



ПРОФЕСИОНАЛНА ГИМНАЗИЯ
ПО ЕЛЕКТРОТЕХНИКА И ЕЛЕКТРОНИКА "АПОСТОЛ АРНАУДОВ"
гр. Русе, ул. "Потсдам" № 3; п.к. 7005, тел. 082/84-60-96; e-mail: electroschool@abv.bg

ДИПЛОМЕН ПРОЕКТ

ЗА ПРИДОБИВАНЕ НА ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА
КВАЛИФИКАЦИЯ

Тема: Разработка на софтуер за генериране
на изпитни билети

Ученик: Даниел Венциславов Василев

Професия: Приложен програмист (код 481030)

Специалност: Приложно програмиране (код 4810301)

Ръководител-консултант: Милена Дамесова

гр. Русе
2023

Съдържание

1	Увод	2
2	Използвани езици и технологии	3
2.1	Rust	3
2.1.1	Отличаващи се особености на езика	3
2.1.2	Enum	3
2.1.3	Option типа	4
2.1.4	Result типа	5
2.2	egui	6
2.2.1	Минимален пример	7
2.3	genpdf	8
2.3.1	Елементи	8
3	Подготовка	9
3.1	Инсталиране на Rust	9
3.2	Избор на IDE	10
3.2.1	Visual Studio Code	11
3.2.2	Vim	11
3.2.3	Neovim	12
3.3	Git	13
4	Разработка	14
4.1	Създаване на проекта	14
4.2	Добавяне на библиотеки	14
4.2.1	Конзолни аргументи	15
4.2.2	Serde	16
4.2.3	Toml	17
4.2.4	genpdf	17
4.3	Създаване на структури от данни	18
4.3.1	Derive macro	18
4.3.2	serde skip	18
4.3.3	serde default	19
4.3.4	enum Question	19
4.4	Генериране на PDF	20
4.5	Създаване на графичен интерфейс	20
4.5.1	Уведомления	21
5	Оптимизации	22
6	Компилиране за различни платформи	23
7	Заклучение	25

1 Увод

В настоящата дисертация ще бъде представен проект, който има за цел да улесни генерирането на изпитни билети за учебни заведения. Програмата е изградена на езика Rust и предлага лесен и ефективен начин за създаване на изпитни билети.

За да може един софтурен продукт да бъде завършен на време, трябва да задачите и целите на продукта да бъдат разделени на по-малки под задачи.

2 Използвани езици и технологии

2.1 Rust

Rust е програмен език от високо ниво създаден през 2006 година от Грейдън Хоаре, който по това време работи за Mozilla. През 2009 година разработката на езика бива спонсорирана от Mozilla, а през 2010 езика е обявен публично. [1]

2.1.1 Отличаващи се особености на езика

Езици като C#, Python и JavaScript използват система за освобождаване на паметта наречена Garbage Collector (GC). За да може да се освободят неизползваните променливи, изпълнението на програмата трябва да бъде спряно на пауза и да се провери дали има заделени региони от паметта, към които вече не се използват или са маркиране за освобождаване от програмиста [2].

Rust използва система наречена borrow checker, която проверява, по време на компилация, дали програмата следва следните принципи:

- Ресурсите (отделената памет за стойността) могат да имат само един собственик и това е самата променлива. Когато променлива вече не може да бъде достъпена ресурсите биват освободени.
- Когато една променлива бъде подадена към някоя функция, собственик на ресурсите става функцията. Ако се пробваме подадем отново променливата, компилатора ще ни каже, че променливата е била преместена (Use of moved value).

2.1.2 Enum

Enum е един от основните типове в Rust. Всеки вариант на enum-а може да има съдържа информация от различен вид [3]. Така са имплементирани някои от най-важните типове: `Option<T>` и `Result<T, E>`.

```

9 fn print_string(data: String) {
8     println!("{}", data);
7 }
6
5 fn main() {
4     let owner_of_data = String::from("Hello, World!"); => String
3
2     print_string(owner_of_data);
1
10    print_string(owner_of_data);
1 }

```

Diagnostics:

1. use of moved value: `owner_of_data`
value used here after move

Фигура 1: Модела на собственик в Rust

```

1 enum Option<T> {
  1     Some(T),
  2     None,
  3 }
  4
  5 enum Result<T, E> {
  6     Ok(T),
  7     Err(E),
  8 }

```

Фигура 2: Стандартната имплементация на Option<T> и Result<T, E>

2.1.3 Option типа

В повечето езици съществува идеята за NULL пойнтери. Когато един pointer е Null това означава, че той сочи към нищо. Идеята за Null на теория е много добра, но на практика създава повече проблеми. Ако се пробваме да достъпим pointer който е Null, програмата ще крашне или в някои езици като C# ще хвърли `NullReferenceException`.

Разработчиците на Rust са намерили много добър заместител на Null и това е Option enum-а, който има два варианта. Това са `Some(T)`, когато имаме някаква стойност и `None`, когато нямаме нищо.

2.1.4 Result типа

Когато програмираме на C# много често ни се случва да хвърляме Exception-и и съответно да ги хващаме с try/catch блока. Exception-ите се ползват, когато в една функция възникне грешка.

