МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра информатики и систем управления

ОТЧЕТ

по лабораторной работе

по дисциплине

Шаблоны проектирования программного обеспечения

РУКОВОДИТЕЛЬ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_ Жевнерчук Д.В,\_\_

(подпись) (фамилия, и.,о.)

СТУДЕНТ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Халеев А.А.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Маясов А.О.

(подпись) (фамилия, и.,о.)

\_\_\_21-ВМз\_\_\_\_\_\_\_\_

(шифр группы)

Работа защищена «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_

С оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Нижний Новгород 2023

**Задание**

**Вариант 6.**

Реализуйте систему генерации шаблона html страницы по описанию:

1. Состава разделов (header, section1, … , section\_n, footer );

2. Структуры каждой секции (количество блоков).

Добавьте опциональную возможность выделять определенный раздел, блок рамкой.

Сгенерированный html код выводите в консоль.

**Проектное решение**

В работе были реализованы следующие паттерны:

1) Порождающие:  
- строитель (Builder);  
- фабричный метод (Factory Method).

2) Структурные:  
- адаптер (Adapter);  
- легковес (Flyweight).

3) Поведенческие:  
- стратегия (Strategy);  
- состояние (State).

**Обоснование выбора паттернов**

***Порождающие***

1. Паттерн Строитель (Builder) используется для создания сложных объектов пошагово. Он разделяет процесс создания объекта от его представления, чтобы один и тот же процесс создания мог создавать разные представления.

В данном проекте класс HtmlBuilder реализует паттерн Строитель. Он используется для построения дерева HTML-элементов. Класс имеет методы add, create\_content, to\_previous и get\_result для добавления элементов, создания содержимого, возврата к предыдущему состоянию и получения результата построения.

Класс HtmlDirector используется для управления процессом построения HTML-дерева. Он содержит экземпляр HtmlBuilder и определяет методы build\_tree и get\_html для создания и получения готового HTML-документа.

Основная идея паттерна Строитель заключается в том, чтобы разделить сложный процесс создания объекта на отдельные шаги и позволить клиенту контролировать этот процесс. В данном случае, HtmlBuilder является строителем, а HtmlDirector - директором, который управляет строителем и определяет последовательность шагов построения.

1. Паттерн Фабричный метод (Factory Method) используется для создания объектов, но делегирует фактическую логику создания подклассам. Он определяет интерфейс для создания объекта, но позволяет подклассам выбирать класс для создания.

Метод create\_content класса HtmlBuilder реализует паттерн Фабричный метод. Он принимает value и specs в качестве аргументов и создает объект типа HtmlTag на основе значения value.

Если value находится в списке HTML\_SINGLES, то создается экземпляр SingleTag.

Если value находится в списке HTML\_DOUBLES, то создается экземпляр DoubleTag.

Если value находится в списке HTML\_UNIQUES, то создается экземпляр UniqueTag.

В противном случае, создается экземпляр TagContent. Возвращается созданный объект.

Непосредственное создание объекта происходит в методе add класса HtmlBuilder, данный метод принимает на вход параметры создаваемого тега и использует фабричный метод для создания объекта на основе предоставленных параметров.

Таким образом, методы add и create\_content класса HtmlBuilder предоставляют гибкость в создании и добавлении различных типов элементов в HTML-дерево, основываясь на значениях их аргументов.

***Структурные***

1. Паттерн Легковес (Flyweight) используется для эффективного разделения объектов на общие и неизменяемые (внутренние) и индивидуальные и изменяемые (внешние) части. Он позволяет экономить память и улучшать производительность, разделяя общие данные между несколькими объектами.

В данном проекте класс UniqueTag реализует паттерн Легковес. Он используется для представления уникальных HTML-тегов, которые могут быть разными объектами, но имеют общие данные и поведение.

Класс UniqueTag имеет поле \_\_instances, которое является словарем, хранящим экземпляры UniqueTag по их именам. Это позволяет обеспечить, что каждый тег имеет только один экземпляр в системе. Когда создается новый тег, проверяется, есть ли уже экземпляр с таким именем в словаре. Если есть, то возвращается существующий экземпляр. Если нет, то создается новый экземпляр и добавляется в словарь.

Суть паттерна Легковес заключается в том, чтобы разделить данные, которые могут быть общими для нескольких объектов, и хранить их в отдельном месте (в данном случае, в словаре \_\_instances). Это позволяет уменьшить потребление памяти и повысить эффективность использования объектов.

Назначение паттерна Легковес состоит в оптимизации работы с большим количеством мелких объектов, снижении потребления памяти и улучшении производительности системы.

1. Паттерн Адаптер используется для преобразования интерфейса одного класса в интерфейс другого класса, с целью обеспечения их совместной работы без изменения исходного кода.

Класс HtmlAdapter представляет реализацию паттерна Адаптер (Adapter). HtmlAdapter обеспечивает адаптацию интерфейса класса HtmlWidget к интерфейсу класса HtmlDirector, чтобы они могли работать вместе.

Конструктор класса HtmlAdapter создает экземпляр класса HtmlDirector, который будет использоваться для построения HTML-страницы.

Метод build\_page принимает объект типа HtmlWidget и использует его свойства (цвет, выравнивание, наличие границ) для создания стиля и передачи его в HtmlDirector, чтобы построить дерево HTML-элементов. Метод get\_html возвращает HTML-код, сгенерированный HtmlDirector.

Метод create\_style является вспомогательным методом, который создает стиль на основе переданных параметров (цвет, выравнивание, наличие границ) и возвращает объект типа Style.

Таким образом, класс HtmlAdapter выполняет функцию адаптера, позволяя классу HtmlWidget работать с классом HtmlDirector, обеспечивая их взаимодействие без изменения исходного кода.

***Поведенческие***

1. Классы Strategy, Node, и Leaf вместе с HtmlBuilder реализуют паттерн Стратегия (Strategy).

