Курсов Проект ООП – Част 1

ДОКУМЕНТАЦИЯ

Явор Йорданов Чамов | СИТ 1 б) | №21621577

Съдържание.

Глава 1. Увод.

Описание и структура на проекта.

Проектът "Приложение за работа с електронни таблици" има за цел да предостави функционалност за отваряне, редактиране и запис на данни в електронни таблици. Потребителят може да извършва операции върху отворените файлове, като добавя, редактира и изтрива данни, както и да ги записва обратно в същия файл или в друг файл, който избере. Има възможност и за затваряне на файловете без запис на промените. При стартиране на приложението, потребителят въвежда команди, които се изпълняват върху отворените файлове. При отваряне на файл, неговото съдържание се зарежда в паметта и файлът се затваря. Всички промени, направени от потребителя, се пазят в паметта, но не се записват обратно, освен ако потребителят явно поиска това. Данните в таблицата се представят в текстов файл, където всеки ред от файла представя отделен ред в таблицата, с данни, разделени със запетаи. Различните типове данни могат да бъдат представени в една таблица.

Цел и задачи на разработката.

Целта на проекта "Приложение за работа с електронни таблици" е да предостави функционалност за отваряне, редактиране и запис на данни в електронни таблици. Освен основните операции като отваряне, затваряне, запис и помощ, приложението трябва да поддържа допълнителни операции. Това включва извеждане на съдържанието на таблицата на екрана, редактиране на отделни клетки чрез въвеждане на ново съдържание и работа с формули. Потребителят може да въведе различни типове данни, като цели числа, дробни числа, низове и формули. Приложението трябва да проверява въведените данни и да извежда подходящи съобщения за грешки, без да прекратява изпълнението си. Формулите могат да използват номерацията на редове и колони в таблицата и да извършват сметки с числови стойности, низове и празни клетки. При наличие на грешка в формулата, приложението трябва да изведе съобщение за грешка и да покаже "ERROR" в съответната клетка на екрана, без да прекъсва своето изпълнение.

Структура на документацията.

Документацията за проекта е структурирана в няколко глави, които представят последователно различни аспекти на разработката. Глава 1, "Увод", включва описание на проекта, неговата идея и целите на разработката. Глава 2, "Преглед на предметната област", представя основни дефиниции, концепции и алгоритми, които ще бъдат използвани, както и анализ на проблемите и подходите за тяхното решаване. В глава 3, "Проектиране", се разглежда общата структура на проекта и се представят диаграми и блок схеми, които илюстрират структурата

и поведението на системата. Глава 4, "Реализация, тестване", се фокусира върху реализацията на класовете, включително важни моменти и кодови фрагменти, както и планирането и създаването на тестови сценарии. Заключителната глава 5, "Заключение", представя обобщение на постигнатите цели и предлага насоки за бъдещо развитие и усъвършенстване на проекта.

Глава 2. Преглед на предметната дейност.

Основни дефинции и концепции.

Калкулатор на формули. (*FormulaCalculator*)

Калкулаторът на формули представлява калкулатор, използван за изчисляване на формули. Предоставя метод за изчисляване, който приема формула като вход и връща изчисления резултат. Формулите могат да включват различни математически операции и препратки към други клетки в таблица. Калкулаторът извършва необходимите изчисления, базирани на подадената формула, и връща резултата.

Команда. (*Command*)

Дефинира структурата и поведението на командите в команден интерфейс. Включва свойството статус код, което представя статусния код на изпълнението на командата, и метод, който е отговорен за изпълнението на логиката на командата. Подкласовете на командата могат да презапишат метода за изпълнение, за да реализират специфична функционалност за командата.

Команда с аргументи. (*ArgumentCommand*)

Командата с аргументи е подклас на командата, който разширява функционалността му, като позволява включването на списък от аргументи. Въвежда свойството аргумети, което е списък от низове, представляващи аргументите, свързани с командата. Тези аргументи предоставят допълнителен вход или параметри за изпълнението на командата. Класът може да се използва, когато командите изискват допълнителна информация или персонализация.

Клетка от таблица. (*TableCell*)

Представлява клетка в таблица. Включва свойствата тип и стойност, които определят типа на данните и стойността, съхранени в клетката, съответно. Свойството тип задава типа на данните, съхранени в клетката, като число, низ или формула. Свойството стойност съдържа реалната стойност на клетката, която може да се достъпва или модифицира при нужда. Класът `*TableCell*` се използва за съхранение и манипулация на отделни клетки в рамките на таблица.

Четец на таблица. (*TableReader*)

Отговорен за четене и обработка на данни от таблица от файл или друг източник на данни. Предоставя метод за четене, който извлича данните от таблицата и ги връща в подходящ формат за по-нататъшна обработка. Класът

`TableReader` се грижи за необходимите операции за достъп и извличане на информация от таблицата от указания източник, позволявайки на други компоненти да работят с получените данни.

Писач на таблица. (*TableWriter*)

Отговорен за записването и запазването на данни от таблица във файл или друга целева дестинация. Предоставя метод за писане, който приема данните от таблицата като вход и ги записва в зададената дестинация. Класът *TableWriter* се грижи за необходимите операции за форматиране и съхранение на данните от таблицата в желания формат. Той позволява на другите компоненти да запазват данните от таблицата и да ги запазят за бъдещо извличане или обработка. Класът играе важна роля в общото управление и запазване на информацията от таблицата в рамките на приложението.

Хранилище на таблицата. (*TableRepository*)

Служи служи като хранилище за управление на таблицата в приложението. Той включва функционалността, свързана със създаването, извличането, обновяването и изтриването на таблицата. Класът *„TableRepository“* предоставя методи за изпълнение на операции като създаване на нова таблица, запазване на промени в съществуваща таблица, извличане на конкретна клетка по идентификатор и изтриване на таблица. Той действа като интерфейс между приложението и основното хранилище или източник на данни, където се запазват таблиците.

Тази документация представя общ преглед на основните дефиниции и концепции, свързани с предоставените класове. Тя обяснява техните цели, свойства и методи, давайки основа за разбиране и работа с тези класове в контекста на софтуерното разработване.

Стандарти.

Следван е стандартът: Семантично версиониране 2.0.0. (Semantic versioning 2.0.0). Семантичното версиониране 2.0.0 (SemVer 2.0.0) е стандарт за версиониране на софтуер, който се основава на задаване на семантичен смисъл на номерата на версиите. Стандартът предоставя конвенции за задаване на версии със семантично значение, което улеснява разбирането на промените и съвместимостта между различни версии на софтуер. Семантичното версиониране 2.0.0 се състои от три числа - MAJOR, MINOR и PATCH: MAJOR версията се увеличава, когато правите неблагоприятни промени, които не са обратно съвместими с предходни версии. MINOR версията се увеличава, когато добавяте нови функционалности, които са обратно съвместими с предходни версии. PATCH версията се увеличава, когато правите неблагоприятни корекции или малки промени, които са обратно съвместими с предходни версии. Проектът съдържа

VERSION текстов файл, който се използва за съхранение на информация относно текущата версия на артифакта или програмата Файлът VERSION съдържа само един ред. В този ред е записана версията формат (например: 1.0.0).

Алгоритми

***"Reverse Polish notation"* - алгоритъм за извършване на изчислителните операции в калкулатора.**

Обратният полски запис (Reverse Polish Notation или RPN) е математическа нотация, която премахва нуждата от скоби и правила за предшествуване на операторите в аритметичните изрази. В RPN операторите се поставят след операндите, вместо между тях. Тази нотация е разработена с цел да опрости изчисляването на математически изрази. За да се оцени израз в RPN, операндите се поставят в стек, и когато се срещне оператор, той се прилага върху върховите елементи на стека. Този процес продължава, докато изразът бъде напълно оценен, а окончателният резултат е стойността, останала на върха на стека. RPN предлага няколко предимства, като отстранява двусмислеността, позволява лесна имплементация в компютърни алгоритми и опростява процеса на изчисление на сложни изрази.

Алгоритъм за скалиране на таблица.

Алгоритъмът се използва за промяна на размера на таблицата до указаните нови размери, като създава нови празни клетки, ако е необходимо. При извикването на алгоритъма, първо се проверява дали предоставените нови стойности за новите редове и колони са валидни. След това алгоритъмът създава нова таблица с предоставените размери. Итерира се през новата таблица и за всяка клетка се проверява дали има съответстваща клетка в текущата таблица, като се сравняват индексите на ред и колона. Ако има съответстваща клетка, съдържанието на клетката от текущата таблица се копира в новата таблица.

Алгоритъм за смаляване на таблицата.

Алгоритъмът премахва празни редове и колони от долната и дясната страна на таблицата. Ако таблицата е напълно празна, този метод не извършва никакви промени. Този алгоритъм намира последния непразен ред и колона в таблицата. След това изчислява новия размер на таблицата, като се използват тези последни непразни индекси. Създава се нова таблица с актуализирания размер и се копират непразните клетки от първоначалната таблица в новата.

Дефиниране на проблеми и сложност на поставената задача

Проблем 1: Реализиране на класова йерархия за всички видове команди

Необходимо е да се реализират различни команди и да се извършват операции в отговор на тях. Въпросът възниква как да се дефинират и управляват тези команди по ефективен и гъвкав начин. Трябва да се преодолеят следните проблеми и сложности:

1. Разделение на отговорностите: Необходимо е да се раздели изпълнението на командите от обектите, които ги извикват. Това поддържа принципа на една отговорност и прави системата по-модулна и поддържаема.
2. Изпълнение на различни команди: Поставя се задачата да се реализират различни команди с различна функционалност и поведение. Възниква нуждата от гъвкав и разширяем механизъм, който да позволява добавянето на нови команди без промяна в съществуващия код.
3. Управление на последователността: В някои случаи е необходимо да се управлява последователността на изпълнение на командите или да се групират в по-големи операции. Това изисква допълнителни механизми за управление на командите и тяхното изпълнение.

