



ВИСШЕ ВОЕННОМОРСКО УЧИЛИЩЕ „Н. Й. ВАПЦАРОВ“

9002 Варна, ул. „В. Друмев“ 73, тел. 052/632-015, факс 052/303-163



“FILII MARIS SUMUS”

ФАКУЛТЕТ „ИНЖЕНЕРЕН“ - КАТЕДРА „ИНФОРМАЦИОННИ ТЕХНОЛОГИИ“



ДИПЛОМНА РАБОТА

за придобиване на ОКС „МАГИСТЪР“

**ТЕМА: РАЗРАБОТКА НА СОФТУЕР ЗА ГЕНЕРИРАНЕ НА
СПРАВКИ И ОТЧЕТИ**

Дипломант: Николай
Живков Николов
Специалност: Киберсигурност
фак. №: 626-201-13

Научен ръководител:
Подполковник
Драгомир Драгнев

гр. Варна
2022 г.



ВИСШЕ ВОЕННОМОРСКО УЧИЛИЩЕ „Н. Й. ВАПЦАРОВ“

9002 Варна, ул. „В. Друмев“ 73, тел. 052/632-015, факс 052/303-163



Заверка на Деканата
Семестриално завършил!

Дата: _____

“FILII MARIS SUMUS”

УТВЪРЖДАВАМ:

ДЕКАН НА ФАКУЛТЕТ ИНЖЕНЕРЕН”

КАПИТАН I РАНГ ДОЦ. Д-Р _____

..... Г.

ЗАДАНИЕ

за дипломна работа на Николай Живков Николов

Тема : РАЗРАБОТКА НА СОФТУЕР ЗА ГЕНЕРИРАНЕ НА СПРАВКИ И ОТЧЕТИ

1. Изходни данни:

Софтуер за генериране на справки и от чети.
Обяснителна записка към дипломната работа.

2. Съдържание на обяснителната записка:

Теоретичен анализ.
Практическа имплементация на задание.

3. Експериментална част:

Софтуер за генериране на справки и отчети.

4. Дата на връчване на заданието: 10.01.2022

5. Дата на предаване на завършения дипломен проект в катедрата:
07.03.2022

Дипломант: Николай Живков
Николов

Н-к катедра „Информационни технологии”

/ / _____

/полк. доц. д-р инж. _____ Юлиан Цонев/

Ръководител: подполк. Драгомир Драгнев

//

Съдържание

1.	Увод	6
2.	Литературен анализ и използвани технологии	7
2.1.	Сравнение на възможни имплементации	7
2.1.1.	Java	7
2.1.2.	C++	9
2.1.3.	Python.....	10
2.1.4.	Избор.....	11
2.2.	Използвани технологии.....	13
2.2.1.	Visual Studio Code.....	13
2.2.2.	GitHub	14
2.2.3.	Модули на Python	15
3.	Имплементация.....	23
3.1.	Обобщение.....	23
3.2.	Логически проблеми.....	24
3.2.1.	Налична информация.....	24
3.2.2.	Нужна информация	24
3.2.3.	Зависимости	25
3.2.4.	Наличие и валидност на данните.....	25
3.2.5.	Конфигурация.....	26
3.2.6.	Събиране на информация	26
3.2.7.	Разпространение	26
3.2.8.	Блок схема на действие.....	27
3.3.	Графичен интерфейс.....	28
3.3.1.	Месечни отчети	29
3.3.2.	Седмични програми & индекс на седмиците	30
3.3.3.	Технически подробности.....	31
3.4.	Прихващане на заявка	33
3.4.1.	За месечни отчети.....	33
3.4.2.	За седмични разписания	33
3.4.3.	Изпълнение на седмични заявки.....	34
3.5.	Филтриране на информация	36
3.5.1.	Основни шаблони и изпълнение.....	37

3.5.2.	Структура на данните и извличане.....	38
3.5.3.	Технически подробности.....	39
3.6.	Експортиране на данни за месец.....	39
3.6.1.	Допълнителна обработка.....	40
3.6.2.	Примерна обработка.....	41
3.6.3.	Попълване на информацията.....	42
3.6.4.	Технически подробности.....	42
3.6.5.	Производителност.....	44
3.6.6.	Точност на данните.....	47
3.7.	Експортиране на програма за седмицата.....	47
3.7.1.	Текстови файлове.....	47
3.7.2.	Файлове на Word.....	48
3.7.3.	Технически подробности.....	48
3.8.	Компиляция към изпълним код.....	48
3.8.1.	Файлове за компиляция.....	49
3.8.2.	Път за изходни файлове.....	50
3.8.3.	Генерация на еднофайлово приложение.....	50
3.8.4.	Прибавяне на файлове.....	50
3.8.5.	Технически подробности.....	50
3.9.	Конфигурация.....	51
3.10.	Logging.....	52
3.11.	Unit testing.....	53
3.12.	Анализ на уязвимостите.....	53
3.12.1.	Уеб уязвимости.....	53
3.12.2.	Изпълним код.....	55
3.12.3.	Свободно разпространение.....	55
3.12.4.	Сканиране за уязвимости.....	56
3.13.	Нереализирани възможности.....	58
3.13.1.	Мобилно приложение.....	58
3.13.2.	Уеб-хостинг.....	58
3.13.3.	Допълнителни файлови типове.....	59
3.13.4.	Хеширане във реално време спрямо GitHub.....	60

3.14.	Наръчник за използване	60
4.	Заключение	62
5.	Библиография	63
6.	Приложение	67
6.1.	Примерни файлове.....	67
6.1.1.	Текстови файл.....	67
6.1.2.	Word файл	68
6.2.	Хеширани файлове	69
6.3.	Уязвимости	70

1. Увод

В този проект ще бъде разгледано реализирането на софтуер за генериране на отчети. Целта на този проект е изготвяне на приложение, с която лесно, бързо и практично да се изготвят месечни отчети за нуждите на преподавателският състав във ВВМУ. Те засягат проведените през предходен период занятия и целят автоматизация на това действие, като се използват предварително налични данни. В краен резултат това ще доведе до спестяване на значителен времеви ресурс при изготвяне на различните видове документи. Цялата информация, която се използва е публично достъпна, не изисква автентификация и не е необходимо да бъде засекретявана. На тази база можем да стигнем до заключението, че е напълно удачно изготвянето на процедурата да бъде автоматизирано. Програмата ще бъде създадена посредством избран език за програмиране след анализ на нейните изисквания. Резултатът от финалният продукт ще бъде оценен спрямо неговите възможности за създаване на отчети, наличието на уязвимости и бързодействие.

Тъй като създаването на софтуер чрез писане на програмен код не кореспондира пряко с дисциплината „киберсигурност“, към дипломната работа ще бъде добавен теоретичен анализ на уязвимостите, който ще служи за допълване на методиката за разработка. Това напълно ще демонстрира придобитите, по време на обучението теоретични и практически знания и умения. Анализът ще включва различни по своя характер уязвимости върху код базиран на езикът **Python**.

2. Литературен анализ и използвани технологии

В изложението е представен анализ на възможните подходи за разработка. Разгледани са използваните технологии и техните функционалности, както и различните модули на проекта. В текста са представени референции към всички използвани източници. Кодът на проекта е достъпен в интернет в поддиректорията `/src` на адрес: <https://github.com/nikolaizhnikolov/NVNA-Schedule-Exporter> (43). В края на изложението е представен анализ на стандартните уязвимости на изпълними програми, както и доколко те засягат даденото приложение.

2.1. Сравнение на възможни имплементации

Създаването на изпълнима програма може да бъде постигнато чрез използването на множество различни инструменти и езици. Спрямо качествата си те биват използвани за различни цели. Някой от тях са специализирани за работа в уеб среда, мобилни или настолни приложения. Други могат да бъдат пригодени към желаната цел, спрямо нуждите на проекта и разработчика. Съответно, специализираните приложения са най-добри в това, за което са създадени. Те биват изключително добре оптимизирани за целта, но не могат да бъдат използвани за нищо друго, което ги ограничава. Спрямо тях генеричните езици имат обширна функционалност и по-малки ограничения, което им позволява да бъдат свободно манипулирани за постигане на крайната цел. Това също дава възможност за по-големя гъвкавост по време на разработка. За сметка на това, в каквато и посока да се използват, те не няма как да изпълняват финалната си цел със същата ефективност колкото специализираните. Нека да разгледаме някои от съществуващите възможности:

2.1.1. Java

Java (1) е програмен език създаден в Sun Microsystems през 1995 година от екип под ръководството на Джеймс Гослинг. Той е един от най-широко използваните езици в компютърната индустрия. Името му идва от кафените зърна отглеждани на остров Джава, около което се изгражда и бранда на езика в последствие (2). Джава е език от високо ниво използващ методологията на обектно-ориентираното програмиране. По този начин всеки различен елемент в програмата се третира като отделен обект със собствени качества и функционалности, дефинирани чрез променливи и методи за тяхната обработка. Съществуват основни атомарни стойности, включително числа, символи и булеви променливи. Заедно със тях се използват сложни обекти, които само по себе си представляват контейнери, в основата на които се комбинират различен набор от съответните базови променливи. Чрез методите за обработка тези променливи биват манипулирани по различен начин, спрямо желаната логика в приложението, за постигане на динамично държане по време на изпълнение. Използването на

обекти е един от най-разпространените начини на работа поради логическото си удобство.

Java няма тясна специализация и може да бъде използван за различни цели, но обикновено бива използван за създаването и поддържането на сървърната част на уеб приложения. Едно от най-големите удобства, което езикът притежава е автоматично почистване на излишни обекти в паметта, познато като *garbage cleaner*. Всеки обект съществува в различна „рамка“, определяща моментното му състояние и необходимост в общото изпълнение на приложението. Той може да преминава в различни функционалности, които влияят на състоянието на обектите. Когато те излязат от своята рамка вече не са нужни за изпълнението на програмата и могат да бъдат премахнати. В програмирането, обаче, това не е просто, като менажирането на памет се явява една от най-сложните задачи. Ако не бъде изпълнено правилно, то може да ограничи изпълнението на една програма и да я забави в пъти, или да я направи напълно неизползваема, като в същото време може да засегне и други приложения, които работят паралелно. Поради тази причина, разработчиците на езика работят дълго време върху функционалността *garbage cleaner* (чистач на боклук), която има за задача да следи всички обекти по време на изпълнението и да наблюдава тяхното състояние. В момента, в който някой от тях излезе от рамката си, той бива добавен към опашка съдържащата всички подобни елементи, където те чакат реда си да бъдат изтрети. Тъй като **Java** е много-нишков език, изпълнението на различни задачи се редува между ядрата на процесора. Когато се освободят ресурси от изпълнението на основната задача, част от тях биват отделени за изпълнение на работата на чистача. В този момент биват затрити всички излишни елементи, а паметта се освобождава. По този начин се заобикаля нуждата програмиста сам да контролира паметта на програмата. От една страна така се намалят значително елементите, които той трябва да менажира, както и се улеснява работата му, тъй като действия от подобно естество изискват задълбочено познание на паметта на компютъра. От друга се ограничава възможността за фин контрол върху действието на програмата което е възможно в други програмни езици. В тях менажирането на паметта е оставено напълно в ръцете на разработчика, което предоставя много повече възможности за ускоряването на изпълнението на програмата, но това увеличава и сложността на нейната реализация.

Производителността на езика зависи от още един фактор, които представлява възможност за мулти-платформено разпространение. Това е направено с цел приложенията, реализирани с него, да притежават способността да бъдат изпълнявани на различни платформи под управлението на различни операционни системи. Това става посредством добавянето на още един елемент, които се грижи за различните имплементации – Джава виртуална машина (**JVM**) (3).

Тя изпълнява по един и същ начин съответният код, без значение от системата, върху която е стартирана. За постигането на тази цел всяка програма, която е написана на **Java**, не се компилира до машинен код, който е директно четим от компютъра, а до Джава байтов (byte) код. Той е специфичен за програми написани на съответният език и представлява списък от инструкции, които виртуалната машина чете по време на изпълнение. Производителността е заменена с удобство и възможност за лесно разпространение.

Поради дългата си история съществува голямо разнообразие от библиотеки, които са построени върху **Java** и доразвиват липсващите или желаните функции на езика. Считайки, че някои от най-големите разработчици и компании в света използват езика за своите проекти се вижда защо той е един от фаворитите.

2.1.2. C++

Доста от споменатото за **Java** важи и за **C++** (4). Методологията на разработка е подобна, основно биваща обектно-ориентирано програмиране. Разликите се намират в компилацията на кода, както и липсата на автоматичен чистач. Компилираният код на **C** бива превърнат в код на Асембли (**Assembly**) (5). Той е език от второ ниво и е един от първите създадени човешко-четими кодове. Неговите инструкции биват директно превърнати в битов, или машинен, код, който се изпълнява пряко от процесора. По този начин производителността на програми написани на Асембли, или езици от по-горни нива инкорпориращи Асембли, е изключително голяма. С напредването на технологиите, обаче, това не винаги е явно, тъй като е възможно програмистите да пишат непроизводителен код. Самото естество на езици от високо ниво от части представлява улеснението, което предлагат в четливостта на кода. По този начин се намалява и бариерата за навлизане в програмирането, което позволява на повече хора да участват. Въпреки това, програмирането на високо ниво означава използването на функционалности, които не са налични на по-ниско. Това води до намаляване на производителността, в случаите, в които тези функционалности не са кодирани достатъчно добре, съгласно общоприетите правила. Тук се намесват вградените оптимизации в компилаторите. Тяхната цел е да компенсират максимално недостатъците в написаният код. Дори при наличието на забавяне поради лоши алгоритми по време на изпълнението, код написан на **C** ще е в пъти по-бърз от такъв, написан на **Java**.

Въпреки това, липсата на чистач на обектите го прави по-труден за контролиране. Езикът не притежава автоматичен мениджър на паметта. Разработчика притежава пълен контрол над съществуването, модификацията и изтриването на обектите в даденото приложение. Това изисква допълнителен набор от знания за начинът, по който компютърът използва паметта по време на изпълнение. Последствията от лошо менажиране на паметта могат да доведат до различни проблеми при изпълнението на програмата. Едно от тях е изчерпване на

работната (RAM) памет на компютъра. Това може доведе до нужда от рестартиране на цялата работна система. Това не е толкова критично в домашни условия, но ако компютърът предоставя услуги на множество потребители паралелно, това ще се отрази на работата на всички участници включени в системата. Освен пълен отказ е възможно и частично забавяне. В зависимост от допусната грешка, последствията могат да бъдат частична загуба на процесорно време и памет или пълен блокаж на цялата система.

C++ също има дълга история, тъй като не винаги е било възможно удобно програмиране посредством автоматичен чистач, а предизвикателството на индивидуалното менажиране от много време привлича опитни програмисти. Езикът притежава голям набор от библиотеки и инструменти и остава ненадминат в разработката на приложения, които се нуждаят от висока производителност, като чертожни програми, 3D визуализации и всички видове софтуер, свързани с математически изчисления. Поради възможността му за пренос на различни платформи продължава приоритетното му използване за програмиране на промишлени контролери и логики. Спрямо повторямата природа на дейността, свързана с изпълнението на този проект, C++ е език, който отговаря на изискванията за реализиране на тази разработка.

2.1.3. Python

Python (6) или Питон е език за програмиране на високо ниво без специализирано предназначение. Създаден е през 90-те години от Гуидо ван Росум и днес е един от най-широко използваните езици. Създателят и екипът му целят да е приятен за използване, което е подчертано от името му, идващо от наименованието на известната британска комедийна труппа Monty Python (7). Отличава се с леснота на използване и модулност. Предназначен е да бъде лесно допълван спрямо нуждите на програмиста. Съдържа разнообразен набор от основни библиотеки (пр. **Datetime** за манипулация на времеви данни), както и голямо количество допълнителни такива, създадени както от професионални програмисти, така и от ентузиаста. Тези модули са разработени с различни цели. Част от тях включват допълване на липсваща функционалност в основните модули за нуждите на професионално разработени проекти, а други представляват разработка за лично използване. По този начин днес **Python** е от най-широко използваните и поддържани езици.

Питон поддържа множество стилове и методологии на програмиране, един от които бива обектно-ориентирано програмиране. Подобно на **Java** има вградени инструменти, които автоматично почистват обекти от паметта след тяхното използване. Особено е, че езикът използва стандарт за кодиране **PEP-8** (8). Той включва структуриране на кода чрез табулатори, които служат за четливост и компилация. С тяхна помощ Питон разбира къде има логическа или структурна промяна. Те заместват стандартните отварящи и затварящи фигурни скоби, с

които се задава „рамката“ на съответният метод или условен блок. Създателят на езика е смятал, че по този начин кодът по-лесно четим.

Спрямо **C++** и **Java**, кодът на Питон се изпълнява по различен начин. В сравнение с тях, Питон може да се счита за интерпретиран, а не компилиран език, въпреки че компилацията е част от процеса. През това време кодът, със стандартно разширение *.py*, преминава през различни етапи. Първоначално бива компилиран до байтов код, подобно на **Java**, който се съхранява с разширение *.pyc*. Този код представлява формат четим от интерпретаторът. По време на изпълнение той превръща кодът в машинен, с допълнителната разлика, че не се изпълнява на процесора, а на графичната карта.

Процесът на интерпретация съдържа допълнителни слоеве в сравнение с другите езици. Така се създава допълнителна сложност, която притежава плюсове и минуси. От една страна интерпретираните приложения са независими от платформата, на която се използват. По този начин може да се разчита на единно поведение при стартиране на различни операционни системи. Друга полза представлява динамичното писане на код по време на разработка. В статичните езици като Си и Джава се задава предварително конкретният тип на всеки обект – число, символ, сложен обект, списък и др. По този начин се гарантира, че няма да има разминаване между използваните типове по време на различните модификационни операции. В динамичните езици като Питон не се задава типът на променливите. Те могат да репрезентират всички видове обекти, както и да придобиват стойност от нов тип по време на манипулация. Валидността на проведените операции се проверява от интерпретатора по време на неговото изпълнение.

От друга страна интерпретираните езици са сравнително по-бавни от статичните (9). Причината Питон да е по-бавен са допълнителните стъпки, които са нужни при компилирането до битов код. Въпреки това съществуват инструменти, които се стараят да компенсират съответните забавяния. Едно от тях е кеширането на информация. То се случва по време на компилация и представлява файл, който може да бъде използван без повторна обработка. По този начин се предотвратява компилацията на непроменени файлове, което ускорява процеса.

2.1.4. Избор

От зададените езици е направен избор спрямо нуждите на приложението. Като критерии са използвани бързодействието, удобството на писане, възможността за лесно инкорпориране на липсващи модули, четливостта и личното предпочитание.

Спрямо производителността, Питон е най-бавният език. Програми написани на него се изпълняват по-бавно от Джава или Си, поради изброените

причини. По време на компиляция на статични езици се задава точният тип на обектите. При изпълнение програмата не губи време в проверки на типовете и директно изчислява резултатът от съответният метод. Спрямо тях, Питон притежава много по-голяма свобода по време на компиляция, но по време на изпълнение губи време в сравнение на типовете данни.

При сравнение на удобството на писане и четливостта, Питон е най-добрият избор от дадените езици. Спрямо другите такива, стандартен код написан на Питон е приблизително 3-5 пъти по-къс от еквивалента му на **Java**, и от 5-10 пъти по-къс от еквивалента му на **C++**. Освен, че е по-къс, той притежава и по-високо ниво на четливост. Поради стандарта на писане и използване на табулации за логическо подреждане на кода се спестяват голям брой редове, които иначе биха били заети от фигурни скоби. Освен това, съответните скоби могат да бъдат разположени по различен начин спрямо навиците на програмиста, което води до възможност за намалено качество на четливостта. Тези фактори позволяват код написан на Питон да се разработва по-бързо, което дава възможност за лесно създаване и модификация на прототипи, което е изключително важно в началните етапи на разработка.