Этот паттерн используется для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Он позволяет изменять поведение объекта во время выполнения программы.

В данном проекте:

Strategy является абстрактным базовым классом, который определяет общий интерфейс для всех стратегий. В данном случае он определяет абстрактный метод add.

Node и Leaf являются конкретными стратегиями, реализующими метод add. Node добавляет новый узел DoubleTag в дерево HTML, а Leaf добавляет одинарный тег HTML.

HtmlBuilder используется для создания HTML. Он содержит метод add, который принимает стратегию в качестве аргумента и использует ее для добавления элементов в дерево HTML. Это позволяет изменить поведение метода add во время выполнения, выбирая различные стратегии.

Таким образом, этот паттерн позволяет изменять алгоритм добавления элементов в дерево HTML на лету, без изменения исходного кода класса HtmlBuilder.

1. Паттерн Состояние (State) позволяет объекту изменять свое поведение в зависимости от своего внутреннего состояния. Это достигается путем делегирования ответственности за определенное поведение состоянию, которое может быть изменено во время выполнения.

В нашем случае, паттерн Состояние реализован внутри класса HtmlBuilder:

Метод to\_previous позволяет объекту HtmlBuilder возвращаться к предыдущему состоянию. Состояния хранятся в стеке (node\_stack), который используется для восстановления предыдущего состояния. Каждый вызов to\_previous приводит к возврату к предыдущему состоянию путем извлечения последнего элемента из стека.

branch\_ptr представляет текущее состояние объекта HtmlBuilder. Он изменяется при вызове метода add и восстанавливается при вызове to\_previous.

Таким образом, в данном случае паттерн Состояние позволяет объекту HtmlBuilder возвращаться к предыдущему состоянию при построении дерева HTML, что дает большую гибкость при создании сложных структур HTML.

Класс HtmlDirector управляет процессом построения HTML, вызывая методы add и to\_previous объекта HtmlBuilder для построения сложного HTML-дерева. Этот класс также демонстрирует, как можно использовать объект HtmlBuilder для создания различных HTML-структур, зависящих от состояния объекта.

***Влияние шаблонов проектирования на расширяемость и межмодульную связность системы***

1. Строитель (Builder): Паттерн Строитель, реализованный в классе HtmlBuilder, облегчает создание сложных объектов, делая код более понятным и поддерживаемым. Это улучшает расширяемость, поскольку можно легко изменять структуру HTML, меняя только код в классе HtmlBuilder.
2. Фабричный метод (Factory Method): Этот паттерн используется в методе create\_content класса HtmlBuilder. Он облегчает создание объектов, делегируя эту задачу специализированным методам. Это уменьшает связанность, так как другие классы не нуждаются в знании о конкретных классах объектов HTML-тегов.
3. Легковес (Flyweight): Класс UniqueTag реализует паттерн Легковес. Этот паттерн сокращает использование памяти путем общего использования объектов для одинаковых данных. Это улучшает производительность, особенно при работе с большим количеством объектов.
4. Адаптер (Adapter): Класс HtmlAdapter реализует паттерн Адаптер. Этот паттерн позволяет классам с несовместимыми интерфейсами работать вместе. Это улучшает расширяемость, позволяя использовать существующий класс в новом контексте без необходимости изменять его код.
5. Стратегия (Strategy): Этот паттерн повышает расширяемость, позволяя легко добавлять новые стратегии без изменения существующего кода. Это также уменьшает связанность, так как классы, использующие стратегии, не нуждаются в знании о конкретных стратегиях.
6. Состояние (State): С помощью паттерна Состояние, состояние объекта управляется внутри самого объекта, что уменьшает зависимость между различными модулями системы.

Все эти шаблоны проектирования вместе улучшают расширяемость и уменьшают межмодульную связанность системы, делая код более модульным, упрощая добавление нового функционала и улучшая производительность. Применение этих шаблонов делает код более чистым, легким для понимания и поддержки, и упрощает процесс тестирования, поскольку каждый модуль или класс можно тестировать независимо.

