Московский государственный технический университет им. Н.Э. Баумана

Факультет “Информатика и системы управления”
Кафедра “Системы обработки информации и управления”

Курс «Парадигмы и конструкции языков программирования»

Отчет по лабораторным работам

Выполнил:
студент группы ИУ5 - 32Б:
Купрюшин Егор А.
Подпись и дата:

Проверил:
преподаватель каф. ИУ5
Гапанюк Ю. Е.
Подпись и дата:

Москва, 2025 г.

Задание

Разработать программу для решения биквадратного уравнения.

1. Программа должна быть разработана в виде консольного приложения на языке Python.
2. Программа осуществляет ввод с клавиатуры коэффициентов A, B, C, вычисляет дискриминант и ДЕЙСТВИТЕЛЬНЫЕ корни уравнения (в зависимости от дискриминанта).
3. Коэффициенты A, B, C могут быть заданы в виде параметров командной строки (вариант задания параметров приведен в конце файла с примером кода). Если они не заданы, то вводятся с клавиатуры в соответствии с пунктом 2. Описание работы с параметрами командной строки.
4. Если коэффициент A, B, C введен или задан в командной строке некорректно, то необходимо проигнорировать некорректное значение и вводить коэффициент повторно пока коэффициент не будет введен корректно. Корректно заданный коэффициент - это коэффициент, значение которого может быть без ошибок преобразовано в действительное число.
5. Дополнительное задание 1 (*). Разработайте две программы на языке Python - одну с применением процедурной парадигмы, а другую с применением объектно-ориентированной парадигмы.
6. Дополнительное задание 2 (*). Разработайте две программы - одну на языке Python, а другую на любом другом языке программирования (кроме C++).

Код

```
package main

import (
	"context"
	"errors"
	"fmt"
	"math"
	"os"
	"strconv"
)

func getCoefficient(ctx context.Context, name string, args []string, index int) float64
{
	for {
		if len(args) > index {
			value, err := strconv.ParseFloat(args[index], 64)
			if err == nil {
				fmt.Printf("Коэффициент %s=%f\n", name, value)
				return value
			}

			err2 := errors.New("error123")

			errors.Is(err, err2)

			fmt.Println("Некорректное значение для", name)
		}
		fmt.Printf("Введите коэффициент %s: ", name)

		var value float64
		if _, err := fmt.Scan(&value); err == nil {
			return value
		}

		fmt.Println("Некорректное значение, попробуйте снова.")
	}
}

func solveBiquadratic(a, b, c float64) []float64 {
	// Discriminant
	d := b*b - 4*a*c
```

```
	fmt.Printf("Дискриминант = %f\n", d)

	// Validate discriminant
	if d < 0 {
		fmt.Println("Действительных корней нет.")
		return []float64{}
	}

	// Use formula
	sqrtD := math.Sqrt(d)
	x1 := (-b + sqrtD) / (2 * a)
	x2 := (-b - sqrtD) / (2 * a)

	// Get roots result as
	roots := []float64{x1}
	if x1 != x2 {
		roots = []float64{x1, x2}
	}

	return roots
}

func main() {
	args := os.Args
	a := getCoefficient("A", args, 1)
	if a == 0 {
		fmt.Println("Коэффициент A в квадратном уравнении не может быть
равен 0.")
		return
	}

	// No rules for B & C
	b := getCoefficient("B", args, 2)
	c := getCoefficient("C", args, 3)

	// Solve
	roots := solveBiquadratic(a, b, c)

	// If ANY roots
	if len(roots) > 0 {
		fmt.Println("Действительные корни:", roots)
		return
```

```
        }

        fmt.Println("Корней нет.")
}
```

Скриншоты работы программы

Рис. 1 – Работа приложения

Лабораторная работа №6

Разработка на языке программирования Rust.

Цель лабораторной работы: изучение возможностей языка программирования Rust.

Требования к отчету:

Отчет по лабораторной работе должен содержать:

1. титульный лист;
2. описание задания;
3. текст программы;
4. экранные формы с примерами выполнения программы.

