Московский Авиационный Институт
(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

Лабораторная работа № 7
по курсу «ООП»

Тема:
Проектирование струткуры классов

Студент:	Козлов А.Д.
Группа:	М80-206Б-18
Преподаватель:	Журавлев А.А.
Вариант:	7
Оценка:	
Дата:	

Москва
2019

1. Код:

document.hpp:

```
#ifndef T_DOCUMENT_HPP
#define T_DOCUMENT_HPP

#include <string>
#include <fstream>
#include <algorithm>

#include "stack.hpp"

#include "figure.hpp"

#include "hexagon.hpp"
#include "octagon.hpp"
#include "triangle.hpp"

namespace editor {
    class operations {
    public:
        operations() {}
        virtual ~operations() {}
        virtual void print() {}
        virtual int get_type_it() {}

        figure* fig = nullptr;
        std::string file_name;
    };
    class operation_import : public operations {
    public:
        operation_import() {}
        operation_import(figure* adding_fig) {
            fig = adding_fig;
        }
        ~operation_import() {
            delete fig;
        }
        void print() override {
            std::cout << "import_doc";
        }
        int get_type_it() override {
            return 1;
        }
};
class operation_export : public operations {
public:
   operation_export() {}
   operation_export(const std::string& fn) {
      file_name = fn;
   }
   void print() override {
      std::cout << "export_doc";
   }
   int get_type_it() override {
      return 2;
   }
};
class operation_create : public operations {
public:
   operation_create() {}
   operation_create(figure* adding_fig) {
      fig = adding_fig;
   }
   void print() override {
      std::cout << "create_doc";
   }
   int get_type_it() override {
      return 3;
   }
};
class operation_delete : public operations {
public:
   operation_delete() = delete;
   operation_delete(figure* adding_fig) {
      fig = adding_fig;
   }
   ~operation_delete() {
      delete fig;
   }
   void print() override {
      std::cout << "delete_doc";
   }
   int get_type_it() override {
      return 4;
   }
};
```

```
class document {
private:
    cont::stack<operations*> oper;
    bool is_create = false;
    figure* current_figure = nullptr;
public:
    document() = default;
    ~document() {
        if(oper.size() != 0) {
            for(auto elem : oper) {
                delete elem;
            }
        }
        if(current_figure != nullptr) {
            delete current_figure;
        }
    }
    void import_doc(const std::string& file_name) { //load
        figure* prev_figure = nullptr;
        if(is_create) {
            std::string command;
            std::string file_name_to_export;

            std::cout << "Document has already existed\n";
            std::cout << "Do you want to save document?\n";

            std::cout << "yes/no/exit\n";
            std::cin >> command;
            if(command == "yes") {
                std::cin >> file_name_to_export;
                this->export_doc(file_name_to_export);
            } else if(command == "no") {
            } else {
                return;
            }
            prev_figure = current_figure;
        }
        std::fstream fin(file_name, std::ios_base::binary | std::ios_base::in);
        if(!fin.is_open()) {
            std::cout << "Error opening file\n";
            return;
        }
        std::string figure_type;
        fin >> figure_type;
        if(figure_type == "hexagon" || figure_type == "1") {
            current_figure = new hexagon<int>;
        } else if(figure_type == "octagon" || figure_type == "2") {
            current_figure = new octagon<int>;
        } else if(figure_type == "triangle" || figure_type == "3") {
            current_figure = new triangle<int>;
        } else {
            std::cout << "Unable to import shape from this file\n";
            current_figure = prev_figure;
            return;
        }

        operations* current_operarion = new operation_import(prev_figure);
        oper.push(current_operarion);

        fin >> *current_figure;
        is_create = true;
        fin.close();
    }
    void export_doc(const std::string& file_name) { //save
        if(!is_create) {
            std::cout << "Document has't been created yet\n";
            return;
        }

        std::fstream fin(file_name, std::ios_base::binary | std::ios_base::out);
        if(!fin.is_open()) {
            std::cout << "Error opening file\n";
            return;
        }

        fin << *current_figure;
        fin.close();

        operations* current_operarion = new operation_export(file_name);
        oper.push(current_operarion);
    }
    void create_doc() {
        figure* prev_figure = nullptr;
        if(is_create) {
```

```
    std::string command;
    std::string file_name_to_export;

    std::cout << "Document has already existed\n";
    std::cout << "Do you want to save document?\n";

    std::cout << "yes/no/exit\n";
    std::cin >> command;
    if(command == "yes") {
        std::cin >> file_name_to_export;
        this->export_doc(file_name_to_export);
    } else if(command == "no") {
    } else {
        return;
    }
    prev_figure = current_figure;
}

std::string command;
std::cin >> command;
while(command == "help") {
    std::cout << "1) hexagon\n";
    std::cout << "2) octagon\n";
    std::cout << "3) triangle\n";
    std::cin >> command;
}
if(command == "hexagon" || command == "1") {
    current_figure = new hexagon<int>;
} else if(command == "octagon" || command == "2") {
    current_figure = new octagon<int>;
} else if(command == "triangle" || command == "3") {
    current_figure = new triangle<int>;
} else {
    std::cout << "Unable create shape\n";
    current_figure = prev_figure;
    return;
}

std::cin >> *current_figure;
is_create = true;

operations* current_operarion = new operation_create(prev_figure);
oper.push(current_operarion);
```

```
    }
    void delete_doc() {
      if(!is_create) {
        std::cout << "Document doesn't exist anymore\n";
        return;
      }

      operations* current_operarion = new
operation_delete(current_figure);
      oper.push(current_operarion);

      is_create = false;
      current_figure = nullptr;
    }
    void print_doc() const {
      if(!is_create) {
        std::cout << "Document hasn't been created yet\n";
        return;
      }
      std::cout << *current_figure << "\n";
    }
    void undo() {
      if(oper.empty()) {
        std::cout << "You don't do anything\n";
        return;
      }
      operations* op = oper.top();
      if(op->get_type_it() == 1) {
        delete current_figure;
        current_figure = op->fig;
        if(current_figure == nullptr) {
          is_create = false;
        }
        op->fig = nullptr;
      } else if(op->get_type_it() == 2) {
        std::fstream fin(op->file_name, std::ios_base::binary |
std::ios_base::out);
        fin.close();
      } else if(op->get_type_it() == 3) {
        delete current_figure;
        current_figure = op->fig;
        if(current_figure == nullptr) {
          is_create = false;
```

```
                }
                op->fig = nullptr;
            } else if(op->get_type_it() == 4) {
                current_figure = op->fig;
                is_create = true;
                op->fig = nullptr;
            } else {
                std::cout << "type_id: " << op->get_type_it() << "\n";
            }
            oper.pop();
            delete op; //delete operations point
            std::cout << "stack operations: ";
            std::for_each(oper.begin(), oper.end(), [](operations* opp) {
                std::cout << "->";
                opp->print();
            });
            std::cout << "\n";
        }
    };
}



#endif
```