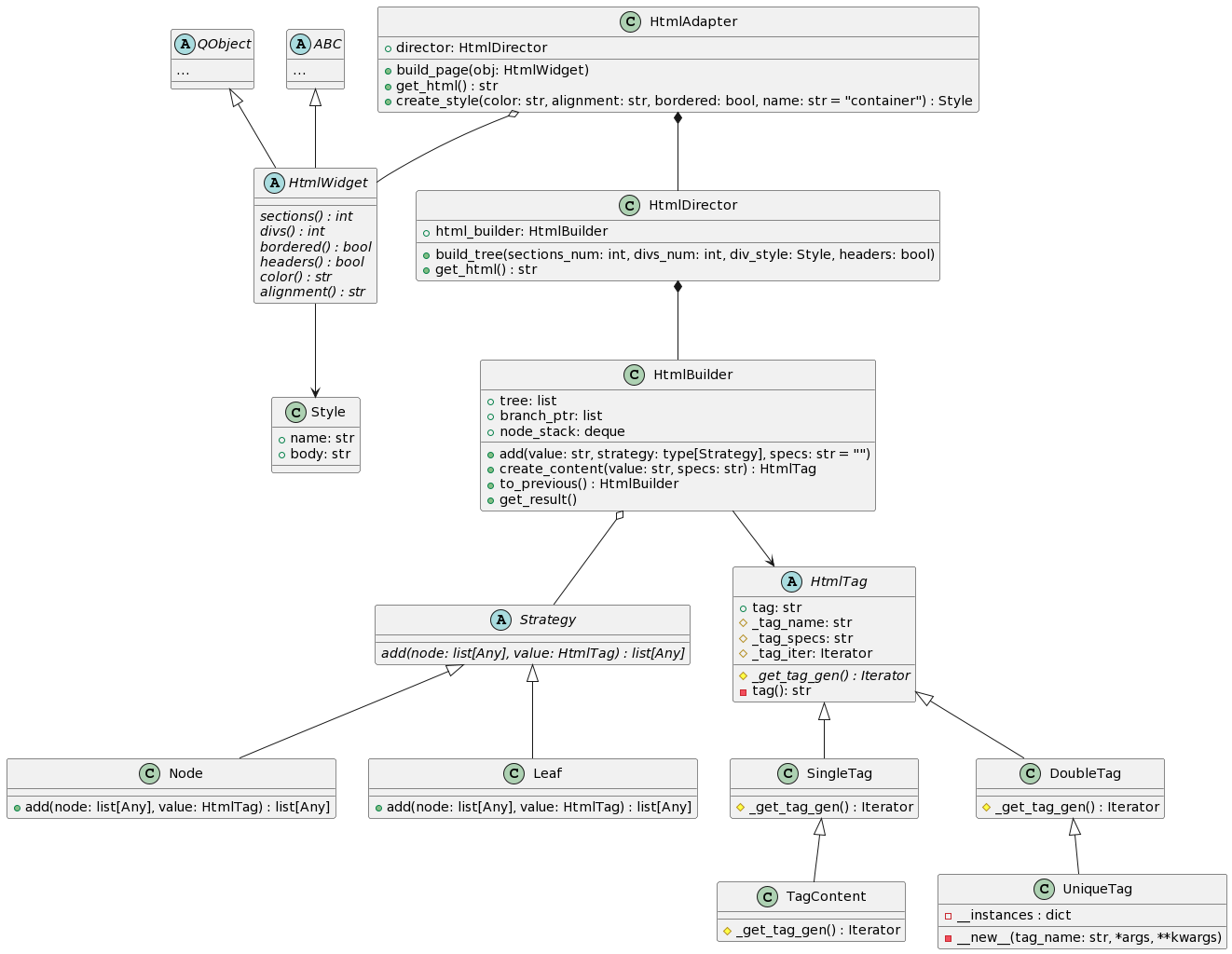


Рис. 1 – Диаграмма классов

**Приложение 1**

**Программный код**

***main.py:***

from sys import argv  
  
from PySide6.QtGui import QIcon  
from PySide6.QtWidgets import QApplication  
  
from source.app import MainWindow  
  
  
def main():  
 app = QApplication(argv)  
 window = MainWindow()  
 window.setWindowIcon(QIcon("designed\_ui/icons/logo.png"))  
 window.show()  
 app.exec()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

***app.py:***

import os  
from enum import IntEnum  
from fnmatch import fnmatch  
  
from PySide6.QtCore import Signal  
from PySide6.QtWebEngineWidgets import QWebEngineView  
from PySide6.QtWidgets import QInputDialog, QMainWindow, QStackedLayout, QTextEdit  
  
from designed\_ui.designed\_interface import Ui\_MainWindow  
from source.html\_utils import HtmlAdapter, HtmlWidget  
  
  
class MainWindow(QMainWindow, Ui\_MainWindow, HtmlWidget):  
 temp\_saved = Signal()  
 temp\_loaded = Signal()  
 temp\_generated = Signal()  
  
 class Mode(IntEnum):  
 TEXT = 0  
 HTML = 1  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.setupUi(self)  
 self.current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  
  
 self.update\_templates()  
 self.text\_edit = QTextEdit()  
 self.html\_render = QWebEngineView()  
 self.html\_render.reload()  
 self.text\_lay = QStackedLayout()  
 self.text\_lay.setStackingMode(QStackedLayout.StackOne)  
 self.text\_lay.addWidget(self.text\_edit)  
 self.text\_lay.addWidget(self.html\_render)  
 self.text\_view.setLayout(self.text\_lay)  
 *# ---------------------------- connections------------------------------------* self.text\_btn.clicked.connect(self.show\_text)  
 self.render\_btn.clicked.connect(self.show\_html)  
 self.generate\_btn.clicked.connect(self.generate)  
 self.clear\_btn.clicked.connect(self.text\_edit.clear)  
 self.clear\_btn.clicked.connect(self.show\_text)  
 self.save\_btn.clicked.connect(self.save\_modal)  
 self.load\_btn.clicked.connect(self.load\_template)  
 self.temp\_saved.connect(self.update\_templates)  
 self.temp\_loaded.connect(self.show\_text)  
 self.temp\_generated.connect(self.show\_text)  
 *# ----------------------------------------------------------------------------* @property  
 def sections(self) -> int:  
 return self.sections\_spin.value()  
  
 @property  
 def divs(self) -> int:  
 return self.divs\_spin.value()  
  
 @property  
 def bordered(self) -> bool:  
 return self.bordered\_check.isChecked()  
  
 @property  
 def color(self) -> str:  
 return self.text\_color\_spin.currentText()  
  
 @property  
 def alignment(self) -> str:  
 return self.alignment\_spin.currentText()  
  
 @property  
 def headers(self) -> bool:  
 return self.headers\_check.isChecked()

def generate(self) -> None:  
 html\_agent = HtmlAdapter()  
 html\_agent.build\_page(self)  
 html\_text = html\_agent.get\_html()  
 self.text\_edit.setPlainText(html\_text)  
 self.temp\_generated.emit()  
  
 def show\_text(self) -> None:  
 self.text\_lay.setCurrentIndex(self.Mode.TEXT)

def show\_html(self) -> None:  
 self.html\_render.setHtml(self.text\_edit.toPlainText())  
 self.text\_lay.setCurrentIndex(self.Mode.HTML)  
  
 def save\_modal(self) -> None:  
 modal = QInputDialog()  
 modal.setWindowTitle("HTML template saving")  
 modal.setLabelText("Enter filename:")  
 modal.exec()  
 if modal.accepted:  
 self.save\_template(modal.textValue())  
  
 def save\_template(self, filename) -> None:  
 with open(f"templates/{filename.rstrip('.html')}.html", "w") as output:  
 output.write(self.text\_edit.toPlainText())  
 self.temp\_saved.emit()  
  
 def update\_templates(self) -> None:  
 self.templates.clear()  
 self.templates.addItems(self.get\_templates())  
  