Задание:

1. Реализуйте любое из заданий курса на языке программирования Rust.
2. Разработайте хотя бы один макрос.
3. Разработайте модульные тесты (не менее 3 тестов).

main.rs

```
use crate::builder::text_builder::TextBuilder;

mod adapter;
mod builder;
mod memento;

fn main() {
    let text = TextBuilder::new()
        .with_author("Egor dehwyy".into())
        .with_headline("It's all about the style".into())
        .with_text(
            "Let's start with some text. And then some more text\nIt's all about the
style".into(),
        )
        .with_pipe(|text| text.to_uppercase())
```

```
        .with_rgb(255, 40, 120) // подсветка розовым
        .build();

    println!("{}", text);
}
```

builder/text_builder.rs

```
use crate::{builder::r#macro, styled};

use super::text;

#[derive(Default)]
pub struct TextBuilder {
    pipes: Vec<Box<dyn Fn(String) -> String>>,
    headline: Option<String>,
    author: Option<String>,
    rgb: Option<(u8, u8, u8)>,
    text: String,
}

impl TextBuilder {
    pub fn new() -> TextBuilder {
        TextBuilder::default()
    }

    pub fn with_pipe(mut self, pipe: impl Fn(String) -> String + 'static) -> Self {
        self.pipes.push(Box::new(pipe));

        self
    }

    pub fn with_headline(mut self, headline: String) -> Self {
        self.headline = Some(headline);

        self
    }

    pub fn with_author(mut self, author: String) -> Self {
        self.author = Some(author);
```

```
        self
    }

    pub fn with_text(mut self, text: String) -> Self {
        self.text = text;

        self
    }

    pub fn with_appended_text(mut self, text: String) -> Self {
        self.text += &text;

        self
    }

    pub fn with_rgb(mut self, r: u8, g: u8, b: u8) -> Self {
        self.rgb = Some((r, g, b));

        self
    }

    pub fn build(self) -> text::Text {
        let mut text = text::Text {
            inner: self.pipes.iter().fold(self.text, |text, pipe| pipe(text)),
            ..Default::default()
        };

        if let Some(author) = self.author {
            text.author = Some(styled!(format!("Author: {author}\n"), 4));
        }

        if let Some(headline) = self.headline {
            text.headline = Some(styled!(format!("Title: {headline}!\n"), 1));
        }

        if let Some((r, g, b)) = self.rgb {
            text.inner = styled!(text.inner, r, g, b);
        }

        text
    }
}
```

builder/text.rs

```
#[derive(Default)]
pub struct Text {
    pub(super) inner: String,
    pub(super) author: Option<String>,
    pub(super) headline: Option<String>,
}

impl std::fmt::Display for Text {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        if let Some(author) = &self.author {
            write!(f, "{}\n", author)?;
        }

        if let Some(headline) = &self.headline {
            write!(f, "{}\n", headline)?;
        }

        write!(f, "{}", self.inner)
    }
}
```

builder/mod.rs

```
pub mod r#macro;
pub mod text;
pub mod text_builder;

#[cfg(test)]
mod tests {
    use super::text::Text;
    use super::text_builder::TextBuilder;
    use std::fmt::Write as _;

    // -------- Helper pipes (fn pointers, не замыкания) --------
    fn trim_pipe(s: String) -> String {
        s.trim().to_string()
    }
    fn upper_pipe(s: String) -> String {
```

```
    s.to_uppercase()
}
fn add_bang_pipe(s: String) -> String {
    format!("{s}!")
}

#[test]
fn build_minimal_empty_by_default() {
    let t = TextBuilder::new().build();
    assert_eq!(t.to_string(), "");
}

#[test]
fn build_with_text_only_no_pipes() {
    let t = TextBuilder::new().with_text("hello".into()).build();

    assert_eq!(t.to_string(), "hello");
}

#[test]
fn build_with_author_and_headline_and_text() {
    let t = TextBuilder::new()
        .with_author("Author Name".into())
        .with_headline("Breaking Headline".into())
        .with_text("Body".into())
        .build();

    // Ожидаемый формат Display:
    // author\n
    // headline\n
    // inner
    assert_eq!(t.to_string(), "Author Name\nBreaking Headline\nBody");
}

#[test]
fn display_with_only_author() {
    let t = TextBuilder::new()
        .with_author("Only Author".into())
        .with_text("Body".into())
        .build();

    assert_eq!(t.to_string(), "Only Author\nBody");
```

```
}

#[test]
fn display_with_only_headline() {
    let t = TextBuilder::new()
        .with_headline("Only Headline".into())
        .with_text("Body".into())
        .build();

    assert_eq!(t.to_string(), "Only Headline\nBody");
}

#[test]
fn pipes_apply_in_insertion_order() {
    // Текст с пробелами вокруг -> trim -> upper -> add !
    let t = TextBuilder::new()
        .with_text("  hello world  ".into())
        .with_pipe(trim_pipe)
        .with_pipe(upper_pipe)
        .with_pipe(add_bang_pipe)
        .build();

    assert_eq!(t.to_string(), "HELLO WORLD!");
}

#[test]
fn with_appended_text_accumulates_before_pipes() {
    let t = TextBuilder::new()
        .with_text("foo".into())
        .with_appended_text("bar".into())
        .with_pipe(upper_pipe)
        .build();

    // "foobar" -> upper -> "FOOBAR"
    assert_eq!(t.to_string(), "FOOBAR");
}

#[test]
fn display_exact_newline_placement() {
    let t = TextBuilder::new()
        .with_author("A".into())
        .with_headline("H".into())
```

```
            .with_text("X".into())
            .build();

        // Проверим точную расстановку переносов строк
        let mut expected = String::new();
        write!(&mut expected, "A\nH\nX").unwrap();

        assert_eq!(t.to_string(), expected);
    }

    #[test]
    fn pipes_on_empty_text_are_ok() {
        let t = TextBuilder::new()
            .with_pipe(add_bang_pipe) // "" -> "!"
            .build();

        assert_eq!(t.to_string(), "!");
    }

    #[test]
    fn multiple_builders_independent_state() {
        let t1 = TextBuilder::new()
            .with_text("alpha".into())
            .with_pipe(upper_pipe)
            .build();

        let t2 = TextBuilder::new()
            .with_text("  beta  ".into())
            .with_pipe(trim_pipe)
            .build();

        assert_eq!(t1.to_string(), "ALPHA");
        assert_eq!(t2.to_string(), "beta");
    }
}
```