Проблем 2: Четене и запис на табличните данни от файл

Необходимо е да се реализира подход за четене и записване на таблични данни от и във външен източник.

Това води до следните проблеми и сложности, които трябва да бъдат решени:

1. Форматиране и обработка на данните: Различните източници на данни могат да използват различни формати или структури на данните. Например, данните могат да бъдат представени в CSV формат, Excel таблица или база данни. Това изисква разбиране на формата на данните и правилното тяхно обработване, за да бъдат извлечени и запазени в правилната структура на табличните данни.
2. Ефективно четене и записване: При големи обеми от данни е важно да се осигури ефективност при операциите на четене и записване. Това включва оптимално управление на паметта и използването на подходящи алгоритми и структури данни, които да позволят бърз достъп до табличните данни и минимално използване на ресурси.
3. Откриване на грешки и обработка на изключения: При операциите на четене и записване може да възникнат различни грешки, като неправилен формат на данните, несъответствие на типовете, недостатъчни права за

достъп и други. Тези грешки трябва да бъдат открити, съобщени на потребителя и обработени по подходящ начин, за да се осигури надеждност и стабилност на операциите.

Проблем 3: Изчисляване на табличните формули

Необходимо е да се реализира подход за изчисляване на математически формули в клетките на таблицата.

Срещат се някои сложности, които трябва да бъдат решени. Това включва следните аспекти:

1. Обработка на формули: Формулите могат да бъдат сложни и да включват различни операции и функции. Изчисляването на формулите изисква правилното разбиране и обработка на тяхната структура. Това включва разпознаване на операциите, извличане на операндите и правилното прилагане на операциите в правилния ред.
2. Управление на зависимости: Формулите могат да зависят от стойностите на други клетки или променливи. За изчисляване на формулите е необходимо управление на зависимостите и следване на промените във входните данни. При промяна на стойността на клетка, зависими формули трябва да бъдат преизчислени автоматично.
3. Обработка на грешки: Възможни са грешки при изчисляването на формули, като деление на нула, невалидни операции или невалидни аргументи за функциите. Тези грешки трябва да бъдат открити и обработени по подходящ начин, за да се предотврати срив на приложението или невалидни резултати.

Подходи, методи за решаване на поставените проблеми

Подход към Проблем 1: Реализиране на класова йерархия за всички видове команди

За решаване на тези проблеми и сложности се използва шаблонът Command. Той предоставя абстракция за командите и ги представя като обекти, които могат да бъдат изпълнени посредством методи като *„execute()“*. Този подход позволява изолирането на логиката на командите, управлението на тяхното изпълнение и лесното добавяне на нови команди. Шаблонът Command допринася за постигане на по-голяма гъвкавост, разширяемост и поддръжаемост в системите, които се нуждаят от управление на команди и операции.

Подход към Проблем 2: Четене и запис на табличните данни от файл

„TableReader“ и *„TableWriter“* са два класа, които се занимават с четенето и записването на таблични данни. Те предоставят функционалност за извличане на данните от външни източници и тяхното записване в подходящ формат. За да решат поставените проблеми и сложности, тези класове трябва да се справят с форматирането и обработката на данните, ефективното управление на операциите на четене и записване и обработката на грешки и изключения. Чрез предоставянето на такива функционалности *„TableReader“* и *„TableWriter“* помагат за улесняване на работата с таблични данни и осигуряват правилното и надеждно манипулиране на тези данни.

Подход към Проблем 3: Изчисляване на табличните формули

Класът *„FormulaCalculator“* се занимава с изчислението на формули. Той предоставя функционалност за приемане на формула и извършване на съответните математически изчисления. За да реши проблемите и сложностите, свързани с формулите, *„FormulaCalculator“* трябва да разбере и обработи структурата на формулите, да управлява зависимостите и да се справи с възможни грешки при изчисленията. Чрез предоставянето на тези функционалности *„FormulaCalculator“* подпомага извършването на изчисления.

Потребителски (функционални) изисквания (права, роли, статуси, диаграми, ...) и качествени (нефункционални) изисквания (скалируемост, поддръжка, ...)

Функционални изисквания:

Приложението трябва да позволява на потребителя да отваря файлове и да извършва операции с тях, като например редактиране на стойности на клетки в таблица.

Потребителят трябва да може да запише промените, направени във файла, или чрез презаписване на оригиналния файл, или като посочи различно име на файл.

Трябва да има опция за затваряне на файла без запазване на промените.

Приложението трябва да предоставя интерфейс на командния ред, където потребителят може да въвежда команди за изпълнение на различни операции.

При отваряне на файл съдържанието му трябва да се зареди в паметта и файлът да се затвори. Всички промени, направени от потребителя след това, трябва да се съхраняват в паметта, но не и да се записват обратно във файла, освен ако не е изрично указано.

Нефункционални изисквания:

Обработка на грешки: Приложението трябва да обработва грешките. Ако възникне грешка по време на зареждане на данни или друга операция, трябва да се покаже подходящо съобщение за грешка и приложението трябва да продължи да работи.

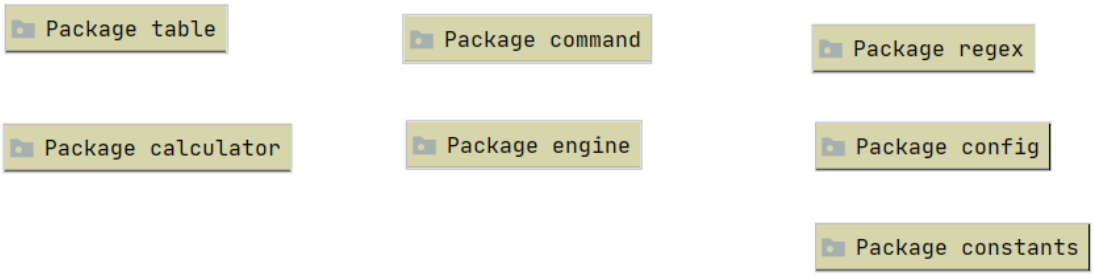
Потребителски интерфейс: Отпечатаният изход на таблицата трябва да бъде добре форматиран с подравнени колони, разделени с вертикални ленти (|).

Типове данни: Приложението трябва да поддържа различни типове данни за клетки на таблица, включително цели числа, числа с плаваща запетая, низове и формули.

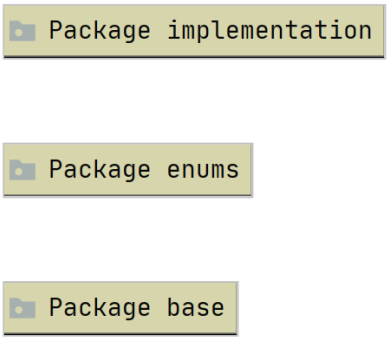
Изчисляване на формули: Формулите трябва да се изчисляват правилно, като се вземат предвид препратките към клетки и се извършват аритметични изчисления. Грешки във формули, като деление на нула, трябва да се обработват чрез показване на „ГРЕШКА“ в съответната клетка при отпечатване на таблицата.

Глава 3. Проектиране.

Обща структура на проекта пакети, които ще се реализират.



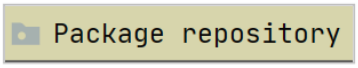
Пакетът *„calculator“* съдържа класа *„FormulaCalculator“*, който е отговорен за изчисляването на математически изрази и формули.
Пакетът *„command”* е разделен на три подпакета:



1. *„base“* - съдржа интерфейси и базови класове за създаване на команди
2. *„implementation“* - съдържа конкретна имплементация на командите
3. *„enums“* - съдържа списък с валидните команди

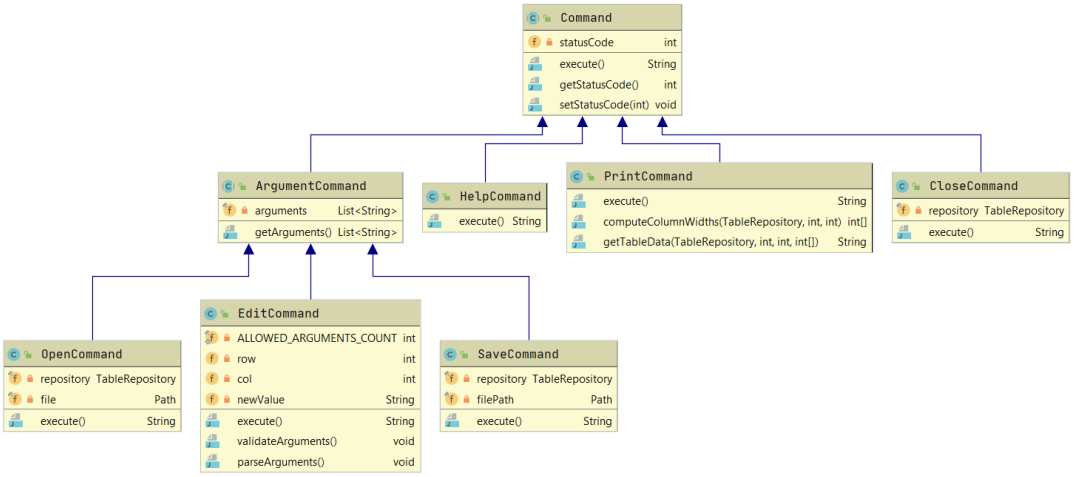
Пакетът *„config“*съдържа клас за управление на конфигурацията на приложението.
Пакетът *„regex“* съдържа Regex шаблоните, използвани в приложението.
Пакетът *„constants”* съдържа всички изходни съобщения.