В идеален случай е най-удобно реализирането на програмата чрез комбинация от езици. Различните модули на приложението могат да бъдат написани на Джава или Си, а Питон да бъде използван като „лепило“ между тях. Така се премахва въпроса за производителността, като тя ще бъде напълно зависима от оптимизирани за това езици. От друга страна се увеличава четливостта, а зависимостта на отделните части на приложението една към друга се намалява, тъй като всяка представлява индивидуален логически модул. Така той може да бъде абстрахиран от специфичните цели на проекта и се превръща в преизползваема функционална единица, която Питон може да комбинира с останалите.

Освен тези сравнения, на практика изборът зависи от реални ограничения като цена, опит, лицензи, както и личен избор. Финансови ограничения за този проект не съществуват. Трите описани езика са, или притежават версия, която е напълно безплатна и разработвана с отворен код. Дори в случаите, когато тази функционалност е намалена спрямо пълните им платени версии, тя е достатъчна за целите на този проект.

На фона на изброените разлики между различните езици, за този проект е избрано да се работи с Питон. Въпреки възможните ниски резултати при производителността, удобството му на използване е счтено като по-важен фактор спрямо времето, което ще е нужно за разработка. Допълнително, никой проект не протича напълно по план. Има възможност за поява на неочаквани

проблеми, които изискват креативни решения. Модулната природа на Питон съответни събития да забавят минимално времето за разработка.

2.2. Използвани технологии

Следва списъкът на всички използвани технологии в разработката на този проект, основните такива биват разпределени в главни отдели, а допълнителните свързани с тях в подотдели.

2.2.1. Visual Studio Code

Visual studio code (10) е платформа, която се използва за написване и компилация на самият код. Теоретично погледнато, кодът може да бъде написан в notepad и компилиран отделно чрез команден ред, но това е непрактично. Чрез нея е удобно визуално разпознаване на различните елементи на кода, като по този начин се увеличава четливостта и съответно скоростта на разработка. Подобно на избраният език, платформата притежава модулност и портативност, като самата тя е портативна версия на основната разработка – **Visual Studio**. Превърната е в олекотена модулна платформа, която може да бъде използвана за голям набор от езици спрямо желанието и нуждите на разработчиците. Не изисква инсталация и по този начин не се обвързва със системни настройки, с изключение от независими временни файлове и кешове.

За да може да разбира написаният код спрямо съответните език за програмиране, платформата съдържа голям набор от „допълнения“, които и позволяват да разпознава елементите на езика. Тези допълнения могат да бъдат намерени във вградения магазин, от които бързо и удобно се инсталират желаните модули. Те могат да бъдат както цели пакети с езици, така и допълнителни инструменти като средства за форматиране и спазване на стандартите. В този случай, **Python** също играе роля на такова допълнение. Той представлява един от многото начини да се използва **Visual Studio Code**. Инсталацията на Питон идва със съответните основни модули, компилатор и интерпретатор, както и всички останали нужни файлове за неговото пълноценно използване. Той, от своя страна, съдържа пакетен мениджър, чрез който се разпространяват и инсталират останалите модули, които са създадени или пригодени за работа с Питон. Те могат да бъдат инсталирани и деинсталирани в рамките на минути или секунди с няколко клика, което позволява и бързото рефакториране на програмата за нови цели, въпреки че това рядко се налага.

Освен това, платформата включва инструмент за интелигентно автоматично довършване на кода по време на писането му – **Intellisense**. Чрез сканиране на вече съществуващите библиотеки, той може да предложи всички налични променливи и методи принадлежащи на един обект, заедно с документация за използването им, ако такава съществува. Той работи подобно на използването на табулатор в конзолна среда за дописване на команди, с разликата,

че предлага повече от едно решение. Улеснява работата на програмиста като спестява време от правописни грешки и пълно изписване. Също така предлага нужната информация за използването на един обект в самата платформа, вместо тя да бъде търсена в първоначалните източници. **Intellisense** е част от всяка подобна платформа в днешно време и е неразделна част от набора инструменти на един програмист. Единственият му недостатък е, че зависи от типът на избраните обекти. Тъй като Питон не е типизиран език, не е ясен видът на обектите извън рамките на тяхното деклариране. Извън тях не може да бъде разпознат техният вид, което съответно не позволява да бъде използвана пълната функционалност на инструмента. Въпреки това полезността му не отпада в случаите, в които манипулацията на елементи не излиза извън съответната рамка.

2.2.2. GitHub

В историята на програмирането има множество проблеми с поддържане на различни версии на приложенията, тяхното споделяне по време на разработка и разпространение след това. За разрешаването им са създадени системи за контрол на версиите (11). Чрез тях е възможно съхранението на различни версии на дадената програмата. Освен това те са способни да създават, менажират и разпространяват всяка от избраните версиите. По този начин е налична подробна история на всички промени направени от първоначалната версия на проекта, както и информация относно разработчикът, който ги е направил. Това позволява внимателно логическо разделение между създаването на различните функционални части на приложението. По-големите промени дори биват разделени в „клони“, които са независими от основното „стебло“. В съответните клони може да се работи върху разработката на определена функционалност с пълна независимост от промените, които се добавят към стеблото. Освен лесно разпространение и история, по този начин се разрешава и проблемът с работа на голям брой хора върху един и същи файл без те да си пречат взаимно и да има нужда от ръчно смесване на проектите в случай на конфликт.

Една от най-известните и дълголетни системи за контрол е **git** (12). Тя се появява след спор между общността на **Linux** и тогавашната им система, след който тя отменя безплатното ползване на инструментите си. Това кара общността, и по-специфично създателят на Линукс – Линус Торвалд, да създаде своя собствена система съобразно научените уроци по време на използване на предишната. В резултат създават изключително бърза система, която е способна успешно да поддържа огромни проекти по ефикасен начин.

Върху **git** са построени някои от най-големите корпоративни проекти, както и множество индивидуални такива. От създаването му са добавени доста инструменти за работа със самата система. Един от тях е разширение с по-удобен графичен интерфейс и логическо разделение на функционалностите. Интерфейсът на оригиналната система е сравнително семпъл, но интеракцията с

нея се прави основно от командният ред, което не е удобно за всеки. Чрез **GitHub** (13) е възможна по-удобна интеракция със системата.

Съхранението на файлове и версии става изключително лесно. Самата програма проследява наличието на промени чрез калкулация на хеш стойностите на всеки един от наличните файлове, използвайки криптиращият алгоритъм *SHA1* (14). По този начин тя поддържа така наречена „снимка“ на състоянието на всеки един от тях и може лесно да сравни и удостовери наличието на промени при разминаване с последните стойности. Така се увеличава производителността, като се избягва нуждата от проверка на самият обем на файловете. От списъкът с всички променени файлове се избират тези, които да бъдат опреснени, добавени, или изтрети преди сливането към предишната версия. По този начин е възможно дори възстановяването на предишно изтрети файлове и се отменя нуждата за съхранение на големи архиви с ненужни файлове. След избирането им те биват слети към временни версии, преди сливане към централен сървър.

Въпреки че системата е изключително удобна, има проблеми, които не успява да разреши. Поради индивидуалната природа на работа при използването на системата е възможно двама човека да правят промени върху един файл паралелно. При опит за сливане това предизвиква конфликт при този, който втори се опита да синхронизира направените промени. За да разрешавани този конфликт той трябва да прегледа разликите между двете версии и да остави само нужните промени. За избягването на последици от подобни случаи, както и по други причини, съществуват два различни вида промени – локални и централни. Основата на проекта се намира на избран сървър, от където той може да бъде изтеглен, след което към него се добавя нова функционалност. В случаят на **GitHub** това е централизиран сървър, в който се съхраняват файловете на потребителите. Преди нова промените на версия да бъдат добавени там се осъществява качването им локално. Създава се нова временна версия, която чака да бъде синхронизирана с основният проект на сървъра. По този начин могат да бъдат натрупани и няколко различни версии на съответният проект при по-дълъг период от работа, и да бъдат добавени към централният сървър едновременно. При съответното добавяне се осъществява проверката за конфликт и сливането се забранява докато този конфликт не бъде разрешен. След това работата протича по очакваният начин. Допълнителна разлика между двата типа версии е, че временните промени могат да бъдат загубени в случай на авария, докато централните не.

2.2.3. Модули на Python

2.2.3.1. Pip installer

Pip installer (15) е част от основните модули на **Python** и се инсталира заедно с него по време на инсталацията му като допълнение във **Visual Studio Code**. Той е пакетен мениджър, чрез който биват добавяни всички останали

модули от библиотеките на Питон. Осигурява достъп до тях, както и лесен мениджмънт. Функционалността му зависи от няколко основни команди:

1. **C:>** py -m pip install sampleproject
2. **C:>** py -m pip install --upgrade sampleproject
3. **C:>** py -m pip uninstall sampleproject

Първата представлява основният метод за добавяне на пакети чрез инсталатора. Втората е идентична, но съдържа допълнителен аргумент *--upgrade*, който ръчно проверява за наличие на нови версии. Ако такива съществуват, ги инсталира върху съответната версия. Ако той не бъде използван, се проверява наличието на пакета като цяло. В случай, че се осъществява опит за използване на нова или променена функционалност, първата команда няма да бъде достатъчна. След нейното изпълнение нищо няма да бъде променено, което може да доведе до проблеми по-късно в реализацията, когато се очаква новата функционалност да е налична. Поради тази причина е препоръчително винаги да се използва даденият аргумент при инсталиране на версии на вече съществуващи библиотеки.

Последната команда премахва съществуващ модул, по същият начин по който се деинсталират допълнения от **Visual Studio Code**. Въпреки че командата рядко бива използвана, има случаи, в които тя е полезна. Подобен случай представлява началото на разработката на този проект, където съществуват различни версии на Питон в работната система. След инсталация на съответните допълнителни модули, програмата не се държеше по желания начин, тъй като и липсват функционалности, които теоретично са инсталирани. След анализ на възможните проблеми се стига до заключението, че съществуват различни версии за съответният модул, а интерпретатора избира версия по подразбиране, която не съответства на желаната за изпълнение. След премахване на дублираните модули между версиите със съответната команда този проблем изчезва.

Освен основните команди съществуват различни аргументи при използването на им, какъвто се явява и *--upgrade*. Те позволяват разнообразни функционалности като задаване на конфигурационен файл при инсталация, избиране на специфична версия за инсталиране и други. Съществуват много други команди спрямо нуждите на използващият го, но те не са нужни за разработката на проекта, затова няма да бъдат разглеждани.

2.2.3.2. pyinstaller

Pyinstaller (16), съкратено от python installer, е модул от допълнителните библиотеки на **Python**, който позволява компилирането на скриптове в изпълними програми, като превръща файловете с *.py* разширения в *.exe*. Отличава се с поддръжка на новите версии на Питон, създаването на по-малки на големината файлове, и консистентно изпълнение на програмата, независимо от платформата,

върху която файловете се използват. Освен това, инсталаторът съдържа функционалност, която му позволява да събере цялата кодирана дейност на една програма в единичен файл, без нуждата от допълнителни директории и разнообразни конфигурационни файлове, които да затормозяват потребителя.

По време на използване, инсталаторът събира всички зададени файлове за превръщане в изпълнима програма, след което проверява всичките им зависимости и навлиза рекурсивно в тях. Събира всички нужни части, като по пътя премахва неизползваните части на модулите, за да не заема излишно пространство и да оптимизира максимално резултата. След обхождането превръща всички събрани елементи, заедно с активният интерпретатор, в изпълнима програма.

За изискванията на този проект, **pyinstaller** ще бъде използван за превръщане на **Python** скриптовете в изпълним файл с графичен интерфейс, който е по-интуитивен за използване от командният ред.

2.2.3.3. **tkinter**

Tkinter (17) или tk interface е стандартно допълнение за функционалност на графичен интерфейс към езикът за програмиране Tcl и собственото му разширение за интерфейс – Tk. **Tkinter** е Питонова обвивка на съответната библиотека. За да може тя да съответства на езика възможно най-добре, тя включва и голямо количество самостоятелна логика, вместо да е просто тънка обвивка около съществуващата библиотека.

Модулът служи за създаване на графични интерфейси за програми. Притежавайки много различни елементи, с нея може да бъде създаден както елементарен интерфейс, като калкулатор, така и сложни интерфейси с множество функционалности, различни екрани и логически разделени пътища на изпълнение, както примерно би бил интерфейсът на един инсталатор. Едно от основните преимущества на библиотеката е, че дава възможност за използване на вече съществуващи елементи, наречени widgets. Това са често използвани елементи в дизайна, като надписи, полета за вход на данни, бутони, изображения и други, характерни със своята преизползваемост и удобство на използване. Това помага за лесното и бързо създаване на прототип, или дори на пълна програма за сравнително по-малко време, отколкото би било нужно при написването на всички тези елементи от нулата. Тъй като не би могло да се разчита единствено на тези елементи, библиотеката позволява специфично наместване на съществуващите елементи по множество начини, включително тяхното точно разположение, размер, промяната на такъв при промяна на размера на екрана и други. Всички тези елементи превръщат библиотеката в изключително полезен инструмент при изграждането на проекта.

2.2.3.4. requests

Библиотеката за интернет заявки – **requests** (18) е допълнителен модул за Питон, който може да бъде инсталиран чрез **pip installer**. Чрез нея можем да осъществяваме всички съществуващи стандартни заявки с протокол – *HTTP/HTTPS* (пр. *GET/POST*) по програмируем начин. Използването на заявки в програма включва много повече сложност, която обикновено остава невидима за потребителите в браузър. Съответната библиотека се грижи за всички тези елементи, които в противен случай биха били спънки по време на разработка. Пример за такъв елемент е автоматично потвърждаване на наличният сертификат при връзка с него. По този начин се удостоверява легитимността му, както и се осъществява последващата сигурна връзка, което помага при проблеми със сигурността. Освен това в случай на големи количества информация тя автоматично бива разделена на парчета, които пристига в целият си вид. Последно, библиотеката позволява създаването на заявки без ръчното създаване и попълване на форма с данни, която да бъде изпратена. Достатъчно е да бъде оказан основният път или домейн, към който биват изпратени заявките, заедно с всички избрани аргументи.

Модулът ще бъде използван за прихващане на съществуващата информация в публичният домейн на университета, относно седмичните програми на учителите. Резултатът от прихващането на заявката ще бъде извлечен като *HTML* код, което ще рече, че тя трябва да бъде детайлно обработена за извличане на желаната информация, след което тя да бъде изчистена и структурирана, преди да е възможно тя да бъде пренасочена и съхранена във файл.

2.2.3.5. re

Re (19), съкратено от *regex*, е основна библиотека от модулите на Питон, която служи за извличане и обработка на стандартни изрази от сложни данни. Името на библиотеката идва от съкращението и обединението на „стандартен израз“ (*regular expression*). Тя съществува в много близки, почти еднакви, състояния в почти всички езици за програмиране. Широко използвана е за намиране на информация, валидация, и други операции включващи регулярни изрази. Въпреки че за по-сложни изрази, нивото на четливост рязко пада, производителността на модулът го кара да бъде често избран (пр. валидация на телефон, пощенски код, имейл). След прихващането на заявка, тази библиотека ще бива използвана за изграждането на регулярни изрази, спрямо които да бъде извлечена необходимата информация.

2.2.3.6. configparser

Config parser (20) е основен модул, който позволява по-детайлна манипулация на файлове от стандартният модул **os**, със специализация към конфигурационни файлове. Той ще бъде използван за съхраняване на настройките на програмата при всеки следващ експорт. За улеснение по време на използването,

той ще запаметява и зарежда последно използваните настройки за експорт в програмата. Това ще доведе до последователното създаване на отчети да бъде сведено до отваряне на програмата, смяна на месеца и натискане на бутон „експорт“, което е доста по-удобно от ръчно въвеждане на цялата информация всеки път.

2.2.3.7. *multiple dispatch*

Multiple dispatch е допълнителен модул, който улеснява използването на Питон по принципите на обектно-ориентираното програмиране. Така се позволява съществуването на функции с еднакви имена и различни параметри, нещо което Питон не позволява по подразбиране. По време на компилация Питон, когато интерпретатора стигне до декларацията на една функция, той задава съответните инструкции като нейна стойност, като по този начин тя бива изпълнена всеки път, когато някой обект я използва. Ако се вземе за пример умножение с две числа, то нека съществува функция, която взема аргументи **A** и **B** и връща резултат на база на манипулацията им. За улеснение на програмиста може да бъде добавена втора функция със същата функционалност, която да работи само с един параметър и да връща резултата от умножението му сам по себе си. В такъв случай ще има проблем и очакваното държание няма да бъде валидно. Питон ще се компилира правилно, но единствената функция за умножение ще бъде тази, която е написана последно в кода. При намиране на първата функция, Питон ще и зададе съответната стойност, но при намирането на втората такава със същото име, тя просто ще бъде презаписана. Това не е грешка в кода на Питон, а просто ограничение, тъй като той не е създаден, за да спазва определени методологии, докато съществуването на множество функции с еднакви имена и различни параметри е функционалност на Обектно-Ориентираното Програмиране, така нареченият *overloading* (21). Този модул ще бъде използван с цел запазването на четливостта и късата дължина на кода.

2.2.3.8. *datetime & time*

Datetime е един от основните времеви модули на **Python** (22), работещ с дати. Тъй като заявките на университета работят с номер на седмица, а времевите отчети са месечни, е необходима такава програма, която надеждно да извършва сложни времеви манипулации. В случая – да изчислява кои седмици участват в даден месец, след което да извлича информацията за всяка една от тях. Тук също така се изисква и фина настройка, според която седмиците не бива да излизат от рамките на месеца, в случаите, в които една седмица бива споделена от два такива. Съответната библиотека позволява подобни изчисления да бъдат извършени сравнително лесно, тъй като повечето от тях вече са налични като вградени функции.

Time е допълнителен модул, с които ще бъде измерено времето, което отнема на програмата за различни основни операции. По този начин ще бъде

сравнена производителността на изпълнението им. **Time** работи с хронологично време, но се използва за точни измервания, като може да бъде настроен какво да включва –избрани процеси на програмата, или в допълнение и стандартните такива като автоматичното чистене на използвани елементи.

2.2.3.9. logging

Python съдържа като всеки друг език за програмиране, библиотека за логване на информация. Чрез нея се проследява всяка част от изпълнението на програмата, както в конзолата по време на разработка, така и в реалното и приложение. Основната полза на логъра е, че записва всичко случващо се, като така значително по-лесно може да се разбере причината за един проблем, когато той се случи. Записва се така нареченият *stack trace*, който съдържа подробна информация за проблема и мястото, от което произлиза. Чрез него може да бъде стеснен кръг на възможностите или директно да бъде намерена причината за него. Библиотеката за логване в Питон се нарича *logging* (23) и е част от основните модули.

2.2.3.10. openpyxl

Името **openpyxl** (24) идва от комбинацията *open source python excel*, - свободно разпространена библиотека за работа с екселски файлове използвайки Питон. Чрез нея се осъществява създаване, манипулация и записване на файлове от съответният тип. Манипулация е възможна на множество нива спрямо нуждите на програмиста – файл, таблица, ред, колона или клетка. Използвайки тази библиотека е възможно цялата информация от заявките да бъде запаметена и форматирана в желания тип файл.

2.2.3.11. python-docx

Python-docx (25) е допълнителен модул, който се занимава със създаването и манипулацията на данни за файлове с разширение *.docx*. Тъй като програмата има възможност за допълнителен експорт на седмични данни за лесна справка, този модул ще бъде използван за експорт в **Word**. Самото име на разширението идва от комбинацията “*xml document – doc/x*”, тъй като самият документ, когато бъде прочетен с елементарен четец като *notepad*, е структуриран във вид на *XML* файл. Характерно за файловете от такъв тип е лесната им четливост и удобство при съхранение на елементи. Вътрешно, всеки елемент се третира като списък от обекти с различни параметри, в който се инкорпорират останалите елементи по рекурсивен начин. Така примерно един параграф може да бъде представен като обект от тип параграф, в който като отделни елементи са добавени неговото заглавие и текст. Обекти от типа на таблици или изображения също ще представляват отделни елементи. Атрибутите на всеки един от тях, като големина и цвят на шрифта, размер и позиция на изображението, или брой клетки и колони на таблицата, ще бъдат представени като аргументи в съответният обект.