builder/macro.rs

```
#[macro_export]
macro_rules! styled {
    // Do not do anything
    ($text:expr,$r:expr, $g:expr, $b:expr) => {
```

```
        format!("\x1b[38;2;{};{};{}m{}\x1b[0m", $r, $g, $b, $text)
    };
    ($text:expr,$style:expr) => {
        format!("\x1b[{}m{}\x1b[0m", $style, $text)
    };
}
```

Adaptar/types.rs

```
pub enum PayeeKind {
    Card,
    Sbp,
}

pub enum PayeeAccount {
    Card {
        card_number: String,
        bank: String,
    },
    Sbp {
        sbp_number: String,
        bank: String,
        owner_name: String,
    },
}

pub enum TransactionState {
    New,
    Success,
    Failed,
    Cancelled,
}

pub struct Transaction {
    pub id: String,

    pub sum: f64,
    pub payee_account: PayeeAccount,
}

pub struct CreateTransactionPayload {
    pub requested_sum: f64,
```

```
    pub payee_kind: PayeeKind,
}

pub struct FinalizeTransactionPayload {
    pub id: String,

    pub new_state: TransactionState,
}
```

Adaptar/test.rs

```
pub use super::{Adapter, sber::*, tbank::*, types::*};

#[cfg(test)]
mod tests {
    use super::*;
    use uuid::Uuid;

    fn make_adapter() -> TBank {
        TBank::new()
    }

    #[test]
    fn create_transaction_card_ok() {
        let mut bank = make_adapter();

        let tx = bank
            .CreateTransaction(CreateTransactionPayload {
                requested_sum: 1234.56,
                payee_kind: PayeeKind::Card,
            })
            .expect("must create");

        assert!(!tx.id.is_empty());
        assert_eq!(tx.sum, 1234.56);
        match tx.payee_account {
            PayeeAccount::Card { ref bank, .. } => assert_eq!(bank, "TBank"),
            _ => panic!("expected card account"),
        }

        // Внутреннее состояние должно содержать статус New
```

```
    let id = Uuid::parse_str(&tx.id).unwrap();
    assert!(matches!(
        bank.transactions.get(&id),
        Some(TransactionState::New)
    ));
}

#[test]
fn create_transaction_sbp_ok() {
    let mut bank = make_adapter();

    let tx = bank
        .CreateTransaction(CreateTransactionPayload {
            requested_sum: 10.0,
            payee_kind: PayeeKind::Sbp,
        })
        .expect("must create");

    match tx.payee_account {
        PayeeAccount::Sbp {
            ref bank,
            ref owner_name,
            ..
        } => {
            assert_eq!(bank, "TBank");
            assert_eq!(owner_name, "Petr Petrov");
        }
        _ => panic!("expected sbp account"),
    }
}

#[test]
fn finalize_transaction_success_updates_state() {
    let mut bank = make_adapter();

    let tx = bank
        .CreateTransaction(CreateTransactionPayload {
            requested_sum: 50.0,
            payee_kind: PayeeKind::Card,
        })
        .unwrap();
```

```
        bank.FinalizeTransaction(FinalizeTransactionPayload {
            id: tx.id.clone(),
            new_state: TransactionState::Success,
        })
        .expect("finalize ok");

        let id = Uuid::parse_str(&tx.id).unwrap();
        assert!(matches!(
            bank.transactions.get(&id),
            Some(TransactionState::Success)
        ));
    }

    #[test]
    fn finalize_transaction_with_invalid_uuid_fails() {
        let mut bank = make_adapter();

        let err = bank
            .FinalizeTransaction(FinalizeTransactionPayload {
                id: "not-a-uuid".to_string(),
                new_state: TransactionState::Failed,
            })
            .expect_err("should fail");

        assert!(
            err.contains("invalid length") || err.contains("ParseError") ||
err.contains("invalid")
        );
    }

    #[test]
    fn finalize_transaction_not_found_fails() {
        let mut bank = make_adapter();
        let random_id = Uuid::new_v4().to_string();

        let err = bank
            .FinalizeTransaction(FinalizeTransactionPayload {
                id: random_id,
                new_state: TransactionState::Cancelled,
            })
            .expect_err("should fail, no such tx");
```

```
        assert_eq!(err, "Transaction not found");
    }
}
```