Във Фигура 3 е даден код, който на пръв поглед изглежда добре, но има скрити бъгове. Какво ще стане, ако потребителят въведе дума вместо число? Ще получим Exception, който ни казва: "Input string was not in a correct format".

```
1  int ReadNumberFromUser()  
1  {  
2      string user_input = Console.ReadLine();  
3      int parsed_int = int.Parse(user_input);  
4      return parsed_int;  
5  }  
6  
7  int number = ReadNumberFromUser();  
8  Console.WriteLine($"{number} * 2 = {number * 2}");
```

Фигура 3: Пример за скрит Exception

```
~/Programming/C#/Scratchpad  
> dotnet run  
word  
Unhandled exception. System.FormatException: Input string was not in a correct format.  
   at System.Number.ThrowOverflowOrFormatException(ParsingStatus status, TypeCode type)  
   at System.Number.ParseInt32(ReadOnlySpan`1 value, NumberStyles styles, NumberFormatInfo info)  
   at System.Int32.Parse(String s)  
   at Program.<<Main>>g__ReadNumberFromUser|0_0() in /home/daniel/Programming/C#/Scratchpad/Program.cs:line 4  
   at Program.<Main>$(String[] args) in /home/daniel/Programming/C#/Scratchpad/Program.cs:line 8
```

Фигура 4: Изход на кода от Фигура 3

Проблемът, е че ние като програмисти не знаем, че *int.Parse* може да хвърли Exception без да се консултираме с документацията [4]. Същият код написан на Rust би изглеждал по следния начин [Фигура 5].

Разликата между C# и Rust, е че Rust кода ни показва типовете при успех и грешка. Функцията връща променлива от тип *Result<T, E>*, където T е променливата от тип *i32* (*int*), ако всичко се и изпълнило без проблем, а E е от тип *ParseIntError*.

```

1 use std::num::ParseIntError;
2
3 fn read_number_from_user() -> Result<i32, ParseIntError> {
4     let mut user_input = String::new(); => String
5     std::io::stdin().read_line(&mut user_input).unwrap(); <- (buf)
6     return user_input.trim().parse();
7 }
8
9 fn main() {
10     let number = read_number_from_user(); => Result<i32, ParseIntError>
11
12     match number {
13         Ok(number) => { => i32
14             // NOTE принтира резултата
15             println!("{}", number * 2);
16         }
17         Err(err) => { => ParseIntError
18             // NOTE принтира грешката в конзолата, но не crash-ва!
19             println!("{:?}", err);
20         }
21     }
22 }

```

Фигура 5: Кода от Фигура 3 написан на Rust

За да използваме резултата от функцията, какъвто и да е той, можем да използваме `match`. С `match` можем да проверим дали резултата е `Ok` или `Err`.

2.2 egui

`egui` е проста, бърза и много преносима библиотека за графични потребителски интерфейси (GUI). Тя работи на много платформи включително: уеб браузъри, като обикновено приложение или в някои `game engine`-а. Написана е на Rust и има много лесен и интуитивен API за разработване.

Главните цели на проекта са:

- Най-лесната за използване GUI библиотека;
- Отзивчив: цели поне 60 FPS при компилация с `Debug` опциите;
- Преносим: кодът да работи в браузър и като собствено приложение;
- Лесен за интегриране във всяка среда;

- Модулен: можете да използвате малки части от egui и да ги комбинирате по нови начини
- Минимален брой зависимости (библиотеки)..-

2.2.1 Минимален пример

egui библиотеката ни дава достъп до App интерфейса. Този интерфейс съдържа една функция `update`. Тя се извиква всеки път когато потребителският интерфейс се е променил или се получи някакво събитие (Event) от мишката или клавиатурата. [Фигура 6]

```

1
2 impl eframe::App for MyApp {
3     fn update(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame) {
4         egui::CentralPanel::default().show(ctx, |ui| {
5             ui.heading("My egui Application");
6             ui.horizontal(|ui| {
7                 let name_label = ui.label("Your name: ");
8                 ui.text_edit_singleline(&mut self.name)
9                     .labelled_by(name_label.id);
10            });
11            ui.add(egui::Slider::new(&mut self.age, 0..=120).text("age"));
12            if ui.button("Click each year").clicked() {
13                self.age += 1;
14            }
15            ui.label(format!("Hello '{}', age {}", self.name, self.age));
16        });
17    }
18 }
19

```

Фигура 6: Имплементация на egui интерфейса

Библиотеката е достатъчно умна сама да прецени дали се нуждае от повторно изобразяване, или не.

egui е GUI библиотека от незабавен режим (Immediate Mode). Това означава, че начина, по който искаме да изглежда графичния интерфейс, се описва, извиквайки методи. По този начин се упростиля разработката на графичния интерфейс. За да покажем един прост бутон се нуждаем от един `if` оператор.

2.3 genpdf

genpdf е библиотека, която абстрахира създаването на PDF файлове. Тя се грижи за оформлението на страницата, подравняването на текста и изобразяването на структурата на документа в PDF файл. Библиотеката както и всичките ѝ зависимости са написани на Rust и следват добрите практики на езика [5].

2.3.1 Елементи

genpdf използва елементи, за да опише оформлението на документа. Всеки елемент имплементира Element интерфейса. Интерфейсът съдържа функция render, която бива извикана всеки път когато елементът трябва да бъде показан в PDF файла [6].