Всичко това позволява лесната му манипулация, считайки че всяка част от него е превърната в програмируем елемент. По този начин се дава възможност за пълен контрол над функциите на модула, което го превръща в удобен инструмент.

2.2.3.12. Bandit

Бандит (26) е допълнителен модул, чрез който могат да бъдат изследвани наличните уязвимости в едно приложение написано на Питон. В основният код на езикът, както и в поддържаните допълнителни модули рядко могат да бъдат намерени уязвимости. Когато това се случи всички усилия се насочват към елиминирането им. Въпреки това, дори когато в тях липсват проблеми е възможно разработчиците да създадат такива по време на писането на код. Поради това е полезно използването на допълнителен модул, който проверява това по време на разработка. Подобни модули обикновено са свързани с бази данни на всички налични уязвимости на съответният език, както и сканиращи инструменти, които намират по-базови грешки. Използвайки съответният набор от инструменти е много по-лесно да бъдат намерени и поправени проблеми, които иначе биха предизвикали много по-дълго и скъпо оправяне по-късно, когато някой злоупотреби с тях.

2.2.3.13. pip-audit

Освен уязвимости в наличният код, както бе споменато, е напълно възможно да има и такива в модулите, на които е базирано приложението. Използването на готов код е стандартна практика, тъй като не може да се очаква всеки разработчик да разчита само на себе си и напълно да написва кода си от нулата. Това би коствало излишно време и ресурси. Допълнено от факта, че всеки разработчик разбира от различни сфери на програмирането и може да допринесе по различен начин, то подобна дейност е напълно излишна. Съответно се пораждат и въпроси относно сигурността на допълнителните модули. С нарастването на функционалностите, на които се разчита, нараства и броят на различните хора, които са ги писали. Поради тази причина е добре да се използва и инструмент, който проверява съответните модули за допълнителни налични уязвимости. Тъй като **bandit** може да проверява само наличният код, е нужен допълнителен инструмент. За този проект, той ще бъде **pip-audit** (27).

2.2.3.14. Стандарт за писане

В написването на една програма е важно да се спазва единен стандарт. Това подпомага яснотата и четливостта на кода както между различните файлове на една програма, така и между всички различни приложения написани на съответният език от други хора. Голяма част от работата не един програмист не е писане и имплементация на нова функционалност, а поддръжка и тестване на стара (28). Съществуването и спазването на стандарти са част от нещата, които помагат това да се случва по-бързо. В случаят на **Python** ще се спазва вече споменатият стандарт **pep8** (8), а инструмента, който ще улесни това е **autopep8**

(29). Той е способен да бъде приложен върху файл с разширение .ru като автоматично поправи всички грешки свързани със стандарта на писане.

3. Имплементация

Реализацията ще разгледа имплементацията на програмата, разрешените проблеми, както и техническите подробности. Ще бъдат правени референции към местата от кода, където се намират съответните части, като самият код може да бъде намерен в приложението. Възможно е логическото обяснение на програмата да е в обратен ред на написаният код, тъй като в **Python** не е възможно да се използва променлива или функция, преди да бъде създадена. Всички детайли на използването трябва да са обявени и инициализирани преди да бъдат извикани.

3.1. Обобщение

Реализацията на проекта се състои от няколко фази. Първоначалната фаза е разработка за дисциплината „курсов проект“ през втори семестър, след което е надградена за целите и нуждите на дипломната работа. Съответната версия съдържа достатъчно сложност дотолкова, че да създаде елементарен експорт на данните за даден месец накуп, без обработка или разделение.

Тези елементи са преразгледани и довършени във втората версия на проекта, която притежава разширена функционалност в количество и качество. Функционалностите засягащи избор на видът отчет са премахнати, като това е сменено с фиксиран вид на отчета под формат *.xlsx*. Останалите формати са добавени към нова функционалност – създаването на преносима седмична програма за допълнително удобство. Използваните данни по време на прототипа представляват нетипизирани списъци от обекти, които биват достъпени спрямо числови индекси. По този начин е намалена четливостта на програмата. Това е нежелан елемент, който ще окаже влияние върху производителността на разработката, като разрастването на програмата ще доведе до логически затруднения свързани с яснотата на написаният код. Поради тази причина съответните структури биват преразгледани и преправени. Те съдържат значително по-голяма яснота при манипулация на данните (пр: взема се `lecturer` – името на водещият лектор, вместо елемент на трето място от масив – `data[2]`).

Преразгледан е и интерфейсът, който придобива по-изчистен вид с няколко допълнителни възможности. Поради добавянето на нова функционалност за създаване на седмични програми се поражда и нуждата от по-голяма яснота за неговото действие. Тъй като седмичните програми се създават с конкретизиране на двата индекса, в рамките на които е желаният резултат, съответните индекси са изкарани в допълнителен екран, в който могат да бъдат сверени.

Третата фаза на проекта е сравнително малка практически, но логически може да бъде разглеждана като отделна – това е превръщането на проекта в изпълнима програма и тестването за желан резултат, както и написването на кратък наръчник за използването и. По този начин трудът по време на разработка

може да бъде превърнат в реална програма с практическа имплементация и конкретна цел, а резултатът от нейното действие може да бъде обективно оценен.

3.2. Логически проблеми

В следващата секция ще бъдат разгледани някои от логическите проблеми, които са срещнати по време на разработка и не са свързани с конкретните технически подробности.

3.2.1. Налична информация

Нека се разгледа информацията, от която зависи цялата програма. Това е публично достъпна информация, която всеки студент и лектор използва за ориентир по време на един семестър. Тя се намира в разписанието (30) и създава заявки към сървъра на университета с три части – номер, седмица, и вид заявка, с избор от:

- класно отделение
- преподавател
- стая

След въвеждане се генерира заявка, която връща таблица с всички занятия през зададената седмица спрямо избраните параметри. В нея съществуват седем различни елемента за всяко едно занятие. Три от тях винаги се срещат:

- номер на час (от 1-13, всеки разделен на 45мин.)
- времетраене (пр: от 10:00 – 11:30)
- име на занятието

А останалите четири зависят от видът на заявката:

- преподавател
- пореден номер на лекцията
- група/класно отделение
- стая

Спрямо видът се появяват тези елементи, които не и съответстват. Тоест, при избиране на разписание за *класно отделение* е логично то да не излиза във финалната таблица, тъй като ще е еднакво навсякъде. Извън това, изключение прави *пореден номер на лекцията*, който се появява само при заявка от вид *преподавател*.

3.2.2. Нужна информация

Нека бъде разгледано и каква информация е нужна за автоматизацията на месечни отчети и изготвянето на програми. За нуждите на този проект бе осигурен шаблон на стандартен месечен отчет във формат, какъвто подават лекторите на

университета. Всички данни в него са примерни. Информацията, която се попълва в тези отчети е:

- номер на дисциплина
- номера на класни отделения
- брой лекции/упражнения на съответната група за месеца
- номер и разделение по групи
- изпити и други свръхнормативни дейности

Видимо е, че информацията е по-обширна от наличната в заявките. За правилното изпълнение на целите на проекта и постигане на максимален процент автоматизация ще са нужни допълнителни средства. След консултация с ръководител се оказва, че такива са налични, отново, в публичното пространство на университета. Те показват съотношението между номерата и наименованията на дисциплините, както и това между номерата, видът и броят на студентите. Чрез тях ще може да бъде попълнена значително по-голяма част от нужната информация, но е невъзможно да бъде достигната надеждна пълна автоматизация. Също така, съответните документи най-вероятно се препокриват през годините, но ще трябва да бъдат преразглеждани или напълно сменяни при всяка следваща година за гарантиране дейността на програмата. Има нужда всеки файл да отговаря на формата, които присъства в сегашните, така че да се разчита на консистентно поведение.

3.2.3. Зависимости

Програмата е значително зависима от състоянието на сайта на университета, както и интернет връзката. Това значи, че всяка промяна в структурата на заявките, или в състоянието на сървъра оказва критично влияние върху функционалността и. При малки промени винаги е възможно кода да бъде променен, така че да пасва на новите изисквания, но програмата би била напълно обезсмислена при пълна смяна на наличният софтуер на университета. Считайки, че това са рядко и трудно случващи се събития, рискът от подобен проблем е нисък.

3.2.4. Наличие и валидност на данните

Заявки към сайта на университета винаги връщат резултат, било то празен или не. Без значение дали съществуват реални данни за съответното класно отделение, преподавател или зала, файл винаги ще бъде създаден. Такова е поведението и в сайта. Тоест, възможно е да съществува празен отчет или програма. В този случай е невъзможно да бъде направена разлика между липсващи данни за седмица и грешно въведени параметри без съответната проверка. След създаването им е добре такава да бъде извършена от потребителя. Така той може да потвърди наличието на информация във файла, както и да поправи и преформатира всякакви нередности.

Също така е възможно да бъде изпратена невалидна информация в заявката, като резултатът отново ще е грешен файл. За да се разреши този проблем е възможно добавянето на валидация на данните. Това става посредством ограничение на входни полета за дължина и формат, както и наличието на подсказки за очакваният вид данни.

3.2.5. Конфигурация

Програмата ще има нужда от конфигурационен файл за максимално улеснение на работата на потребителя. Самият файл няма да служи за настройване на системни елементи като големина на екрана или качество, а за записване на последно използваната информация. Ако бъде представен един реален случай на използване, то това може да бъде преподавател, който месец за месец, или седмица за седмица използва програмата за създаване на отчети. Единственото нещо, което той ще промени, при всяко следващо използване, е номера на съответният месец или седмица. Водено от този ред на мисли, има логика да бъде създаден конфигурационен файл, който да записва последно зададената стойност на всички полета (номер на преподавател, вид заявка, папка за съхранение и др.), за да спести време и да улесни работата на крайният потребител.

3.2.6. Събиране на информация

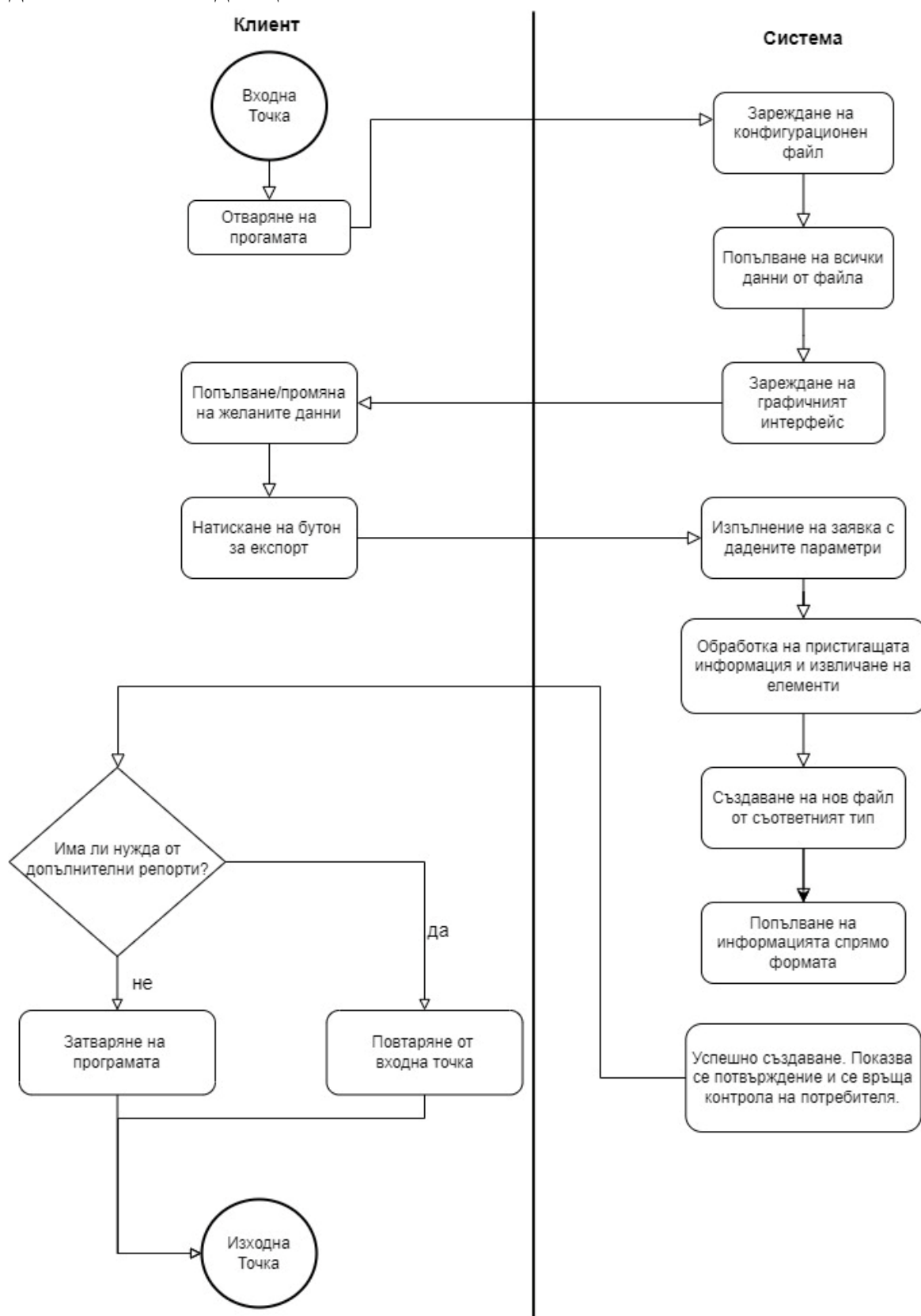
Както вече бе споменато, записването на техническа информация по време на процеса на работа е изключително полезно. Чрез него може да се проследи дали изпълнението на програмата протича правилно, и ако не, да се анализира какъв е проблема. Важно е да се отбележи, че събираната информация е чисто техническа, по никакъв начин не се събира или анализира информация за потребителя, служи само и единствено за проследяване на държанието на програмата в случай на проблем.

3.2.7. Разпространение

След създаването на програмата е нужен и начин за разпространение. Желателно е той да е удобен, бърз и разбираем, без наличието на хардуерни устройства, тъй като това добавя допълнителна сложност и проблеми свързани със сигурността. Програмата ще трябва да се разпространява заедно с наръчник, който оказва очакваният начин на работа, и стъпките, които да се следват за достигането и, както и какво да се прави при наличието на проблем.

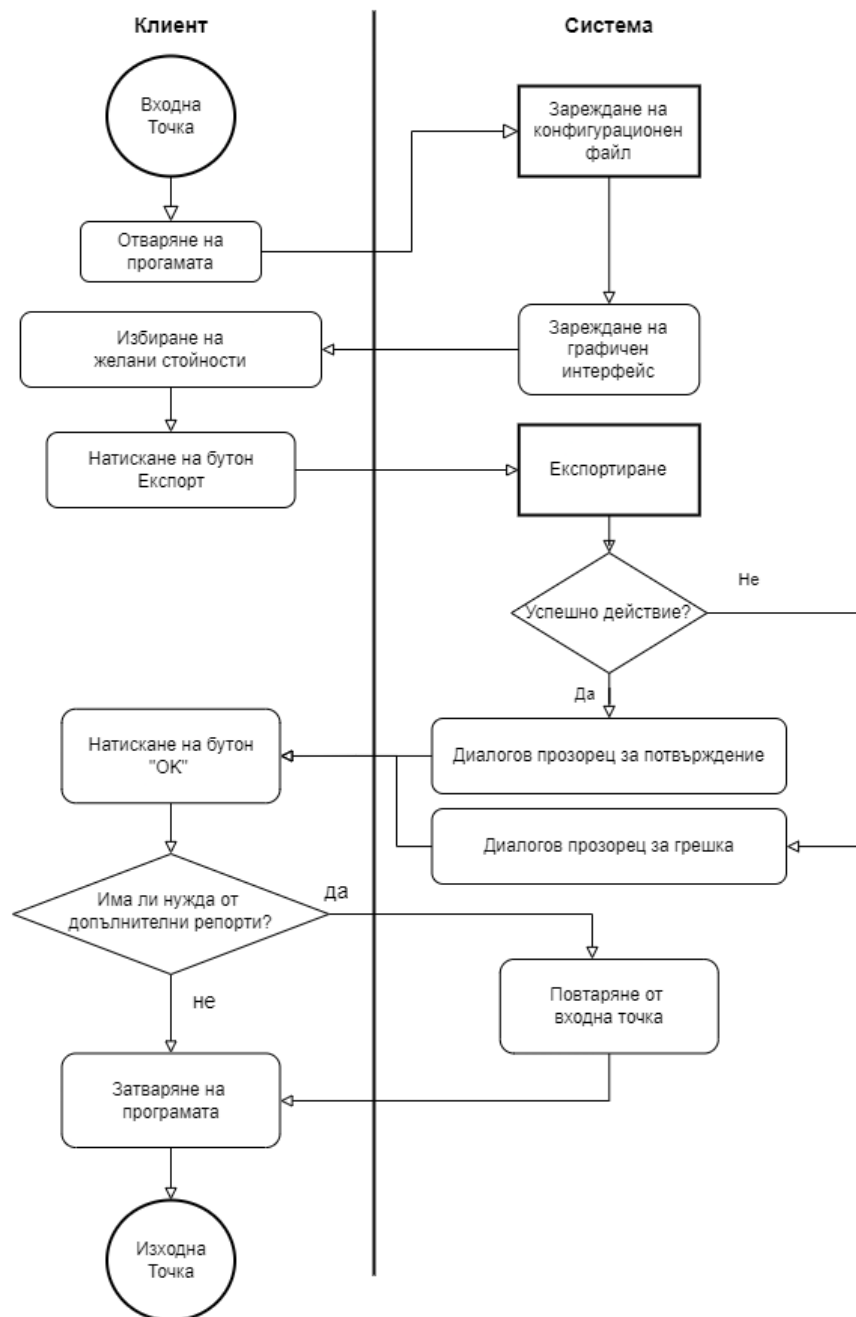
3.2.8. Блок схема на действие

Представена е обща блок схема, чиито елементи ще бъдат индивидуално допълнени в следващите части на изложението.



3.3. Графичен интерфейс

Програмата започва логически от графичният интерфейс в *ExporterInterface.py*. Той се изисква да бъде удобен за използване, с разпознаваеми елементи, които могат да бъдат използвани интуитивно. Пътят на изпълнението на програмата трябва да е възможно най-ясен и кратък. Представена е подробна блокова схема на функционалността (всеки прозорец с удебелена граница бива развит в отделна блокова схема):



Графичният интерфейс се изгражда от правоъгълна матрица от блокове, в които могат да бъдат вграждани графичните елементи. От една страна по този начин е изключително лесно да се скицира и одобри промяна по дизайна. Всеки елемент има строго определено място и може лесно да бъде разместен. Така не изисква повторяемо програмно тестване при всяка дребна промяна, тъй като

основният дизайн е първоначално изграден върху хартия. От друга страна – матричният подход е един от трите вградени начина за подредба на визуални елементи на използваната библиотека **Tkinter** (31). Използването на обособени и тествани инструменти помага за лесно пренасяне на написаната на хартия логика върху дигитална медия. В резултат се придобива близък до пълен процент сходство между прототипа и финалният продукт:

Група:		
Вид заявка:		
Месец:		
Папка:		Избери...
Експортиране		

3.3.1. Месечни отчети

Дизайнът на месечните отчети е продължение на прототипът.