 @staticmethod  
 def get\_templates() -> tuple[str, ...]:  
 return tuple(entry for entry in os.listdir("templates")

if fnmatch(entry, "\*.html"))  
  
 def load\_template(self) -> None:  
 template\_name = str(self.templates.currentText())  
 with open(f"templates/{template\_name}", "r") as temp:  
 template\_text = temp.read()  
 self.text\_edit.setPlainText(template\_text)  
 self.temp\_loaded.emit()

***html\_utils.py:***

from abc import ABC, ABCMeta, abstractmethod  
from collections import deque  
from dataclasses import dataclass  
from typing import Any, Union  
  
from PySide6.QtCore import QObject  
  
from source.html\_tags import HtmlTag, DoubleTag, SingleTag, UniqueTag, TagContent, HTML\_SINGLES, \ HTML\_DOUBLES, HTML\_UNIQUES  
  
  
class \_ABCQObjectMeta(type(QObject), ABCMeta): ...  
  
  
class HtmlWidget(QObject, ABC, metaclass=\_ABCQObjectMeta):  
 @property  
 @abstractmethod  
 def sections(self) -> int: ...  
  
 @property  
 @abstractmethod  
 def divs(self) -> int: ...  
  
 @property  
 @abstractmethod  
 def bordered(self) -> bool: ...  
  
 @property  
 @abstractmethod  
 def headers(self) -> bool: ...  
  
 @property  
 @abstractmethod  
 def color(self) -> str: ...  
  
 @property  
 @abstractmethod  
 def alignment(self) -> str: ...  
  
  
@dataclass  
class Style:  
 name: str  
 body: str  
  
  
class HtmlAdapter:  
 *""" This class is used to implement the "adapter" pattern"""* def \_\_init\_\_(self) -> None:  
 self.director = HtmlDirector()  
  
 def build\_page(self, obj: HtmlWidget) -> None:  
 style = self.create\_style(color=obj.color, alignment=obj.alignment, bordered=obj.bordered)  
 self.director.build\_tree(sections\_num=obj.sections, divs\_num=obj.divs, div\_style=style,  
 headers=obj.headers)  
  
 def get\_html(self) -> str:  
 return self.director.get\_html()  
  
 @staticmethod  
 def create\_style(\*, color: str, alignment: str, bordered: bool, name: str = "container") -> "Style":  
 style = f".{name} {{"  
 if color:  
 style += f"color: {color}; "  
 if alignment:  
 style += f"text-align: {alignment}; "  
 if bordered:  
 style += "border: 1px solid black; "  
 style += "}"  
 return Style(name, style)

class Strategy(ABC):  
 *""" This class is a head of hierarchy, that provide to implement a part of the strategy pattern"""* @staticmethod  
 @abstractmethod  
 def add(node: list[Any], value: HtmlTag) -> list[Any]:  
 pass  
  
  
class Node(Strategy):  
 @staticmethod  
 def add(node: list[Any], value: HtmlTag) -> list[Any]:  
 if isinstance(value, DoubleTag):  
 new\_node = [value]  
 node.append(new\_node)  
 return new\_node  
 else:  
 return Leaf.add(node, value)  
  
  
class Leaf(Strategy):  
 @staticmethod  
 def add(node: list[Any], value: HtmlTag) -> list[Any]:  
 node.append(value)  
 return node  
  
  
class HtmlBuilder:  
 *""" This class is used to implement those patterns:  
 - factory method  
 - builder  
 - strategy (a part of)  
 - state  
 """* def \_\_init\_\_(self) -> None:  
 self.tree: list[Any] = list()  
 self.branch\_ptr: list[Any] = self.tree  
 self.node\_stack: deque = deque()  
  
 def add(self, value: str, \*, strategy: type[Strategy], specs: str = "") -> None:  
 *""" Implements a part of "strategy" pattern"""* self.node\_stack.append(self.branch\_ptr)  
 content = self.create\_content(value, specs) *# a part of "factory method" pattern* self.branch\_ptr = strategy.add(self.branch\_ptr, content)  
  
 @staticmethod  
 def create\_content(value: str, specs: str) -> HtmlTag:  
 *""" Implements "factory method" pattern"""* if value in HTML\_SINGLES:  
 return SingleTag(value, tag\_specs=specs)  
 elif value in HTML\_DOUBLES:  
 return DoubleTag(value, tag\_specs=specs)  
 elif value in HTML\_UNIQUES:  
 return UniqueTag(value, tag\_specs=specs)  
 else:  
 return TagContent(value)  
  
 def to\_previous(self) -> "HtmlBuilder":  
 *""" Implements "state" pattern - we can return to previous state using node callstack"""* if self.node\_stack:  
 last\_branch = self.node\_stack.pop()  
 if last\_branch is self.branch\_ptr:  
 self.to\_previous() *# recursive call* else:  
 self.branch\_ptr = last\_branch  
 if not self.node\_stack:  
 self.node\_stack.append(self.branch\_ptr)  
 return self  
  
 def get\_result(self):  
 return self.tree

class HtmlDirector:  
 def \_\_init\_\_(self) -> None:  
 self.html\_builder = HtmlBuilder()  
  