Използвайки Element интерфейса, разработчиците на genpdf са ни предоставили най-често използваните елементи:

- Контейнери:

LinearLayout: подрежда елементите си последователно;

TableLayout: подрежда елементите си в колони и редове;

OrderedList/UnorderedList: подреждат елементите им последователно с bullet-и.

- Текст:

Text: един ред текст;

Paragraph: подравнен параграф.

- Обвивки:

FramedElement: елемент с рамка;

PaddedElement: добавя разстояние между елементите;

StyledElement: задава стил по подразбиране за обвития елемент и неговите деца (елементите, които му принадлежат).

- Други:

Image: снимка с описание;

Break: добавя прекъсвания на редове като разделител;

PageBreak: добавя принудително прекъсване на страницата.

3 Подготовка

Преди да започнем с разработката на софтуера трябва да настроим нашата среда за действието. Тя включва Rust компилатора, текстов редактор (IDE) и Git за контрол на версиите.

3.1 Инсталиране на Rust

Rust е език, който използва LLVM за обръщането на код в машинни инструкции. LLVM е набор от технологии за компилиране, които позволява да бъдат написани различни frontend-ове за всеки език и backend-ове за всяка хардуерна архитектура. Благодарение на този факт, Rust може да работи на всички модерни операционни системи като Windows, MacOS, Linux, OpenBSD и още много други.

За операционните системи базирани на UNIX принципите, като MacOS и Linux можем да инсталираме Rust с една проста команда:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

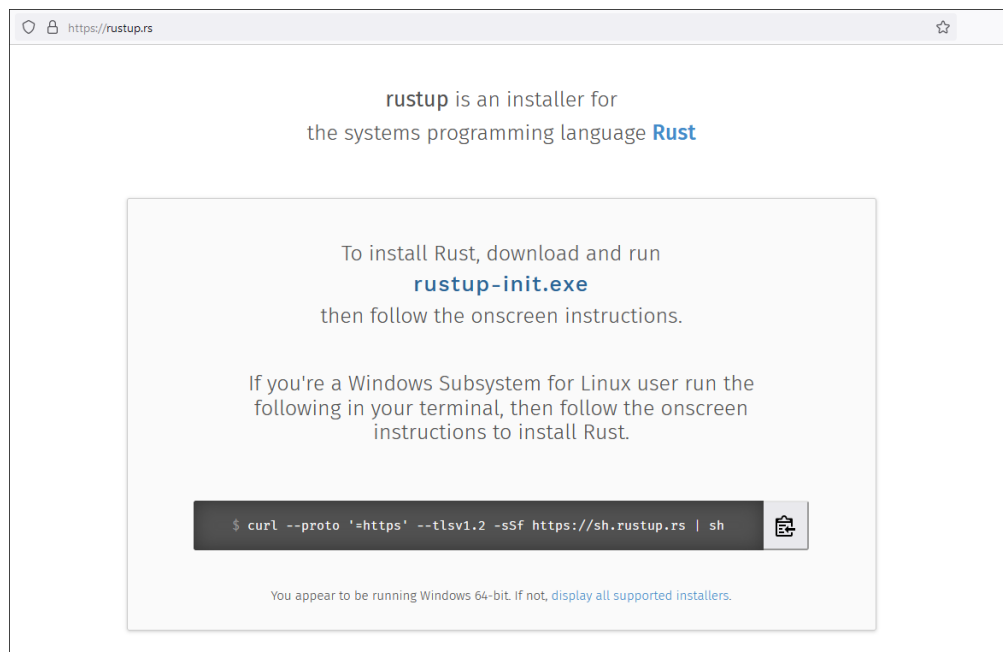
Или през package manager-a на операционната система. В MacOS можем да използваме Homebrew. За да инсталираме Rust, трябва да изпълним следните команди:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew  
/install/HEAD/install.sh)"  
  
