**Ордена Трудового Красного Знамени**

Федеральное Государственное

бюджетное образовательное учреждение

высшего образования

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

**МОСКОВСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**СВЯЗИ И ИНФОРМАТИКИ**

Кафедра Информационной Безопасности

**ЛАБОРАТОРНАЯ РАБОТА**

**№ 8**

по дисциплине «Объектно-ориентированное программирование система защиты информации»

на тему:

**«Изучение паттернов проектирования»**

Вариант №16

Выполнил: студ. гр. БПЗ1501

Романюк И. Ф.

Проверил: ассисент кафедры ИБ

Барков В. В.

Москва 2018

**Цель работы**

Научиться применять распространённые паттерны проектирования (Стратегия и фабричный метод) при создании своих проектов

**Выполнение**

Задание

Используя паттерн проектирования «Стратегия» отделить логику хранения в отдельную иерархию классов, с базовым классом Storage. 2 стратегии взять из предыдущих двух лабораторных работ, третью стратегию разработать. Используя паттерн проектирования «Фабричный метод» создать иерархию классов для создания объектов класса с базовым классом Storage. Создать класс ДСД, указанный в индивидуальном задании.

Исходный код:

Node.h

#pragma once

#include "iostream"

#include "conio.h"

using namespace std;

template<typename T>

struct Node

{

T data;

Node<T> \*previous, \*next;

};

UnidirectionalList.h

#pragma once

#include "Node.h"

template<typename T>

class UnidirectionalList

{

Node<T> \*head;

Node<T> \*tail;

int size = 0;

public:

UnidirectionalList(); //инициализация

UnidirectionalList(const UnidirectionalList &other); //копирование

UnidirectionalList(UnidirectionalList &&other); //перемещение

UnidirectionalList &operator=(const UnidirectionalList &other); //присваивание

UnidirectionalList &operator=(UnidirectionalList &&other); //присваивание с еремещением

~UnidirectionalList(); //деструктор

int GetSize();

Node<T> \*Find(int number);

void Push(T element, int number); //Вставка (добавление) элемента

T Remove(int number); //Удаление (взятие) элемента

T Peek(int number); //Просмотр (взятие без удаления) элемента

bool CheckNoEmptyList(); //Проверка наличия элементов

friend ostream & operator << <T>(ostream & stream, const UnidirectionalList<T> &a); //Вывод

};

template<typename T>

UnidirectionalList<T>::UnidirectionalList()

: head(NULL)

{

head = new Node<T>;

tail = new Node<T>;

head->next = tail;

}

template<typename T>

UnidirectionalList<T>::~UnidirectionalList() //деструктор

{

Node<T> \*node = head;

while (node->next != tail)

{

Node<T> \*deleteNode = node;

node = node->next;

delete deleteNode;

}

delete node;

delete tail;

}

template<typename T>

UnidirectionalList<T>::UnidirectionalList(const UnidirectionalList &other) //конструктор копирования

{

Node<T> oldOldNode = other.head;

head = other.head;

tail = other.tail;

size = other.size;

Node<T> newOldNode = head;

while (newOldNode->next != tail)

{

Node<T> \*newNewNode;

newOldNode->next = newNewNode;

newOldNode = newOldNode->next;

oldOldNode = oldOldNode->next;

newOldNode->data = oldOldNode->data;

}

newOldNode->next = tail;

}

template<typename T>

UnidirectionalList<T>::UnidirectionalList(UnidirectionalList &&other) //перемещение

{

head = other.head;

tail = other.tail;

size = other.size;

other.head = nullptr;

}

template<typename T>

UnidirectionalList<T> &UnidirectionalList<T>::operator=(const UnidirectionalList &other) //присваивание

{

if (this == &other)

{

return \*this;

}

delete head;

delete tail;

head = new Node<T>;

tail = new Node<T>;

Node<T> \*node = head;

while (node->next != tail)

{

Node<T> \*newNode = new Node<T>;

Node<T> \*oldNode = node->next;

node->next = newNode;

newNode->data = oldNode->data;

}

node->next = tail;

}

template<typename T>

UnidirectionalList<T> &UnidirectionalList<T>::operator=(UnidirectionalList &&other) //присваивание с перемещением

{

if (this == &other)

{

return \*this;

}

delete head;

head = other.head;

other.head = nullptr;

}

template<typename T>

int UnidirectionalList<T>::GetSize()

{

return this->size;

}

template<typename T>

Node<T> \*UnidirectionalList<T>::Find(int number)