Пакетът *„table”* е разделен на пет подпакета:

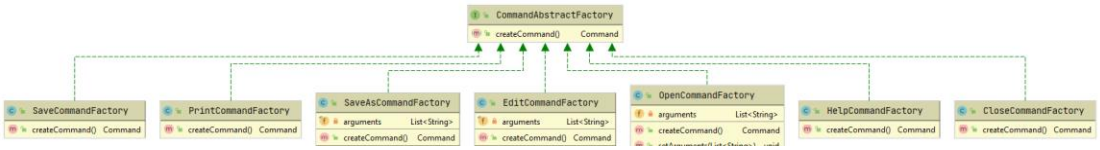


1. Пакетът *“repository”*съдържа класа TableRepository, който отговаря за управлението на състоянието на таблицата и файла, с който е свързана.
2. Пакетът *“reader”* съдържа класа TableReader, който отговаря за четенето на данни от текстов файл и преобразуването му в обекти TableCell.
3. Пакетът *"writer"* съдържа имплементацията на класа TableWriter. Този клас предоставя функционалност за запис на съдържанието на таблица във файл.
4. Пакетът *"cell"* съдържа класа TableCell, който представлява една клетка в таблица на електронна таблица.
5. Пакетът *“util”* съдържа класа CellTypeUtil, който предоставя помощни методи, свързани с изброяването на CellType. Класът CellTypeUtil е отговорен за предоставянето на методи, които помагат при идентифицирането на типа на стойността на клетката.

Диаграми/Блок схеми (на структура и поведение - по обекти, слоеве с най-важните извадки от кода)

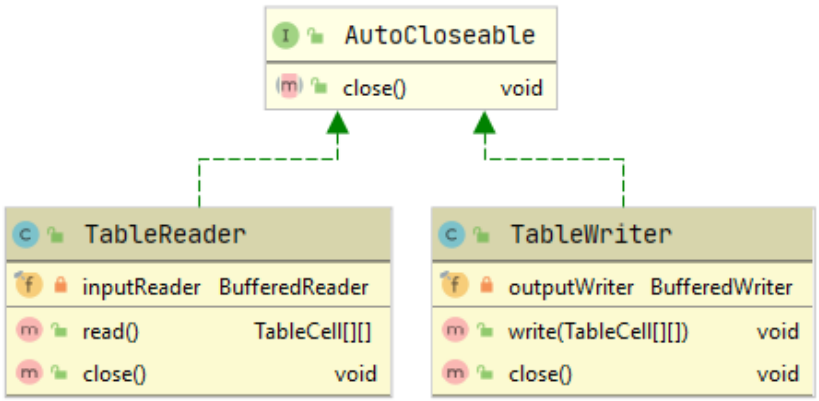


Йерархията на показаните класовете е свързана с модела на командите, който е модел на поведенчески дизайн, който позволява да се дефинира йерархия от команди, да капсулираме всяка една и да ги направим взаимозаменяеми. В горната част на йерархията имаме команден абстрактен клас, който дефинира метода за изпълнение, който трябва да бъде приложен от всяка конкретна команда, както и върнатия статус код. Абстрактният клас ArgumentCommand разширява командния интерфейс и предоставя имплементация за анализиране на аргументите, изисквани от неговите подкласове. Неговите подкласове, EditCommand, SaveCommand и OpenCommand, са конкретни команди, които изискват допълнителни аргументи за изпълнение. Класовете HelpCommand, PrintCommand и CloseCommand са конкретни команди, които не изискват допълнителни аргументи за изпълнение, следователно не наследяват от ArgumentCommand. Тази йерархия позволява отделянето на командите от логиката, която ги изпълнява, което улеснява добавянето на нови команди в бъдеще, без да се променя съществуващият код.



Тази йерархия на класове имплементира шаблона за проектиране на фабричния метод. Интерфейсът *„AbstractCommandFactory“* е абстрактната фабрика, която дефинира метода *„createCommand“*, който връща обект *„Command“*. Конкретните фабрични класове *„HelpCommandFactory“*, *„PrintCommandFactory“*, *„CloseCommandFactory“*, EditCommandFactory, SaveCommandFactory, OpenCommandFactory и SaveAsCommandFactory всички

имплементират интерфейса AbstractCommandFactory и всеки замества метода createCommand, за да създаде конкретен команден обект. Тези конкретни фабрични класове са отговорни за създаването на екземпляри на HelpCommand, PrintCommand, CloseCommand, EditCommand, SaveCommand, OpenCommand и SaveAsCommand, съответно чрез внедряване на метода createCommand на интерфейса AbstractCommandFactory. Този подход ни позволява да капсулираме създаването на командни обекти и гарантира, че създаването на всяка команда се обработва от съответния фабричен клас.



Интерфейсът AutoCloseable е част от Java API и предоставя начин за автоматично освобождаване на ресурси, когато вече не са необходими. Интерфейсът декларира един метод close(), който освобождава всички ресурси, използвани от обект, който го прилага. Класовете TableWriter и TableReader реализират интерфейса AutoCloseable, което означава, че те могат да се използват в *"try-with-resources"* блок, за да се гарантира, че всички ресурси, които използват (напр. файлови потоци), се освобождават автоматично при излизане от блока. Класът TableWriter отговаря за записването на данни във файл, докато класът TableReader е отговорен за четенето на данни от файл. И двата класа използват съответно класовете BufferedWriter и BufferedReader за четене и запис на данни от/във файл. Тъй като и двамата обработват файлов I/O, важно е да се уверим, че всички ресурси, които използват, са правилно освободени, за което служи интерфейсът AutoCloseable.

Глава 4. Важни моменти при реализация, алгоритми, оптимизации.

Абсрактен команден клас, използван за създаването на всички команди.

```
public abstract class Command {
   private int statusCode = UNSUCCESSFUL_STATUS_CODE;
   public abstract String execute();
   public int getStatusCode() {
     return statusCode;
   }
   protected void setStatusCode(int statusCode) {
     this.statusCode = statusCode;
   }
}
```

Командният клас определя поведението на команда, която може да бъде изпълнена от приложението. Притежава статусен код, който представя резултата от операция. По подразбиране е настроен като успешен статусен код. Методът execute изпълнява командата и връща съобщение, показващо резултата от изпълнение на командата. Този клас се наследява от всички командни класове.

Абстрактен клас, който разширява командния с добавянето на аргументи.

В допълнение на абстрактният клас Command, съществува и команден абстрактен клас, който разполага с допълнително свойство за приемане на списък от низови променливи.

```
public abstract class ArgumentCommand extends Command {
   private final List<String> arguments;
   protected ArgumentCommand() {
     this.arguments = new
ArrayList<>(Collections.singletonList(DEFAULT_TABLE_FILENAME))
;
}
   protected ArgumentCommand(List<String> arguments) {
     this.arguments = arguments;
   }

   protected List<String> getArguments() {
     return arguments;
   }
}
```

Абстрактен базов клас за команди, които приемат списък от низови аргументи. Подкласовете трябва да наследяват абстрактният клас Command и дефинират как да се изпълни командната логика. Този клас предоставя конструктор по

подразбиране, който инициализира аргументите със стойност по подразбиране, и конструктор което позволява предаване на потребителски списък с аргументи. Той също така предоставя метод за извличане на текущия списък с аргументи. Подкласовете могат да добавят допълнителни полета, конструктори и методи, ако е необходимо, за да реализират специфичното си поведение.

За имплементирането на Abstract Factory шаблонът се използва interface CommandAbstractFactory. Реализациите на този интерфейс трябва да предоставят начин за създаване на нова инстанция на обект Command, които след това могат да бъдат изпълнени от приложението.

Интерфейс за създаване на команди използвайки *"Abstract Factory"*.

```
public interface CommandAbstractFactory {
   Command createCommand();
}
```

Ползите от този дизайнов шаблон са следните:

1. Инкапсулиране: Абстрактният фабричен модел капсулира създаването на обекти от команди. Той покрива сложността на създаването на обекти и предоставя единен интерфейс за създаване на команди, без да разкрива подробностите на конкретната им реализация.

2. Гъвкавост: Абстрактният фабричен модел позволява лесно подмяна на различни типове команди. Чрез използване на общ интерфейс за създаване на команди между тях можем да превключваме различни конкретни фабрики, които произвеждат различни типове команди, без да повлияваме на клиентския код.

3. Разширяемост: Абстрактният фабричен модел улеснява въвеждането на нови типове команди без модификация на съществуващия код. Просто създаваме нова конкретна фабрика, която произвежда нов тип команда, без да е необходима промяна на клиентския код, който използва абстрактния фабричен интерфейс.

Това е примерна имплементация на команден клас, който отваря файл с таблица и записва съдържанието му в хранилището на таблицата. Целта е да изпълнява командата, като отваря файла с таблицата, посочен в аргументите, чете съдържанието му, и го зарежда в хранилището на таблицата.

Създаване на шаблон за команда с аргументи.

```
public class OpenCommand extends ArgumentCommand {

private final TableRepository repository =
TableRepository.getInstance();

private final File with path;

OpenCommand(List<string> arguments) {
  super(arguments);
  this.file = Paths.get(RESOURCES_DIRECTORY,
arguments.get(0));
}

@Override
public String execute() {
if (repository.isTableOpened()) {
  return TABLE_ALREADY_OPENED_MESSAGE;
}
try (TableReader tableReader = new TableReader(file)) {
  repository.loadData(tableReader.read());

repository.setTableFileName(file.getFileName().toString());
setStatusCode(SUCCESSFUL_STATUS_CODE);
        return TABLE_OPENED_SUCCESSFULLY_MESSAGE;
      } catch (IOException | IllegalArgumentException e) {
        return String.format(ERROR_OPENING_TABLE_MESSAGE,
e.getMessage());
      } catch (ArrayIndexOutOfBoundsException e) {
        return INSUFFICIENT_ROW_OR_COLUMNS_ERROR_MESSAGE;
      }
}
```

Използването на „*try-with-resources*“ в този код има следните ползи:

1. Автоматично затваряне на ресурси: Синтаксисът на try-with-resources автоматично затваря TableReader обекта след приключването на блока на try. Това гарантира, че ресурсът ще бъде правилно освободен, без да се налага изрично да се извиква методът close() в кода.

2. Гарантирано използване на ресурса: Независимо от това дали блока на try приключи нормално или по средата на него бъде хвърлено изключение, ресурсът (TableReader) ще бъде правилно затворен. Това предотвратява утечки на ресурси и гарантира, че ресурсът няма да бъде оставен отворен в случай на грешка.