Adapter/tbank.rs

```
use super::{Adapter, types::*};
use std::collections::HashMap;
use uuid::Uuid;

pub struct TBank {
    pub transactions: HashMap<Uuid, TransactionState>,
}

impl TBank {
    pub fn new() -> Self {
        Self {
            transactions: HashMap::new(),
        }
    }

    pub fn GetFreeCard(&self) -> Option<PayeeAccount> {
        Some(PayeeAccount::Card {
            card_number: String::from("5536 9100 0000 0000"),
            bank: String::from("TBank"),
        })
    }

    pub fn GetFreeSBP(&self) -> Option<PayeeAccount> {
        Some(PayeeAccount::Sbp {
            sbp_number: String::from("+7 900 111-22-33"),
            bank: String::from("TBank"),
            owner_name: String::from("Petr Petrov"),
        })
    }

    /// Returns new transaction id
    pub fn NewPayment(&mut self) -> Result<Uuid, String> {
        println!("TBank initiate payment");
        let id = Uuid::new_v4();
        if let Some(_) = self.transactions.insert(id, TransactionState::New) {
            return Err(String::from("Transaction already exists"));
```

```
        }
        Ok(id)
    }

    pub fn UpdatePayment(&mut self, id: &Uuid, state: TransactionState) -> Result<(),
String> {
        println!("TBank update payment");
        match self.transactions.get_mut(id) {
            Some(v) => {
                *v = state;
                Ok(())
            }
            None => Err(String::from("Transaction not found")),
        }
    }
}

impl Adapter for TBank {
    fn CreateTransaction(
        &mut self,
        payload: CreateTransactionPayload,
    ) -> Result<Transaction, String> {
        let payee_account = match payload.payee_kind {
            PayeeKind::Card => self.GetFreeCard(),
            PayeeKind::Sbp => self.GetFreeSBP(),
        };

        if payee_account.is_none() {
            return Err(String::from("Payee account not found"));
        }

        let tx_id = self.NewPayment().map_err(|err| {
            eprintln!("{}", err);
            err
        })?;

        Ok(Transaction {
            id: tx_id.to_string(),
            sum: payload.requested_sum,
            payee_account: payee_account.unwrap(),
        })
    }
```

```
    fn FinalizeTransaction(&mut self, payload: FinalizeTransactionPayload) ->
Result<(), String> {
        let parsed_id = uuid::Uuid::parse_str(&payload.id).map_err(|err|
err.to_string())?;
        self.UpdatePayment(&parsed_id, payload.new_state)
    }
}
```

