.2. UML-діаграма шаблону “****Adapter****”*

*Структурний шаблон*. Забезпечує спільну роботу класів з несумісними інтерфейсами шляхом створення спільного об’єкта, через який вони можуть взаємодіяти.

*Структура*.Конкретні адаптери - **ImageResizer , ImageResizer, ImageRotator, ImageFliper, ImageFilterer, ImageBrightener, ImageContraster, ImageSharpner.**

*Обґрунтування використання даного шаблону*. Реалізований для того, щоб мати один інтерфейс, через який можна безпечно та зручно взаємодіяти зі всією системою операцій. Операції, які виконуються над зображеннями реалізовані різними способами, а деякі з них проводяться за допомогою посторонніх бібліотек, тому цілком доцільно було уніфікувати інтерфейс усіх операцій.

1. **Прототип**

****

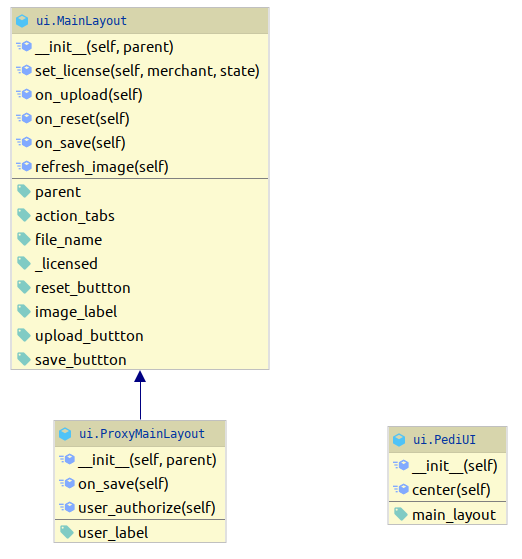
*Рис. 2.1.3. UML-діаграма шаблону “****Prototype****”*

*Структурний шаблон*. Задає види створюваних об’єктів за допомогою екземпляра-прототипу і створює нові об’єкти шляхом копіювання цього прототипу.

*Структура*. **ICloneable** – головний інтерфейс даного шаблону. **PediImage —** клас, конструктор якого приймає шлях до зображення, яке треба зчитати і має метод *clone(),* який дозволяє створити новий об’єкт зображення без звертання до файлової системи.

*Обґрунтування використання даного шаблону*. Створення зображення шляхом читання його з файлової системи набагато затратніше, аніж його створення шлях копіювання уже зчитаного зображення.

1. **Proxy**

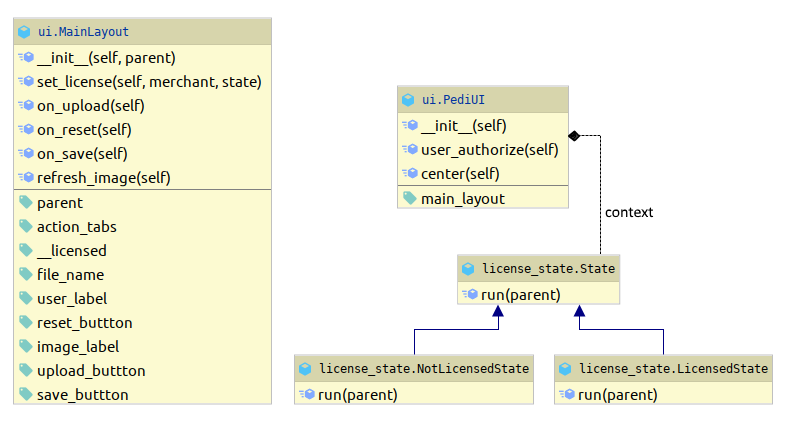
*Рис. 2.1.4. UML-діаграма шаблону “****Proxy****”*

*Структурний шаблон*. Обертає корисний об'єкт або сервіс спеціальним об'єктом-замінником, який “прикидається» оригіналом і перехоплює всі виклики до нього, а потім, після деякої обробки, направляє їх до обгорнутого об'єкту.

*Структура*. **MainLayout —** клас, стан якого змінює його поведінку. **PediUI —** його заступник, який огортає та містить поле типу **MainLayout** та імітує собою клас, для яйого є заступником.

*Обґрунтування використання даного шаблону*. **PediUI** робить вигляд, що є **MainLayout,** але на початку роботи змушує користувача авторизуватися.