В него са описани нужните полета за създаване на заявки, като някои от нужните стойности биват директни логически преводи, а други транслирани. По този начин, примерно, се изписва номер на класно отделение или преподавател, като финалната му стойност бива директно предадена за по-нататъшно изпълнение. Спрямо тях, обаче, не съществува поле за директно вписване на желана седмица, тъй като отчетите биват месечни. Потребителят избира желаният месец, след което той бива превърнат в списък с номера на седмици при изпълнение. Редовните полета са допълнени с такива, чиято функция е

улеснението на работата на крайният потребител. Добавени са входни полета с избор на име на крайният файл, както и папката, където да бъде създаден.

Съществува разминаване между използваните говорими езици – български и английски. Сървърът, към който биват изпратен заявките, работи с латински стойности. Допълнително, низове с кирилизирани символи създават допълнителна сложност при написването на програмата. В същото време, за потребителят е важно да разпознава лесно елементите на дизайна, което ще бъде по-лесно на кирилица. За разрешаване на този проблем са добавени помощни елементи във формата на конвертори, които се намират в допълнителен файл (*ExporterUtil.py*) и се използват в графичният интерфейс. Чрез конверторите е възможно потребителят да избере спокойно стойности на български, след което програмата да се погрижи за подаването на нужните латински стойности към хода на програмата без допълнително усложняване. Освен старите полета са добавени и нови такива – избор на име на файл и неговото разширение.

3.3.2. Седмични програми & индекс на седмиците

Дизайнът на седмичните програми е почти идентичен на този на отчетите, като месеците са сменени за седмици, а предишно заключени полета представляват списък от стойности.

Тъй като в този случай програмата може да бъде използвана както от преподавателите, така и от студентите, видът заявка може да бъде избран. Допълнително, оригиналното разширение от прототипа *.xlsx* не е подходящо за създаване на седмични програми, така че бива заменен от *.txt/.docx* – съответно елементарен текстови файл, и такъв на **Word**.

Считайки, че е неразумно да се очаква потребителят да знае седмичните индекси, се добавя допълнителен екран за помощ, в който лесно може да се свери принадлежността на седмица към месец и да се идентифицират желаните индекси.

Индексите се генерират при стартиране на програмата за денят, в който е отворена. Съответно те са валидни за годината към която принадлежат.



3.3.3. Технически подробности

Интерфейсът е създаден с помощта на **Tkinter** (17) и хартиени източници за прототипи. Всички елементи са част от библиотеката на модула, като стойността и подредбата им се задава преди стартирането на програмата, след което се извиква `root_frame.mainloop()` (ExporterInterface.py, ред 286). Извикването на тази функция казва на библиотеката да поддържа избраните елементи на екрана. Подобно на филмите, чрез актуализация на всеки кадър програмата изглежда постоянна на екрана, противно на това да се загаси веднага след стартиране.

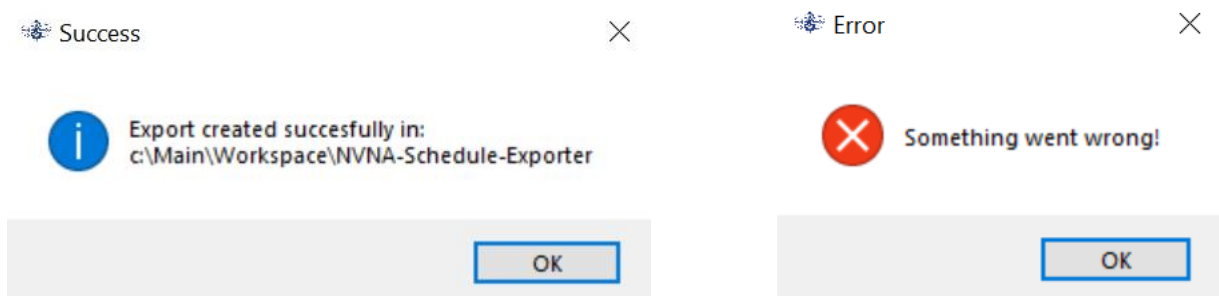
3.3.3.1. Конфигурация

При стартиране на програмата, първоначално се вземат всички съществуващи елементи в конфигурационния файл. Това позволява консистентен графичен вид между различните изпълнения на програмата, както и удобство на потребителя. В идеален случай ще е нужно само промяна на месеца/седмицата при следващото стартиране.

3.3.3.2. Функционалности

Повечето елементи на екрана са входни полета за информация и етикети. Функционалните елементи са бутоните на програмата, които изпълняват съответната команда при натискане. Освен избирането на директория за съхранение, те изпълняват и самите експорти, като са обвързани с функции, извикващи останалите логически части на програмата. След изпълнение на експорт, потребителят получава известие във формата на диалогов прозорец. То информира, заедно с името на създадения файл, дали действието е успешно или не, след което връща контрола на програмата на потребителя. В случай на успех

не се изискват допълнителни действия, но в случай на проблем ще бъде показано общо съобщение за грешка. Подробности за нея няма да бъдат показани на потребителят, тъй като не е препоръчително да се препращат стек трейсове и кодове за грешки директно към допирната точка на потребителите с програмата. Въпреки липсата и в графичният интерфейс, информация за грешката се съхранява със събирането на информация. В случай на грешка потребителят може да изпрати информационният файл към разработчик, който да прегледа и идентифицира грешката.



3.3.3.3. Валидация

Тук се засяга и валидацията на елементите. Не всичко, което потребителят може да впише във входните полета се счита за валидна информация. Въпреки това, за него не съществува полза от използването на такава, освен като опит програмата да бъде счупена. Дори при успех, нейното действие при следващо стартиране няма да бъде нарушено, като в резултат играта с данни по този начин не води до никаква облага, и не може да бъде използвана за злоупотреба. Спрямо този факт, и допълнителната сложност, която валидацията изисква при използването и с избраната библиотека, такава не е добавена. Вместо това се разчита на диалоговият прозорец за грешка, когато потребителят трябва да разпознае, че не е въвел правилните елементи.

3.3.3.4. Време на изпълнение

Един от недостатъците на графичният интерфейс е липсата на прогресен показател. Добавянето на такъв чрез **tkinter** е възможно, но за точно проследяване се изисква редовното му опресняване. Това се оказва трудно, тъй като създаването на месечни отчети минава през няколко фази, някои от които са времеемки. Точно разделение на времето продължение на всеки елемент няма как да бъде направен, а в допълнение за правилното функциониране на подобен показател се изисква обвързването на всяка част от програмата с интерфейса. По този начин всяка промяна свързана с него ще се отрази и на тях, което не е добра практика. Потребителят няма контрол над програмата по време на създаване на отчет или програма, от което следва, че всякакви опити за интеракция с елементи от интерфейса няма да са успешни. Това може да покаже, че програмата все още работи, но може и да обърка използващият, който не знае дали действието

наистина се извършва или не. За да се компенсира за това са добавени диалогови прозорци за финалния резултат от действията.

Към същата категория приспада и пускането на програмата. Тъй като **Python** не е типизиран език не е оптимизиран толкова добре по време на пускане на програмата. Началното зареждане отнема до няколко секунди в зависимост от машината, на която е пуснато приложението. Това време позволява на потребителя да постави под въпрос правилното изпълнение на програмата, в случай, че тя не дава ясни признаци на действие. Поради тази причина е добавен начален екран представляващ логото на университета, които автоматично изчезва при зареждане.

3.4. Прихващане на заявка

Прихващането на заявките става в следващата част на програмата (`ExporterRequestProcess.py`), която приема зададените параметри от графичният интерфейс и формулира серия от заявки спрямо тях.

3.4.1. За месечни отчети

В случаят на месечните отчети, първо се изпълнява вземане името на месеца и се превръща в индекс с помощта на вградените функции на времевата библиотека (22). Той се използва за намиране на индекса на първата му седмица, тъй като следващите функции на библиотеката, които ще бъдат използвани работят с номер, а не с наименование на месец. След това по подобен начин биват намерени и общият брой седмици в съществуващият месец. Генерира се списък с желаните седмични индекси, след което продължава изпълнението.

Ако се вземе за пример месец **Февруари** на **2022**, името ще бъде преобразувано в индекс - **2**, след което ще бъде взет индекса на първата му седмица, които се явява номер **6**. **Февруари** се изчислява да притежава общо **5 седмици**, което значи, че заявки ще бъдат генерирани за следният списък от седмици – **[6, 7, 8, 9, 10]**.

3.4.2. За седмични разписания

В случаят на седмичните разписания, самият списък вече е генериран от потребителя, когато посочи начало и край на разписанието. Хубаво е да се знае, че самият сървър на университета успешно тълкува седмичните индекси без значение дали стойността им превишава **52**. Това е често срещан проблем при работа с време, където последните няколко дни от една година могат да прелеят в седмица **53**, или в нулева **0** от следващата година. Тъй като този проблем вече е разрешен вътрешно от непознат инструмент на сървъра, програмата няма нужда да се справя с него. След избиране на индексите продължава изпълнението.

3.4.3. Изпълнение на седмични заявки

Без значение от използваният вид функционалност, изпълнението на седмични заявки се осъществява по един и същи начин с подаване на списък от седмични индекси. Създава се първоначално празна структура от данни, в която да се съдържа ежедневната информация. Започвайки от индекса на първата зададена седмица, заявки се формулират по следната формула:

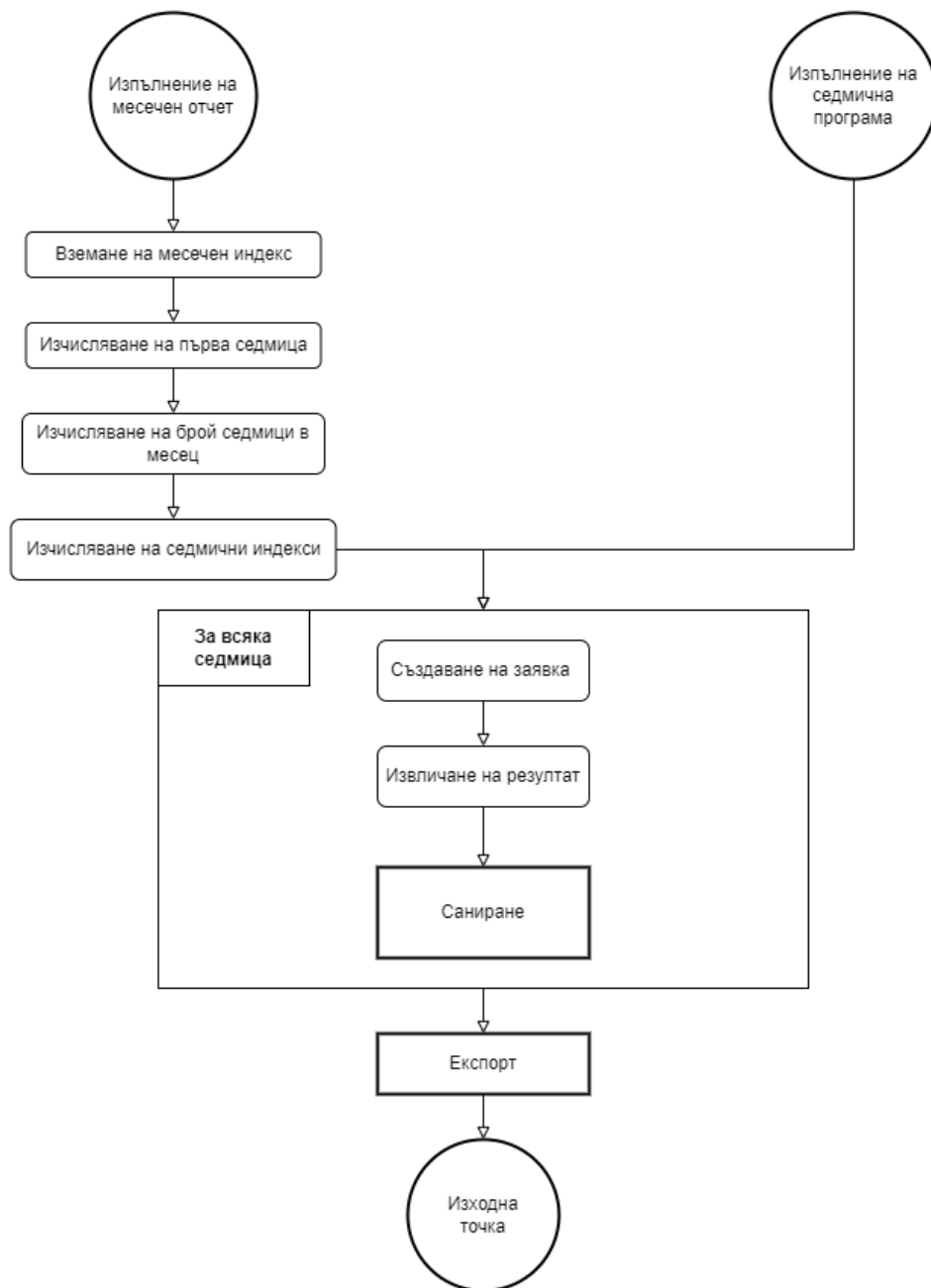
Основен път към сайта + идентификационен номер + вид заявка + номер на седмица.

Ако искаме да направим заявка за **класно отделение**, с номер **626201** за първата седмица на Февруари с индекс **6**, тя ще изглежда така:

<https://nvna.eu/wp/?group=626201&queryType=group&Week=6>

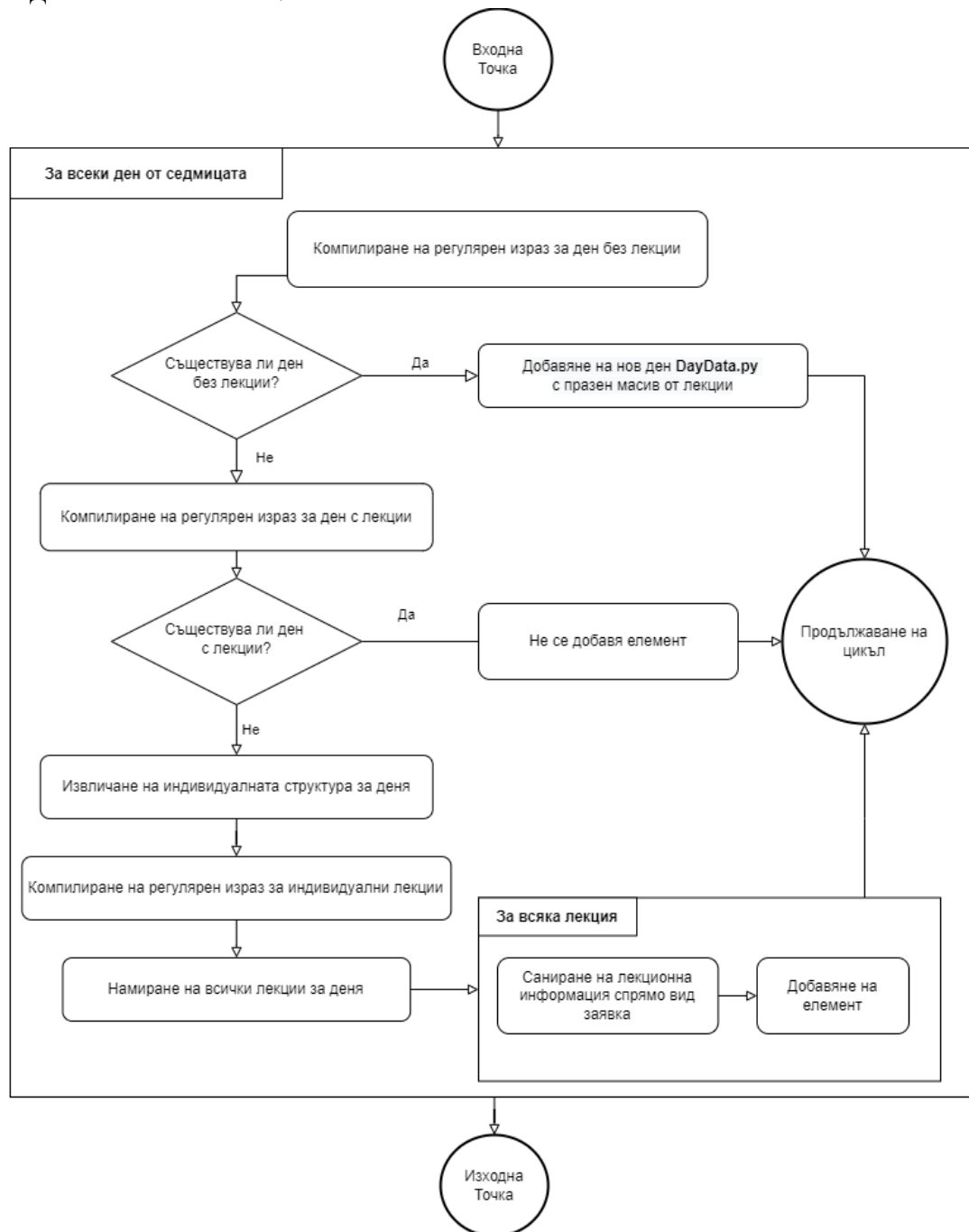
Февруари от 2021 има 5 седмици, съответно ще бъдат създадени 5 различни заявки в подобна форма. Всяка заявка бива създадена и изпълнена с помощта на модул **requests** (18). Отговорът от всяка е *HTML* кодът на цялата страница. Това, което иначе би видял потребителят в сайта, е показано в оригиналният му вид преди браузърът да го превърне в човешки четим. За да бъде полезна, информацията трябва да се преработи, което се случва по време на филтрацията.

Преди продължаване дейността на програмата се извършва проверка за върнатият код от сървъра. Всяка заявка в интернет връща един от множество кодове, определени по стандарт (32). Стандартната очаквана стойност при успешен резултат е код 200. Проверката удостоверява наличността му, след което се продължава нормалното изпълнение. В противен случай се повдига код за грешка и изпълнението се връща до графичният интерфейс, а грешката се съхранява в информационният файл. Прихващането на заявки може да бъде разгледано като блок схема:



3.5. Филтриране на информация

След прихващане на общите данни за седмица, информацията се препраща към санираща функция *sanitize_weekly_data(raw_data, month, query_type)* (*ExporterRequestProcess.py*, ред 69), където се извлича информацията за всеки ден и се добавя към временна седмична структура. След края на всяка седмична заявка структурата бива добавена към пълна месечна или много-седмична, след което автоматично се унищожава от вградените инструменти в **Python** преди създаването на следващата заявка. Филтрацията може да бъде разгледана в следната блок схема:



Извличането става чрез използването на модул **re** (19). Използват се няколко различни регулярни изрази, в зависимост от състоянието на информацията в таблицата, и наличието или липсата на данни. Всеки израз се състои от комбинация изрично обособени или шаблонни части. Обособените части представляват конкретно зададени символи поредици или изрази. При търсене на програма за „понеделник“ ще бъде търсено мястото в таблицата, където съществува символен низ съдържащ точно думата „понеделник“. Шаблонните части съдържат по-голяма свобода на тълкуване, като позволяват разпознаването на различни вариации обособени изрази. Примерно при търсене на дата ще се използва шаблонна комбинация от символи от избран вид (числа, букви, празно място и др.) с определена дължина. Ако първият понеделник на Февруари е във формат **2022-02-07**, тогава регулярният израз, на който отговаря е `[0-9]{4}-[0-9]{2}-[0-9]{2}`. Това значи, че модулът ще търси текст с поредица от четири цифри, тире, две цифри, тире, и отново две цифри. На него ще отговаря датата за съответният ден, както и всяка друга такава. По този начин може да се разчита, че за всеки ден от седмицата ще бъде намерен израз, който съответства на името на деня в комбинация с дата, където единият елемент е конкретно известен, а другият не. По подобен начин са формулирани и останалите изрази, но тъй като биват сравнително по-сложни, а регулярните изрази не са известни с лесната си четливост, те нямат да бъдат разглеждани детайлно. Вместо това ще бъде коментиран начинът им на използване. Съществуват три случая, които трябва да бъдат покрити от изразите:

- ден без налични лекции/упражнения
- дни със налични занятия
- лекции

3.5.1. Основни шаблони и изпълнение

Спрямо наличието на занятия самият ден има различна структура в суровият *HTML* код, което изисква третирането му по различен вид. На база на това са изградени четири различни шаблонни изрази, които да прихващат всички разновидности на информация. Те могат да бъдат намерени във *ExporterUtil.py* (започвайки от ред 133) и са следните:

- *daily_regex_template(day, month)* – структура на заглавен ред на ден от седмицата в таблицата.
- *daily_schedule_regex_template()* – обща структура на ден с занятия
- *no_lecture_regex_template()* – обща структура на ден без занятия
- *lecture_regex_template()* – структура на занятие.