{

Node<T> \*node = head;

for (int i = 0; i < number; i++)

{

node = node->next;

}

return node;

}

template<typename T>

void UnidirectionalList<T>::Push(T element, int number) //Вставка (добавление) элемента

{

Node<T> \*previousNode = this->Find(number);

Node<T> \*nextNode = previousNode->next;

Node<T> \*newNode = new Node<T>;

newNode->data = element;

previousNode->next = newNode;

newNode->next = nextNode;

size++;

}

template<typename T>

T UnidirectionalList<T>::Remove(int number) //Удаление (взятие) элемента

{

if ((size > 0) && (number > 0) && (number <= size))

{

Node<T> \*oldNode = this->Find(number);

Node<T> \*nextNode = oldNode->next;

Node<T> \*previousNode = this->Find(number - 1);

T element = oldNode->data;

previousNode->next = nextNode;

delete oldNode;

size--;

return element;

}

return 0;

}

template<typename T>

T UnidirectionalList<T>::Peek(int number) //Просмотр (взятие без удаления) элемента

{

Node<T> \*node = this->Find(number);

return node->data;

}

template<typename T>

bool UnidirectionalList<T>::CheckNoEmptyList() //Проверка наличия элементов

{

return (size > 0);

}

template<typename T>

ostream & operator << (ostream & stream, const UnidirectionalList<T> &a) //Вывод

{

UnidirectionalList<T> b = a;

stream << "=====\n";

for (int i = 0; i < b.size; i++)

{

stream << "| " << b.Remove() << "\n";

}

stream << "=====\n";

return stream;

}

BidirectionalCircularList.h

#pragma once

#include "Node.h"

template<typename T>

class BidirectionalCircularList

{

Node<T> \*head;

int size = 0;

public:

BidirectionalCircularList(); //инициализация

BidirectionalCircularList(const BidirectionalCircularList &other); //копирование

BidirectionalCircularList(BidirectionalCircularList &&other); //перемещение

BidirectionalCircularList &operator=(const BidirectionalCircularList &other); //присваивание

BidirectionalCircularList &operator=(BidirectionalCircularList &&other); //присваивание с еремещением

~BidirectionalCircularList(); //деструктор

int GetSize();

Node<T> \*Find(int number);

void Push(T element, int number); //Вставка (добавление) элемента

T Remove(int number); //Удаление (взятие) элемента

T Peek(int number); //Просмотр (взятие без удаления) элемента

bool CheckNoEmptyList(); //Проверка наличия элементов

friend ostream & operator << <T>(ostream & stream, const BidirectionalCircularList<T> &a); //Вывод

};

template<typename T>

BidirectionalCircularList<T>::BidirectionalCircularList()

: head(NULL)

{

head = new Node<T>;

Node<T> \*node = new Node<T>;

head->next = node;

head->previous = node;

node->next = head;

node->previous = head;

}

template<typename T>

BidirectionalCircularList<T>::~BidirectionalCircularList() //деструктор

{

Node<T> \*node = head;

while (node->next != head)

{

node = node->next;

delete node->previous;

}

delete head;

}

template<typename T>

BidirectionalCircularList<T>::BidirectionalCircularList(const BidirectionalCircularList &other) //конструктор копирования

{

Node<T> oldOldNode = other.head;

head = other.head;

Node<T> newOldNode = head;

size = other.size;

while (oldNode->next != other.head)

{

Node<T> \*newNewNode;

newOldNode->next = newNewNode;

newNewNode->previous = newOldist;

newOldNode = newOldNode->next;

newOldNode->data = oldOldNode->data;

}

newOldNode->next = head;

head->previous = newOldNode;

}

template<typename T>

BidirectionalCircularList<T>::BidirectionalCircularList(BidirectionalCircularList &&other) //перемещение

{

head = other.head;

size = other.size;

other.head = nullptr;

}

template<typename T>

BidirectionalCircularList<T> &BidirectionalCircularList<T>::operator=(const BidirectionalCircularList &other) //присваивание

{

if (this == &other)

{

return \*this;

}

delete head;

head = new Node<T>;

Node<T> \*node = head;

while (node->next != head)

{

Node<T> \*newNode = new Node<T>;

Node<T> \*oldNode = node->next;

node->next = newNode;

newNode->previous = node;

newNode->data = oldNode->data;

}

}

template<typename T>

BidirectionalCircularList<T> &BidirectionalCircularList<T>::operator=(BidirectionalCircularList &&other) //присваивание с перемещением

{

if (this == &other)

{

return \*this;