figure.hpp:

```
#ifndef T_FIGURE_HPP
#define T_FIGURE_HPP

#include <iostream>
#include "point.hpp"

class figure {
public:
    figure() {};
    virtual ~figure() {};

    virtual int get_size() const = 0;

    virtual void read(std::istream&) = 0;
    virtual void write(std::ostream&) const = 0;

    virtual double area() = 0;
    virtual point<double> center() = 0;
protected:
    template<typename T>
    static double polygon_area(const point<T>*, const int);
    template<typename T>
    static point<double> polygon_center(const point<T>*, const int);
};

template<typename T>
double figure::polygon_area(const point<T>* peaks, const int size) {
    //std::cout << "Size = " << Size << "\n";
    double A = 0.0;
    if(size < 3) {
        return A;
    }
    for(int i = 0; i < size - 1; ++i) {
        A = A + (peaks[i].x * peaks[i + 1].y);
    }
    A = A + (peaks[size-1].x * peaks[0].y);
    for(int i = 0; i < size - 1; ++i) {
        A = A - (peaks[i+1].x * peaks[i].y);
    }
    A = A - (peaks[0].x * peaks[size-1].y);
    return A >= 0 ? (A * 0.5) : (-A * 0.5);
}

template<typename T>
point<double> figure::polygon_center(const point<T>* peaks, const int size) {
    point<double> res;
    double x = 0;
    double y = 0;
    for(int i = 0; i < size; ++i) {
        x = x + peaks[i].x;
        y = y + peaks[i].y;
    }
    res.x = x / size;
    res.y = y / size;
    return res;
}

std::istream& operator >>(std::istream& is, figure& f) {
    f.read(is);
    return is;
```

```
}
std::ostream& operator <<(std::ostream& os, figure& f) {
    f.write(os);
    return os;
}

#endif
```