Логическата последователност на обхождането на седмична информация е следната – за всеки ден от седмицата се намира структурата на ден без лекции. Ако е намерена изпълнението продължава към следващият ден, като в масивът с

обекти се попълва единствено името и дата на деня с празен списък от лекции. Ако не е намерен, се търси обща структура на ден с лекции, след което тя се извлича за изолирана обработка, тъй като в противен случай следващата част на изпълнението неправилно засяга и лекции от други дни. След извличането и се прилага лекционна структурата, която връща като резултат всички налични занятия за даденият ден, елементите на които са логически разделени на групи. Всяка лекция се препраща за индивидуално извличане и преработка към `sanitize_weekly_data()`(ред 47).

3.5.2. Структура на данните и извличане

В прототипа на програмата всеки отделен елемент бива добавян като символен низ към масив. Това бива достатъчно за нуждите на дисциплината „Курсов проект“, но надграждането му изисква независима структура от данни. Тъй като различните видове заявки имат различни елементи, не може да се разчита на вземането на елементи от масив по индекс, тъй като не се знае какво представлява всеки един от тях. Съответната структура трябва да е способна да съдържа желаните елементи поименно.

За тази цел са създадени класовете *Day & Lecture*, намиращи се в *DayData.py*. Те са контейнери, които позволяват лесно и удобно съхранение на използваната информация. Всеки обект от тип *Day* съдържа списък от лекции, името и датата на съответният ден, които по-късно служат за вписване в седмичните разписания, както и в логването на информация. Всеки елемент от списъкът е от тип *Lecture*, като самият клас е направен да поддържа всички седем вида информация, които могат да бъдат извлечени. Спрямо видът заявка, различни полета от лекцията се попълват (пр. при лекция от тип *преподавател*, полето за преподавател ще остане празно). Съответните класове позволяват смислено и удобно използване на информацията.

След структурно извличане на съществуваща лекция във формат на *HTML* код, информацията в него бива санирана спрямо вида заявка. Това става посредством използване на групи в регулярните изрази. Това са логически разделени части от израза, обособени в самият него чрез използването на скоби. По този начин структурата на една лекция съдържа общо седем групи. В тях са включени редовните:

- номер на час
- продължителност
- наименование на дисциплина

И условните:

- номер на класно
- преподавател

- пореден номер на занятие
- номер на стая.

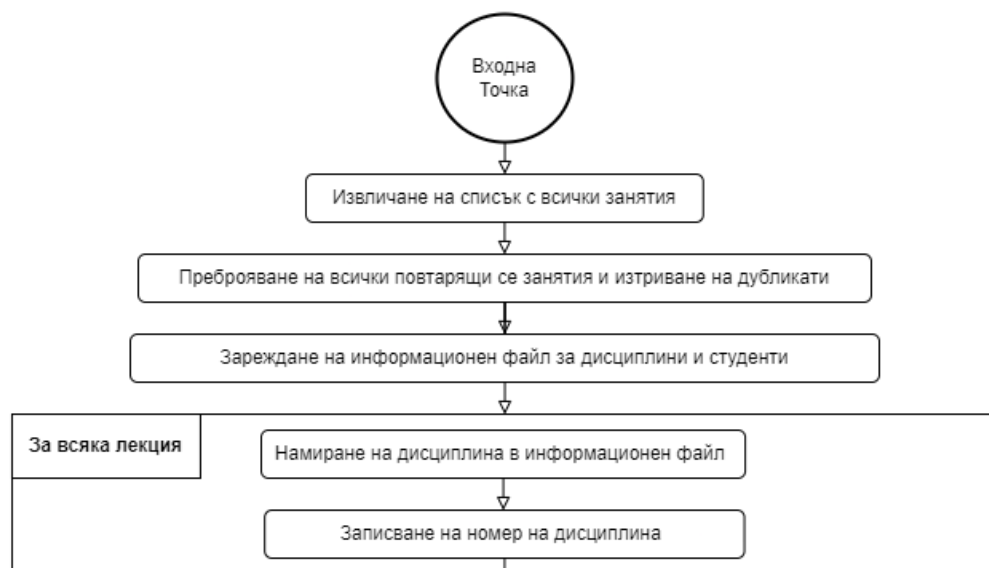
Попълват се задължителните полета, след което се проверява видът заявка преди попълване на условните.

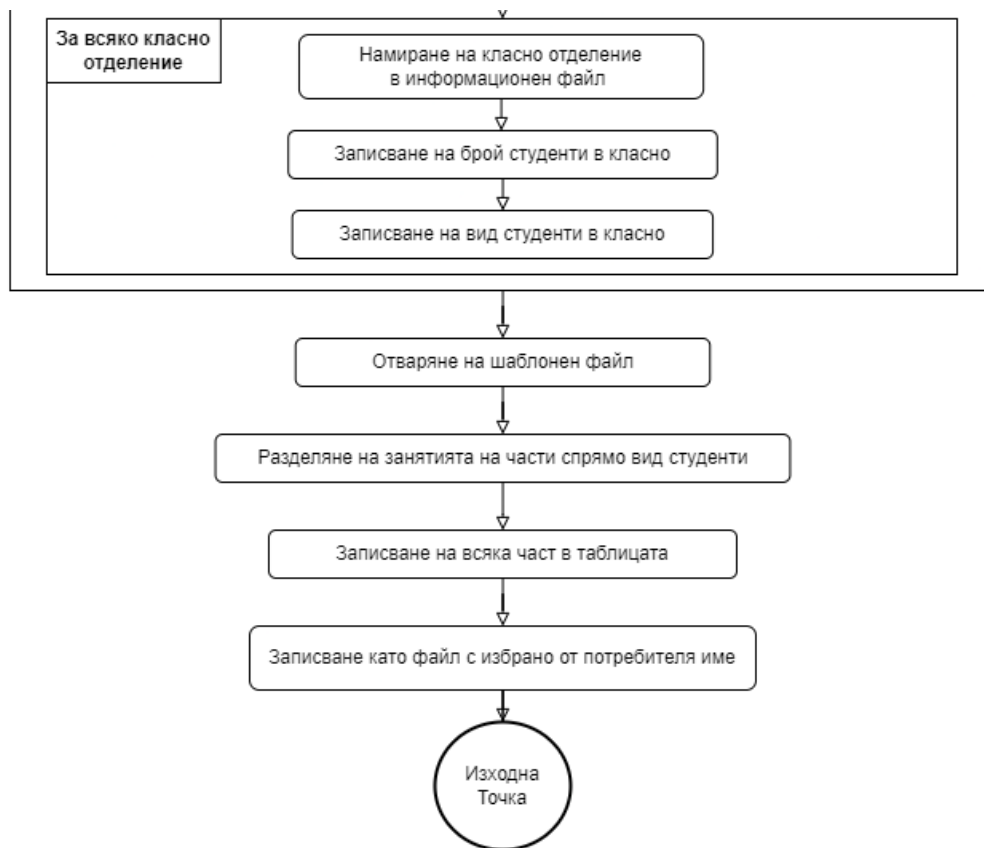
3.5.3. Технически подробности

По време на разработката се стигна до информация, съдържащата неправилно форматираните елементи – всичко, което е на български език бива третирано като битова поредица от символи, тъй като по подразбиране се използва вид кодиране, което не поддържа кирилица. С тази цел са направени две промени: всеки използван файл по време на изпълнението на програмата (лог, експорт и др.) бива отворен с кодиране от тип UTF-8 (33). Инсталиран е и допълнителен модул **regex** (34), който допълва липсващи функции на оригиналният **re**, като добавя флаг за *UNICODE* символи, като по този начин става възможно да се извлича безпроблемно кирилизирани информация от заявките.

3.6. Експортиране на данни за месец

След обхождане на цялата входна информация и извличане на всички занятия, програмата преминава към експортиране на данните към желанят формат, кодът за което се намира в *FileExporter.py*. Месечните отчети включват голямо количество сложност в сравнение със седмичните програми. За по-лесно проследяване е предоставена блок схема на процеса:





3.6.1. Допълнителна обработка

Първоначално се взема филтрираната информация, подадена като списък от *DayData* обекти - *format_lecture_data_for_monthly_report(data)* (ред 48). Тъй като месечният отчет изисква много различни стойности, които не са налични в първоначалната информация, тя трябва да се преработи повторно. Подобно на класовете *Day* и *Lecture* е нужна допълнителен клас за работа със занятия. За това е създаден класът *LectureDataForExport.py*. В него се съхраняват всички възможни стойности, които могат да бъдат извлечени от дадена заявка и допълнителните файлове. Всяка лекция бива преобразувана в такъв елемент.

При създаването на всеки такъв елемент се записват името на дисциплината и списък от номерата на групите. Допълнително от името се извличат ключови думи показващи дали съответното занятие е лекционно, практическо или изпит, и се записват като негов тип в обекта. Съществуват 3 вида занятие индексирани със символи (*LectureDataForExport.py*, ред 4):

- **L** – lecture
- **P** – practice
- **E** - exam

Всички останали предишни елементи са ненужни и биват изтрети. Тъй като **Python** не е типизиран език, същата променлива, която предишно сочи към списъкът с непереработени обекти, сега ще представлява списък с преработени

такива. След изхвърлянето на излишните елементи, много от новите обекти придобиват еднакви стойности, което е удобно за следващата част от процеса – сортиране и преброяване на обектите. От една страна това помага за оптимизацията, като значително намаля броя на обектите, които трябва да бъдат обхождани, от друга страна е нужно, защото един от елементите на отчета е брой на дадените занятия през месеца. След сортирането, всеки дублиран обект бива преброен и добавен като стойност, преди дубликатите да бъдат премахнати.

След това се зарежда файл с данни, служещите за придобиване на допълнителна информация. Отново е от тип *.xlsx* и съдържа два списъка – един с кодовете и имената на дисциплините, и втори с броят и типът на студентите за всяко класно отделение. Обхождат се всички лекции и всички редове в съответната колона, като се търси символният низ репрезентиращ името на дисциплината във файла. Не се търси точно равенство, тъй като името, което дисциплината притежава в оригиналният си вид, идващ от заявките, е комбинация с типът му – лекция, упражнение или изпит. При успешно равенство се взема стойността на колоната до името, която показва идентификационният номер на дисциплината, и се вписва като следващ елемент в обекта.

По подобен начин се обхожда и вторият списък, които съдържа информацията за студентите. Разликата е в увеличеният брой обхождания, тъй като търсене трябва да бъде направено за всяко класно отделение присъствало на занятие. След намиране, в съответният обект се попълват сумата от броя на студентите и техният тип – бакалаври, магистри или курсанти. Те се индексират с цифри по следният принцип (*LectureDataForExport.py*, ред 10):

- 0 – курсанти
- 1 – бакалаври редовно
- 2 – бакалаври задочно
- 3 – магистри

3.6.2. Примерна обработка

Ако имаме обект *DayData*, в който имаме налична следната информация за практическо занятие:

- **Ден:** Понеделник
- **Дата:** 2022-02-21
- **Списък от лекции с една налична:**
 - **Продължителност:** 9:50-11:30
 - **Дисциплина:** Курсов проект от дисциплина от 2ри или 3ти семестър - пз група 1
 - **№ на занятие:** 4
 - **Зала:** виртуална стая
 - **Групи:** група/групи – 126202

След преработка ще имаме обект *LectureDataForExport* с данни за дисциплина, групите и видът на занятието, като видът се удостоверява от наличието на „пз“ в името на дисциплината. След вземане на данните от файловете ще бъде намерено съдържанието на „Курсов проект от дисциплина от 2ри или 3ти семестър“ в наименованието на дисциплината под код „26131“, а до номера на класното отделение ще бъдат взети броя на студенти – „26“, и номер „1“. Финалният обект ще има данни със следните стойности:

- **Дисциплина:** Курсов проект от дисциплина от 2ри или 3ти семестър - пз група 1
- **Номер на дисциплина:** 26131
- **Групи:** [126202]
- **Тип занятие:** Р (Практическо)
- **Брой студенти:** 26
- **Тип студенти:** 1 (Бакалаври редовно)

3.6.3. Попълване на информацията

След извличането на тази информация се продължава към попълването в шаблонният файл. Редът на попълване е строго определен от него. Файлът е разделен на части спрямо видовете студенти (пр. курсанти), затова и попълването на информацията е разделена. В самият файл не се прави разлика между редовни и задочни бакалаври, затова номера показващи „1“ и „2“ в типовете студенти се комбинират. Вземат се три неизменими индекса показващи номера на реда, от който всеки от типовете трябва да започне да се попълва. За да не се развали този ред, попълването става в обратен ред от магистри към курсанти.

Попълват се неизменимите полета – име на дисциплината и списък от групи, след което се проверява вида занятие. Ако е лекция се попълват следващите полета за общ брой студенти и занятия, ако е упражнение се взема половината брой студенти и същият брой занятия.

3.6.4. Технически подробности

3.6.4.1. Извличане на вид занятие

Тъй като не е наличен конкретен индикатор показващ видът занятие, отново се разчита на библиотеката **re** (19), с която се търсят низове съдържащи думите „пз“, „практическо занятие“ или „упражнение“. В случай на правописна грешка или друг термин, проверката няма да сработи, което ще доведе до допълнителна работа при поправката на файла. По същият начин се извличат и групите от съдържащите ги низове, но в техният случай извличането винаги е сигурно, тъй като форматът, в който биват придобити от заявките е един.

3.6.4.2. Сравняване на обекти

Докато сравняването на атомарни променливи като числа и низове е вградено във всеки език, сравняването на обекти по подразбиране се базира не на

стойността им, а на адресите им в паметта. Ако два обекта са създадени един след друг с еднаква стойност, то всеки от тях заема различна част от нея и притежава различен адрес, което води до логическо разминаване в очакваният резултат при сравнение. Това е проблем при сравнението на обекти, който се решава чрез пренаписването на някои от вградените функции в **Python**. Те присъстват във всеки сложен обект и служат за инициализация, сравнение, сортиране и други основни операции. В случаят, се интересуваме от две определени – хеш функцията - `__hash__()` и тази за сравнение - `__eq__()` (основните функции в Питон могат да се разпознаят по наличието на двойни долни черти от всяка страна на името).

Хеширащата функция (`__hash__(self)`, ред 70) се използва за създаване на еднопосочна числена стойност наречена „хеш“, произлизащата от комбинацията на всички изчислени хешове на вече съществуващите елементи в един обект. По този начин се стига до хеш на атомарни стойности, които имат конкретна имплементация и връщат очакван резултат – числата връщат себе си, а символните низове връщат манипулация от числените стойности на отделните символи. Чрез съответната преработка се стига до единствено число, което винаги ще бъде едно и също при еднакви входни данни. Чрез хеширане на елементите на класът *LectureDataForExport.py* се възвръща логическата точност при сравняване на елементи от съответният клас.

Подобно на това се имплементира и функцията за сравнение (`__eq__(self, other)`, ред 73), която приема като параметър втори обект. Основната проверка в случаят е видът на вторият обект, след което се сравняват комбиниранияте стойности на двата класа (`__key__(self)`, ред 67).

3.6.4.3. Зависимости

Важно е да се отбележи, че функционалността на попълването е строго зависима от състоянието на шаблонният файл. Тя е направена специфично с даденият шаблон в предвид, и всяка промяна в неговата структура има голям шанс да засегне успеха на експортирането. Това е така основно поради наличието на видове студенти – курсанти, бакалаври, магистри. Всяко занятие на определената група си има място в шаблона, поради което номерата на съответните места (редове в **Excel**) са строго зададени в кода на програмата. В заден план, това е разпознато като грешен подход на действие, като едно от възможните решения за това е да се добави конфигурационна секция, в която да бъдат попълвани ръчно подобни настройки. По този начин те ще могат лесно да бъдат променени от опитните потребители на програмата, но едновременно с това няма да присъстват в графичният интерфейс, тъй като ще представляват излишни елементи.

3.6.5. Производителност

Скоростта на алгоритмите в програмирането се мери с горната граница на изпълнение, или най-лошият случай (35). При анализ на кода от *FileExporter.py* (ред 48) се вижда, че има налични няколко вложени цикъла. Всеки от тях разчита на различни променливи, така че за изчисленията ще бъдат използвани следните символи:

- m – брой дни
- n – брой занятия
- g – брой групи
- r – брой редове
- c – брой колони

Резултатът може да се разглежда по следната формула:

$$m * n + n + n * r * c * (1 + g)$$

Считайки, че се търси най-лошият случай, търсим опростена версия на формулата, която да зависи от най-важните елементи. Броят на дните никога не може да е по-голям от броят лекции, а обхожданията винаги се случват върху една колона. Тези елементи отпадат и тогава формулата може да бъде опростена до следният вариант, който показва, че най-лошият вариант за скорост зависи от броят лекции, редове и групи:

$$\begin{aligned} m * n + n + n * r * c * (1 + g) &= \\ 1 * n + n + n * r * 1 * g &= \\ n * r * g \end{aligned}$$

Това е най-времеемката част от програмата, потвърдено от следните наблюдения направени при многократно изследване на производителността:

№ тест	1	2	3	4	5	6
Вид дейност	Отчет	Отчет	Отчет	Програма	Програма	Програма
Заявка	Преподавател	Преподавател	Преподавател	Преподавател	Преподавател	Преподавател
Номер	2613	2613	2613	2613	2613	2613
Период	Януари	Февруари	Март	1-5	1-10	1-20

№ тест	1	2	3	4	5	6
Седмици	5	5	5	5	10	20
Занятия	36	61	68	34	99	228
Повторения	10	10	10	10	10	10
T мин (сек.)	1.578	2.062	2.015	0.926	1.966	4.353
T макс (сек.)	1.828	2.593	2.328	1.276	2.597	4.949
T средно (сек.)	1.670	2.237	2.202	1.052	2.323	4.759

В сравнение с месечните отчети, седмичните за същото количество седмици се изпълняват два пъти по-бързо. Спрямо направените изследвания за един стандартен месец със 5 седмици, средната скорост за създаване на месечен отчет е 2,06 секунди, а тази на създаването на програма за същото време е 1,05 секунди. Разбира се, скоростта зависи основно от броя занятия, а не от броя на седмиците. Празна седмица би създавала празен експорт за много по-кратко време, но точното такова не е подлагано на тест. От резултатите на тестовете се наблюдава линейна зависимост между занятия и скорост. В таблицата могат да бъдат видени резултатите за тестове 4-6, където времетраенето се увеличава със сравнителна точност спрямо количеството занятия. Ако времето за минаване през всички други стъпки на програмата не се взема под внимание, може да се стигне до заключението, че средната тежест на едно занятие се явява на **0,024** секунди. Ако се вземат наличните занятия в тестваните месеци – Януари, Февруари и Март на 2022 година – се стига до **55** средни занятия на месец на брой. Със средно отклонение **11%**, изчислено на база направените тестове, може да се заключи, че средното нужно време за създаването на месечен отчет (**T**) е в рамките на:

$$T = 0,024 * 55 = 1,32 \text{ sec}$$

$$\sigma = 1,32 * 0,11 = 0,132 \text{ sec}$$

$$T = [1,32 \text{ sec} \pm \sigma]$$

$$T = [(1,32 - 0,132) \dots (1,32 + 0,132)]$$

$$T = [1,198 \dots 1,452]$$

Разгледани са няколко варианта за максимално редуциране на даденото време за създаване на месечен отчет. Преразглеждането на кода на програмата и промяната на файловата структура или избраната библиотека за обхождане. В началото на разработката библиотеката **Openpyxl** е избрана поради удобството и на създаване на файлове от съответният тип. След използване на вторичните файлове си проличават и недостатъците и. Единственият начин за обхождане е чрез избиране на съответният списък и минаването през него по редове и колони, което гарантира начална сложност от $O(n^2)$. За програма, чието основно действие засяга съответното обхождане многократно при всяко използване означава, че голяма част от работата по време на изпълнение ще бъде прекарано в повторяема дейност.