При update-ването на статус кода в горния код се постигат няколко ползи. Статус кодът представлява информация за резултата от изпълнението на операцията и е от голямо значение за потребителя или други части на системата, които се нуждаят от тази информация. Като update-ваме статус кода, предоставяме ясен и конкретен начин за обратна връзка, който може да бъде използван за вземане на решения или за визуално представяне на резултата на операцията.

Реално създаване на обект от тип команда чрез *"Abstract Factory"*

```
private void executeOpenCommand(List<String> arguments) {

      OpenCommand open = (OpenCommand)
CommandFactory.getCommand(new OpenCommandFactory(arguments));
      System.out.println(open.execute());

      if (open.getStatusCode() != SUCCESSFUL_STATUS_CODE) {
        executeExitCommand(Collections.emptyList());
      }
    }
```

Даденият код представлява метод, който извършва следните действия:

1. Създава инстанция на обект от тип "OpenCommand" чрез извикване на статичния метод "getCommand" на класа "CommandFactory". Този метод приема фабрика за съответния тип команда (в случая "OpenCommandFactory") и създава съответния обект.

2. След създаването на обекта се извиква методът "execute" върху него, който изпълнява командата "OpenCommand". Резултатът от изпълнението на командата се извежда на стандартния изход

3. След изпълнението на командата "OpenCommand" се проверява статус кодът на командата чрез използването на методът "getStatusCode()". Ако статус кодът не е "SUCCESSFUL_STATUS_CODE", се извиква методът "executeExitCommand" с празен списък от аргументи, който изпълнява командата "ExitCommand" за излизане от програмата.

Метод за изчисляване на формули в клетките. Reverse Polish Noation.

```
public double evaluate(String formula) throws
ArithmeticException {

    List<String> tokens = tokenize(formula);
    Deque<Double> values = new ArrayDeque<>();
    Deque<Character> ops = new ArrayDeque<>();

    for (String token : tokens) {
      if (isCellReference(token)) {
        double cellValue = getCellValue(token);
        values.push(cellValue);
      } else if (isInteger(token) ||
isFractionalNumber(token)) {
        values.push(Double.parseDouble(token));
      } else if (isOperator(token)) {
        char op = token.charAt(0);
        while (!ops.isEmpty() && hasPrecedence(ops.peek(),
op)) {
          double b = values.pop();
          double a = values.pop();
          char prevOp = ops.pop();
          double result = applyOperation(prevOp, b, a);
          values.push(result);
        }
        ops.push(op);
      }
    }

    while (!ops.isEmpty()) {
      double b = values.pop();
      double a = values.pop();
      char op = ops.pop();
      double result = applyOperation(op, b, a);
      values.push(result);
    }

    return values.pop();
  }
```

Даденият код представлява метод с име "evaluate", който извършва изчисляване на математическа формула и връща резултата от изчисленото.

1. Създаване на списък от токени (лексеми) - формулата се разделя на отделни части (токени) чрез използване на метода "tokenize". Токените се съхраняват в списък "tokens".

2. Създаване на стекове за стойности и операции - създават се два стека, "values" и "ops", които ще се използват при изчисляване на формулата.

3. Итерация през токените - за всеки токен в списъка "tokens" се извършва следната проверка: Ако токенът е препратка към клетка (cell reference), се взема стойността на клетката чрез метода "getCellValue" и се поставя в стека "values". Ако токенът е цяло число или дробно число, се парсира в double и се поставя в стека "values". Ако токенът е оператор, се извършва следното: Ако стекът "ops" не е празен и операторът има по-висок или равен приоритет от оператора на върха на стека "ops", се извършва следното: Изваждат се две стойности (b и a) от стека "values". Изважда се предходният оператор от стека "ops". Извършва се операцията с предходния оператор върху стойностите b и a. Резултатът се поставя в стека "values". Операторът се добавя в стека "ops".

4. Изчистване на останалите операции - след като са обходени всички токени, се извършва следното: Докато стекът "ops" не е празен, се извършва следното: Изваждат се две стойности (b и a) от стека "values". Изважда се оператор от стека "ops". Извършва се операцията с оператора върху стойностите b и a. Резултатът се поставя в стека "values". Връщане на резултат - накрая, връща се стойността, която остава във върха на стека "values".

Извличане на токените (лексемите) за калкулация.

```
private List<String> tokenize(String formula) {
    List<String> tokens = new ArrayList<>();
    StringBuilder currentToken = new StringBuilder();
    for (int i = 0; i < formula.length(); i++) {
      char currentChar = formula.charAt(i);
      if (Character.isDigit(currentChar) || currentChar ==
'.') {
        currentToken.append(currentChar);
      } else if (Character.isAlphabetic(currentChar)) {
        currentToken.append(currentChar);
      } else {
        if (currentToken.length() > 0) {
          tokens.add(currentToken.toString());
          currentToken.setLength(0);
        }
        tokens.add(Character.toString(currentChar));
      }
    }
    if (currentToken.length() > 0) {
      tokens.add(currentToken.toString());
    }
    if (tokens.size() > 0 && tokens.get(0).equals("=")) {
      tokens.remove(0);
    }
    return tokens;
  }
```

Даденият код представлява метод с име "tokenize", който разделя математическата формула на отделни токени (лексеми) и ги връща в списък.

1. Създаване на списък и текущ токен - създава се празен списък "tokens", който ще съдържа разделените токени. Създава се също и StringBuilder "currentToken", който се използва за съставяне на текущия токен.

2. Итерация през символите на формулата - за всеки символ във формулата се извършва следното: Ако символът е цифра или точка, той се добавя към текущия токен, като се използва методът append на StringBuilder. Ако символът е буква, той се добавя към текущия токен, като се използва методът append на StringBuilder. В противен случай, ако текущият токен има дължина по-голяма от 0, се добавя към списъка "tokens" стойността на текущия токен.

3. След това текущият токен се изчиства (сетва се с празен StringBuilder), а символът се добавя като нов токен в списъка "tokens". Добавяне на последния токен - след като са обходени всички символи, се извършва следното: Ако текущият токен има дължина по-голяма от 0, се добавя

към списъка "tokens" стойността на текущия токен. Премахване на първия токен, ако е "=" - ако списъкът "tokens" има поне един елемент и първият елемент е "=", той се премахва от списъка. Връщане на списъка с токените - връща се сформираният списък с токените. Този код имплементира алгоритъм за токенизация на формула, като разделя символите на формулата в отделни токени в списък. Токените представляват числа, оператори или променливи и са необходими за по-нататъшна обработка и изчисление на математическата формула.

Статичен блок със зареждане на външна конфигурация.

```
static {
    Properties properties = new Properties();
    try {
      properties.load(new
FileInputStream("config.properties"));
    } catch (IOException e) {
      System.out.println(e.getMessage());
    }

    ROWS = Integer.parseInt(properties.getProperty("rows",
"100"));
    COLS = Integer.parseInt(properties.getProperty("cols",
"100"));
    CELLS_SEPARATOR =
properties.getProperty("cells.separator", ",");
    RESOURCES_DIRECTORY =
properties.getProperty("resources.directory",
            new
File(Config.class.getProtectionDomain().getCodeSource().getLoc
ation().getPath()).getParent());
    DEFAULT_TABLE_FILENAME =
properties.getProperty("default.table.filename", "data.csv");
    CELL_ERROR_VALUE =
properties.getProperty("cell.error.value", "ERROR");
  }
```

Даденият код представлява статичен блок и се използва за инициализация на статични полета в класа. Създава се обект от тип Properties, който се използва за зареждане на стойностите от конфигурационния файл "config.properties". Този файл трябва да се намира в същата директория, от която се изпълнява програмата. Опитът за зареждане на конфигурационния файл се обвива в try-catch блок, за да се обработи възможното изключение при грешка при четенето на файла. Ако зареждането на конфигурационния файл премине успешно, стойностите от файла се използват за инициализация на статични полета в класа. Тези полета са: ROWS: броят на редовете на таблицата, взет от стойността на ключа "rows" в конфигурационния файл. Ако ключът не съществува, се използва стойността по подразбиране "100". COLS: броят на

колоните на таблицата, взет от стойността на ключа "cols" в конфигурационния файл. Ако ключът не съществува, се използва стойността по подразбиране "100". CELLS_SEPARATOR: разделителят за клетките в таблицата, взет от стойността на ключа "cells.separator" в конфигурационния файл. Ако ключът не съществува, се използва стойността по подразбиране "," (запетая). RESOURCES_DIRECTORY: директорията за ресурсите, взета от стойността на ключа "resources.directory" в конфигурационния файл. Ако ключът не съществува, се използва директорията на текущия клас, взета чрез Config.class.getProtectionDomain().getCodeSource().getLocation().getPath(). Това осигурява, че програмата ще търси ресурси като файлове в директорията, от която е стартирана. DEFAULT_TABLE_FILENAME: името на файла по подразбиране за таблицата, взето от стойността на ключа "default.table.filename" в конфигурационния файл. Ако ключът не съществува, се използва стойността по подразбиране "data.csv". CELL_ERROR_VALUE: стойността, която се използва за обозначение на грешки в клетките на таблицата, взета от стойността на ключа "cell.error.value" в конфигурационния файл. Ако ключът не съществува, се използва стойността по подразбиране "ERROR". Този код предоставя гъвкавост при конфигурирането на различни параметри на приложението. Стойностите, прочетени от конфигурационния файл, могат да бъдат променени без промяна на изходния код на приложението, което улеснява настройката и поддръжката на програмата.