hexagon.hpp:

```
#ifndef T_HEXAGON_HPP
#define T_HEXAGON_HPP

#include "point.hpp"
#include "figure.hpp"

template<typename T>
class hexagon : public figure {
public:
    hexagon() = default;
    hexagon(point<T>&, point<T>&, point<T>&, point<T>&, point<T>&,
point<T>&);
    int get_size() const override;
    double area() override;
    point<double> center()  override;
    void write(std::ostream&) const override;
    void read(std::istream&) override;
private:
    point<T> p1, p2, p3, p4, p5, p6;
};

template<typename T>
hexagon<T>::hexagon(point<T>& p1_, point<T>& p2_, point<T>& p3_,
point<T>& p4_, point<T>& p5_, point<T>& p6_)
            : p1(p1_), p2(p2_), p3(p3_), p4(p4_), p5(p5_), p6(p6_) {};

template<typename T>
int hexagon<T>::get_size() const {
    return 6;
}

template<typename T>
double hexagon<T>::area() {
    point<T> points[6] = {p1, p2, p3, p4, p5, p6};
    return polygon_area(points, 6);
}

template<typename T>
point<double> hexagon<T>::center() {
    point<T> points[6] = {p1, p2, p3, p4, p5, p6};
    return polygon_center(points, 6);
}

template<typename T>
void hexagon<T>::write(std::ostream& os) const {
    os << "hexagon "<< p1 << " " << p2 << " " << p3 << " " << p4 << " " <<
p5 << " " << p6;
}

template<typename T>
void hexagon<T>::read(std::istream& is) {
    point<T> p1_, p2_, p3_, p4_, p5_, p6_;
    is >> p1_ >> p2_ >> p3_ >> p4_ >> p5_ >> p6_;
    *this = hexagon(p1_, p2_, p3_, p4_, p5_, p6_);
}

template<typename T>
std::ostream& operator <<(std::ostream& os, const hexagon<T>& hex) {
    hex.write(os);
    return os;
}

template<typename T>
std::istream& operator >>(std::istream& is, hexagon<T>& hex) {
    hex.read(is);
    return is;
}

#endif
```

octagon.hpp:

```
#ifndef T_OCTAGON_HPP
#define T_OCTAGON_HPP

#include "point.hpp"
#include "figure.hpp"
```

```
template<typename T>
class octagon : public figure {
public:
    octagon() = default;
    octagon(point<T>&, point<T>&, point<T>&, point<T>&, point<T>&,
point<T>&, point<T>&, point<T>&);
    int get_size() const override;
    double area() override;
    point<double> center() override;
    void write(std::ostream&) const override;
    void read(std::istream&) override;
private:
    point<T> p1, p2, p3, p4, p5, p6, p7, p8;
};

template<typename T>
octagon<T>::octagon(point<T>& p1_, point<T>& p2_, point<T>& p3_,
point<T>& p4_, point<T>& p5_, point<T>& p6_, point<T>& p7_,
point<T>& p8_)
            : p1(p1_), p2(p2_), p3(p3_), p4(p4_), p5(p5_), p6(p6_), p7(p7_), p8(p8_)
{};

template<typename T>
int octagon<T>::get_size() const {
    return 8;
}

template<typename T>
double octagon<T>::area() {
    point<T> points[8] = {p1, p2, p3, p4, p5, p6, p7, p8};
    return polygon_area(points, 8);
}

template<typename T>
point<double> octagon<T>::center() {
    point<T> points[8] = {p1, p2, p3, p4, p5, p6, p7, p8};
    return polygon_center(points, 8);
}

template<typename T>
void octagon<T>::write(std::ostream& os) const {
    os << "octagon " << p1 << " " << p2 << " " << p3 << " " << p4 << " " <<
p5 << " " << p6 << " " << p7 << " " << p8;
}

template<typename T>
void octagon<T>::read(std::istream& is) {
   point<T> p1_, p2_, p3_, p4_, p5_, p6_, p7_, p8_;
   is >> p1_ >> p2_ >> p3_ >> p4_ >> p5_ >> p6_ >> p7_ >> p8_;
   *this = octagon(p1_, p2_, p3_, p4_, p5_, p6_, p7_, p8_);
}

template<typename T>
std::ostream& operator <<(std::ostream& os, const octagon<T>& oct) {
   oct.write(os);
   return os;
}

template<typename T>
std::istream& operator >>(std::istream& is, octagon<T>& oct) {
   oct.read(is);
   return is;
}

#endif
```