 def build\_tree(self, \*, sections\_num: int, divs\_num: int, div\_style: Style, headers: bool) -> None:  
 self.html\_builder.add("!DOCTYPE", strategy=Leaf, specs="html")  
 self.html\_builder.add("html", strategy=Node)  
 self.html\_builder.add("head", strategy=Node)  
 self.html\_builder.add("style", strategy=Node, specs='type="text/css"')  
 self.html\_builder.add(div\_style.body, strategy=Leaf)  
 self.html\_builder.to\_previous().to\_previous()  
 self.html\_builder.add("body", strategy=Node)  
 self.html\_builder.add("header", strategy=Leaf)  
 self.html\_builder.add("main", strategy=Node)  
 for s\_num in range(1, sections\_num + 1):  
 self.html\_builder.add("section", strategy=Node)  
 for d\_num in range(1, divs\_num + 1):  
 self.html\_builder.add("div", strategy=Node, specs=f"class={div\_style.name}")  
 if headers:  
 h\_level = d\_num if d\_num <= 6 else 6  
 self.html\_builder.add(f"h{h\_level}", strategy=Node)  
 self.html\_builder.add(f"section-{s\_num} div-{d\_num} message", strategy=Leaf)  
 self.html\_builder.to\_previous()  
 else:  
 self.html\_builder.add(f"section-{s\_num} div-{d\_num} message", strategy=Leaf)  
 self.html\_builder.to\_previous()  
 self.html\_builder.to\_previous()  
 self.html\_builder.to\_previous()  
 self.html\_builder.add("footer", strategy=Leaf)  
  
 def get\_html(self) -> str:  
 res = ""  
 space\_tab = " "  
  
 def tree\_traversal(node: list[Union[HtmlTag, Any]], level: int = -1) -> None:  
 *""" It's a function that traverses the html tree recursively to write result to nonlocal "res" variable """* nonlocal res  
 first\_tag = node[0]  
 res += space\_tab \* level + first\_tag.tag + '\n'  
 for child in node[1:]:  
 match child:  
 case SingleTag():  
 res += space\_tab \* (level + 1) + child.tag + '\n'  
 case DoubleTag():  
 res += space\_tab \* (level + 1) + child.tag + child.tag + '\n'  
 case list():  
 tree\_traversal(child, level + 1) *# recursive case* if isinstance(first\_tag, DoubleTag):  
 res += space\_tab \* level + first\_tag.tag + '\n'  
  
 tree\_traversal(self.html\_builder.tree)  
 return res

***html\_tags.py:***

from abc import ABC, abstractmethod  
from itertools import cycle, repeat  
from typing import Iterator  
  
HTML\_SINGLES = {'!DOCTYPE', 'area', 'base', 'br', 'col', 'command', 'embed', 'hr', 'img', 'input',   
 'keygen', 'link', 'meta', 'param', 'source', 'track', 'wbr'}  
  
HTML\_DOUBLES = {"a", "abbr", "address", "article", "aside", "audio", "b", "bdi", "bdo",   
 "blockquote", "button", "canvas", "caption", "cite", "code", "data", "datalist",   
 "dd", "del", "details", "dfn", "div", "dl", "dt", "em", "fieldset", "figcaption",   
 "figure", "footer", "form", "h1", "h2", "h3", "h4", "h5", "h6", "hgroup", "i",   
 "iframe", "ins", "kbd", "label", "legend", "li", "map", "mark", "menu",   
 "menuitem", "meter", "nav", "noscript", "object", "ol", "optgroup", "option",   
 "output", "p", "pre", "progress", "q", "rp", "rt", "ruby", "s", "samp", "script",   
 "section", "select", "small", "span", "strong", "style", "sub", "summary", "sup",

"table", "tbody", "td", "textarea", "tfoot", "th", "thead", "time", "title", "tr",

"u", "ul", "var", "video"}  
  
HTML\_UNIQUES = {"html", "head", "body", "header", "main", "footer"}  
  
  
class HtmlTag(ABC):  
 def \_\_init\_\_(self, tag\_name: str, \*, tag\_specs: str = ""):  
 self.\_tag\_name = tag\_name  
 self.\_tag\_specs = tag\_specs  
 self.\_tag\_iter = self.\_get\_tag\_gen()  
  
 def \_\_repr\_\_(self):  
 return f"{type(self).\_\_name\_\_}({self.\_tag\_name})"  
  
 @abstractmethod  
 def \_get\_tag\_gen(self) -> Iterator: ...  
  
 @property  
 def tag(self) -> str:  
 return next(self.\_tag\_iter)  
  
  
class SingleTag(HtmlTag):  
 def \_get\_tag\_gen(self) -> Iterator:  
 return repeat(f"<{self.\_tag\_name}{' ' \* bool(self.\_tag\_specs)}{self.\_tag\_specs}>")  
  
  
class DoubleTag(HtmlTag):  
 def \_get\_tag\_gen(self) -> Iterator:  
 return cycle((f"<{self.\_tag\_name}{' ' \* bool(self.\_tag\_specs)}{self.\_tag\_specs}>",

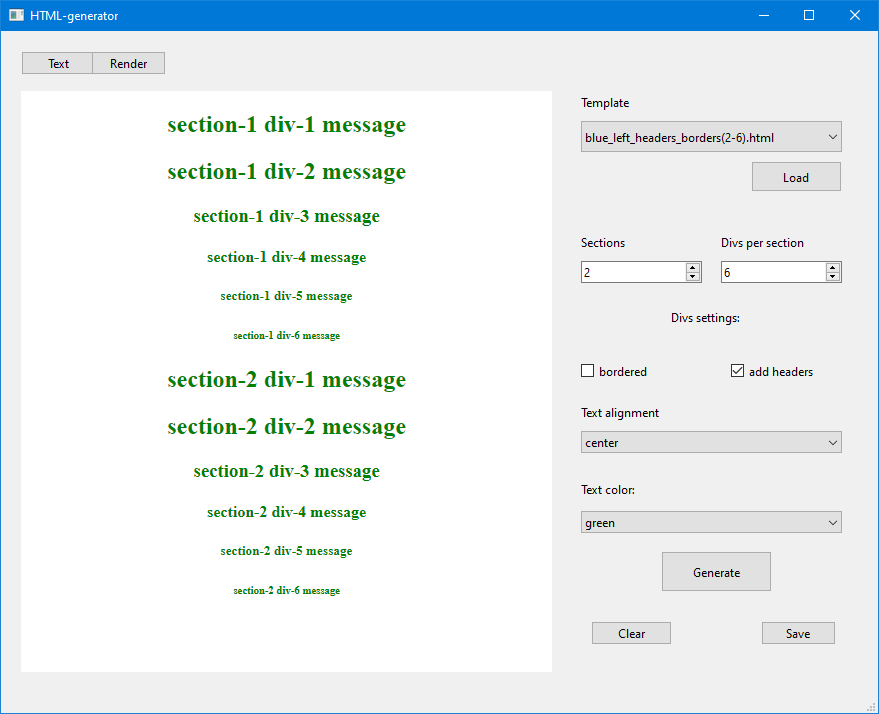
f"</{self.\_tag\_name}>"))  
  