1. **State**

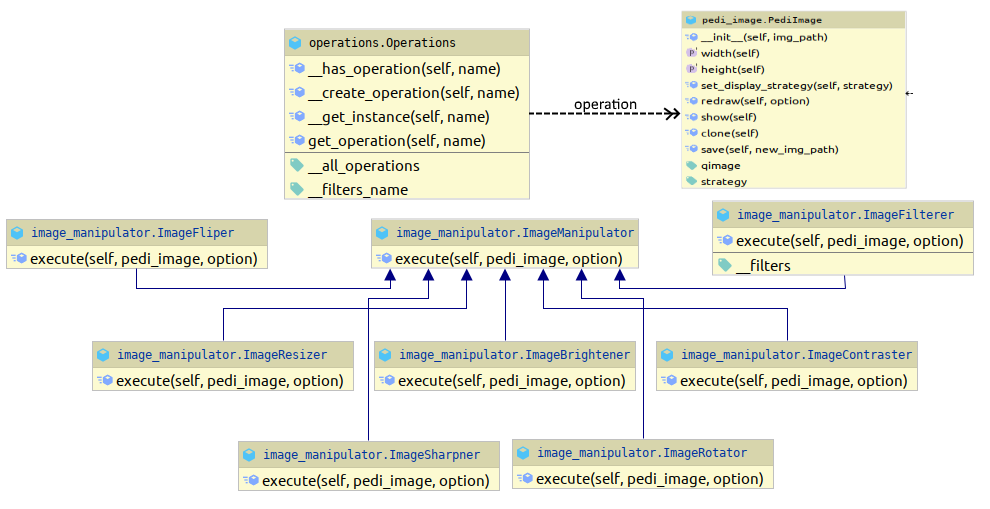
*Рис. 2.1.5. UML-діаграма шаблону “****State****”*

*Поведінковий шаблон*. Створює механізм підписки, за допомогою якого одні об'єкти можуть підписуватися на оновлення, що відбуваються в інших об'єктах.

*Структура*. **MainLayout** – основний компонент системи, поведінка якого залежить від його стану в поточний момент часу (ліцензія активована/не активована). **LicensedState** i **NotLicensedState LicensedState** класи, які описують поведінку в залежності від стану. **PediUI —** єдиний клас, який може змінювати стан.

*Обґрунтування використання даного шаблону*.У мові Python більша частина за приватність внутрішніх полів та методів класу лежить на відповідальності самих програмістів, тому було прийнято рішення розробити таку систему паттерну “Стан”, де стан об’єкту **MainLayout** має змогу змінювати лише об’єкт **PediUI.**

1. **Flyweight**

****

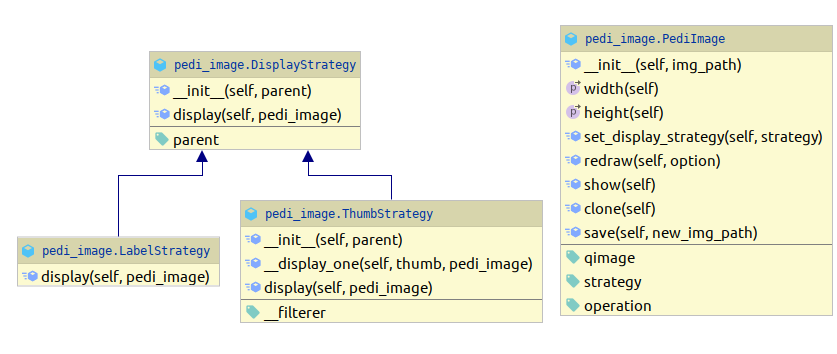
*Рис. 2.1.6. UML-діаграма шаблону “****Flyweight****”*

*Структурний шаблон*. Структурує об'єкти таким чином, що з них створюється лише обмежений набір екземплярів замість великої множини об’єктів. Полегшує повторне використання багатьох малих об’єктів, роблячи використання великої кількості об’єктів більш ефективною.

*Структура*. **NotifyMenu** – клас-обгортка для класу **Menu**, який водночас є представником підписника на події і декоратором відповідного класу. Об’єкт даного класу містить у собі об’єкт класу **Menu** (в тому числі – потенційно можливого меню, яке реалізовує інтерфейс **IPageMenu**) і переадресовує усі базові операції, які виконуються з меню, до представника, для якого він слугує “обгорткою”. Додатково, цей клас розширює функціонал меню і дозволяє меню виступати у якості підписника на події.

*Обґрунтування використання даного шаблону*. В графічному редакторі доволі часто виконуються маніпуляції з зображеннями, тому недоцільно створювати окремий об’єкт операції кожного разу.