triangle.hpp:

```
#ifndef T_TRIANGLE_HPP
#define T_TRIANGLE_HPP

#include "point.hpp"
#include "figure.hpp"

template<typename T>
class triangle : public figure {
public:
   triangle() = default;
   triangle(point<T>&, point<T>&, point<T>&);
   int get_size() const override;
   double area() override;
   point<double> center() override;
   void write(std::ostream&) const override;
   void read(std::istream&) override;
private:
   point<T> p1, p2, p3;
```

```
};

template<typename T>
triangle<T>::triangle(point<T>& p1_, point<T>& p2_, point<T>& p3_)
            : p1(p1_), p2(p2_), p3(p3_) {};

template<typename T>
int triangle<T>::get_size() const {
   return 3;
}

template<typename T>
double triangle<T>::area() {
   point<T> points[3] = {p1, p2, p3};
   return polygon_area(points, 3);
}

template<typename T>
point<double> triangle<T>::center() {
   point<T> points[3] = {p1, p2, p3};
   return polygon_center(points, 3);
}

template<typename T>
void triangle<T>::write(std::ostream& os) const {
   os << "triangle " << p1 << " " << p2 << " " << p3;
}

template<typename T>
void triangle<T>::read(std::istream& is) {
   point<T> p1_, p2_, p3_;
   is >> p1_ >> p2_ >> p3_;
   *this = triangle(p1_, p2_, p3_);
}

template<typename T>
std::ostream& operator <<(std::ostream& os, const triangle<T>& tri) {
   tri.write(os);
   return os;
}

template<typename T>
std::istream& operator >>(std::istream& is, triangle<T>& tri) {
    tri.read(is);
    return is;
}

#endif
```

point.hpp:

```
#ifndef T_POINT_HPP
#define T_POINT_HPP

#include <iostream>

template<typename T>
struct point {
    T x;
    T y;
    point<T> operator +(point<T>&);
    point<T> operator -(point<T>&);
};

template<typename T>
point<T> point<T>::operator +(point<T>& other) {
    return {x + other.x, y + other.y};
}

template<typename T>
point<T> point<T>::operator -(point<T>& other) {
    return {x - other.x, y - other.y};
}

template<typename T>
std::ostream& operator <<(std::ostream& os, const point<T>& p) {
    os << p.x << " " << p.y;
    return os;
}

template<typename T>
std::istream& operator >>(std::istream& is, point<T>& p) {
    is >> p.x >> p.y;
    return is;
}

#endif
```