  
*# flyweight*class UniqueTag(DoubleTag):  
 *""" This class is used to implement "flyweight" pattern """* \_\_instances: dict[str, HtmlTag] = dict()  
  
 def \_\_new\_\_(cls, tag\_name: str, \*args, \*\*kwargs):  
 return cls.\_\_instances.setdefault(tag\_name, super().\_\_new\_\_(cls))  
  
  
class TagContent(SingleTag):  
  
 def \_get\_tag\_gen(self) -> Iterator:  
 return repeat(self.\_tag\_name)

**Приложение 2**

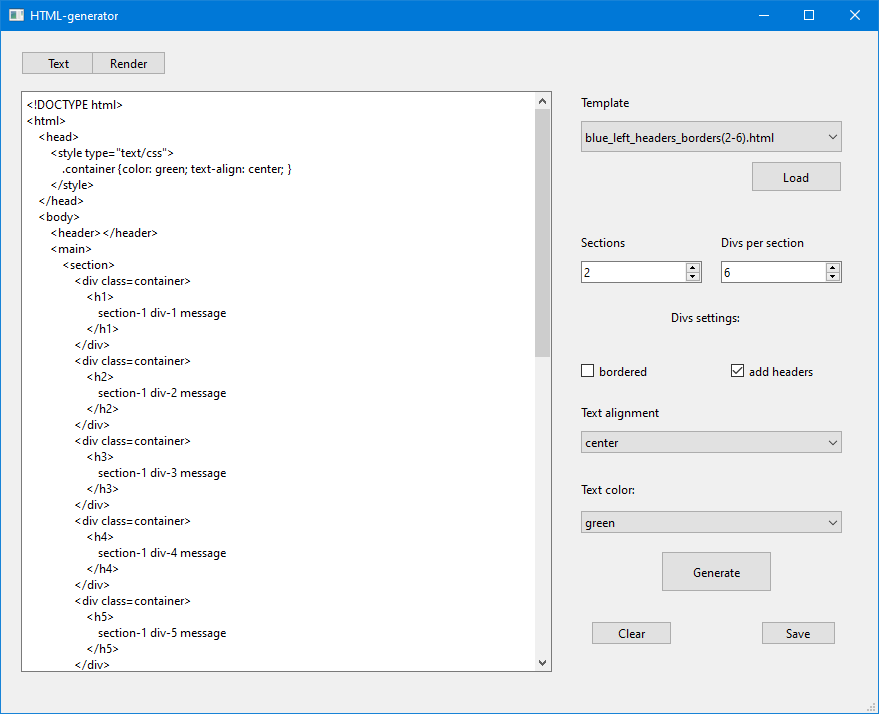
**Результаты тестирования**

*Генерация шаблона без рамки, с разноуровневыми заголовками зеленого цвета, по центру*