Adapter/sber.rs

```
use super::{Adapter, types::*};
use std::collections::hash_map::HashMap;
use uuid::Uuid;

pub struct Sberbank {
    pub transactions: HashMap<Uuid, TransactionState>,
}

impl Sberbank {
    pub fn new() -> Sberbank {
        Sberbank {
            transactions: HashMap::new(),
        }
    }

    pub fn GetFreeCard(&self) -> Option<PayeeAccount> {
        Some(PayeeAccount::Card {
            card_number: String::from("5469 3800 0000 0000"),
            bank: String::from("Sberbank"),
        })
    }

    pub fn GetFreeSBP(&self) -> Option<PayeeAccount> {
        Some(PayeeAccount::Sbp {
            sbp_number: String::from("+7 985 172-66-30"),
            bank: String::from("Sberbank"),
            owner_name: String::from("Ivan Ivanov"),
        })
    }

    ///  Returns new transaction id
```

```
    pub fn NewPayment(&mut self) -> Result<Uuid, String> {
        println!("Sberbank initiate payment");

        let id = Uuid::new_v4();
        if let Some(_) = self.transactions.insert(id.clone(), TransactionState::New) {
            return Err(String::from("Transaction already exists"));
        }

        Ok(id)
    }

    pub fn UpdatePayment(&mut self, id: &Uuid, state: TransactionState) -> Result<(),
String> {
        println!("Sberbank update payment");

        match self.transactions.get_mut(id) {
            Some(v) => (*v = state),
            None => return Err(String::from("Transaction not found")),
        }

        Ok(())
    }
}

impl Adapter for Sberbank {
    fn CreateTransaction(
        &mut self,
        payload: CreateTransactionPayload,
    ) -> Result<Transaction, String> {
        let payee_account = match payload.payee_kind {
            PayeeKind::Card => self.GetFreeCard(),
            PayeeKind::Sbp => self.GetFreeSBP(),
        };

        if payee_account.is_none() {
            return Err(String::from("Payee account not found"));
        }

        let tx_id = self.NewPayment().map_err(|err| {
            eprintln!("{}", err);
            err
        })?;
```

```
        Ok(Transaction {
            id: tx_id.to_string(),
            sum: payload.requested_sum,
            payee_account: payee_account.unwrap(),
        })
    }

    fn FinalizeTransaction(&mut self, payload: FinalizeTransactionPayload) ->
Result<(), String> {
        let parsed_id = Uuid::parse_str(&payload.id).map_err(|err| {
            eprintln!("{}", err);
            err.to_string()
        })?;

        self.UpdatePayment(&parsed_id, payload.new_state)
    }
}
```

Adapter/mod.rs

```
pub mod sber;
pub mod tbank;
pub mod test;
pub mod types;

use types::*;

pub trait Adapter {
    fn CreateTransaction(
        &mut self,
        payload: CreateTransactionPayload,
    ) -> Result<Transaction, String>;

    fn FinalizeTransaction(&mut self, payload: FinalizeTransactionPayload) ->
Result<(), String>;
}
```

Скриншоты работы программы

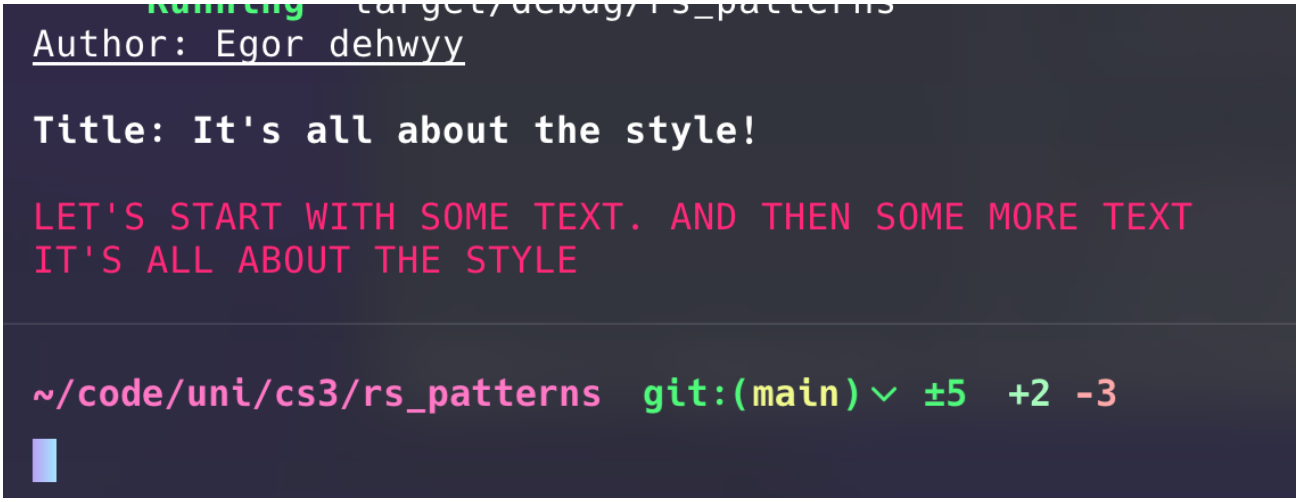

Рис. 2 – работа приложения