brew install rust
```

Когато използваме Linux и искаме да инсталираме компилатора, трябва да се съобразим с това каква дистрибуция използваме. Най-често използваните такива са:

- Arch Linux - pacman -S rustup
- Debian Linux - apt-get install rustup
- Fedora Linux - dnf install rust cargo

За да инсталираме Rust на Windows, трябва да изтеглим 64 или 32 битовия инсталационен файл от сайта на Rust: <https://rustup.rs/>.



Фигура 7: Сайт за изтегляне на инсталационния файл на Rust за Windows

3.2 Избор на IDE

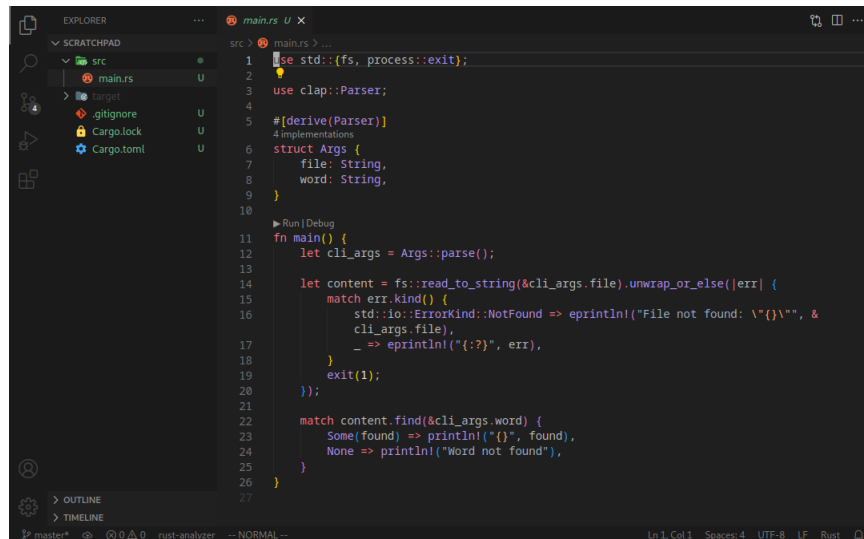
За да можем да създаваме софтуер на Rust, по най-ефективния начин, се нуждаем от текстов редактор, който поддържа LSP (Language Server Protocol). Този протокол е създаден от Microsoft за Visual Studio Code и служи за комуникация между текстовия редактор и специализирани програми, които анализират кода, който пишем и показват къде има грешки, предложения как да бъдат поправени, допълнение на код, подчертаване на синтаксиса [7].

3.2.1 Visual Studio Code

Visual Studio Code е текстов редактор с отворен код, създаден от Microsoft. Той използва Electron за графична библиотека и работи на операционните системи: Windows, Linux и MacOS. През 2022 година в допитване до потребителите на Stack Overflow Visual Studio Code е класиран като най-популярният текстов редактор сред 71 010 респонденти [8].

Visual Studio Code със заводските си настройки не може да прави почти нищо. За да получим всички полезни функционалности на LSP, трябва да инсталираме така наречените плъгини.

Един от недостатъците на Visual Studio Code обаче е високите системни изисквания за нормална работа и моментално време за реакция при въвеждане на текст.



Фигура 8: Visual Studio Code

3.2.2 Vim

Vim е текстов редактор с отворен код, първоначално написан за настолния компютър Amiga през 1991 година от Брам Моленар. За разлика от Visual Studio Code, Vim е текстов редактор, който е бил замислен да работи не само в графични среди, но и в терминални среди [9].

Най-привекателната част от Vim е начина за навигация. В повечето редактори се навигира чрез мишката и няколко клавишни комбинации, докато Vim използва само клавишни комбинации. По този начин ръцете ни остават на клавиатурата и няма нужда да отделяме време за навигиране с мишка.

Vim разполага с 3 режима за работа и това са:

- Normal - В нормалния режим можем само да манипулираме текст;
- Visual - В визуалния режим можем да избираме по-големи региони от текст;
- Command - В командния режим можем да изпълняваме команди за манипулиране на текст, настройване на редактора и изпълнение на команди в операционната система.

Vim успява да използва по-малко ресурси от Visual Studio Code, без да прави компромиси от към функционалности. За да може да бъде постигнат Vim е написан на C и потребителят трябва ръчно да си настрои редактора, използвайки специално направения език VimScript. Този процес е прекалено сложен за повечето потребители, за това те предпочитат да използват Visual Studio Code, дори и да използва повече ресурси.

3.2.3 Neovim

Neovim е копие на Vim, което се стреми да подобри скоростта и поддръжката на Vim. Някои добавени функционалности на копието включват вградена поддръжка на LSP и поддръжка за Lua скриптове като заместител на VimScript [10].

Проектът Neovim стартира през 2014 година, като някои членове на Vim общността предлагат помощ в усилията за основно рефакториране на кода, за да осигурят по-добри езици за скриптове, плъгини и интеграция с модерни графични потребителски интерфейси. Проектът е с отворен код, който е достъпен в GitHub. [11]

Откъм производителност Neovim е малко по-бавен от предшественика си Vim, но все пак е в пъти по-бърз от главния си конкурент Visual Studio Code, затова аз се спрях на Neovim.