}

delete head;

head = other.head;

other.head = nullptr;

}

template<typename T>

int BidirectionalCircularList<T>::GetSize()

{

return this->size;

}

template<typename T>

Node<T> \*BidirectionalCircularList<T>::Find(int number)

{

Node<T> \*node = head->next;

for (int i = 0; i < number; i++)

{

node = node->next;

}

return node;

}

template<typename T>

void BidirectionalCircularList<T>::Push(T element, int number) //Вставка (добавление) элемента

{

Node<T> \*previousNode = this->Find(number);

Node<T> \*nextNode = previousNode->next;

Node<T> \*newNode = new Node<T>;

newNode->data = element;

previousNode->next = newNode;

newNode->previous = previousNode;

newNode->next = nextNode;

nextNode->previous = newNode;

size++;

}

template<typename T>

T BidirectionalCircularList<T>::Remove(int number) //Удаление (взятие) элемента

{

if (size > 0)

{

Node<T> \*oldNode = this->Find(number);

Node<T> \*nextNode = oldNode->next;

Node<T> \*previousNode = oldNode->previous;

T element = oldNode->data;

nextNode->previous = previousNode;

previousNode->next = nextNode;

delete oldNode;

size--;

return element;

}

return 0;

}

template<typename T>

T BidirectionalCircularList<T>::Peek(int number) //Просмотр (взятие без удаления) элемента

{

Node<T> \*node = this->Find(number);

return node->data;

}

template<typename T>

bool BidirectionalCircularList<T>::CheckNoEmptyList() //Проверка наличия элементов

{

return (size > 0);

}

template<typename T>

ostream & operator << (ostream & stream, const BidirectionalCircularList<T> &a) //Вывод

{

BidirectionalCircularList<T> b = a;

stream << "=====\n";

for (int i = 0; i < b.size; i++)

{

stream << "| " << b.Remove() << "\n";

}

stream << "=====\n";

return stream;

}

Storage.h

#pragma once

#include "iostream"

#include "conio.h"

using namespace std;

template<typename T>

class Storage

{

public:

virtual void Push(T element) = 0; //Вставка (добавление) элемента

virtual T Pop() = 0; //Удаление (взятие) элемента

virtual T Peek() = 0; //Просмотр (взятие без удаления) элемента

virtual bool CheckNoEmptyQueue() = 0; //Проверка наличия элементов

};

Queue.h

#pragma once

#include "Storage.h"

template<typename T>

class Queue

{

Storage<T> \*queue;

public:

~Queue() {}

void setStrategy(Storage<T> \*q)

{

queue = q;

}

void Push(T element)

{

queue->Push(element);

}

T Pop()

{

return queue->Pop();

}

T Peek()

{

return queue->Peek();

}

bool CheckNoEmptyQueue()

{

return queue->CheckNoEmptyQueue();

}

};

QueueBasedOnArray.h

#pragma once

#include "Storage.h"

template<typename T>

class QueueBasedOnArray : public Storage<T>

{

T \*array;

int count;

int end = 0, start = 0, size = 0;

public:

QueueBasedOnArray(int count); //инициализация

QueueBasedOnArray(const QueueBasedOnArray &other); //копирование

QueueBasedOnArray(QueueBasedOnArray &&other); //перемещение

QueueBasedOnArray &operator=(const QueueBasedOnArray &other); //присваивание

QueueBasedOnArray &operator=(QueueBasedOnArray &&other); //присваивание с еремещением

~QueueBasedOnArray(); //деструктор

int GetSize();

void Push(T element) override //Вставка (добавление) элемента

{

if (size < count)

{

array[end] = element;

end = (end + 1) % count;

size++;

}

};

T Pop() override //Удаление (взятие) элемента

{

if (size > 0)

{

T element = array[start];

start = (start + 1) % count;

size--;

return element;

}

return 0;

};

T Peek() override //Просмотр (взятие без удаления) элемента

{

return array[start];

};

bool CheckNoEmptyQueue() override //Проверка наличия элементов

{

return (size > 0);

}

friend ostream & operator << <T>(ostream & stream, const QueueBasedOnArray<T> &a); //Вывод

};

template<typename T>

QueueBasedOnArray<T>::QueueBasedOnArray(int count)

: count(count)

{

array = new T[count];

}

template<typename T>

QueueBasedOnArray<T>::~QueueBasedOnArray()

{

delete[] array;

}

template<typename T>

QueueBasedOnArray<T>::QueueBasedOnArray(const QueueBasedOnArray &other)