Задаване стойност на клетката

```
private void setValue(String data) {
    if (data.trim().isEmpty()) {
      this.value = null;
    } else if (isInteger(data)) {
      this.value = Integer.parseInt(data);
    } else if (isFractionalNumber(data)) {
      this.value = Double.parseDouble(data);
    } else if (isString(data)) {
      data = removeQuotes(data).trim();
      if (data.isEmpty()) {
        this.value = null;
        return;
      }
      if (!areAllQuotesEscaped(data)) {
        throw new IllegalArgumentException(data +
UNESCAPED_QUOTES_ERROR_MESSAGE);
      }
if (!areAllBackslashesEscaped(data)) {
        throw new IllegalArgumentException(data +
UNESCAPED_BACKSLASH_ERROR_MESSAGE);
      }
      this.value = parseEscapedString(data);
    } else if (isFormula(data)) {
      this.value = data;
```

```
    } else {
      throw new IllegalArgumentException(data +
UNKNOWN_DATA_TYPE_MESSAGE);
    }
  }
```

Даденият код представлява метод setValue, който служи за задаване на стойност на клетката. Ако подаденият низ data е празен или съдържа само интервали (след извличането на интервалите с trim()), това означава, че няма зададена стойност, и полето value се задава на null. Ако data е цяло число, полето value се задава като цяло число, използвайки метода Integer.parseInt(data). Ако data е десетично число, полето value се задава като десетично число, използвайки метода Double.parseDouble(data). Ако data е символен низ, то може да има различни случаи: Ако низът е празен след премахването на кавичките и интервалите (с removeQuotes(data).trim()), това означава, че няма зададена стойност, и полето value се задава на null. Ако в низа има некоректно ескейпнати кавички (директно преди тях няма обратна наклонена черта), хвърля се изключение от тип IllegalArgumentException със съобщение, което включва data и съобщението за грешка UNESCAPED_QUOTES_ERROR_MESSAGE. Ако в низа има некоректно ескейпнати обратни наклонени черти (недостатъчен брой обратни наклонени черти пред друга наклонена черта), хвърля се изключение от тип IllegalArgumentException със съобщение, което включва data и съобщението за грешка UNESCAPED_BACKSLASH_ERROR_MESSAGE. Ако няма грешки в ескейпирането на кавичките и обратните наклонени черти, низът се извлича с правилното ескейпиране чрез метода parseEscapedString(data) и се задава като стойност на полето value. Ако data е формула, т.е. не отговаря на нито един от предходните случаи, но е разпознато като формула, то data се задава директно като стойност на полето value. Ако data не съответства на нито един от горните случаи, хвърля се изключение от тип IllegalArgumentException със съобщение, което включва data и съобщението за грешка UNKNOWN_DATA_TYPE_MESSAGE.

Метод, проверяващ дали всички кавички са *escape-нати*

```
private boolean areAllQuotesEscaped(String input) {
    for (int i = 0; i < input.length(); i++) {
      char c = input.charAt(i);
      if (c == '\"' && i == 0) {
        return false;
      }
      if (c == '\"' && input.charAt(i - 1) != '\\') {
        return false;
      }
    }
    return true;
  }
```

Функцията areAllQuotesEscaped проверява дали всички кавички (") в подадения текстов низ са обработени с обратна наклонена черта (\). Кодът извършва следните действия: Итерира през всички символи на входния низ input чрез цикъл. За всяка позиция i в низа се извлича символа с на тази позиция. Проверява се дали първата кавичка на позиция 0 не е обработена с обратна наклонена черта. Ако не е обработена, функцията връща false, защото първата кавичка трябва да бъде escape-ната. Проверява се дали текущата кавичка на позиция i не е escape-ната, като се проверява дали символът преди нея (на позиция i - 1) е обратна наклонена черта. Ако текущата кавичка не е escape-ната, функцията връща false, защото всички кавички трябва да бъдат обработени. Ако нито едно от горните условия не е изпълнено за някоя от кавичките в низа, функцията връща true, което означава, че всички кавички са обработени коректно.

Метод, прверяващ дали всички наклонени черти са escape-нати.

```
private boolean areAllBackslashesEscaped(String input) {
    input = input.replaceAll(BACKSLASH_ESCAPING_QUOTE, "");
    input =
input.replaceAll(BACKSLASH_ESCAPING_INPUT_SEPARATOR, "");

    for (int i = 0; i < input.length(); i++) {
      char c = input.charAt(i);
      if (c == '\\') {
        if (i == input.length() - 1) {
          return false;
        } else if (input.charAt(i + 1) != '\\') {
          return false;
        }
        i++;
      }
    }
  return true;
  }
```

Функцията areAllBackslashesEscaped проверява дали всички обратни наклонени черти (\\) в подадения текстов низ са escape-нати. Кодът извършва следните действия: Заменя двоен обратен наклонен слеш (\\) с празен низ ("") във входния низ input. Това се прави с цел премахване на обратния наклонен слеш, който е представен като екраниращ символ за кавичка (\") и символ за разделител (\,). Така се премахва потенциалното екраниране на кавичките и разделителите, за да може правилно да се провери екранирането на обратните наклонени черти. За всяка позиция i в низа input се извлича символът на тази позиция. Проверява се дали текущият символ с е обратна наклонена черта. Ако е, се извършват допълнителни проверки. Ако текущата обратна наклонена черта е последният символ в низа, функцията връща false, защото не може да има незавършено екраниране. Ако следващият символ след текущата обратна наклонена черта

(input.charAt(i + 1)) не е обратна наклонена черта, функцията връща false, защото трябва да има двойна обратна наклонена черта, за да се екранира символът. Ако горните условия не са изпълнени, се прескача следващата позиция i, тъй като текущата обратна наклонена черта е валидно екранирана със следващата. Ако нито едно от горните условия не е изпълнено за някоя от обратните наклонени черти в низа, функцията връща true, което означава, че всички обратни наклонени черти са правилно екранирани.

Функция за зареждане на данните от файл в паметта.

```
public void loadData(TableCell[][] data) {
   int maxRow = findMaxRowIndex(data);
   int maxCol = findMaxColIndex(data);
   TableCell[][] newData = createNewDataArray(maxRow, maxCol);
   copyNonNullValues(data, newData, maxRow, maxCol);
   table = newData;
   isTableOpened = true;
 }
```

Функцията loadData се използва за зареждане на данни в двумерен масив TableCell[][] - представяне на таблица. Кода извършва следните действия: Намира най-голямите индекси на редове и колони, съдържащи непразни (различни от null) елементи в подадения масив data. Това се прави чрез извикване на функциите findMaxRowIndex и findMaxColIndex, които връщат индекса на последния непразен ред и колона в масива. Създава нов двумерен масив newData с размери, определени от най-големите индекси на редове и колони. Това се прави чрез извикване на функцията createNewDataArray, която създава нов масив с подходящ размер, и копира непразните елементи от първоначалния масив data в новия масив newData. За копирането се използва функцията copyNonNullValues. Присвоява новия масив newData на полето table на обекта и задава флагът isTableOpened на true, указвайки, че таблицата е заредена успешно. Този код е отговорен за зареждането на данни във вътрешното представяне на таблицата и осигурява, че само непразните елементи се копират и запазват. Това помага за оптимизиране на използваната памет и обработката на таблицата.

Метод за скалиране *(scale out)* на таблицата.

```
public void scale(int newRowSize, int newColSize) {
    if (newRowSize < 0 || newColSize < 0) {
      throw new
IllegalArgumentException(String.format(INVALID_SIZE_ERROR_MESS
AGE, newRowSize, newColSize));
    }
```

```
    TableCell[][] newTable = new
TableCell[newRowSize][newColSize];

    for (int row = 0; row < newRowSize; row++) {
      for (int col = 0; col < newColSize; col++) {
        if (row < table.length && col < table[row].length) {
          newTable[row][col] = table[row][col];
        } else {
          newTable[row][col] = new TableCell();
        }
      }
    }

    table = newTable;
  }
```

Функцията scale се използва за промяна на размерите на таблицата на посочени нови размери. Кода извършва следните действия: Проверява се дали новите размери newRowSize и newColSize са невалидни (отрицателни). Ако един от тях или и двата са отрицателни, се хвърля изключение от тип IllegalArgumentException със съобщение за невалидни размери. Създава се нов двумерен масив newTable с размери, зададени от новите размери на таблицата. Обхожда се новият масив newTable с два вложени цикъла, като се проверява дали текущият индекс (row, col) е в границите на старата таблица. Ако е, стойността на съответния елемент в новата таблица се присвоява от съществуващия елемент в старата таблица (table[row][col]). Ако не е, се създава нов обект от класа TableCell и се присвоява на съответния елемент в новата таблица. Накрая, полето table се актуализира с новата таблица newTable. Този код позволява на таблицата да бъде мащабирана до нови размери, като запазва съществуващите данни, ако е възможно. В случай че новите размери са по-големи от текущите размери на таблицата, се добавят нови клетки, ако са по-малки - се изтриват клетки. Това позволява гъвкавост при манипулирането на размерите на таблицата и запазването на съществуващите данни.

Метод за свиване *(scale in)* на таблицата.

```
public void shrink() {

    int lastNonEmptyRow = findLastNonEmptyRow();
    int lastNonEmptyCol = findLastNonEmptyColumn();

    if (lastNonEmptyRow < 0) {
      return;
    }

    TableCell[][] newTable =
createNewDataArray(lastNonEmptyRow, lastNonEmptyCol);
```

```
    copyFilledCells(newTable, lastNonEmptyRow,
lastNonEmptyCol);

    table = newTable;
  }
```

Функцията shrink се използва за смаляване на размерите на таблицата, като премахва ненужните празни редове и колони от таблицата. Кода извършва следните действия: Намира последния непразен ред и последната непразна колона в текущата таблица, като извиква функциите findLastNonEmptyRow и findLastNonEmptyColumn съответно. Това се прави, за да се определи горната граница на новата по-малка таблица. Проверява се дали lastNonEmptyRow е по-малко от нула. Ако е, това означава, че таблицата е пълна с празни клетки и няма нужда от намаляване. В този случай се връща от функцията без да се извършват други действия. Създава се нов двумерен масив newTable с размери, определени от lastNonEmptyRow и lastNonEmptyCol. Копират се само запълнените клетки от текущата таблица в новата таблица, като се извиква функцията copyFilledCells. Това осигурява копирането само на необходимите данни и изключва празните клетки. Накрая, полето table се актуализира с новата, по-малка таблица newTable. Този код позволява на таблицата да бъде смалена до минимални размери, като се премахват ненужните празни редове и колони. Това води до оптимизация на паметта и се осигурява по-компактно съхранение на данните в таблицата.