```
1 use std::{fs, process::exit};
2
3 use clap::Parser;
4
5 #[derive(Parser)]
6 struct Args {
7     file: String,
8     word: String,
9 }
10
11 fn main() {
12     let cli_args = Args::parse(); => Args
13
14     let content = fs::read_to_string(&cli_args.file).unwrap_or_else(|err| { <- (op, path) => String, Error
15         match err.kind() {
16             std::io::ErrorKind::NotFound => eprintln!("File not found: \"{}\"", &cli_args.file),
17             _ => eprintln!("{}", err),
18         }
19         exit(1); <- (code)
20     });
21
22     match content.find(&cli_args.word) {
23         Some(found) => println!("{}", found), => usize
24         None => println!("Word not found"),
25     }
26 }
```

NORMAL P master main.rs • rust Top 1:1

Фигура 9: Neovim

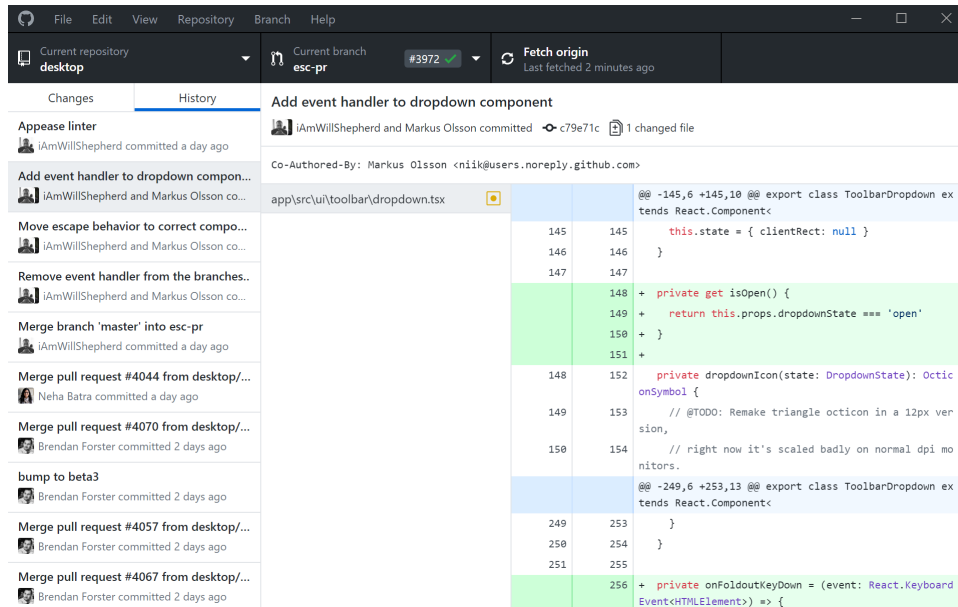
3.3 Git

Git е система за контрол на версиите, която проследява промените във всеки набор от компютърни файлове. Обикновено се използва за координиране на работата между програмистите, които съвместно разработват изходния код по време на разработката на софтуер. Целите на системата включват скорост, цялост на данните и поддръжка за разпределени, нелинейни работни потоци (хиляди паралелни клонове, работещи на различни системи).

Git първоначално е създаден от Линус Торвалдс, през 2005 година за разработване на Linux ядрото, като инструмент за други разработчици, които допринасят за първоначалното му развитие. От 2005 година Джуино Хамано е основният разработчик. Както при повечето други разпределени системи за контрол на версиите и за разлика от повечето системи клиент-сървър, всяка Git директория на всеки компютър е пълноценно хранилище с пълна история и пълни възможности за проследяване на версиите, независимо от достъпа до мрежата или централен сървър. Git е безплатен софтуер с отворен код, разпространяван само под лиценз GPL-2.0. [12]

Git е конзолно приложение, но съществуват и приложения с графичен интерфейс. Най-често използваното такова е GitHub Desktop. Това приложение ни дава

възможност да управляваме разработката на проекта, да създаваме потребителски истории и тяхното управление. [Фигура 10] [13]



Фигура 10: Github Desktop в Windows

4 Разработка

4.1 Създаване на проекта

За да създадем нов Rust проект, първо трябва да отворим конзолата и да изпълним следната команда:

```
cargo new test-generator
```

Тя ще генерира папка с името test-generator, в която се намира проекта. В src папката се намират файловете, в които пишем кода, а в Cargo.toml файла се намира конфигурацията на проекта.

За да отворим проекта в текстов редактор, може да напишем командата 'code .' за да го отворим във Visual Studio Code или 'nvim .' за да го отворим във Neovim.

4.2 Добавяне на библиотеки

За да добавим библиотека в Rust трябва да я добавим в Cargo.toml файла под секцията наречена dependencies. Синтакса за добавяне е много прост, а именно

```

→ ~ cargo new test-generator
   Created binary (application) `test-generator` package
→ ~ tree test-generator
test-generator
├── Cargo.toml
└── src
    └── main.rs
→ ~

```

Фигура 11: Файловата структура на Rust проекта

в ляво името на библиотеката, посредата знака за равно и вдясно версията на библиотеката.

```

14 [[dependencies]]
1  anyhow = "1.0"      ✓ 1.0.70
2  serde = { version = "1.0", features = ["derive"] }
3  toml = "0.7.2"      ✓ 0.7.3
4  clap = { version = "4.1.8", features = ["derive"] }
5  rckive-genpdf = "0.4.0" ✓ 0.4.0
6  rand = "0.8.5"      ✓ 0.8.5
7  egui = "0.21.0"     ✓ 0.21.0
8  eframe = "0.21.3"   ✓ 0.21.3
9  egui-notify = "0.6.0" ✓ 0.6.0
10 ilog = "1.0.1"      ✓ 1.0.1

```

Фигура 12: Ситаксис за добавяне на библиотеки

4.2.1 Конзолни аргументи

Аргументите на конзолата са параметри, предавани на програмата преди изпълнение и в командния ред. В Rust аргументите могат да бъдат достъпени чрез функцията `std::env::args()`, която връща итератор над аргументите като списък от низове.

За да вземем подходящата информация за приложението, може да напишем наш собствен анализатор или да използваме една от многото различни библиотеки за работа с конзолни аргументи. Една от най-често използваните библиотеки е *clap* (Console Line Argument Parser).

Clap ни предоставя с `clap::Parser` макрото, което при компилирането на програмата анализира структурата от данни и автоматично търси командните аргументи при изпълнение на програмата. Също така, проверява кои аргументи са марки-

рани като задължителни или такива със стойност по подразбиране. [Фигура 13]

```
1 use clap::Parser;
2
3 #[derive(Parser)]
4 struct Args {
5     path: String,
6 }
7
8 fn main() {
9     let args = Args::parse(); => Args
10    println!("Path => {}", args.path);
11 }
```

Фигура 13: Четене на командни аргументи с *clap*

При въвеждане на грешни аргументи или при липсата на задължителните такива, *clap* показва автоматично генерираното помощно съобщение на потребителя.