Възможно решение за редуцирането на това време е изнасянето на всички данни във елементарен текстови файл *.txt*, които да поддържа информацията като двойки ключ и стойност, като по този начин се намали сложността и съответно се ускори изпълнението. От друга страна оригиналните използвани файлове са с разширение *.xls* и са с минимални промени. В случаят се задава въпросът дали е по-добре всеки нов файл да замени с минимални промени предишните, или да бъде изцяло пренаписван в друг формат.

Освен това, промяната на видът файл не елиминира многократното обхождане. Дори неговото времетраене да бъде намалено е желателно да се елиминират логически възможно най-голям брой цикли. За разрешаването на този проблем е възможна преработката на вторичните файлове по време на първоначалното пускане на програмата и преобразуването им в дневник. При стандартните списъци с елементи сложността на добавяне на елемент е константа $O(1)$ като нови елементи се добавят директно в края на опашката, тъй като се знае кой е следващият празен елемент и се достъпва директно по адрес в паметта. Скоростта на търсене, обаче, е линейна и се равнява на броят елементи $O(n)$. В случаят е нужна структура, която да е удобна за еднократно добавяне, тъй като файловете не търпят чести промени и биват заредени веднъж в началото на изпълнение, и многократно обхождане. В програмирането дневниците, или също познати като карти, са такива структури, които позволяват лесен достъп на елементите им (36). Разграничават се с ниска производителност при добавяне на елементи, тъй като всеки нов елемент заема изчислено, а не конкретно място в паметта. Те притежават изключително висока при достъпването им, тъй като и двете операции се базират на стойността на елементите. Тази стойност отново е изчисляването на хеш, според който се задава първоначалната им стойност, и според които след това те бива лесно намерени. В случай на дублирана стойност в дневника, мястото, където се съхранява първоначалният добавен елемент се превръща в стандартен списък, в който биват добавяни всички елементи със същата стойност.

Чрез използването на дневник е изключително удобно да се намали общото време на изпълнение. Въпреки че програмата все още ще бъде зависима от наличието на файловете, тяхното общо влияние над производителността ще намалее драстично. Няма как да бъде спестено обхождането над имената на дисциплините, тъй като извлечените имена не отговарят точно на файловете, поради наличието на видът занятие, но групите могат да бъдат директно достъпвани. Това би намалило времето за обхождане и намиране на всички данни за групите до $O(n)$, където n е броят налични групи във всички лекции. Единствената причина този метод да не е инкорпориран в програмата е липсата на време.

3.6.6. Точност на данните

При автоматизацията на процеса има голямо количество несигурност и зависимости от трети лица – състояние на сървър, наличие и формат на файлове и други. Това води до липсата на начин за достигане на пълна автоматизация на отчетите, поне чрез средствата достъпни на външен разработчик спрямо системата на университета. Тъй като всички достъпни данни имат конкретен източник е възможно той да е един същ във всички случаи, и да е идва от вътрешни системи. В такъв случай, проблемите с точността на приложението могат да бъдат разрешени с осигуряването на достъп до системата. Това от своя страна създава въпроси свързани със сигурността, която трябва да бъде осигурена при съответният достъп, както и публичното разпространение на програмата. Предприемането на такова решение изисква преразглеждането на цялата програма. Като частично решение ще бъдат разгледани възможните грешки, които програмата може да направи при създаване на отчет, и ще бъдат добавени към наръчника за използване.

3.7. Експортиране на програма за седмицата

Създаването на седмични програми работи по различен начин от седмичните отчети, като основната разлика е изборът на начална и крайна точка на списъкът с желани седмици, вместо посочването на месец. След обработка на данните се извиква `export_simple_report(data, folder, file_name, file_type, weekly_indices)` (*FileExporter.py*, ред 35). От него се викат различни функции спрямо типа файл, който потребителят е избрал. Има опции за файлове от тип *.docx* и *.txt*.

3.7.1. Текстови файлове

При избиране на нормален текстови файл, информацията за всеки ден бива записана във файл със стандартно разширение. По този начин желаните данни за седмиците биват съхранени в един от най-широко използваните формати по цял цвят, който също така бива четен и поддържан от изключително много от съществуващите програми. Самият формат на записване се състои от заглавен ред, който показва общият брой и индексите на избраните седмици, след което се записва всяка седмица и всеки ден с разстояние между тях за визуална четливост. Информацията, която се добавя е предварително форматирана в методите *info()* на класовете *Day* и *Lecture*, които връщат наличните си данни във формиран начин. По този начин се премахва необходимостта от допълнително форматиране след обработка. Тъй като самите обекти са създадени за по-удобна манипулация на обектите, има логика самите те да притежават функция, която постига съответният ефект. При извикването и се връща символен низ съдържащ цялата желана информация, след което той просто бива добавен към създадения файл. Примерен файл може да бъде намерен в приложението.

3.7.2. Файлове на Word

При избиране на файл от тип *.docx* информацията се разделя на таблици за всяка седмица. Създава се заглавен ред, в който се съхранява информация за номера на седмицата, както и началната и крайната и дата. След това за всяко занятие се добавя нов ред, в който се записва съответната му информация. След записване на цялата информация, всички клетки в таблицата се обхождат и стилизират, като всеки заглавен ред бива стилизиран по различен начин от информационните. По време на стилизирането се настройват формата, цвета и големината на клетките, техният фон и широчината на колоните. Финалният резултат е примерен и е извлечен от стойностите на съответните атрибути след първоначално ръчно наместване. Примерен файл отново може да бъде намерен в приложението.

3.7.3. Технически подробности

Формата, който се използва при създаване на файлове за **Word** – *.docx* следва стандарта за разработка на файлове във формат *XML*, като използва библиотеките на **Word** за да създаде елементи, които могат да бъдат разчетени от програмата. Това прави използваната библиотека по програмен начин - **python-docx** (25). Тя позволява лесното отваряне и внасяне на елементи. За съжаление не е напълно разработена, като има липсващи функционалности за стилизиране и форматиране. Някои от стандартните възможности не са имплементирани и липсват, докато други са сериозно ограничени. За да компенсира това, тя позволява директното вграждане на *XML* тагове, което позволява използването на всички стандартно налични елементи. Тъй като резултатът от всички нейни функции е *XML* код, вграждането е един вид прескачане на тези функции, или компенсирането за тяхната липса, където ги няма. По този начин е използвано и стилизирането на заглавни редове, като техният цвят е вкаран допълнително с таг `<w:shd {} w:fill="A6A6A6"/>` (*FileExporter.py*, ред 221), където стойността на *w:fill* е шестнайсетично число оказващо цвета на клетката във формат *RGB*.

Отново може да се счита, че настройките на таблицата могат да бъдат добавени в отделна част на конфигурационният файл, където да бъдат избирани настройки като цвят, големина и шрифт на текста, но по време на проекта не е държало висок приоритет.

3.8. Компиляция към изпълним код

След като кодът е написан и тестван, той трябва да превърнат в изпълнима програма. Това става чрез използването на модул **pyinstaller** (16). Той се използва от командният ред, като му се задават скриптовете, които да бъдат превърнати в програма, заедно с всички желани настройки. Ще бъдат разгледани тези, които са използвани в проекта и техният резултат.

Има няколко начина да се задейства създаването на програма през командният ред. Първият е директно вписване на името и желаните настройки, заедно с техните стойности. По този начин беше компилирана програмата по време на етап „курсов проект“, когато имаше ограничена функционалност и пониски цели. След разрастването на програмата се появи нужда от по-удобен начин на изпълнение поради използването на повече настройки и честото тестване. Едно от решенията е копирането на използваната команда в текстови файл и повторното и използване, но самият модул позволява по елегантни решения.

Има два варианта, от които може да бъде избрано решение. Първият е да се използва *.spec* файл, в който са описани същите настройки по формат, който модулет разбира. При стандартното вписване в команден ред, първото нещо, което се случва е да се генерира такъв файл със зададените стойности, така че този вариант може да се разглежда като премахване на горният слой и по-директна манипулация. Този вариант е препоръчан при по-голямо наличие на настройки и нужда от по-фина манипулация, но е и по-труден за използване.

Вторият вариант е подобен на вписването в текстови файл, но е по-удобен за използване от него и елиминира сложността на *.spec* файла. Това е пренасянето на командите в изпълним *.bat* файл, който да бъде задействан от командният ред с елементарно извикване. В него са посочени всички команди, настройки и техните стойности. При желание за промяна различни части от него могат да бъдат добавяни или коментирани, а самият файл се съхранява с програмата в системата за контрол на версиите. След като се знае начинът на създаване, нека се разгледат споменатите настройки:

```
1. pyinstaller src/ExporterInterface.py ^
2. --distpath dist ^
3. --workpath build ^
4. --specpath spec ^
5. --onefile ^
6. --noconsole ^
7. --add-data "../assets;assets" ^
8. --splash "../assets/logo_bg.png" ^
9. --icon "../assets/logo.ico" ^
10.--hidden-import=pyi_splash
```

3.8.1. Файлове за компилация

Основното, което се избира при компилация е файлът, който да бъде превърнат в програма (ред 1). Могат да бъдат избрани няколко такива, но в случаят логическото начало на програмата се бележи от графичният интерфейс. Всеки друг скрипт е свързан директно или индиректно с него, което ще рече, че добавянето им е излишно и би добавило ненужна сложност.

3.8.2. Път за изходни файлове

По време на компилация се създават много временни файлове, както и се подменят последните такива от предишното изпълнение. Тези файлове затормозяват съхранението чрез **GitHub** като запълват папката за съхранение с множество вторични файлове. По този причина са изместени пътищата на папките за временни и изходни файлове (ред 2-4). Оказани са пътища в самостоятелни папки, където могат да бъдат разделени трите основни типа файлове. В папката *build* се намират временните файлове за компилация. В папката *src* се намира конфигурационният файл, който беше споменат по рано. Той се създава спрямо зададените настройки в началото на компилация и не съдържа нищо друго. В последната папка *dist* се намират изходните файлове от компилацията. Там се намира и самата изпълнима програма.

3.8.3. Генерация на еднофайлово приложение

За най-голямо удобство на потребителя и абстракция от допълнителни файлове, програмата се създава като еднофайлово приложение (ред 5). Също така при стартирането на програмата се създава и конзола, която може да се използва за проследяване на случващото се. Тъй като това нарушава визуалният вид на приложението, а същото проследяване може да се види в логовите файлове, се използва допълнителна команда (ред 6), която я премахва.

3.8.4. Прибавяне на файлове

По време на втората фаза са добавени много вторични файлове за използване на програмата. Към това число се включват изображенията и шаблоните. Чрез тях се задава логото на графичният интерфейс, изображението налично в екраните за отчети и програми, както и цялата информация, която се ползва при създаването на отчети. Без нея то няма да може да работи, като не може да се очаква потребителя да ги предоставя. Тези елементи са добавени в последните 4 реда на скрипта, като по този начин те се предоставят на модула за използване по време на компилация.

Те служат за още една цел. Приложението стартира сравнително бавно , когато се изпълнява от файл, спрямо изпълнение от програмна среда. За да не е объркващо за потребителя е добавен начален екран, който стои отворен докато приложението стартира нормално и графичният интерфейс се задейства.

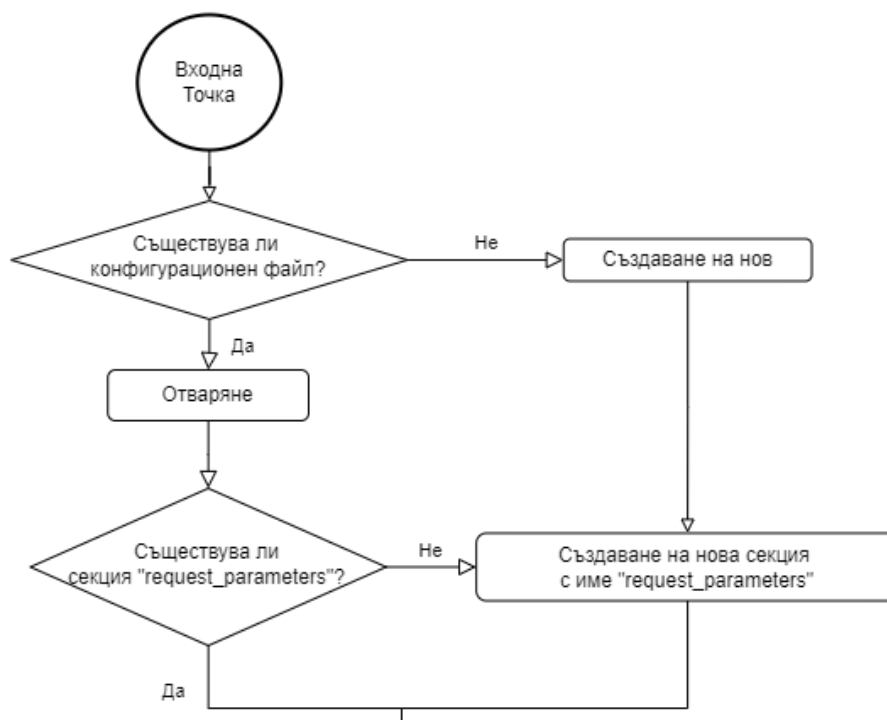
3.8.5. Технически подробности

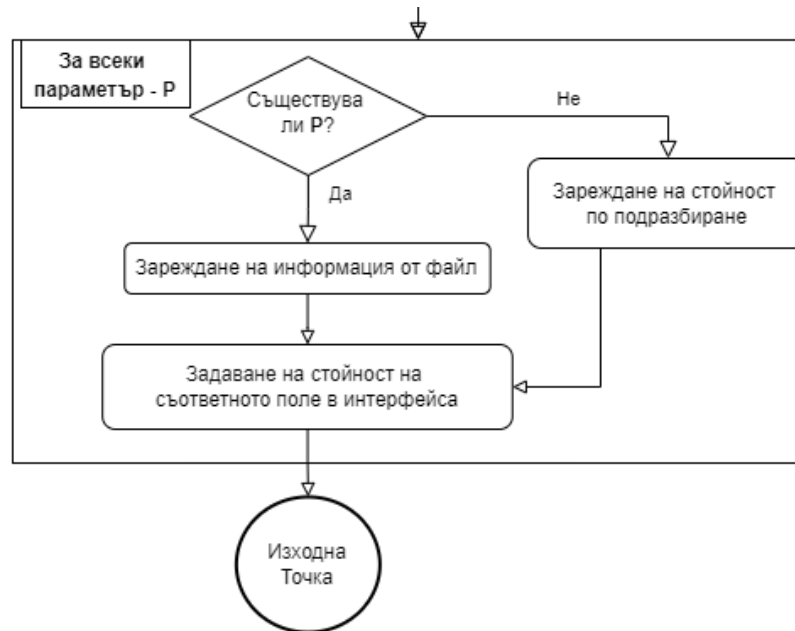
След успешна компилация приложението се генерира, но при стартиране то няма да може да се задейства успешно поради липсата на вторични файлове. Беше очаквано, че добавянето им в скрипта ще ги добави към самото еднофайлово приложение за вътрешно използване. Това не беше истинският резултат, поради което те са добавени в папка със същото наименование, което имат и в

програмната среда. Те вървят ръка за ръка с приложението, и по този начин дори могат да бъдат лесно обменяни, без нужда от нова компилация.

3.9. Конфигурация

Конфигурацията (*ExporterConfig.py*) се грижи за създаване и запазване на използваните параметри за по-лесно бъдещо използване. Тъй като е най-добре програмата да се разпространява с минимално количество файлове, тя не разчита, че потребителят има или знае как да конфигурира съответният файл. При всяко действие се изпълнява функция, която запамятава новите стойности на параметрите, като първо се проверява дали файлът съществува. Ако не, се създава нов такъв с разширение *.cfg*, след което се добавя секция с име *[request_parameters]*, където ще бъдат съхранявани всички желани стойности. След това продължава стандартното изпълнение на програмата. По този начин също се елиминира и нуждата от пренасяне на шаблонен конфигурационен файл с изпълнимата програма, като може да се разчита, че при неговата липса той ще бъде създаден. Обратно на казаното, въпреки че може да се разчита липсата на файл да не е проблемна, наличието на такъв може да се окаже полезно при споделянето на файл между преподаватели, като новият му потребител ще има предварително заредени данни и ще може да го използва с минимални промени. Представена е блок схема на процеса на конфигурация:





3.10. Logging

По време на изпълнение на приложението е полезно събирането на техническа информация за случващото се на всяка стъпка. Това бива изпълнено с помощта на логъри, които са вградени във всяка логически обособена част от скриптовете. Основната логика за конфигурацията му се съдържа в отделен файл (*ExporterLogger.py*), към които реферират съответните части, когато съществува изходна информация за записване. Този файл използва оригиналният модул за логване на **Python** - **logging** (23) и служи за единна точка на комуникация с библиотеката. По този начин той бива отделна логическа единица с очаквано поведение и не се изисква настройка от всеки друг файл, който го реферира. Различните му елементи са конфигурирани спрямо нуждите на приложението. Стандартният изход на информацията е конзолата, като за приложението той е дублиран към текстови файл. Неговото съдържание е индивидуално за всяко пускане на програмата и по този начин се запазва информацията след приключване на изпълнение. Съответно то бива презаписано с всяко следващо пускане, което изисква копирането му при наличие на проблем.

Съществуват различни нива на записване (пр.: INFO), наименувани спрямо сериозността на съответното действие. Честа практика е записването на всяка информация от INFO нагоре, считайки че всяко ниво има и съответстващият му индекс. От възможните нива за този проект са използвани две – INFO, ERROR, и е добавено трето – PERF, тъй като съществуващите не отразяват достатъчно добре логическото му използване. Чрез първите две нива се записва стандартна информация и такава при грешка, а с новодобавеното – информация за производителността на приложението. Използван е стандарт за кодиране от вид

UTF-8 за поддръжка на изходящата информация, тъй като тя включва кирилизирани низове. Форматът на изходящите данни е следният:

НИВО:ИМЕ_НА_ЛОГЪР:ИНФОРМАЦИЯ

Информацията, която се записва зависи от логическите граници на приложението. Логват се данни при стартирането на програмата, при изпълнение на експорт, в случай на грешка и нарушаване на стандартният ход на изпълнение. Записва се и самата информация от заявките, филтрирана и подредена за всеки ден. По този начин може лесно да се проследи точно къде е настъпил проблемът.

3.11. Unit testing

За установяване на грешки в кода е честа практика да се тества. Тестването в изолация помага за ранното намиране на такива преди те да стигнат до крайният потребител. Въпреки това, за този проект не остава достатъчно време за детайлен тест. Приложението е тествано по стандартен начин с проба и грешка по време на работа, а за информация в случай на грешка се разчита на логър и конзола. Единственият клас, който е детайлно тестван е при създаване на обекти от тип *LectureDataForExport*, като се тества типът занятие след създаване. По време на разработката съществуват проблеми с правилното удостоверяване на типовете, но тестването с нов отчет всеки път не е достатъчно ясно и забавя процеса. Също така минава през няколко стъпки на обработка, които влияят на резултата. Поради тази причина проблемът е изолиран и разрешен в тестови клас. Това е показателно за нуждата от наличие на тестове още при създаването на всеки скрипт и ще бъде взето под по-строго внимание при следващата разработка.