Стартиране на програмата от клас *Engine*.

```
public void start() {

    printStartupHeader();

    try (Scanner scanner = new Scanner(System.in)) {
      while (true) {
        List<String> input = new
ArrayList<>(Arrays.asList(scanner.nextLine().trim().split(DO_N
OT_SPLIT_IF_ENCLOSED_IN_QUOTES_PATTERN)));
        String menuChoice = input.get(0);

        if (menuChoice.trim().isEmpty()) continue;

        Commands command;
        try {
          command =
Commands.valueOf(menuChoice.toUpperCase());
        } catch (IllegalArgumentException e) {
          System.out.printf(COMMAND_NOT_FOUND_MESSAGE + "%n",
menuChoice);
          continue;
        }
```

```
        commandExecutor.executeCommand(command,
input.subList(1, input.size()));
      }
    }
  }
```

Методът start в класа Engine представлява главната функция, която управлява стартирането на програмата. Кода извършва следните действия: Извежда заглавието на програмата чрез извикване на метода printStartupHeader. Това служи за представяне на стартирането на програмата на потребителя. Създава се инстанция на Scanner, свързана със стандартния вход (System.in), чрез конструкцията try-with-resources. Това гарантира, че ресурсът (в случая скенерът) ще бъде правилно затворен след приключване на изпълнението на блока. Итерира се в безкраен цикъл, като вътре се извършват следните действия: Чете се вход от потребителя с помощта на метода nextLine на скенера и се премахват водещите и завършващите интервали с trim. Входът се разделя на отделни аргументи с помощта на метода split и се съхранява в списъка input. Определя се първият елемент от списъка input като menuChoice. Ако menuChoice е празен (не съдържа символи), се продължава със следващата итерация на цикъла без изпълнение на останалите действия. Създава се променлива command, в която се опитва да се присвои стойността на една от константите в енумерацията Commands, като се използва методът valueOf. Ако възникне изключение IllegalArgumentException, това означава, че въведената команда не съответства на нито една от валидните команди в енумерацията. В този случай се извежда съобщение за грешка и се продължава със следващата итерация на цикъла. Изпълнява се командата чрез commandExecutor.executeCommand, като се предават аргументите, които са останалите елементи от списъка input. Цикълът продължава безкрайно, докато програмата не бъде спряна от потребителя или от друга част на кода.

Класът Engine има няколко ползи: Централизирана логика за управление: Класът Engine служи като централна точка за управление на основните операции и процеси в приложението. Той събира различни компоненти и модули на системата и координира техните взаимодействия. Това помага за поддържане на добра организация и структура на кода. Разделяне на отговорности: Чрез изолиране на логиката за управление в отделен клас като Engine, се постига разделяне на отговорностите между различните компоненти на системата. Това прави кода по-модулен и по-лесно поддържаем, тъй като всяка компонента се фокусира върху своите специфични функции. Инкапсулация на сложни операции: Класът Engine може да обвие сложни операции и процеси, като ги скрие от външния свят. Това позволява да се предоставят по-абстрактни и удобни интерфейси за използване на функционалността на системата, без да се излагат всички детайли на реализацията.

Това е начинът по който се стартира цялата програма.

```
public class Application {
```

```
  public static void main(String[] args) {
    Engine engine = new Engine();
    engine.start();
  }
}
```

Метод за извеждане на таблицата от програмата към външен файл.

```
public void write(TableCell[][] tableData) throws IOException
{

    for (TableCell[] tableDatum : tableData) {
      StringBuilder lineBuilder = new StringBuilder();
      for (int col = 0; col < tableDatum.length; col++) {
        TableCell cell = tableDatum[col];

        if (cell.getType() == CellType.STRING) {
          String value = cell.getValueAsString();
          value = value.replace(SINGLE_BACKSLASH,
DOUBLE_BACKSLASH);
          value = value.replace(NON_UNESCAPED_QUOTE,
ESCAPED_QUOTE);
          value = value.replace(CELLS_SEPARATOR,
ESCAPED_CELLS_OUTPUT_SEPARATOR);

lineBuilder.append(NON_UNESCAPED_QUOTE).append(value).append(N
ON_UNESCAPED_QUOTE);
        } else {
          lineBuilder.append(cell.getValueAsString());
        }
        if (col < tableDatum.length - 1) {
          lineBuilder.append(CELLS_SEPARATOR);
        }
      }

      outputWriter.write(lineBuilder.toString());
      outputWriter.newLine();
    }

    outputWriter.flush();
  }
```

Този код представлява метод, който записва данни от двумерен масив (TableCell[][]) в изходен поток (например файл). Итерира през всеки ред на двумерния масив. Създава StringBuilder, който ще съхранява текущия ред от данни за запис. Итерира през колоните на текущия ред и построява реда за запис, като проверява типа на всяка клетка и обработва стойността й. Добавя

разделител между клетките. Записва реда в изходния поток. Повтаря стъпките за всеки ред на двумерния масив. Извиква flush() на изходния поток, за да се увери, че всички буферирани данни са записани. Този метод е отговорен за преобразуването на масива от клетки във формат, който може да бъде записан във файл. Той обработва различни типове клетки (например стрингови и числови) и извършва съответните операции за форматиране и ескейпване на данните, за да гарантира коректен запис. След изпълнение на метода, данните от масива ще бъдат записани в изходния поток в съответния формат, готови за използване или запазване.

Затваряне на входно / изходен поток.

```
@Override
  public void close() throws IOException {
    outputWriter.close();
  }
```

Този код представлява имплементация на метода close() от интерфейса Closeable или AutoCloseable. В случая outputWriter е обектът за запис, който трябва да бъде затворен. Ето как функционира кодът: Методът close() бива извикан и неговата имплементация изпълнява затварянето на outputWriter. Затварянето на outputWriter се осъществява чрез извикване на метода close() на обекта. В случай че има отворени ресурси или буферирани данни в outputWriter, те ще бъдат изчистени и затворени. Този код е полезен, когато работим с ресурси, които изискват затваряне (например файлове), тъй като гарантира, че ресурсите се освобождават правилно след приключване на използването им. В този случай, outputWriter се затваря, което може да включва изчистване на буфери и освобождаване на свързани ресурси.

Regex шаблони.

Клас "Patterns" съдържа различни модели на регулярни изрази и низови константи. Тези шаблони и константи се използват за съпоставяне и манипулиране на низове, свързани със стойности на клетки и математически изрази.

Примери:

```
public static final String CELL_REFERENCE_PATTERN =
"[rR](\\d+)[cC](\\d+)";

public static final String
INTEGER_OR_FRACTIONAL_NUMBER_PATTERN = "^[+-
]?\\d+(\\.\\d+)?$";
```

Извеждане на съдържанието в табличен вид на козолата.

```
private String getTableData(TableRepository table, int
numRows, int numCols, int[] colWidths) {

    StringBuilder tableData = new StringBuilder();

    for (int i = 0; i < numRows; i++) {
      tableData.append("| ");
      for (int j = 0; j < numCols; j++) {
        TableCell cell = table.getCell(i, j);
        int colWidth = colWidths[j];
        if (cell.getType() != CellType.EMPTY) {
          String cellValue = cell.getValueAsString();
          if (cell.getType() == CellType.FORMULA) {
            try {
              cellValue =
String.valueOf(FormulaCalculator.getInstance().evaluate(cellVa
lue));
            } catch (ArithmeticException e) {
              cellValue = CELL_ERROR_VALUE;
            }
          }
          if (j == 0) {
            tableData.append(String.format("%-" + colWidth +
"s", cellValue));
          } else {
            tableData.append(String.format("%" + colWidth +
"s", cellValue));
          }
        } else {
          tableData.append(String.format("%" + colWidth + "s",
""));
        }
        tableData.append(" | ");
      }
      tableData.append(System.lineSeparator());
    }

    if (tableData.toString().trim().isEmpty()) {
      return NO_DATA_MESSAGE;
    }

    return tableData.toString().trim();
  }
```

Кодът представлява метод с име getTableData, който генерира представянето на данните от таблица във форматиран вид. Създава се обект от класа StringBuilder,

който ще съхранява резултатната таблица. Започва се итерация през редовете на таблицата в диапазона от 0 до numRows. За всеки ред, се добавя първата част на клетката, която представлява символа | за отделяне на клетките в таблицата. След това се извършва итерация през колоните на таблицата в диапазона от 0 до numCols. За всяка клетка, се извлича стойността на клетката чрез метода getValueAsString(). Ако типът на клетката не е празен (различен от CellType.EMPTY), се извършва допълнителна обработка на стойността на клетката. Ако типът на клетката е формула (CellType.FORMULA), стойността на клетката се преобразува като се изчисли формулата чрез FormulaCalculator и полученият резултат се използва вместо оригиналната стойност на клетката. Ако възникне ArithmeticException по време на изчислението на формулата, стойността на клетката се замества с CELL_ERROR_VALUE. Ако текущата колона е първата колона (j == 0), стойността на клетката се добавя към резултатната таблица с форматиране, което осигурява зададената ширина на колоната (colWidth) и ляво подравняване на стойността. В противен случай, стойността на клетката се добавя към резултатната таблица с форматиране, което осигурява зададената ширина на колоната (colWidth) и дясно подравняване на стойността. Ако типът на клетката е празен (CellType.EMPTY), се добавя празна стойност с форматиране, което осигурява зададената ширина на колоната (colWidth). След добавянето на стойността на клетката, се добавя и втората част на клетката, която е отново символа |, за да се завърши форматирането на клетката. След завършване на итерацията през колоните, се добавя нов ред в резултатната таблица чрез System.lineSeparator(). След приключване на итерацията през редовете, се проверява дали резултатната таблица е празна (не съдържа данни). Ако резултатната таблица е празна, се връща съобщение за няма данни (NO_DATA_MESSAGE). В противен случай, се връща резултатната таблица като низ, като преди това се премахват празните места от началото и края на таблицата чрез trim().