*Окно “Render”:*

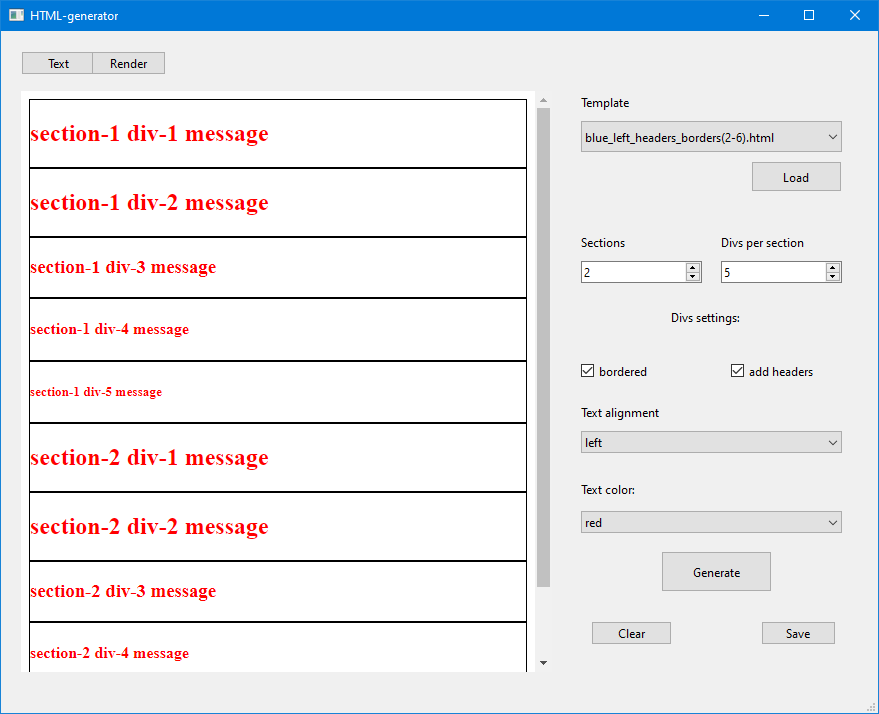
****

*Окно “Text”:*

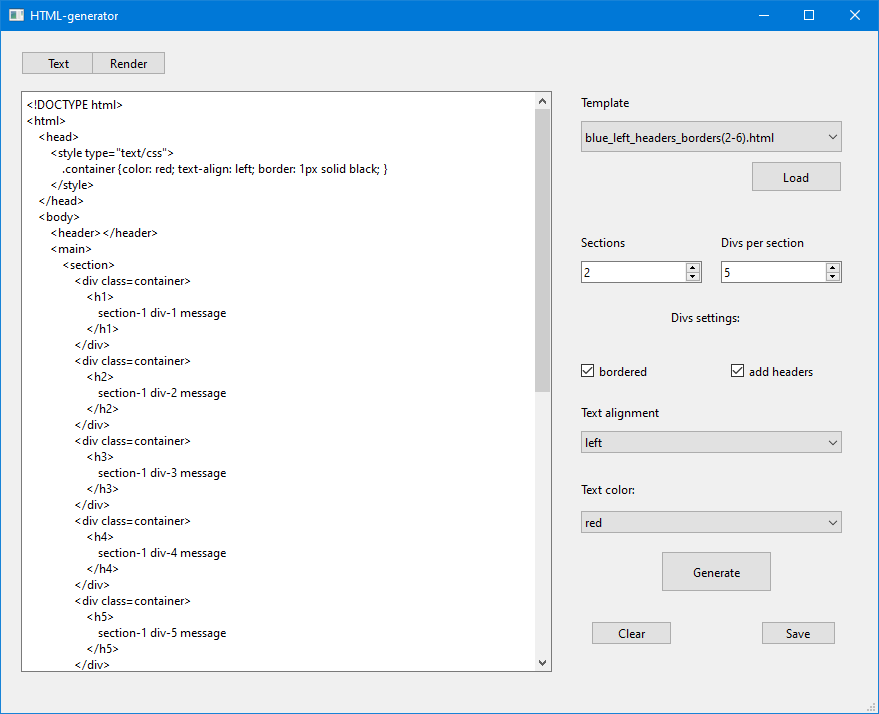
****

*Генерация шаблона c рамками, с разноуровневыми заголовками красного цвета, слева*