stack.hpp:

```
#ifndef CONT_STACK_HPP
#define CONT_STACK_HPP

#include <exception>
#include <memory>

namespace cont {
    template<class T>
    class stack {
    private:
        class stack_node;
        std::shared_ptr<stack_node> head;
        std::shared_ptr<stack_node> tail;
    public:
        class iterator;
        class const_iterator;
        stack();
        stack(const stack&) = delete;
        stack& operator =(const stack&) = delete;

        bool empty() const;
        void push(const T&);
        void pop();
        T& top();
        size_t size() const;

        iterator begin();
        iterator end();
        const_iterator begin() const;
        const_iterator end() const;

        void insert(iterator, const T&);
        void erase(iterator);
    };

    template<typename T>
    struct stack<T>::stack_node {
        stack_node() = default;
        stack_node(T new_value) : value(new_value) {}

        T value;
        std::shared_ptr<stack_node> next = nullptr;
        std::weak_ptr<stack_node> prev;
    };

    template<typename T>
    stack<T>::stack() {
        head = std::make_shared<stack_node>();
        tail = head;
    }

    template<typename T>
    bool stack<T>::empty() const {
        return head == tail;
    }

    template<typename T>
    void stack<T>::push(const T& value) {
        std::shared_ptr<stack_node> new_elem =
std::make_shared<stack_node>(value);
        if(empty()) {
            head = new_elem;
            head->next = tail;
            tail->prev = head;
        } else {
            new_elem->next = head;
            head->prev = new_elem;
            head = new_elem;
        }
    }

    template<typename T>
    void stack<T>::pop() {
        if(empty()) {
            throw std::out_of_range("Pop from empty stack");
        }
        head = head->next;
    }

    template<typename T>
    T& stack<T>::top() {
        return head->value;
    }
    template<typename T>
    size_t stack<T>::size() const {
        size_t size = 0;
        for(auto i : *this) {
            ++size;
        }
        return size;
    }

    template<typename T>
    typename stack<T>::iterator stack<T>::begin() {
        return iterator(head, this);
    }
    template<typename T>
    typename stack<T>::iterator stack<T>::end() {
        return iterator(tail, this);
    }
    template<typename T>
    typename stack<T>::const_iterator stack<T>::begin() const {
        return const_iterator(head, this);
    }
    template<typename T>
    typename stack<T>::const_iterator stack<T>::end() const {
        return const_iterator(tail, this);
    }

    template<typename T>
    void stack<T>::insert(iterator it, const T& value) {
        if(it.collection != this) {
            throw std::runtime_error("Iterator does not belong to this collection");
        }
        std::shared_ptr<stack_node> it_ptr = it.node.lock();
        if(!it_ptr) {
            throw std::runtime_error("Iterator is corrupted");
        }
        if(it == begin()) {
            push(value);
            return;
        }
        std::shared_ptr<stack_node> new_elem = std::make_shared<stack_node>(value);
        if(it == end()) {
            it_ptr->prev.lock()->next = new_elem;
            new_elem->prev = it_ptr->prev;
            new_elem->next = it_ptr;
            it_ptr->prev = new_elem;
        } else {
            std::shared_ptr<stack_node> next_ptr = it_ptr->next;
            std::weak_ptr<stack_node> prev_ptr = it_ptr;
            new_elem->next = next_ptr;
            next_ptr->prev = new_elem;
            new_elem->prev = prev_ptr;
            prev_ptr.lock()->next = new_elem;
        }
    }

    template<typename T>
    void stack<T>::erase(iterator it) {
        if(it.collection != this) {
            throw std::runtime_error("Iterator does not belong to this
collection");
        }
        std::shared_ptr<stack_node> it_ptr = it.node.lock();
        if(!it_ptr) {
            throw std::runtime_error("Iterator is corrupted");
        }
        if(it == end()) {
            throw std::runtime_error("Erase of end iterator");
        }
        if(it == begin()) {
            pop();
        } else {
            std::shared_ptr<stack_node> next_ptr = it_ptr->next;
            std::weak_ptr<stack_node> prev_ptr = it_ptr->prev;
            next_ptr->prev = prev_ptr;
            prev_ptr.lock()->next = next_ptr;
        }
    }

    template<typename T>
    class stack<T>::iterator {
        friend stack<T>;
    public:
        using value_type = T;
        using reference = T&;
        using pointer = T*;
        using difference_type = ptrdiff_t;
        using iterator_category = std::forward_iterator_tag;

        iterator(std::shared_ptr<stack_node> init_ptr, const stack<T>* ptr) :
node(init_ptr), collection(ptr) {}
        iterator(const iterator& other) : node(other.node),
collection(other.collection) {}

        iterator& operator =(const iterator&);
        bool operator ==(const iterator&) const;
        bool operator !=(const iterator&) const;
        iterator& operator ++();
        iterator operator ++(int);
        T& operator *() const;
    private:
        std::weak_ptr<stack_node> node;
        const stack<T>* collection;
    };

    template<typename T>
    typename stack<T>::iterator& stack<T>::iterator::operator =(const
iterator& other) {
        node = other.node;
        return *this;
    }
    template<typename T>
    bool stack<T>::iterator::operator ==(const iterator& other) const {
        auto lhs = node.lock();
        auto rhs = other.node.lock();
        if (lhs && rhs) {
            return lhs.get() == rhs.get();
        }
        return false;
    }
    template<typename T>
    bool stack<T>::iterator::operator !=(const iterator& other) const {
        return !(*this == other);
    }
    template<typename T>
    typename stack<T>::iterator& stack<T>::iterator::operator ++() {
        std::shared_ptr<stack_node> tmp = node.lock();
        if(tmp) {
```

```
      if(tmp->next == nullptr) {
        throw std::out_of_range("Going out of container boundaries");
      }
      tmp = tmp->next;
      node = tmp;
      return *this;
    } else {
      throw std::runtime_error("Element pointed by this iterator doesnt
exist anymore");
    }
  }
  template<typename T>
  typename stack<T>::iterator stack<T>::iterator::operator ++(int) {
    iterator result(*this);
    ++(*this);
    return result;
  }

  template<typename T>
  T& stack<T>::iterator::operator *() const {
    std::shared_ptr<stack_node> tmp = node.lock();
    if(tmp) {
      if(tmp->next == nullptr) {
        throw std::runtime_error("Dereferencing of end iterator");
      }
      return tmp->value;
    } else {
      throw std::runtime_error("Element pointed by this iterator doesnt
exist anymore");
    }
  }

  template<typename T>
  class stack<T>::const_iterator {
  private:
    std::weak_ptr<stack_node> node;
    const stack<T>* collection;
  public:
    using value_type = T;
    using reference = T&;
    using pointer = T*;
    using difference_type = ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;
```

```
        const_iterator(std::shared_ptr<stack_node> init_ptr, const stack<T>*
ptr) : node(init_ptr), collection(ptr) {}
        const_iterator(const const_iterator& other) : node(other.node),
collection(other.collection) {}

        const_iterator& operator =(const const_iterator&);
        bool operator ==(const const_iterator&) const;
        bool operator !=(const const_iterator&) const;
        const_iterator& operator ++();
        const_iterator operator ++(int);
        T& operator *() const;
    };

    template<typename T>
    typename stack<T>::const_iterator& stack<T>::const_iterator::operator
=(const const_iterator& other) {
        node = other.node;
        return *this;
    }
    template<typename T>
    bool stack<T>::const_iterator::operator ==(const const_iterator& other)
const {
        auto lhs = node.lock();
        auto rhs = other.node.lock();
        if (lhs && rhs) {
            return lhs.get() == rhs.get();
        }
        return false;
    }
    template<typename T>
    bool stack<T>::const_iterator::operator !=(const const_iterator& other)
const {
        return !(*this == other);
    }
    template<typename T>
    typename stack<T>::const_iterator& stack<T>::const_iterator::operator +
+() {
        std::shared_ptr<stack_node> tmp = node.lock();
        if(tmp) {
            if(tmp->next == nullptr) {
                throw std::out_of_range("Going out of container boundaries");
            }
            tmp = tmp->next;
            node = tmp;
            return *this;
        } else {
            throw std::runtime_error("Element pointed by this iterator doesnt
exist anymore");
        }
    }
    template<typename T>
    typename stack<T>::const_iterator stack<T>::const_iterator::operator ++
(int) {
        const_iterator result(*this);
        ++(*this);
        return result;
    }

    template<typename T>
    T& stack<T>::const_iterator::operator *() const {
        std::shared_ptr<stack_node> tmp = node.lock();
        if(tmp) {
            if(tmp->next == nullptr) {
                throw std::runtime_error("Dereferencing of end iterator");
            }
            return tmp->value;
        } else {
            throw std::runtime_error("Element pointed by this iterator doesnt
exist anymore");
        }
    }
}

#endif
```