Рекурсивно изчисляване на формули.

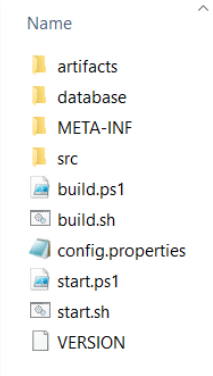
```
private double getCellValue(String cellRef) {
    String[] parts = cellRef.split("R|C");
    int row = Integer.parseInt(parts[1]) - 1;
    int col = Integer.parseInt(parts[2]) - 1;
    TableCell cell;
    try {
      cell = TableRepository.getInstance().getCell(row, col);
    } catch (IllegalArgumentException e) {
      return 0;
    }
    ...
    if (cell.getType() == CellType.FORMULA) {
      return this.evaluate(cell.getValueAsString());
    }

    return 0;
  }
```

В кода съществува рекурсия. Рекурсията се наблюдава в метода evaluate, където има извикване на самия себе си в ред, ако се срещне клетъчна референция, която е от тип формула (CellType.FORMULA). В този случай методът evaluate се извиква рекурсивно, за да се пресметне стойността на формулата, която се намира в съответната клетка. Това изпълнение на метода evaluate вътре в самия себе си се повтаря, докато не се достигне крайното условие или базовия случай.

```
public double evaluate(String formula) throws
ArithmeticException {

    List<String> tokens = tokenize(formula);
    Deque<Double> values = new ArrayDeque<>();
    Deque<Character> ops = new ArrayDeque<>();

    for (String token : tokens) {
      if (isCellReference(token)) {
        double cellValue = getCellValue(token);
        values.push(cellValue);
      } else if (isInteger(token) ||
isFractionalNumber(token)) {
        values.push(Double.parseDouble(token));
      } else if (isOperator(token)) {
        char op = token.charAt(0);
        while (!ops.isEmpty() && hasPrecedence(ops.peek(),
op)) {
          double b = values.pop();
          double a = values.pop();
          char prevOp = ops.pop();
          double result = applyOperation(prevOp, b, a);
          values.push(result);
        }
        ops.push(op);
      }
    }

    while (!ops.isEmpty()) {
      double b = values.pop();
      double a = values.pop();
      char op = ops.pop();
      double result = applyOperation(op, b, a);
      values.push(result);
    }

    return values.pop();
  }
```

Глава 5. Работа с програмата. Изисквания за изпълнение на програмата.

Проектът включва PowerShell и bash скриптове, които се използват за създаване и стартиране на програмата. При извършване на операция за създаване на програмата (build), новата версия на програмния файл (jar файл) се добавя в директорията "artifacts/". Стартирането на последно качената версия на програмата се извършва чрез изпълнение на скриптовете "start.sh" или "start.ps1". Има възможност и за конкретизиране на точно определена версия. За успешното изпълнение на програмата е необходимо наличието на Java Runtime Environment (JRE) версия 1.8 или по-нова. Скриптовете се изпълняват от основната директория на проекта. При стартиране на програмата се зареждат конфигурациите от файла "config.properties". Важни полета от този файл са "resources.directory" и "default.table.filename", които определят директорията, в която се съхраняват данните, и името на файла с данни по подразбиране.



Необходима файлова структура, за build и стартиране на програмата.

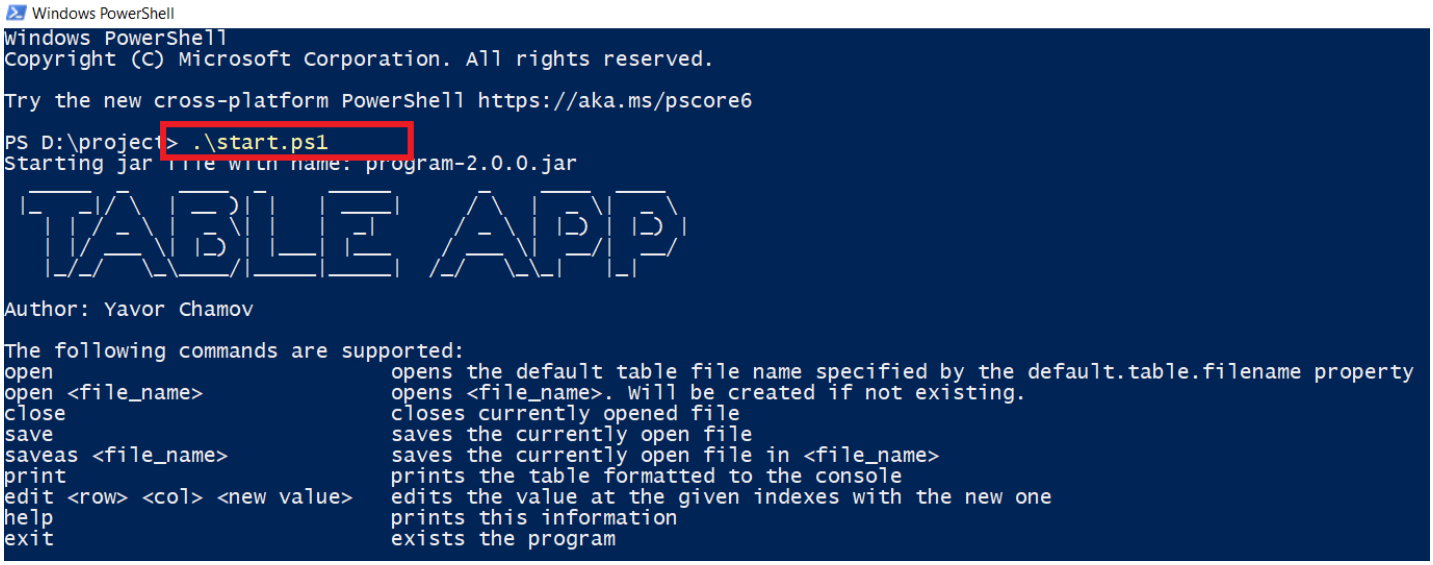
Примерно съдържание на config.properties

```
rows=10
cols=10
cells.separator=,
resources.directory=./database
default.table.filename=data.csv
cell.error.value=ERROR
```

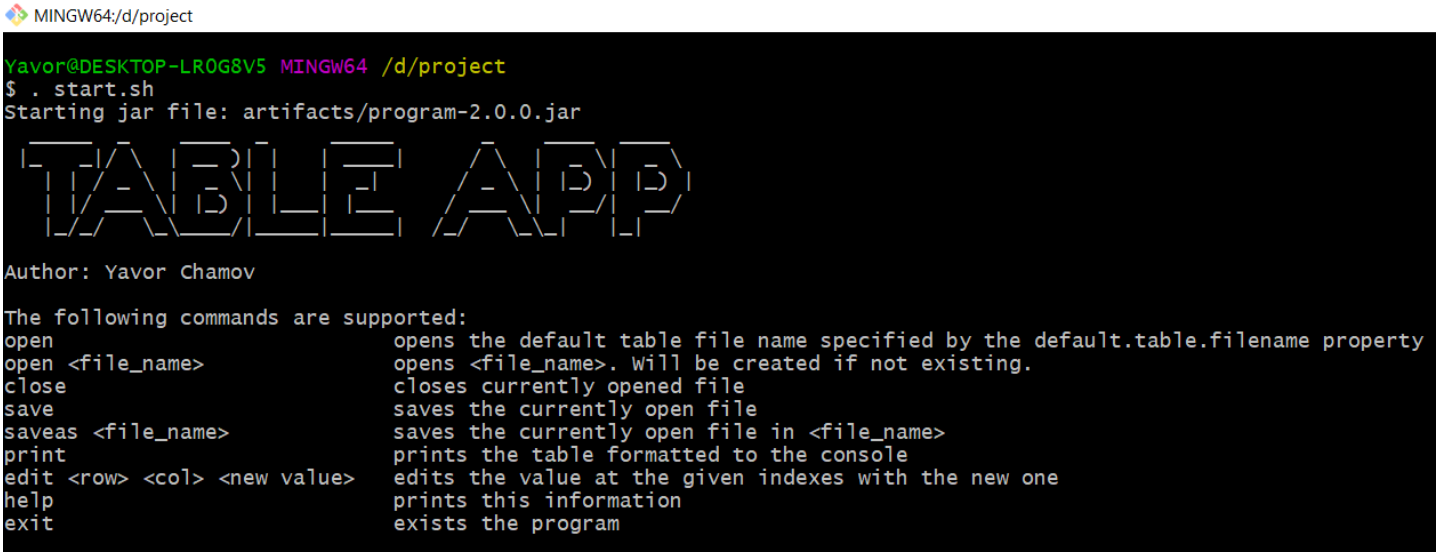
Bash скриптове за build-ване и стартиране на програмата.

```
#!/bin/bash

versionFile="VERSION"
version=$(cat "$versionFile")
jarName="program"
jarNameWithVersion="$jarName-$version"
artifacts="artifacts/"

if [[ ! -d "$artifacts" ]]; then
  mkdir -p "$artifacts"
fi

find . -type f -name "*.java" > classes.txt
javac -encoding UTF-8 -d bin @"classes.txt"
jar cmvf META-INF/MANIFEST.MF "$jarNameWithVersion.jar" -C
bin/ .
cp "$jarNameWithVersion.jar" "$artifacts"
rm "$jarNameWithVersion.jar"
rm -rf bin
rm classes.txt
```

```
#!/bin/bash

artifacts="artifacts"
pattern="program-[0-9]+\.[0-9]+\.[0-9]+\.jar"

if [[ $# -eq 0 ]]; then
  files=($(find "$artifacts" -name "*.jar" -type f -regex
".*/$pattern" | sort -V))
  latestFile=${files[-1]}
else
  version="$1"
  files=($(find "$artifacts" -name "*.jar" -type f -regex
".*/program-$version\.jar"))
  latestFile=${files[0]}
fi

if [[ -z $latestFile ]]; then
  echo "No matching jar file found."
else
  echo "Starting jar file: $latestFile"
  java -jar "$latestFile"
fi
```



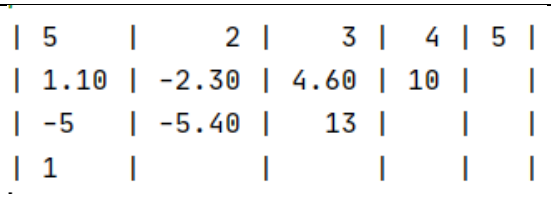
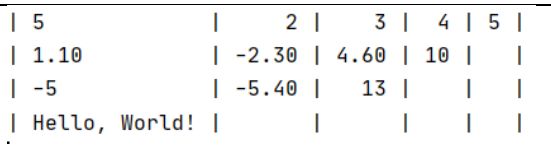
Windows OS



Unix-based OS

Глава 6. Тестови сценарии.