1. **Strategy**

****

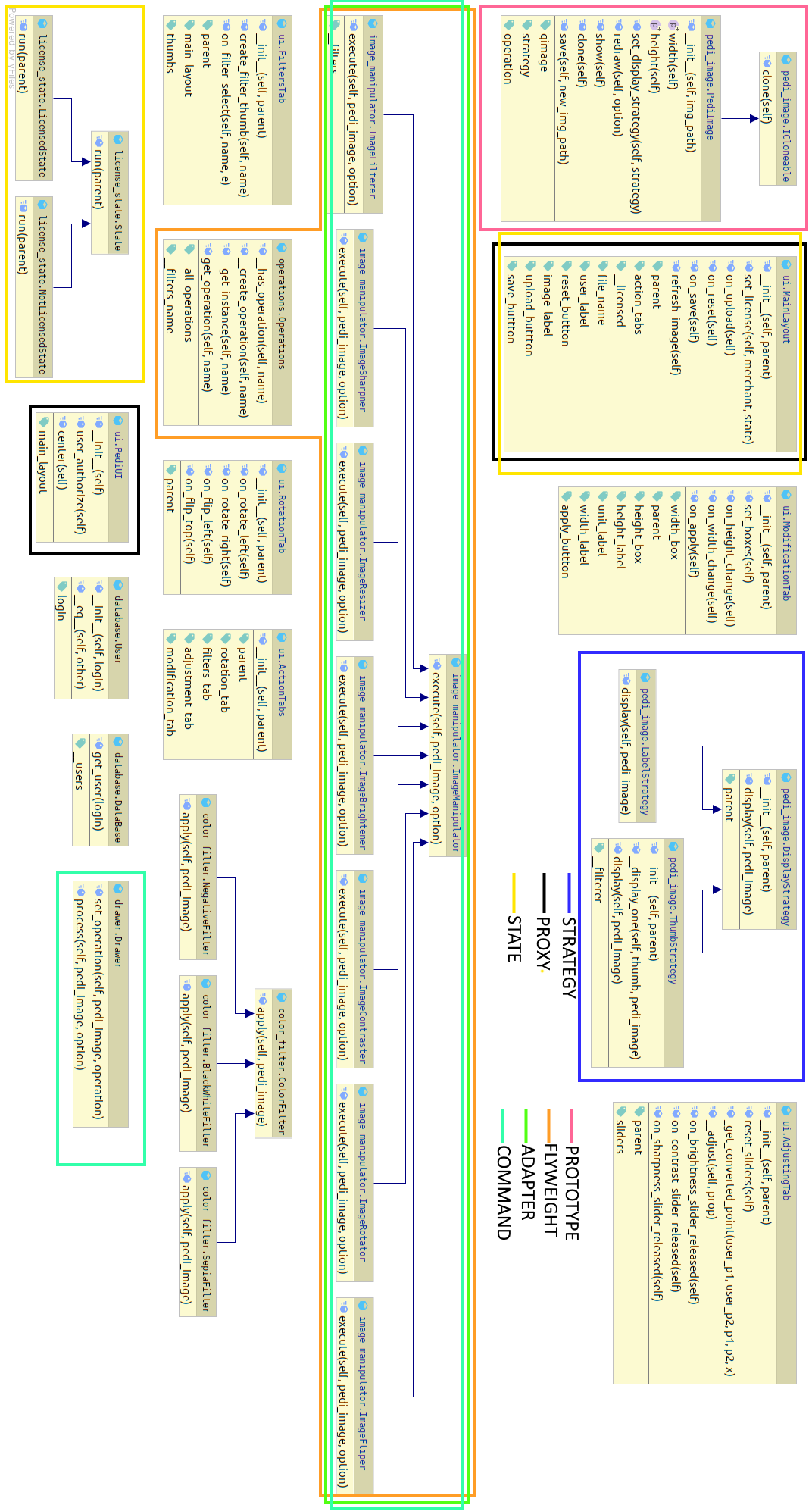
*Рис. 2.1.7. UML-діаграма шаблону “****Strategy****”*

*Поведінковий шаблон*. Визначає сімейство алгоритмів, інкапсулює кожен з них та робить їх взаємозамінними. Дозволяє змінювати алгоритми незалежно від коду клієнтів. Також відомий як Policy. Якщо в системі є алгоритми, які часто можуть використовуватися повторно в різних частинах програми, зручно їх виділити в окрему сутність, параметризувати та запускати там, де це потрібно, не дублюючи сам код.

*Структура*. **LabelStrategy** i **ThumbStrategy** — описують стратегії відображення зображення на екрані в різних місцях та в різних цілях. **PediImage** — має поле, яке відповідає за поле стратегії.

*Обґрунтування використання даного шаблону*. Алгоритми відображення зображення у різних місцях дуже сильно різняться між собою, тому доцільно було відокремити їх та інкапсулювати в методи з однаковими назвами.