3.12. Анализ на уязвимостите

Проекта представлява създаването на изпълнима програма спрямо зададени входни данни. Спрямо тях, тя взаимодейства с брой елементи, всеки от които може да бъде уязвим по един или друг начин. Програмата има достъп до интернет и изпраща заявки, съдържа изпълним код във формат *.exe*, и се разпространява публично в интернет пространството, като не се изисква автентификация при изтеглянето и. Ако се приеме, че оригиналният и източник е сигурен, нека бъдат разгледани възможните атаки към всеки един от тези елементи, и решенията, които могат да бъдат предприети.

3.12.1. Уеб уязвимости

Програмата има достъп до интернет, което автоматично увеличава броят възможни уязвимости. Заявките се изпълняват по конкретно определен начин, което увеличава и предсказуемостта им. Възможни са атаки от тип Man in the middle (37) или Denial of service (38) .

Атака от тип „човек по средата“ (Man in the middle) представлява прихващане на трафик между потребител и сървър, след което се осъществява

неправомерен достъп до споделената информация. По този начин тя може да бъде подслушвана или подменена, без никоя от двете страни да знае за това. След прихващане на информацията, от нея могат да бъдат извлечени всички данни, които биват препратени, включително имена, пароли, имейли и банкови сметки. Природата на прихваната информация прави подобни атаки изключително опасни. Такава атака може да бъде изпълнена чрез IP Spoofing, където нападателят променя данните в изпратените пакети, като по този начин потребителят бива препратен директно към нападателя, когато се опитва да достъпи съответният сървър. След прихващане е нужно информацията, която бива споделена, да се декриптира, когато тя не се изпраща в суров вид. Декриптирането на информация може да стане по няколко подхода, спрямо начина, по който е криптирана.

Атаки от тип „отказ на услугата“ (Denial of service) представлява спиране на машина или мрежа, чрез наводняване на сървърите с продължителни заявки, или изпращане на необработваема информация, правейки съответната цел недостъпна за крайните потребители. По време на атаката трудно биват откраднати лична данни, но производителността може да бъде сведена до нула за неопределен период от време, поради което цели на тази атака често биват високопрофилни организации като банки, правителствени инструменти и други. Лишаването на съответните групи от действието на инструментите им дори за кратко време може да предизвика големи парични и времеви загуби.

Подобни атаки са възможни при всяко приложение, което има нещо общо с интернет. В случаят, комуникативният елемент на програмата е създаването и изпращането на заявки. Самата тя не разчита на сървър, тъй като работи локално на компютър, затова не е директно уязвима, но е зависима от сървъра на университета, който е уязвим към подобен вид атаки. Ако той бъде нападнат, това ще окаже директно влияние и върху работата на програмата.

Нека бъде разгледана отново информацията, която се достъпва от приложението. Тя е публично достъпна и не изисква автентификация. Може да бъде директно достъпена както от самата страница за седмичните програми, така и от приложението. В противен случай, то щеше да притежава допълнителна логическа част, занимаваща се със осигуряването на сигурен достъп до данните. Успешното прихващане и обработване на данните показва, че такъв не е нужен. Спрямо природата на данните и описаните видове атаки, се счита, че съществува единствено минимален риск от нападение от тип „човек по средата“. В случаят, поради предсказуемостта на заявките е възможно да бъдат прихванати, но въпреки това информацията протичаща между двете страни ще бъде напълно криптирана. Допълнително, подслушването на информацията няма да донесе нищо полезно на нападателят, тъй като единствените данни, които биват пратени

са вече публично достъпни. Не се изпращат лични или конфиденциални данни, което свежда последствията от успешна атака до минимални.

3.12.2. Изпълним код

Компилирането на кодът в изпълнима програма води до няколко точки на уязвимост. По време на работа се използва модулът **pyinstaller** за компилацията, което прави програмата зависима от неговата функционалност. Това води до подлагане под въпрос на количеството доверие, което може да се гласува на чужд модул при работа с него. Първият начин, по който това може да бъде измерено е чрез анализ на кода на приложението, екипът му и времето, от което съществува. Въпреки че няма как целият код да бъде прегледан, лесно може да се види, че кодът е изцяло публичен и видим на **GitHub** страницата на модула (39). Екипът му се състои от няколко основни персонажа, но бива допълнен и от голямо количество самостоятелни разработчици, като на страницата са оказани всичките 384 сътрудници към момента на писане. Модулът започва съществуването си в публичното пространство малко преди 2006 година. За предишна история не е ясно, но приблизително тогава е направено първо качване на сървърите на **GitHub**. Публичната видимост на кода, дългата история и голямото количество поддръжници вдъхват доверие в използването на модулът.

Въпреки това е напълно възможно да има чисто технически проблеми. Не най-малко поради наличието на толкова много сътрудници. След сверяване с два сайта за проверка на уязвимостите (40) (41) е идентифицирана една такава. Уязвимост **CVE-2019-16784** е от тип ескалация на привилегии (42) и е валидна за програми стартирани на операционна система Windows, които са компилирани с опция *–onefile* (с каквато е компилиран и този проект). Чрез нея е възможен непривилегирован достъп, когато програмата е пусната от системен акаунт като сървиз или с планирано стартиране, с допълнението, че уязвимостта е валидна единствено при вторично стартиране. Казано по друг начин, ако компилираният код се пуска със стартирането на компютъра той няма да бъде уязвим, но след рестартирането му има такава опасност. Уязвимостта е идентифицирана за версии на модула от 3.6. надолу. Използваната версия за този проект е последната налична към момента на разработка – 4.9., което води до заключението, че тя не представлява риск за изпълнението на програмата.

3.12.3. Свободно разпространение

Изпълнимото приложение и оригиналният код на програмата, заедно с цялата документация са налични на сървъра на **GitHub**, където те се поддържат по време на разработката. Всеки е свободен да изтегли, използва и модифицира програмата по желан начин, което поражда проблеми при споделяне на файловете извън оригиналните способности. Възможно е директно или индиректно да бъдат споделени модифицирани файлове съдържащи вируси и рутките. Това може да се случи поради наличието на оригиналните скриптове в сървъра - възможна е

както директна промяна и компилиране на нов изпълним файл, така и декомпиляция на съществуващият такъв със същата цел. Новите програми ще се държат по непредсказуем начин спрямо модификациите направени от нападателя и могат да нанесат вреди на използващите ги.

Възможността за подобно събитие е малка, но рискът, който то носи е сравнително по-голям. Поради тази причина е препоръчително единственият източник на файловете да бъде официалният му контейнер на сървъра (43). По този начин се намаляват максимално възможните точки на провал, като реалистично биват сведени до една. Въпреки това е добавена допълнителна защита към програмата. Тя представлява скрипт, чиято функция е хеширане на стойностите на изпълнимата програмата, вторичните файлове и всички оригинални .ру скриптове. Използван е алгоритъм **SHA-256** (14). Сметнато е, че притежава висока сложност и ниска предсказуемост, което не позволява на случайни файлове да притежават едни и същи хеш стойности. Изчислените суми за всеки файл се съхраняват в текстови файл и могат лесно да бъдат пресметнати с инструменти каквито се намират на всяка операционна система. Примерни сравнения между резултат от скрипта и от вградени функции на **Windows** могат да бъдат намерени в приложението. За целите е използвана команда:

certutil -hashfile [file location] SHA256

Където *certutil* е името на конкретната имплементация, *-hashfile* е атрибут показващ файлът, който да бъде хеширан, след което е зададена и стойността му. Накрая е зададен алгоритъмът, по който да бъде извършена операцията. На други операционни системи е възможно същото нещо.

Linux (44): *\$ sha256sum [file]*

Mac (45): *\$ shasum -a 256 [file]*

Със посочените команди е възможно да бъде проверен интегритета на програмата преди да бъде изтеглен. Чрез отваряне на текстовият файл, съдържащ всички генерирани суми, може да бъде сравнена тяхната стойност без наличието на каквото и да е опасност. Единственото нещо, което е нужно за подобно държане е редовното наличие на последни версии на стойностите. При всяка промяна с използваните файлове трябва да бъдат генерирани нови стойности. Възможно решение е вграждането на скрипта в редовното изпълнение на програмата, но към момента това не е изпълнено за програмата. В случай на съмнение за модификация се разчита на потребителя да направи ръчна проверка.

3.12.4. Сканиране за уязвимости

Освен оказаните проблеми е възможно да са налични уязвимости, които не са очаквани. Те могат да са налични в основният код на Питон, в допълнителните модули, но най-вече в кодът, който е написан по време на проекта. Той е най-нов

и не е детайлно тестван, тъй като не е предназначен за широка употреба. Допълнено с факта, че е разработван само от един човек, което трудно позволява втора гледна точка да повлияе на процеса, той е най-възможно да съдържа проблеми. Поради тази причина са използвани средства за сканиране на съществуващият код за възможни уязвимости.

Написаният код е сканиран с помощта на допълнителният модул **bandit** (26). Той позволява проверка на цялата налична база код за възможни уязвимости. Има възможност за използване на множество аргументи спрямо нуждите на разработката. За този проект се изискват следните:

- Покритие на уязвимости от всички нива на сериозност
- Изключване на наличните тестове с цел избягване на фалшиви уязвимости
- Наименование на сканираните файлове

Използвана е следната команда, в която са оказани нужните настройки:

```
bandit './src' -r --severity-level 'all' -x './src/LectureDataForExportTest.py'
```

Резултатите от сканирането не показват никакви налични уязвимости в съществуващият код и могат да бъдат намерени в приложението.

Освен наличният код е възможно да има проблеми с някой от използваните допълнителни модули. Те не могат да бъдат проверявани ръчно, тъй като това трябва да се случва при всяка промяна. Така тази проверка става прекалено времеемка, като е високо възможно по този начин да бъде допусната човешка грешка по време на проверката. За това се използва допълнителен инструмент – модул **pip-audit**, който автоматично проверява всички модули, на които разчита написаният код. За да може да бъде използван първо се използва команда, която описва всички налични модули и техните версии. Въпреки че може да бъде описан списък само с използваните такива за проекта, това изисква създаването на отделна виртуална среда – нещо, което не е било налично по време на началото на разработката. Тъй като е излишно готов проект да бъде местен в нова среда, списъкът с генерирани модули включва всички стандартни инсталирани модули, които идват с Питон, освен допълнителните такива. Въпреки липсата на изолирана среда, списъкът ще бъде използван за проверка на съществуващите уязвимости, като от резултата ще бъдат филтрирани тези, които са част от неизползвани модули. Използвана е следната команда:

```
pip-audit -r './requirements.txt'
```

Спрямо наличните модули, уязвимости не са намерени. В заключение – програмата не притежава познати уязвимости, които биха попречили на стандартното и изпълнение или биха допринесли за успешна атака от страна на

нападател. Възможни са такива, в случай на целенасочена атака и при условие, че се разчита на човешка грешка (пр: изтегляне на програмата от трето лице).

3.13. Нереализирани възможности

В следващите параграфи ще бъдат обсъдени възможности, за които не е имало достатъчно време за имплементация по време на разработка. Други такива вече са обяснени в секциите „технически подробности“ под съответното заглавие.

3.13.1. Мобилно приложение

Създаването на месечни отчети е действие, стандартно изпълнимо на настолен компютър, където големината на екрана позволява удобно сверяване и поправяне на файлове. Също така, въпреки че съществуват приложения за отваряне на засегнатите типове файлове за телефон, те биват рядко използвани и в сравнение не са толкова удобни. От друга страна, след автоматизация на месечните отчети не е нужно същото количество работа при сверяването му. Финалните отчети могат да бъдат приготвени и изпратени с минимални усилия, а преносимата природа на мобилните устройство позволява изпълняването на това действие навсякъде, където има достъп до интернет. Също така, използването на настолен компютър или лаптоп за целите на стандартно сверяване на седмична програма от преподавател или студент е излишно. Това често пъти бива правено от мобилно устройство чрез директно достъпване на програмата в сайта. Поради тази причина е удобно съществуването на втора версия на програмата, компилирана под друг формат, който може да бъде изпълнен на телефон. След кратък обзор на ресурсите са намерени следните инструменти, с които това може да бъде изпълнено: **BeeWare** (46), модул **python-for-android** (47) и модул **Buildozer** (48). Наличните средства за това съществуват и позволяват изпълнението на дадената задача. Чрез тяхното използване може вече съществуващият код да бъде пренесен върху мобилни устройство с минимални усложнения.

3.13.2. Уеб-хостинг

Подобно на създаването на версия за телефон, мобилност на приложението може да бъде осигурено чрез създаването на веб версия на приложението. Тя ще има отделен графичен интерфейс, логически подобен на първоначалният, който ще бъде хостван в публичното пространство и ще позволява публичен достъп до възможностите на приложението на всеки, който има нужда от него. По този начин се премахва нуждата от теглене на нови версии при разпространението на приложението и се осъществява единна точка на достъп. По този начин всеки с достъп до интернет ще може да използва желаната функционалност.

Този метод има своите плюсове и минуси. Позитивните му страни вече са изброени, но отрицателните такива включват: допълнително време за разработка, цена на поддръжка, и нови проблеми свързани със сигурността. Тъй като трябва

да бъде написан графичен интерфейс изцяло наново, това изисква допълнително време, дори и той логически да бъде идентичен с оригиналният. Също така това включва наличието на допълнителни технологии, с които да бъде написана както статичната, така и динамичната част на интерфейса, както и средства, с които да бъде поддържан сайта в публичното пространство. Пример за това е наемането на сървър и работата с него. Този проблем би могъл да бъде разрешен чрез хостване на приложението на сървърите на университета при придобиване на позволение. Допълнително, уеб-базирано приложение е постоянно налично в интернет пространството, а не само по време на заявка. По този начин то бива допълнително уязвимо при атаки. Считайки, че то отново ще създава файлове за справки и програми, хакването на сайта би било доста по-сериозно отколкото прихващането на заявки в стандартното приложение. В такъв случай нападателят би могъл да изпрати каквато иска информация на потребителите, с която да придобие достъп на техните системи или да злоупотреби с информацията им по друг начин.

3.13.3. Допълнителни файлови типове

Към приложението могат да бъдат добавени допълнителни файлови типове. Към момента съществуващите такива изпълняват достатъчно добре зададената задача, но не са удобни за повторно използване или поддържане в дълъг план. Въпреки че форматът им позволява използването върху мобилни устройства, те не целят подобно използване. В случай на създаване на мобилно приложение ще са нужни по-удобни формати на използване и визуализация, като това засяга най-много седмичните програми. Пример за по-удобен формат може да бъде PDF, за който съществува и съответната библиотека – **pyPdf** (49).

Разработката на допълнителни файлове в други формати включва нова функционалност, всяка от която ще бъде изпълнена по различен начин, и ще има своя собствена сложност. Въпреки това, информацията, която идва винаги ще бъде една и съща. С цел – четливост на кода, удобна поддръжка и премахване на излишната сложност, ще бъде полезно да бъде инкорпорирана единна структура, от която всяка логически отделна част да извлича информацията. Към момента това се осъществява чрез масив от вътрешни класове, специфично създадени за приложението. Ако то се разрасне, или съществува на няколко медии едновременно – компютър, телефон и уеб-хостинг, тогава съответният стандарт ще бъде удобен за използване преди препращане за обработка. Такъв може да бъде *XML* стандартът, който е широко използван, и за когото съществуват множество библиотеки, като освен самите данни, в него могат да бъдат съхранени и нужните атрибути са пресъздаване на желаният формат – шрифт, цвят, дебелина на границите и други. След това остава само прочитането на информацията и записването и по желаният начин.

3.13.4. Хеширане във реално време спрямо GitHub

С цел допълнителна сигурност, съществува хеширане на избрани файлове от приложението. Това хеширане работи по строго определен начин и изисква интеракция от потребителя, за да бъде напълно функционално. Това може да бъде счтено като проблем за разрешаване. Неговото решение се крие в автоматизацията на проверките на хеш кодове в две стъпки. Като първа стъпка се закача съществуващият скрипт към стартирането на програмата. Преди зареждането на конфигурационните файлове и всичко останало се прави проверка на съществуващите хеш стойности с нови такива, генерирани на момента от скрипта. Ако са намерени разлики програмата хвърля грешка и спира, а потребителят се предупреждава за възможна модификация.

Проблемът с този подход е, че цялата логика бива направена излишна при модификация. Проверката зависи изцяло от съществуващите вътрешни файлове на приложението, което значи, че един опитен нападател просто ще изтрие съществуващата проверка, заедно с желаните си модификации. След това ще добави фалшив екран показващ липсата на модификация, и потребителят ще вярва, че всичко е наред. Решение на това може да се намери чрез закачането на проверката към наличните хеш стойности на сървъра. Тъй като програмата вече зависи от наличието на интернет връзка, това няма да бъде допълнителна зависимост. Чрез сравняване на генерирани на момента стойности, с вече съществуващите на сървъра може да бъде елиминиран елемента на уязвимост. Единственият проблем, който остава, е че скрипта и стойностите трябва внимателно да бъдат опреснявани при всяка промяна, след което работата на програмата зависи от скоростта, с която файловете ще бъдат синхронизирани на сървъра. Освен времето, причина тази функционалност да не е инкорпорирана по време на разработка са затрудненията и забавянето, които се появяват от съответната синхронизация.

3.14. Наръчник за използване

Следните указания посочват препоръчаният начин на използване на приложението в реална среда и грешки, които могат да настъпят по време на изпълнение.

Приложението, се намира на публично достъпен домейн - github.com/nikolaizhnikolov/NVNA-Schedule-Exporter. При достъпване на страницата се изтегля цялото съдържание на папка *dist*, където е самата изпълнима програма, заедно с изображения и вторични файлове в папката *dist/assets*. Приложението се стартира от *NvnaScheduleExporter.exe*, и се изчаква неговото зареждане. В случай на липсващи изображения програмата няма да може да стартира правилно.

При съмнение за модификация или получаването на програмата от място, което не е официалният му източник е препоръчително сверяването на съответните хеш стойности с тези на сървъра. Ако съвпадат приложението е сигурно за използване. Проверка може да бъде направена със следните команди, спрямо операционната система:

Windows: `certutil -hashfile [file] SHA256`

Linux (44): `$ sha256sum [file]`

Mac (45): `$ shasum -a 256 [file]`

След стартиране се въвеждат желаните данни и се натиска бутон експорт. Резултатът ще бъде изписан на екрана. При успех резултатният файл може да бъде намерен в избраната папка. След първото използване ще бъдат създадени два файла с разширение *.cfg* и *.txt*. В първият файл се съхраняват конфигурационните настройки за следващото пускане, а в текстовият файл информацията от последното изпълнение. При местене на програмата се премества и конфигурационният файл. В противен случай нов такъв ще бъде създаден в новата директория, а последно запазените данни ще бъдат затрити. Логовите файлове не са нужни при успешни действия, в случай на грешка информацията за нея може да се намери вътре.

След създаване на месечен отчет е препоръчителна проверката на верността на информация, както и ръчното допълване на липсващите елементи. Програмата не поддържа:

- Всички видове свръхнормативна заетост
- Студенти на индивидуално обучение
- Студенти с повишен риск
- Автоматично вписване на име на преподавател
- Практически занятия на групи с втори номер
- Извънредни занятия
- Изплатена заетост

Създаването на седмични програми работи по подобен начин, но информацията в него е точна. Индекси на всяка седмица от годината могат да бъдат намерени в третият таб. В случай на неочаквани или неидентифицирани проблеми е желателно да бъде осъществена връзка с разработчик.