CSV файл	CLI команда	Изход	Коменар
1, 2, 3, 4, 5 1.1, -2.3, 4.6 -5, -5.4, 13 1	open	Table opened successfully.	Отваряне на таблица с валидни целочислени и дробни данни. Положителни и отрицаелни. Отваря data.csv файла по default.
	print	| 1 | 2 | 3 | 4 | 5 | | 1.10 | -2.30 | 4.60 | | | | -5 | -5.40 | 13 | | | | 1 | | | | |	Отпечатване на таблицата на екрана. Стойностите в първата колона са подравнени в ляво, а в останалите вдясно.
	edit 1 1 5	Cell (1, 1) updated from "1" to "5"	Промяна на елемент в ред 1 и колона 1 с нова стойност 5
	print	| 5 | 2 | 3 | 4 | 5 | | 1.10 | -2.30 | 4.60 | | | | -5 | -5.40 | 13 | | | | 1 | | | | |	Отпечатване на таблицата след

			команда edit.
	edit 2 4 10	Cell (2, 4) updated from "" to "10"	Добавяне на елемент на някоя от празните позиции
	print		Отпечатване на таблицата след команда edit.
	edit 4 1 “Hello, World!”	Cell (4, 1) updated from "1" to "Hello, World!"	Промяна на типа на данните от целочислен към низ.
	print		Успешно промених ме типа на данните в клетката.
	edit 1 1 ““Tu-Varna”“	Error editing the cell: "Tu-Varna" has unescaped quotes.	Проверка за escape-нати кавички. Вътрешните кавички не са escape-нати и получавам е съобщение за грешка.

	edit 1 1 “\”Tu-Varna\””	Cell (1, 1) updated from "5" to ""Tu-Varna""	Правилно escape-нати кавички.
	print		Успешно обновихме елемента на ред 1 и колона 1.
	edit 1 2 “\Test\”	Error editing the cell: \Test\ has unescaped backslash.	Проверка за escape-нати наклонени черти \
	edit 1 2 “\\Test\\”	Cell (1, 2) updated from "2" to "\Test\"	Правилно escape-нати наклонени черти.
	edit 4 2 =5+2*3	Cell (4, 2) updated from "" to "=5+2*3"	Прилагане на формула с директно зададени стойности без да се реферират клетки. Операциите се прилагат в правилен ред.
	print		
	edit 4 2 =42/0	Cell (4, 2) updated from "=5+2*3" to "=42/0"	Добавяне на формула с

			делене на 0.

	print	`| "Tu-Varna" | \Test\ | 3 | 4 | 5 |`
		`| 1.10 | -2.30 | 4.60 | 10 | |`
		`| -5 | -5.40 | 13 | | |`
		`| Hello, World! | ERROR | | | |`
			Визуализиране на ERROR заради делението на 0.

	edit 4 3 =R2C2+R3C3	Cell (4, 3) updated from "" to "=R2C2+R3C3"	Прилагане на формула с реферирането на клетки с дробни числа.

	print	`| "Tu-Varna" | \Test\ | 3 | 4 | 5 |`
		`| 1.10 | -2.30 | 4.60 | 10 | |`
		`| -5 | -5.40 | 13 | | |`
		`| Hello, World! | ERROR | 10.7 | | |`
			Успешно изпълнена формула с реферирани клетки.

	edit 4 4 =R4C3*2	Cell (4, 4) updated from "" to "=R4C3*2"	Формула с реферирането към клетка, която също съдържа формула.

	print	`| "Tu-Varna" | \Test\ | 3 | 4 | 5 |`
		`| 1.10 | -2.30 | 4.60 | 10 | |`
		`| -5 | -5.40 | 13 | | |`
		`| Hello, World! | ERROR | 10.70 | 21.4 | |`
			Успешно приложена **рекурсия**.

	edit 4 5 =R4C4*R1C1	Cell (4, 5) updated from "" to "=R4C4*R1C1"	Умножаваме числена стойност с низ, който не може да се

			превърне в число.
	print	| "Tu-Varna" | \Test\ | 3 | 4 | 5 | | 1.10 | -2.30 | 4.60 | 10 | | | -5 | -5.40 | 13 | | | | Hello, World! | ERROR | 10.70 | 21.4 | 0.0 |	Получаваме 0, тъй като низът се представя като 0 по условие.
	edit 3 5 “2”	Cell (3, 5) updated from "" to "2"	Добавяме числова стойност под формата на низ.
	edit 2 5 =R3C1/R3C5	Cell (2, 5) updated from "" to "=R3C1/R3C5"	Умножаваме две числови стойности, като едната от тях е под формата на низ.
	print	| "Tu-Varna" | \Test\ | 3 | 4 | 5 | 1.10 | -2.30 | 4.60 | 10 | -2.5 | -5 | -5.40 | 13 | | 2 | Hello, World! | ERROR | 10.70 | 21.4 | 0.0	Получаваме правилен резултат в следствие на cast.
	edit 8 8 “Scale”	Cell (8, 8) updated from "" to "Scale"	Динамично скалиране на таблицата с допълнителни редове и колони.

<table>
<tr><td></td><td>print</td><td>| "Tu-Varna" | \Test\ | 3 | 4 | 5 | | | |
1.10	-2.30	4.60	10	-2.5			
-5	-5.40	13		2			
Hello, World!	ERROR	10.70	21.4	0.0			
							Scale |</td><td>Таблицата успешно скалира, за да приеме новата стойност.</td></tr>
<tr><td></td><td>edit 8 8 “”</td><td>Cell (8, 8) updated from "Scale" to ""</td><td>Динамично намляване на таблицата до последният ред и колона, в която има стойност.</td></tr>
<tr><td></td><td>print</td><td>	"Tu-Varna"	\Test\	3	4	5
1.10	-2.30	4.60	10	-2.5
-5	-5.40	13		2
Hello, World!	ERROR	10.70	21.4	0.0	</td><td>Таблицата е успешно смалена и паметта освободена</td></tr>
<tr><td>"Hello\, World!", "Numbers from 1 to 5: 1\,2\,3\,4\,5 "</td><td>print</td><td>| Hello, World! | Numbers from 1 to 5: 1,2,3,4,5 |</td><td>Успешно escape-ваме запетаите, които участват в низовете.</td></tr>
<tr><td>-</td><td>help</td><td>The following commands are supported:
open <file> opens <file>
close closes currently opened file
save saves the currently open file
saveas <file> saves the currently open file in <file>
print prints the table formatted to the console
edit <row> <col> <new value> edits the value at the given indexes with the new one
help prints this information
exit exists the program</td><td>Output от команда help.</td></tr>
<tr><td>-</td><td>print</td><td>print
No table is currently opened.</td><td>Output при липсваща отворена таблица.</td></tr>
</table>

-	save	`Table saved to file "src\bg\tu_varna\sit\b1\f21621577\resources\data.csv"`	Output при успешна save операция.
-	saveas	`saveas data2.csv` `Table saved to file "src\bg\tu_varna\sit\b1\f21621577\resources\data2.csv"`	Output при успешна saveas опеарация.

Глава 7. Заключение.

Създадох приложение "Приложение за работа с електронни таблици", което има за цел да предостави функционалност за отваряне, редактиране и запис на данни в електронни таблици. Освен основните операции като отваряне, затваряне, запис и помощ, приложението поддържа допълнителни операции. Това включва извеждане на съдържанието на таблицата на екрана, редактиране на отделни клетки чрез въвеждане на ново съдържание и работа с формули. Показах архитектурата, най-важните моменти, алгоритми и стандарти в имплементацията. За финал приложението може да се *build-не* и стартира с помощта на скриптове.

Насоки за бъдещо развитие и усъвършенстване:

1. Разширяване на функционалността
    a. добавяне на допълнителни функции, които ще подобрят работата и удобството на потребителите. (сортиране на данните, филтриране, изчисления с формули със скоби, форматиране на клетките и други.)

2. Интерактивен потребителски интерфейс (GUI)

3. Доразвиване на механизмите за валидация на въведените данни, за да предотвратим неправилни операции и грешки при обработката на формули и типовете данни.

4. Ясни съобщения за грешки, които да помогнат на потребителите да открият и отстранят проблемите.

5. Поддръжка на повече файлови формати - възможностите за импорт и експорт на данни в различни файлови формати, като например Excel и други.

Източници:

Reverse Polish Notation:

Wikipedia: https://en.wikipedia.org/wiki/Polish_notation, https://en.wikipedia.org/wiki/Reverse_Polish_notation

Geeksforgeeks: https://www.geeksforgeeks.org/evaluate-the-value-of-an-arithmetic-expression-in-reverse-polish-notation-in-java/

Regex:

Regex for splitting a string using space when not surrounded by single or double quotes - https://stackoverflow.com/questions/366202/regex-for-splitting-a-string-using-space-when-not-surrounded-by-single-or-double

Семантично версиониране 2.0.0

https://semver.org/lang/bg/