{

array = new T[other.count];

count = other.count;

end = other.end;

start = other.start;

size = other.size;

for (int i = 0; i < count; ++i)

{

array[i] = other.array[i];

}

}

template<typename T>

QueueBasedOnArray<T> &QueueBasedOnArray<T>::operator=(const QueueBasedOnArray &other)

{

if (this == &other)

{

return \*this;

}

delete[] array;

array = new T[other.count];

count = other.count;

for (int i = 0; i < count; ++i)

{

array[i] = other.array[i];

}

}

template<typename T>

QueueBasedOnArray<T> &QueueBasedOnArray<T>::operator=(QueueBasedOnArray &&other)

{

if (this == &other)

{

return \*this;

}

delete[] array;

array = other.array;

count = other.count;

other.array = nullptr;

}

template<typename T>

QueueBasedOnArray<T>::QueueBasedOnArray(QueueBasedOnArray &&other)

{

array = other.array;

count = other.count;

other.array = nullptr;

}

template<typename T>

int QueueBasedOnArray<T>::GetSize()

{

return this->size;

};

template<typename T>

ostream & operator << (ostream & stream, const QueueBasedOnArray<T> &a)

{

QueueBasedOnArray<T> b = a;

stream << "=====\n";

for (int i = 0; i < b.count; i++)

{

stream << "| " << b.Pop() << "\n";

}

stream << "=====\n";

return stream;

}

QueueBasedOnUnidirectionalList.h

#pragma once

#include "Storage.h"

#include "UnidirectionalList.h"

template<typename T>

class QueueBasedOnUnidirectionalList : public Storage<T>

{

UnidirectionalList<T> \*Queue;

int count;

public:

QueueBasedOnUnidirectionalList(int count)

: Queue(), count(count) { }

QueueBasedOnUnidirectionalList(const QueueBasedOnUnidirectionalList &other) //конструктор копирования

{

Queue = other.Queue;

count = other.count;

}

QueueBasedOnUnidirectionalList(QueueBasedOnUnidirectionalList &&other)

{

Queue = other.Queue;

count = other.count;

}

QueueBasedOnUnidirectionalList<T> &operator=(const QueueBasedOnUnidirectionalList &other)

{

if (this == &other)

{

return \*this;

}

count = other.count;

Queue = other.Queue;

}

QueueBasedOnUnidirectionalList<T> &operator=(QueueBasedOnUnidirectionalList &&other)

{

if (this == &other)

{

return \*this;

}

count = other.count;

Queue = other.Queue;

}

~QueueBasedOnUnidirectionalList();

void Push(T element) override //Вставка (добавление) элемента

{

if (Queue->GetSize() < count)

{

Queue->Push(element, Queue->GetSize());

}

}

T Pop() override //Удаление (взятие) элемента

{

return Queue->Remove(1);

}

T Peek() override //Просмотр (взятие без удаления) элемента

{

return Queue->Peek(1);

}

bool CheckNoEmptyQueue() override //Проверка наличия элементов

{

return Queue->CheckNoEmptyList();

}

friend ostream & operator << <T>(ostream & stream, const QueueBasedOnUnidirectionalList<T> &a); //Вывод

};

template<typename T>

QueueBasedOnUnidirectionalList<T>::~QueueBasedOnUnidirectionalList()

{

delete Queue;

}

template<typename T>

ostream & operator << (ostream & stream, const QueueBasedOnUnidirectionalList<T> &a) //Вывод

{

QueueBasedOnUnidirectionalList<T> b = a;

stream << "=====\n";

for (int i = 0; i < b.count; i++)

{

stream << "| " << b.Pop() << "\n";

}

stream << "=====\n";

return stream;

}

QueueBasedOnBidirectionalCircularList.h

#pragma once

#include "Storage.h"

#include "BidirectionalCircularList.h"

template<typename T>

class QueueBasedOnBidirectionalCircularList : public Storage<T>

{

BidirectionalCircularList<T> \*Queue;

int count;

public:

QueueBasedOnBidirectionalCircularList(int count)

: Queue(), count(count) { }

QueueBasedOnBidirectionalCircularList(const QueueBasedOnBidirectionalCircularList &other) //конструктор копирования

{

Queue = other.Queue;

count = other.count;

}

QueueBasedOnBidirectionalCircularList(QueueBasedOnBidirectionalCircularList &&other)

{

Queue = other.Queue;

count = other.count;

}

QueueBasedOnBidirectionalCircularList<T> &operator=(const QueueBasedOnBidirectionalCircularList &other)