```
→ scratchpad git:(master) x cargo run -q
error: the following required arguments were not provided:
  <PATH>

Usage: scratchpad <PATH>

For more information, try '--help'.
→ scratchpad git:(master) x cargo run -q -- /home/user
Path => /home/user
→ scratchpad git:(master) x
```

Фигура 14: Генерираното помощно съобщение от *clap*

4.2.2 Serde

Serde е framework за ефективно сериализиране и десериализиране на структури от данни в Rust.

Екосистемата на *Serde* се състои от структури от данни, които знаят как да сериализират и десериализират себе си заедно с формата на данните. *Serde* предос-

тавя слой, чрез който тези две групи взаимодействат помежду си, позволявайки всяка поддържана структура от данни да бъде сериализирана и десериализирана с помощта на всеки поддържан формат на данни.

Докато много други езици разчитат на runtime среда (като Dotnet) за сериализиране на данни, *Serde* вместо това е изградена върху много добрата интерфейс система на Rust.

Структура от данни, която знае как да сериализира и десериализира сама себе си, е тази, която използва интерфейсите на *Serde* за сериализиране и десериализиране (или използва атрибута *derive* на *Serde* за автоматично генериране на интерфейси по време на компилация). По този начин се избягват забавянето от употребата на runtime среда.

Всъщност в много ситуации взаимодействието между структурата на данните и формата на данните може да бъде напълно оптимизирано от Rust компилатора, оставяйки сериализацията на *Serde* да се изпълни със същата скорост като ръчно написан сериализатор в езици от по-ниско ниво като *C*.

4.2.3 Toml

Toml (Tom's Obvious Minimal Language) е файлов формат за съхранение на софтуерни конфигурации. Този формат ще бъде използван за съхраняване на настройките, въпросите и друга информация за тестовете. За да добавим поддръжка за този формат в Rust, трябва да инсталираме *Toml* библиотека, която използва *Serde* за преобразуването на файл в обект и обратно.

4.2.4 genpdf

genpdf е библиотека за генериране на PDF документи на високо ниво, изградена от две по-малки библиотеки от по-ниско ниво. Тя се грижи за оформлението на страницата, подравняването на текста и изобразява дървовидната структура на документа в PDF файла.

4.3 Създаване на структури от данни

4.3.1 Derive macro

In Rust, a derive macro is a feature that allows you to automatically generate code for a type by annotating it with a special attribute. The annotation tells the Rust compiler to generate code for the annotated type based on a predefined set of rules.

Derive macros are implemented as procedural macros, which means that they take Rust code as input, transform it, and produce new Rust code as output. The output can be either an implementation of a trait or a completely new type definition.

To use a derive macro in Rust, you simply add the `#[derive]` attribute followed by the name of the macro to the definition of a struct, enum, or other type. For example, to derive the `Debug` trait for a struct, you can write:

This will automatically generate an implementation of the `Debug` trait for `MyStruct`, which will print out the values of its fields when the `println!()` macro is used with an instance of the struct.

Derive macros are very powerful and can be used to generate code for many other common traits and features, such as `PartialEq`, `Eq`, `Clone`, and `Hash`. They can also be used to generate code for custom traits and features that you define yourself.

Overall, derive macros are a powerful feature in Rust that allows you to write less code and reduce boilerplate. They are easy to use and can be a great way to generate common functionality for your types.

4.3.2 serde skip

When a field is annotated with `#[serde(skip)]`, `serde` will not include that field in the serialized output, and will skip over it when deserializing input data. This can be useful when you have a struct or enum with fields that you do not want to include in the serialized output, or when you want to ignore certain fields in input data.

4.3.3 serde default

When a field is annotated with `#[serde(default)]`, serde will use the default value of the field if it is missing from the input data. This can be useful when you have a struct or enum with optional fields, or when you want to ensure that certain fields always have a value.

4.3.4 enum Question

Потребителят ще може да избира между два вида въпроси - с отворен отговор (Input) и с избиране на подточки (Selection). Разликата между въпросите с отворен отговор и избиране, са че за отворения отговор трябва да знаем колко реда за отговор да бъдат дадени, а при въпроса с избиране трябва потребителя да каже кои са правилните отговорите.