*Окно “Render”:*

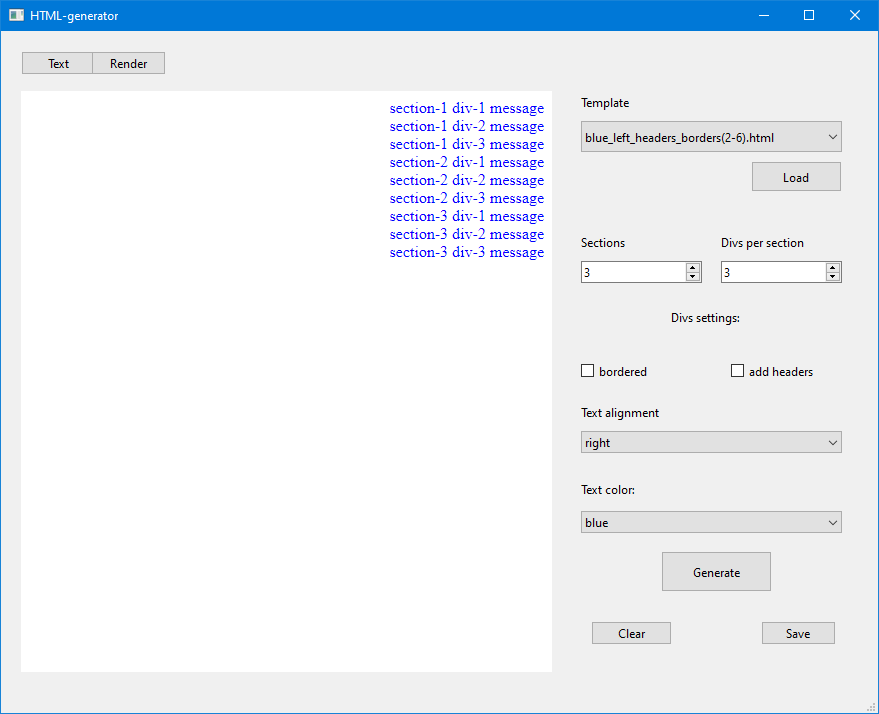
****

*Окно “Text”:*

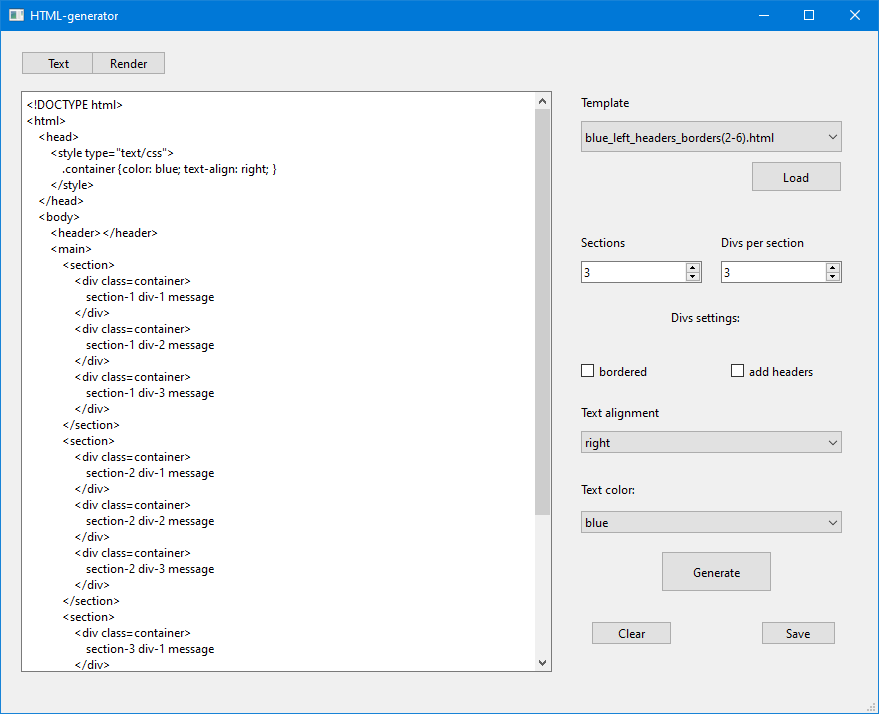
****

*Генерация шаблона без рамок, без заголовков, с текстом синего цвета, справа*

*Окно “Render”:*

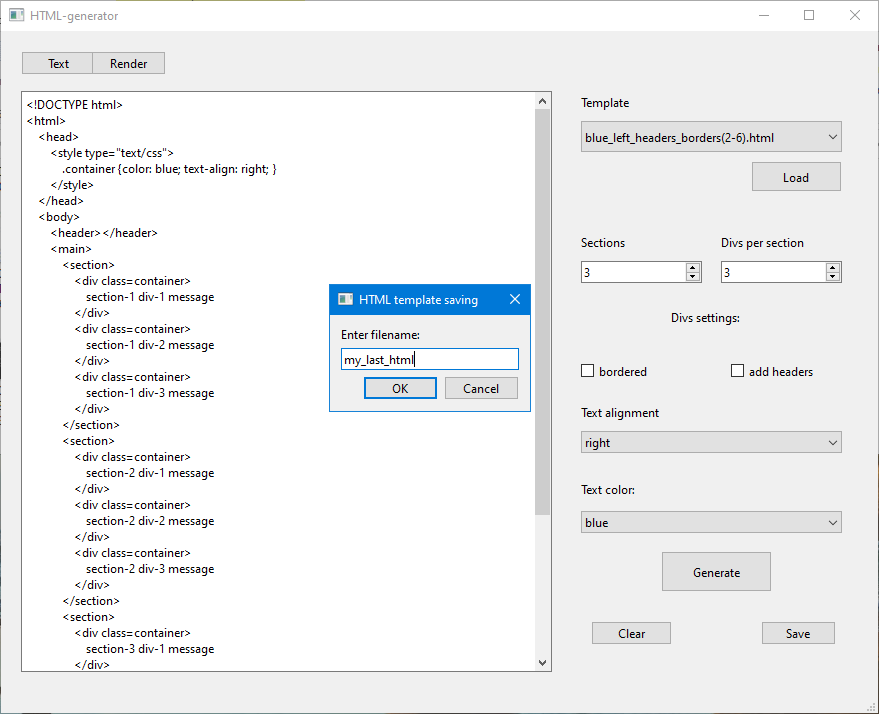
****

*Окно “Text”:*

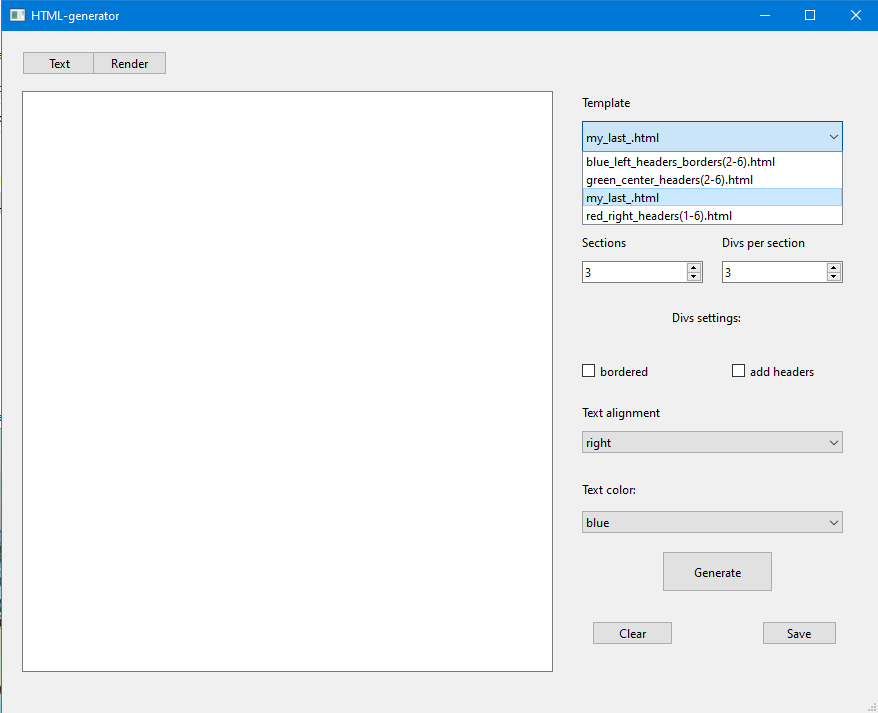
****

*Проверка работоспособности элементов сохранения и загрузки шаблонов*

*Результат нажатия кнопки “Save”:*

****

*Результат нажатия кнопки “Load”:*

****

**Вывод:**

Разработанное приложение работает без ошибок, полностью выполняет задуманную функциональность. С помощью программы можно генерировать шаблоны разного уровня вложенности, с возможностью добавления заголовков убывающего уровня, выбором цвета текста а также позиции на странице. Также есть возможность очистки окна ввода html-кода, загрузки и сохранения шаблонов.

Использование шаблонов проектирования в проекте позволило улучшить читаемость кода, повысить производительность, гибкость системы, а так же сделало ее расширяемой, позволило снизить межмодульную связность, что положительно сказалось на возможности тестирования системы, ее использования и доработки.