{

if (this == &other)

{

return \*this;

}

count = other.count;

Queue = other.Queue;

}

QueueBasedOnBidirectionalCircularList<T> &operator=(QueueBasedOnBidirectionalCircularList &&other)

{

if (this == &other)

{

return \*this;

}

count = other.count;

Queue = other.Queue;

}

~QueueBasedOnBidirectionalCircularList();

void Push(T element) override //Вставка (добавление) элемента

{

if (Queue->GetSize() < count)

{

Queue->Push(element, Queue->GetSize());

}

}

T Pop() override //Удаление (взятие) элемента

{

return Queue->Remove(1);

}

T Peek() override //Просмотр (взятие без удаления) элемента

{

return Queue->Peek(1);

}

bool CheckNoEmptyQueue() override //Проверка наличия элементов

{

return Queue->CheckNoEmptyList();

}

friend ostream & operator << <T>(ostream & stream, const QueueBasedOnBidirectionalCircularList<T> &a); //Вывод

};

template<typename T>

QueueBasedOnBidirectionalCircularList<T>::~QueueBasedOnBidirectionalCircularList()

{

delete Queue;

}

template<typename T>

ostream & operator << (ostream & stream, const QueueBasedOnBidirectionalCircularList<T> &a) //Вывод

{

QueueBasedOnBidirectionalCircularList<T> b = a;

stream << "=====\n";

for (int i = 0; i < b.count; i++)

{

stream << "| " << b.Pop() << "\n";

}

stream << "=====\n";

return stream;

}

FactoryMethod.h

#pragma once

#include "QueueBasedOnArray.h"

#include "QueueBasedOnUnidirectionalList.h"

#include "QueueBasedOnBidirectionalCircularList.h"

template<typename T>

class Creator

{

public:

virtual Storage<T>\* factoryMethod(int count) = 0;

};

template<typename T>

class ConcreteCreatorQueueBasedOnArray : public Creator<T>

{

public:

~ConcreteCreatorQueueBasedOnArray() {};

Storage<T>\* factoryMethod(int count) override

{

return new QueueBasedOnArray<T>(count);

}

};

template<typename T>

class ConcreteCreatorQueueBasedOnUnidirectionalList : public Creator<T>

{

public:

~ConcreteCreatorQueueBasedOnUnidirectionalList() {};

Storage<T>\* factoryMethod(int count) override

{

return new QueueBasedOnUnidirectionalList<T>(count);

}

};

template<typename T>

class ConcreteCreatorQueueBasedOnBidirectionalCircularList : public Creator<T>

{

public:

~ConcreteCreatorQueueBasedOnBidirectionalCircularList() {};

Storage<T>\* factoryMethod(int count) override

{

return new QueueBasedOnBidirectionalCircularList<T>(count);

}

};

main.cpp

#include "Queue.h"

#include "QueueBasedOnArray.h"

#include "QueueBasedOnUnidirectionalList.h"

#include "QueueBasedOnBidirectionalCircularList.h"

#include "FactoryMethod.h"

void main()

{

setlocale(LC\_ALL, "Rus");

int n = 10;

//патерн "Стратегия"

Queue<int> queue;

QueueBasedOnArray<int> queue1(n);

QueueBasedOnUnidirectionalList<int> queue2(n);

QueueBasedOnBidirectionalCircularList<int> queue3(n);

queue.setStrategy(&queue1);

queue.Push(2);

queue.Push(5);

cout << queue.Pop() << "\n";

cout << queue.Peek() << "\n";

queue.setStrategy(&queue2);

queue.Push(3);

queue.Push(6);

cout << queue.Pop() << "\n";

cout << queue.Peek() << "\n";

queue.setStrategy(&queue3);

queue.Push(8);

queue.Push(1);

cout << queue.Pop() << "\n";

cout << queue.Peek() << "\n";

delete &queue;

//патерн "Фабричный метод"

ConcreteCreatorQueueBasedOnArray<int> Creator1;

ConcreteCreatorQueueBasedOnUnidirectionalList<int> Creator2;

ConcreteCreatorQueueBasedOnBidirectionalCircularList<int> Creator3;

Creator<int> \*creators[3] = { &Creator1, &Creator2, &Creator3 };

for (int i = 0; i < 3; i++)

{

Storage<int> \*queue = creators[i]->factoryMethod(n);

queue->Push(4);

queue->Push(7);

cout << queue->Pop() << "\n";

cout << queue->Peek() << "\n";

delete queue;

}

\_getch();

}