4. Заключение

В тази дипломна работа е представен проект за разработка на програма, която служи за автоматично създаване на месечни отчети и седмични програми. Целта на проекта е да спести време на учителите във Военноморско училище „Никола Йонков Вапцаров“, като създаде инструмент, който да автоматизира подаването на съответните отчети. Разгледани са различни подходи и технологии, които могат да бъдат използвани за създаването на такава програма, и е направен избор спрямо техните възможности и личното предпочитание на разработчика. Направен е анализ на всички построени модули за програмата и са обосновани приетите подходи на разработка. Добавен е и подробен анализ на уязвимостите на приложението, с цел информираност на крайният потребител и затвърждаване на придобитите знания по време на обучение. Неговото заключение е, че уязвимости с висока опасност и последствия не съществуват в приложението, а малките такива зависят от дейностите на потребителя. В резултат на проекта е построено приложение, което успява частично да автоматизира създаването на месечни отчети, с допълнителната функционалност да създава удобни седмични програми. Поради липса на достъп до вътрешната мрежа и наличните ресурси на университета се стига до заключението, че пълна и точна автоматизация е невъзможна с публично достъпни ресурси. Въпреки това, с наличните такива е доказано, че дори и частично това би било полезно. След изследване на производителността на приложението е показано, че времето на създаване на такъв отчет е сведено до броени секунди. Ако се приеме, че един отчет отнема средно между 15-30 минути, когато тези минути се умножат по броят преподаватели спестеното време всеки месец е значително.

5. Библиография

43. **Николов, Николай.** Nvna Schedule Exporter. *GitHub*. [Онлайн] 24 February 2022 г. [Цитирано: 24 February 2022 г.] <https://github.com/nikolaizhnikolov/NVNA-Schedule-Exporter>.
1. **Oracle.** What is java? *Java*. [Онлайн] [Цитирано: 10 January 2022 г.] https://www.java.com/en/download/help/whatis_java.html.
2. **Murphy, Kieron.** So why did they decide to call it Java? *Info World*. [Онлайн] 1996 г. [Цитирано: 10 January 2022 г.] <https://www.infoworld.com/article/2077265/so-why-did-they-decide-to-call-it-java-.html>.
3. **Hartman, James.** What is the JVM? *guru99*. [Онлайн] 12 February 2022 г. [Цитирано: 20 February 2022 г.] <https://www.guru99.com/java-virtual-machine-jvm.html>.
4. **The C++ Resources Network.** Welcome to cplusplus.com. *cplusplus*. [Онлайн] [Цитирано: 22 February 2022 г.]
5. **Pedamkar, Priya.** What is assembly language. *educba*. [Онлайн] 2020 г. [Цитирано: 10 February 2022 г.] <https://www.educba.com/what-is-assembly-language/>.
6. **Python Software Foundation.** Python. *python*. [Онлайн] [Цитирано: 14 December 2021 г.] <https://www.python.org/>.
7. **Python (Monty) Pictures Limited.** Monthly Python's Official Website. *Monty Python*. [Онлайн] Python (Monty) Pictures Limited. [Цитирано: 13 December 2021 г.] <http://www.montypython.com/>.
8. **Rossum, Guido van, Warsaw, Barry и Coghlan, Nick.** Style Guide for Python Code. *Python*. [Онлайн] 05 July 2001 г. [Цитирано: 13 December 2021 г.] <https://www.python.org/dev/peps/pep-0008/>.
9. **Brihadiswaran, Gunavaran.** A Performance Comparison Between C, Java, and Python. *Medium*. [Онлайн] July 2020 г. [Цитирано: 10 February 2022 г.] <https://medium.com/swlh/a-performance-comparison-between-c-java-and-python-df3890545f6d>.
10. **Microsoft.** Visual Studio Code. *Visual Studio Code*. [Онлайн] [Цитирано: 13 December 2021 г.] code.visualstudio.com.
11. **Git.** 1.1 Getting Started - About Version Control. *git*. [Онлайн] [Цитирано: 10 February 2022 г.] <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>.

12. **Software Freedom Foundation.** *git. git.* [Онлайн] <https://git-scm.com/>.
13. **GitHub, Inc.** GitHub. *GitHub.* [Онлайн] [Цитирано: 13 December 2021 r.] <https://github.com>.
14. **Eastlake, Donald и Hansen, Tony.** *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF).* неизв. : IETF, 2011.
15. **The pip developers.** pip documentation v21.3.1. *pip.* [Онлайн] [Цитирано: 13 December 2021 r.] <https://pip.pypa.io/en/stable/>.
16. **PyInstaller Development Team.** PyInstaller. *PyInstaller.* [Онлайн] PyInstaller Development Team, 11 August 2021 r. [Цитирано: 15 December 2021 r.] <https://www.pyinstaller.org/>.
17. **Python Software Foundation.** tkinter — Python interface to Tcl/Tk¶. *Python Documentation.* [Онлайн] [Цитирано: 15 December 2021 r.] <https://docs.python.org/3/library/tkinter.html>.
18. **Reitz, Kenneth.** Requests: HTTP for Humans™. *Requests.* [Онлайн] [Цитирано: 14 December 2021 r.] <https://docs.python-requests.org/en/latest/>.
19. **Python Software Foundation.** re — Regular expression operations. *Python Documentation.* [Онлайн] [Цитирано: 14 December 2021 r.] <https://docs.python.org/3/library/re.html>.
20. —. configparser — Configuration file parser. *Python Documentation.* [Онлайн] [Цитирано: 15 December 2021 r.] <https://docs.python.org/3/library/configparser.html>.
21. **W3Schools.** Java Method Overloading. *W3Schools.* [Онлайн] [Цитирано: 16 February 2022 r.] https://www.w3schools.com/java/java_methods_overloading.asp.
22. **Python Software Foundation.** datetime — Basic date and time types¶. *Python Documentation.* [Онлайн] [Цитирано: 15 December 2021 r.] <https://docs.python.org/3/library/datetime.html>.
23. **The Python Software Foundation.** logging - Logging Facility For Python. *logging library documentation.* [Онлайн] [Цитирано: 15 January 2022 r.] <https://docs.python.org/3/library/logging.html>.
24. **Gazoni, Eric и Clark, Charlie.** openpyxl - A Python library to read/write Excel 2010 xlsx/xlsm files. *openpyxl.* [Онлайн] [Цитирано: 15 December 2021 r.] <https://openpyxl.readthedocs.io/en/stable/>.
25. **Canny, Steve.** python-docx. *python-docx 0.8.11 documentation.* [Онлайн] 2013 r. [Цитирано: 15 January 2022 r.] <https://python-docx.readthedocs.io/en/latest/>.


26. **PyCQA**. bandit 1.7.3. *pypi*. [Онлайн] 2022 г. [Цитирано: 10 February 2022 г.]
27. **Woodruff, William**. pip-audit 2.0.0. *pypi*. [Онлайн] 2022 г. [Цитирано: 10 February 2022 г.] <https://pypi.org/project/pip-audit/>.
28. **Grams, Chris**. How Much Time Do Developers Spend Actually Writing Code? *The New Stack*. [Онлайн] 15 October 2019 г. [Цитирано: 22 February 2022 г.] <https://thenewstack.io/how-much-time-do-developers-spend-actually-writing-code/>.
29. **Hattori, Hideo**. autopep8 1.6.0. *pypi*. [Онлайн] 4 October 2021 г. [Цитирано: 14 December 2021 г.] <https://pypi.org/project/autopep8/>.
30. **ВВМУ "Никола Йонков Вѡпцаров"**. Разписание. *Nikola Vaptsarov Naval eAcademy*. [Онлайн] [Цитирано: 14 December 2021 г.] <https://nvna.eu/wp/>.
31. **Roseman, Mark**. The Grid Geometry Manager. *TkDocs Tutorial*. [Онлайн] [Цитирано: 12 December 2021 г.] <https://tkdocs.com/tutorial/grid.html>.
32. **MDN Contributors**. HTTPS status response codes. *MDN Web Docs*. [Онлайн] 18 February 2022 г. [Цитирано: 22 February 2022 г.] <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>.
33. **Yergeau, F**. UTF-8, a transformation format of ISO 10646. *Internet Engineering Task Force*. [Онлайн] January 1998 г. [Цитирано: 15 February 2022 г.] <https://www.ietf.org/rfc/rfc2279.txt>.
34. **Barnett, Matthew**. regex 2022.1.18. [Онлайн] January 2022 г. [Цитирано: 15 December 2021 г.] <https://pypi.org/project/regex/>.
35. **Huang, Shen**. What is Big O Notation Explained: Space and Time Complexity. *freeCodeCamp*. [Онлайн] 16 January 2020 г. [Цитирано: 10 February 2022 г.] <https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/>.
36. **Python Software Foundation**. 5. Data Structures. *Python Docs*. [Онлайн] [Цитирано: 10 February 2022 г.] <https://docs.python.org/3/tutorial/datastructures.html>.
37. **Imperva**. Man in the Middle (MITM) attack. *Imperva*. [Онлайн] [Цитирано: 10 February 2022 г.] <https://www.imperva.com/learn/application-security/man-in-the-middle-attack-mitm/>.
38. **paloalto networks**. What is a Denial of Service Attack (DOS)? *paloalto*. [Онлайн] [Цитирано: 10 February 2022 г.] <https://www.paloaltonetworks.com/cyberpedia/what-is-a-denial-of-service-attack-dos>.

39. **PyInstaller Development Team.** pyinstaller GitHub page. *GitHub*. [Онлайн] [Цитирано: 24 February 2022 r.] <https://github.com/pyinstaller/pyinstaller/pulse>.
40. **Vulmon.** pyinstaller vulnerabilities. *Vulmon*. [Онлайн] [Цитирано: 10 February 2022 r.] <https://vulmon.com/vulnerabilitydetails?qid=CVE-2019-16784&scoretype=cvssv3>.
41. **Snyk.** pyinstaller vulnerabilities. *Snyk*. [Онлайн] [Цитирано: 10 February 2022 r.] <https://snyk.io/vuln/pip:pyinstaller>.
42. **Cynet.** Understanding Privilege Escalation and 5 Common Attack Techniques. <https://www.cynet.com/network-attacks/privilege-escalation/>. [Онлайн] [Цитирано: 20 February 2022 r.] <https://www.cynet.com/network-attacks/privilege-escalation/>.
44. **baeldung.** Generating an SHA-256 Hash From the Command Line. *baeldung*. [Онлайн] [Цитирано: 10 February 2022 r.] <https://www.baeldung.com/linux/sha-256-from-command-line>.
45. **Classroom.** How to verify checksum on a Mac. *Classroom*. [Онлайн] [Цитирано: 10 February 2022 r.] <https://dyclassroom.com/howto-mac/how-to-verify-checksum-on-a-mac-md5-sha1-sha256-etc>.
46. **Keith-Magee, Russell.** BeeWare. *BeeWare*. [Онлайн] 2021 г. [Цитирано: 10 February 2022 r.] <https://beeware.org/>.
47. **Taylor, Alexander.** python-for-android. *python-for-android docs*. [Онлайн] 2015 г. [Цитирано: 10 February 2022 r.] <https://python-for-android.readthedocs.io/en/latest/>.
48. **Kivy Developers.** Welcome to Buildozer's documentation! *Buildozer docs*. [Онлайн] 2014 г. [Цитирано: 10 February 2022 r.] <https://buildozer.readthedocs.io/en/latest/index.html>.
49. **Fenniak, Mathieu.** pyPdf 1.13. *pyPdf*. [Онлайн] 5 December 2010 г. [Цитирано: 10 February 2022 r.] <https://pypi.org/project/pyPdf/>.
50. **The Python Software Foundation.** Comparing Python to Other Languages. *python*. [Онлайн] [Цитирано: 10 February 2022 r.] <https://www.python.org/doc/essays/comparisons/>.

6. Приложение

6.1. Примерни файлове

6.1.1. Текстови файл

 2613 Седмици 5-9.txt - Notepad

File Edit Format View Help

Програма за седмици: 5 - 9

Седмица: 5

2022-01-31 - 2022-02-06

Понеделник:

9:50-11:30 Курсов проект от дисциплина от 2ри или 3ти семестър - пз група 1
група/групи - 126202 виртуална стая

11:45-13:20 Компютърни мрежи: Connecting Networks - лекция в поток
група/групи - 10391, 11091, 12291, 12691 виртуална стая

13:50-15:25 Курсов проект от дисциплина от 2ри или 3ти семестър - пз група 1
група/групи - 130201 виртуална стая

15:40-19:10 Компютърни мрежи: Connecting Networks - пз в поток
група/групи - 722211, 726211 зала 1405

Вторник:

9:50-11:30 Курсов проект от дисциплина от 2ри или 3ти семестър - пз група 1
група/групи - 126202 виртуална стая

11:45-13:20 Курсов проект от дисциплина от 4ти или 5ти семестър - пз група 2
група/групи - 12691 виртуална стая

13:50-15:25 Курсов проект от дисциплина от 2ри или 3ти семестър - пз група 1
група/групи - 130201 виртуална стая

Сряда:

9:50-11:30 Администриране на UNIX и Linux операционни системи - пз група 2
група/групи - 12291 виртуална стая

11:45-13:20 Администриране на UNIX и Linux операционни системи - пз група 2
група/групи - 12691 виртуална стая

13:50-15:25 Учебна практика - I част - пз група 1
група/групи - 126211 виртуална стая

Четвъртък:

8:00-9:35 Компютърни мрежи: Connecting Networks - практическо занятие /пз/
група/групи - 10391 зала 1406

9:50-11:30 Компютърни мрежи: Connecting Networks - практическо занятие /пз/
група/групи - 11091 зала 1406

<

6.1.2. Word файл

Седмица 5	От	2022-01-31	До	2022-02-06
Понеделник	9:50-11:30	Курсов проект от дисциплина от 2ри или 3ти семестър - пз група 1	група/групи - 126202	виртуална стая
	11:45-13:20	Компютърни мрежи: Connecting Networks - лекция в поток	група/групи - 10391, 11091, 12291, 12691	виртуална стая
	13:50-15:25	Курсов проект от дисциплина от 2ри или 3ти семестър - пз група 1	група/групи - 130201	виртуална стая
	15:40-19:10	Компютърни мрежи: Connecting Networks - пз в поток	група/групи - 722211, 726211	зала 1405
Вторник	9:50-11:30	Курсов проект от дисциплина от 2ри или 3ти семестър - пз група 1	група/групи - 126202	виртуална стая
	11:45-13:20	Курсов проект от дисциплина от 4ти или 5ти семестър - пз група 2	група/групи - 12691	виртуална стая
	13:50-15:25	Курсов проект от дисциплина от 2ри или 3ти семестър - пз група 1	група/групи - 130201	виртуална стая
Сряда	9:50-11:30	Администриране на UNIX и Linux операционни системи - пз група 2	група/групи - 12291	виртуална стая
	11:45-13:20	Администриране на UNIX и Linux операционни системи - пз група 2	група/групи - 12691	виртуална стая
	13:50-15:25	Учебна практика – I част - пз група 1	група/групи - 126211	виртуална стая
Четвъртък	8:00-9:35	Компютърни мрежи: Connecting Networks - практическо занятие /пз/	група/групи - 10391	зала 1406
	9:50-11:30	Компютърни мрежи: Connecting Networks - практическо занятие /пз/	група/групи - 11091	зала 1406
	11:45-13:20	Курсов проект от дисциплина от 4ти или 5ти семестър - практическо занятие /пз/	група/групи - 11091	зала 1402

6.2. Хеширани файлове

```

checksums.txt - Notepad
File Edit Format View Help
NvnaScheduleExporter.exe: fb6b69aefe4adad06bf160b7977cc9bee528e6aec0b05a127941f0da9ea53749
data.xlsx: 26f34ef7fbdd718157f3ad541523fc7f6d2286987a19acf0328ae50de620c162
logo.ico: 2fffba8c8745a462f03111f1868efcad42a2c3b6c392683a780ec0fb87102c0c
logo.png: e8cb9acdb1b3e706e6e948da2de3efc1fc503337ac09975c4eb088e4c96ed151
logo_bg.png: cba875c6cf39bdb726efa8b2eae6571197719adbb909e9e8fb5ac69687d1e5c2
template.xlsx: b8a0776728f9149a644578fdaccec5bd4ddaab96d4f1c299652b8aa8b2ebb8290
DayData.py: da15c1ee4ddddd79ca3785e4cd214e77c5b2bde96d4faf66687fc3ee4a4f9444d
ExporterConfig.py: 97ef9fe6326ccc7aa1d571d35e9955d584776d645955c1f022c327c95fab0d76
ExporterHashChecker.py: fb8bf9ee38c93b049b84c9ad1060ae853931ffa6e1784a329f8d41d589fac5e0
ExporterInterface.py: 09670b2c84cbf3d957ae64d0352f5f8ef87fba34469f8e94769c5990ae13de47
ExporterLogger.py: 266893802cabbe33c1e07aa7ea425644a3b735fb96db86f51296c334382b16a6
ExporterRequestErrorMessages.py: f971e4fd507ae014e1846863809a1df7d9df552bdf7763f3368daf3a8715f3d7
ExporterRequestProcess.py: 66e3fc178ce5e47b75d70e8fb9a2cf345f18c063f14d5092d1d82e0359d4e43e
ExporterUtil.py: bce3c72222c1d0a1c334216b30cde65fa9e61b5e2d028dfc8764c9a6c89294d3
FileExporter.py: 26b19c7359c4d24cf3ff708eae85aebd8d7aaa4e084014e91bf53a0aff336b3
LectureDataForExport.py: 674ce38816bc3c5265515805e4dec4227705291af3aa4a0b9c93ebb9fd83daf
LectureDataForExportTest.py: 918104960ad73683ccdc200634e1a7c36237e803789115734689bd817cec9eb9
OpenPyxlRowInsertOverride.py: 7a61dd0a2f83c1d413afe20a4b3e540eb310fea936495f459e8e8b5aa03267ef

```

```

Command Prompt
Microsoft Windows [Version 10.0.19042.1526]
(c) Microsoft Corporation. All rights reserved.

C:\Users\nnikolov28> certutil -hashfile C:\Main\Workspace\NVNA-Schedule-Exporter\dist\NvnaScheduleExporter.exe SHA256
SHA256 hash of C:\Main\Workspace\NVNA-Schedule-Exporter\dist\NvnaScheduleExporter.exe:
fb6b69aefe4adad06bf160b7977cc9bee528e6aec0b05a127941f0da9ea53749
CertUtil: -hashfile command completed successfully.

C:\Users\nnikolov28> certutil -hashfile C:\Main\Workspace\NVNA-Schedule-Exporter\dist\assets\logo.png SHA256
SHA256 hash of C:\Main\Workspace\NVNA-Schedule-Exporter\dist\assets\logo.png:
e8cb9acdb1b3e706e6e948da2de3efc1fc503337ac09975c4eb088e4c96ed151
CertUtil: -hashfile command completed successfully.

C:\Users\nnikolov28> certutil -hashfile C:\Main\Workspace\NVNA-Schedule-Exporter\dist\assets\data.xlsx SHA256
SHA256 hash of C:\Main\Workspace\NVNA-Schedule-Exporter\dist\assets\data.xlsx:
26f34ef7fbdd718157f3ad541523fc7f6d2286987a19acf0328ae50de620c162
CertUtil: -hashfile command completed successfully.

C:\Users\nnikolov28> certutil -hashfile C:\Main\Workspace\NVNA-Schedule-Exporter\src\ExporterInterface.py SHA256
SHA256 hash of C:\Main\Workspace\NVNA-Schedule-Exporter\src\ExporterInterface.py:
09670b2c84cbf3d957ae64d0352f5f8ef87fba34469f8e94769c5990ae13de47
CertUtil: -hashfile command completed successfully.

C:\Users\nnikolov28>

```

6.3. Уязвимости

```
PS C:\Main\Workspace\NVNA-Schedule-Exporter> bandit '.\src' -r --severity-level 'all' -x '.\src\LectureDataForExportTest.py'
[main] INFO profile include tests: None
[main] INFO profile exclude tests: None
[main] INFO cli include tests: None
[main] INFO cli exclude tests: None
[main] INFO running on Python 3.10.1
Run started:2022-03-01 12:11:20.500739

Test results:
    No issues identified.

Code scanned:
    Total lines of code: 1095
    Total lines skipped (#nosec): 0

Run metrics:
    Total issues (by severity):
        Undefined: 0
        Low: 0
        Medium: 0
        High: 0
    Total issues (by confidence):
        Undefined: 0
        Low: 0
        Medium: 0
        High: 0
Files skipped (0):
```