```
1  #[derive(Deserialize, Serialize, Clone)]
2  pub struct SelectionQuestion {
3      #[serde(skip)]
4      pub question_buf: String,
5      pub question: String,
6      pub correct: Vec<String>,
7      pub incorrect: Vec<String>,
8      #[serde(default = "default_points")]
9      pub points: u8,
10 }
11
12 #[derive(Deserialize, Serialize, Clone)]
13 pub struct InputQuestion {
14     #[serde(skip)]
15     pub question_buf: String,
16     pub question: String,
17     pub number_of_lines: u16,
18     #[serde(default = "default_points")]
19     pub points: u8,
20 }
21
22 pub enum Question {
23     Selection(SelectionQuestion),
24     Input(InputQuestion),
25 }
```

Фигура 15: Въпроси

4.4 Генериране на PDF

За да генерираме PDF файл със `genpdf` библиотеката, трябва да създадем нов обект от тип `genpdf::Document`. Конструктора на този обект иска да му бъде подадена пътека към фонта, който да бъде използван за генерирането. Библиотеката ни позволява да добавим и `PageDecorator`, чрез който можем да добавим поле около страницата, заглавия и `footer`-и за номера на страницата. В този случай ще използваме само полето около страницата.

След като сме подготвили обекта, който представлява PDF документа, можем да започнем да добавяме нашите собствени елементи. Първите такива елементи ще бъдат заглавието на теста, името на ученика, клас и номер в клас. Заглавието на теста ще е с най-големия шрифт и ще е центриран, а под него името, класа и номер на един ред.

4.5 Създаване на графичен интерфейс

Най-важната част от графичният интерфейс са табовете. Чрез тях се избира коя част от графичния интерфейс да бъде показана - въпросите, конфигурацията на проекта или настройките на графичния интерфейс.

За да знаем кой от трите таба е отворен използваме Rust `enum`. `Egui` библиотеката ни позволява директно да и подадем глобален обект от този тип и тя автоматично ще му смени варината когато някой бутон е натиснат [Фигура 16]. А във функцията който изобразява елементите използваме `match` за да извикаме съответната функция за избраният таб [Фигура 18].

```

60 ui.selectable_value(
1   &mut self.gui_state.opened_tab,
2   OpenedTab::Questions,
3   "Questions",
4 );
5 ui.selectable_value(
6   &mut self.gui_state.opened_tab,
7   OpenedTab::Configuration,
8   "Configuration",
9 );
10 ui.selectable_value(
11   &mut self.gui_state.opened_tab,
12   OpenedTab::Settings,
13   "Settings",
14 );

```

Фигура 16: Flower one.

```

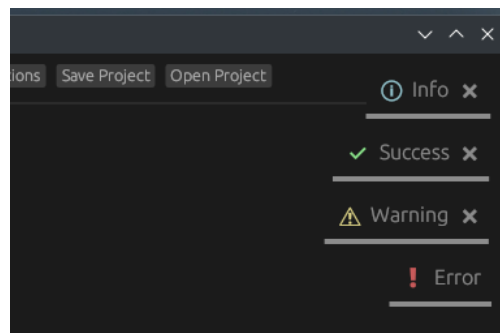
281 ScrollArea::vertical().show(ui, |ui| {
1   match self.gui_state.opened_tab {
2       OpenedTab::Questions => self.draw_questions(ui),
3       OpenedTab::Configuration => self.draw_configuration(ui),
4       OpenedTab::Settings => self.draw_settings(ui, ctx),
5   }
6 });

```

Фигура 17: Flower two.

4.5.1 Уведомления

За да съобщим на потребителя за статуса на програмата ще използваме `egui-notify` библиотеката. Тя предоставя обект, който съдържа няколко метода: `info`, `success`, `warning` и `error`. Тези методи ще покажат на потребителя различните видове уведомления.

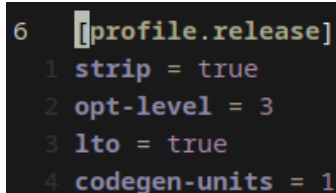


Фигура 18: Различните видове уведомления от `egui-notify`

5 Оптимизации

По подразбиране Rust компилра кода без никакви оптимизации и с много символи които помагат за по лесното дебъгване на програмата. Когато компилираме кода за употреба от нашите потребители, искаме нашата програма да е възможно най-бърза и с най-малък размер. Но преди това трябва да кажем на компилатора как по-точно искаме да бъде оптимизирана програмата. Това се случва като добавим следните 5 реда в Cargo.toml [Фигура 19].

- strip - премахва символите за повечето дебъгване;
- opt-level - колко пъти компилатора да се пробва да оптимизира кода;
- lto - извършва оптимизации при свързването (linking) на всички библиотеки с програмата;
- codegen-units - колко паралелни нишки да бъдат създадени за по-бързо компилиране, когато се използва само 1 нишка кода може да бъде оптимизиран по-добре.



```
6 [[profile.release]]
1 strip = true
2 opt-level = 3
3 lto = true
4 codegen-units = 1
```

Фигура 19: Допълнителни настройки за оптимизиране на кода

За да кажем на компилатора да компилира програмата с всички оптимизации, можем да подадем `--release` флага на компилатора.

```
cargo build --release
```

Разликата между двата файла е огромна. Не оптимизирания файл е цели 272 мегабайта, а оптимизирания едва 10 мегабайта [Фигура 20]. Освен птимизации за размер, Rust прилага и оптимизации за време.

```

→ test-generator git:(master) x du -h target/debug/test-generator
273M   target/debug/test-generator
→ test-generator git:(master) x du -h target/release/test-generator
9.8M   target/release/test-generator
→ test-generator git:(master) x █

```

Фигура 20: Разлика в размера при Debug и Release компилиране

Тестовите за време са извършени при 1000 итерации и времето е изчислено следно-аритметично и като медиана. Средното времето за изпълнение пада от 20 милисекунди на по-малко то 2 милисекунди, а медианата от почти 43 милисекунди на 3. Между двата начина за тестване, скоростта на изпълнение се увеличава повече от 10 пъти [Фигура 21].