CmakeLists.txt:

```
cmake_minimum_required(VERSION 3.0)
project(oop_exercize_07)
set(CMAKE_CXX_STANDART 17)

add_executable(oop_exercize_07 main.cpp ${FIGURE})
```

2. Ссылка на репозиторий на GitHub.

https://github.com/ArtemKD/oop_exercise_07

3. Набор testcases.

test_01.txt:

```
new
3
0 0 1 1 2 2
save copy
new
no
3
3 3 4 4 5 5
load copy
no
delete
undo
undo
```

test_02.txt:

```
new
1
0 0 1 1 2 2 3 3 4 4 5 5
save copy_2
delete
undo
```

4. Результаты выполнения тестов.

test_01.txt:

```
Document has already existed
Do you want to save document?
yes/no/exit
Document has already existed
Do you want to save document?
yes/no/exit
```

test_02.txt:

5. Объяснение результатов работы программы.

В программе реализовано меню с пунктами:

1) new — создание фигуры
2) delete - удаление фигуры
3) laod — загрузка фигуры из файла
4) save — сохранение фигуры в файл
5) undo — шаг назад

6) pirnt — вывести фигуру
7) help — помощь в командах

6. Вывод.

Выполняя данную лабораторную я получил навыки проектирования структуры классов Создал шаблонный класс document для работы с документами. Реализовал создание новой фигуры, удалиние, запись в файл и чтение из него. Также реализовал функцию undo(шаг назад) и вывод текущей фигуры в стандартный поток.