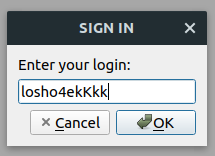
* 1. **Діаграма класів**



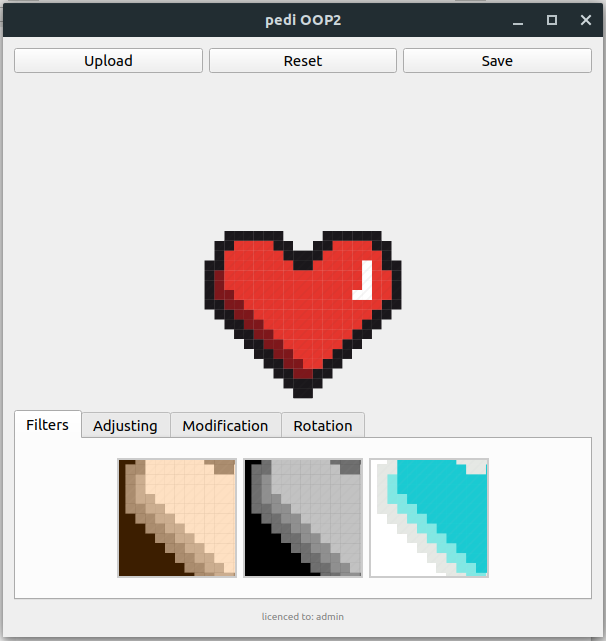
*Рис. 2.2.1. Діаграма класів програми*

* 1. **Результати роботи програми**

Початок роботи графічного редактору починається з вікна, де користувачу надається можливість авторизуватися:

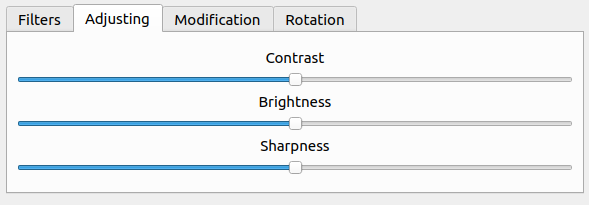
*Рис. 2.3.1. Вікно авторизації*

Після вибору зображення воно відображається на екрані:



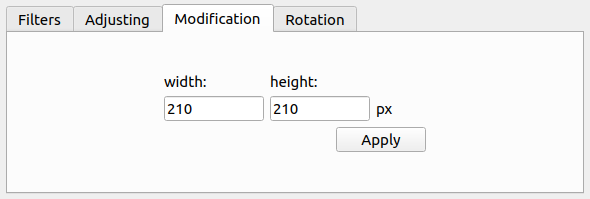
*Рис 2.3.2. Відображення зображення на екрані (вкладка Filters)*

На вкладці *Adjusting* є можливість змінити яскравість, контраст та чіткість:

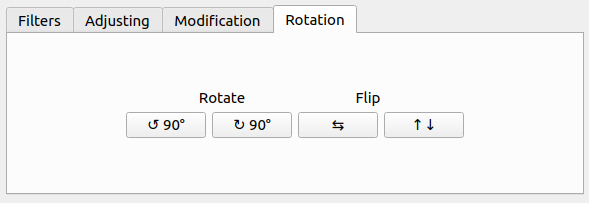


*Рис 2.3.3. Вкладка Adjusting*

На владці *Modification* можна змінити розмір зображеня у пікселях:

*Рис 2.3.4. Владка Modification*

На вкладці *Rotation* можна повернути/відобразити зображення.



*Рис 2.3.4. Вкладка Ratation*

**ВИСНОВКИ**

Метою даної курсової роботи було розроблення програмного забезпечення графічного редактора з використанням шаблонів проектування. Підставою для розроблення стало завдання на виконання курсової роботи з дисципліни «Об’єктно-орієнтоване програмування» студентами ІІ курсу кафедри ПЗКС НТУУ «КПІ».

Для досягнення поставленої мети у повному обсязі виконано завдання, визначені у аркуші завдання на курсову роботу; розроблено графічні матеріали; реалізовано всі вимоги до програмного продукту, наведені у технічному завданні; створено відповідну документацію. Розроблене програмне забезпечення дозволяє користувачу редагувати зображення (різні кольорні фільтри, зміна розміру/яскравості/контрасту/чіткості, обертання/відображення).

Програму створено на основі використання шаблонів проектування. Зокрема, до структури програмного забезпечення входить реалізація семи шаблонів, які належать до різних груп.

Для розроблення програмного забезпечення була використана мова програмування Python.

**СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ**

1. Язык шаблонов. Города. Здания. Строительство. / Кристофер Вольфганг Александер. – 1977. – 1096 с.
2. Приёмы объектно-ориентированного проектирования. Паттерны проектирования / Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. – 1994. – 395 с.
3. Руководство Microsoft по проектированию архитектуры приложений. / С. Сомасегар, Скотт Гатри, Девид Хилл. – 2009. – 529 с.