```

→ test-generator git:(master) x cargo run -q -- --perf-test project.toml
Testing PDF generation 1000 times
Median: 43.8387ms
Average: 20.759313ms
→ test-generator git:(master) x cargo run -q --release -- --perf-test project.toml
Testing PDF generation 1000 times
Median: 3.572469ms
Average: 1.76593ms
→ test-generator git:(master) x █

```

Фигура 21: Разлика в скоростта на изпълнение при Debug и Release компилиране

6 Компилиране за различни платформи

За да компилирате Rust за различни платформи, ще трябва да инсталирате toolchain и след това компилираме проекта. За да видим различните видове toolchain-и, можем да използваме следната команда:

```
rustup target list
```

Тази команда ще изведе имената за над 90 различни поддържани платформи, но ще се фокусираме само върху Windows и Linux. Те отговарят на имената "x86-64-pc-windows-gnu" и "x86_64-unknown-linux-gnu".

За автоматичното компилиране и архивиране на файловете за различните платформи можем да създадем един Bash скрипт [Фигура 22]. Този скрипт автоматично ще инсталира toolchain-а ако вече не е, компилира проекта и създава zip архив със файла за изпълнение.


```

1  #!/bin/sh
2
3  set -e
4
5  build_for() {
6      TOOLCHAIN=$1
7      EXE=$2
8
9      rustup target add $TOOLCHAIN
10     cargo build --target "$TOOLCHAIN" --release
11
12     rm $1.zip
13     7z a $1.zip ./target/$TOOLCHAIN/release/$EXE ./assets
14 }
15
16 rustup default stable
17 build_for "x86_64-pc-windows-gnu" "test-generator.exe"
18 build_for "x86_64-unknown-linux-gnu" "test-generator"

```

Фигура 22: Bash скрипт за компилиране и архивиране на проекта

На 3ти ред във Фигура 22 дефинираме функция *build_for* която приема като аргументи името на toolchain-a и името на компилирания файл. Функцията компилира и проекта използвайки всички оптимизации и го архивира във файл със същиятот име като името на toolchain-a.

Тези архиви могат да бъдат разпространени в интернет и всеки потребител да изтегли съответната версия за неговата платформа и да използва софтуерния продукт.

7 Заключение

Литература

- [1] Wikipedia contributors. *Rust (programming language)* — *Wikipedia, The Free Encyclopedia*. 2023. URL: [https://en.wikipedia.org/w/index.php?title=Rust_\(programming_language\)&oldid=1146879721#Origins_\(2006%E2%80%93932012\)](https://en.wikipedia.org/w/index.php?title=Rust_(programming_language)&oldid=1146879721#Origins_(2006%E2%80%93932012)).
- [2] Wikipedia contributors. *Garbage collection (computer science)* — *Wikipedia, The Free Encyclopedia*. 2023. URL: [https://en.wikipedia.org/w/index.php?title=Garbage_collection_\(computer_science\)&oldid=1146816153](https://en.wikipedia.org/w/index.php?title=Garbage_collection_(computer_science)&oldid=1146816153).
- [3] Rust contributors. *Defining an Enum*. 2021. URL: <https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html>.
- [4] Microsoft. *Int32.Parse Method*. 2023. URL: <https://learn.microsoft.com/en-us/dotnet/api/system.int32.parse?view=net-8.0>.
- [5] ireas. *genpdf-rs - User-friendly PDF generator written in pure Rust*. 2021. URL: <https://git.sr.ht/~ireas/genpdf-rs>.
- [6] ireas. *docs.rs - Trait genpdf::Element*. 2021. URL: <https://docs.rs/genpdf/0.2.0/genpdf/trait.Element.html>.
- [7] Wikipedia contributors. *Language Server Protocol* — *Wikipedia, The Free Encyclopedia*. 2023. URL: https://en.wikipedia.org/w/index.php?title=Language_Server_Protocol&oldid=1146435481.
- [8] Wikipedia contributors. *Visual Studio Code* — *Wikipedia, The Free Encyclopedia*. 2023. URL: https://en.wikipedia.org/w/index.php?title=Visual_Studio_Code&oldid=1147584690.
- [9] Wikipedia contributors. *Vim (text editor)* — *Wikipedia, The Free Encyclopedia*. 2023. URL: [https://en.wikipedia.org/w/index.php?title=Vim_\(text_editor\)&oldid=1147168005](https://en.wikipedia.org/w/index.php?title=Vim_(text_editor)&oldid=1147168005).
- [10] Wikipedia contributors. *Neovim* — *Wikipedia, The Free Encyclopedia*. 2023. URL: [https://en.wikipedia.org/w/index.php?title=Vim_\(text_editor\)&oldid=1147168005#Neovim](https://en.wikipedia.org/w/index.php?title=Vim_(text_editor)&oldid=1147168005#Neovim).

- [11] Neovim contributors. *neovim - GitHub User*. 2023. URL: <https://github.com/neovim/neovim/>.
- [12] Wikipedia contributors. *Git — Wikipedia, The Free Encyclopedia*. 2023. URL: <https://en.wikipedia.org/w/index.php?title=Git&oldid=1145875787>.
- [13] Microsoft. *GitHub Desktop / Simple collaboration from your desktop*. 2023. URL: <https://desktop.github.com>.