Картина, която съдържа текст, графична колекция

Описанието е генерирано автоматичноТЕХНИЧЕСКИ УНИВЕРСИТЕТ – ВАРНА

Факултет по изчислителна техника и автоматизация

Катедра „Софтуерни и интернет технологии“

**ДОКУМЕНТАЦИЯ ЗА ПРОЕКТ ПО ОБЕКТНО-ОРИЕНТИРАНО ПРОГРАМИРАНЕ ПЪРВА ЧАСТ**

на тема: „Растерна графика”

|  |  |
| --- | --- |
| Изготвил: Петър Борисов Сталев | Проверил: |
| Специалност: СИТ |  |
| Група: 1а |  |
| Факултетен номер: 23621693 |  |

Структура на документацията

[Глава 1. Увод 3](#_Toc198760824)

[1.1 Описание и идея на проекта 3](#_Toc198760825)

[1.2 Цел и задачи на разработката 3](#_Toc198760826)

[Глава 2. Преглед на предметната област 3](#_Toc198760827)

[2.1 Основни концепции и алгоритми, които ще бъдат използвани 3](#_Toc198760828)

[2.2 Подходи, методи за решаване на поставените задачи 5](#_Toc198760829)

[2.3 Функционални изисквания (потребителски изисквания, права, роли, статуси, диаграми и качествени (нефункционални) изисквания) 6](#_Toc198760830)

[Глава 3. Проектиране 7](#_Toc198760831)

[3.1 Обща структура на проекта 7](#_Toc198760832)

[3.2 Диаграми/Блок схеми 8](#_Toc198760833)

[Глава 4. Реализация 23](#_Toc198760834)

[4.1. Реализация на класове 23](#_Toc198760835)

[4.2 Алгоритми и оптимизации. 31](#_Toc198760836)

[Глава 5. Тестване 35](#_Toc198760837)

[5.1. Планиране, описание и създаване на тестови сценарии (създаване на примери за всички възможни входове на командите) 35](#_Toc198760838)

[Глава 6. Заключение 35](#_Toc198760839)

[6.1. Обобщение на изпълнението на началните цели 35](#_Toc198760840)

# Глава 1. Увод

## 1.1 Описание и идея на проекта

Проектът представлява софтуерна система за обработка на растерни изображения, реализирана като конзолно приложение с интерфейс базиран на команден ред. Приложението предоставя възможност за работа с три основни графични формата: PPM (цветни изображения), PGM (изображения в градации на сивото) и PBM (черно-бели бинарни изображения). Системата поддържа множество потребителски сесии, като всяка сесия може да съдържа различни набор от изображения и приложени графични трансформации. Функционалностите включват зареждане и запазване на файлове, прилагане на различни графични филтри (градации на сивото, негатив, монохром), геометрични трансформации (завъртане) и създаване на колажи от множество изображения.

Проектът следва строго принципите на обектно-ориентираното програмиране, като прилага абстракция за различните типове изображения, капсулация на логиката за обработка на файлове и графични трансформации, и полиморфизъм за еднообразно прилагане на операциите върху различните формати. Архитектурата на системата е проектирана да бъде модулна, скалируема и лесно разширяема, позволявайки добавяне на нови файлови формати и графични операции без промяна на съществуващата функционалност. Чрез ясно дефинирани интерфейси и добре структуриран код, приложението осигурява стабилна и надеждна основа за последващо развитие и поддръжка.

## 1.2 Цел и задачи на разработката

Основната цел на разработката е да се демонстрира как чрез съвременни обектно-ориентирани практики може да се създаде ефективна и гъвкава система за обработка на графични изображения. Проектът има за цел да покаже как различни компоненти - файлове, трансформации, сесии и команди - могат да взаимодействат помежду си по ясен и структуриран начин, като същевременно се поддържа висока степен на модулност и възможност за разширяване.

Задачите на разработката включват създаване на интуитивен команден интерфейс, реализация на поддръжка за трите графични формата с техните специфики, разработка на система за управление на множество сесии, имплементация на графични трансформации с поддръжка за обратни операции (undo), и създаване на механизми за валидация и обработка на грешки. Допълнителна задача е реализацията на функционалност за създаване на колажи, която демонстрира способността на системата да комбинира и обработва множество изображения едновременно. Системата трябва да бъде проектирана така, че да осигурява висока производителност, коректна работа с големи файлове и лесно разширяване с нови функционалности в бъдеще.

# Глава 2. Преглед на предметната област

## 2.1 Основни концепции и алгоритми, които ще бъдат използвани

Проектът използва съвременни обектно-ориентирани подходи и дизайн патерни за реализация на функционалностите за обработка на растерни изображения. Основните концепции включват:

* **Command Pattern** за имплементация на потребителските команди, като всяка операция (load, save, grayscale, rotate и др.) се реализира като отделен клас, имплементиращ общ интерфейс. Това позволява лесно добавяне на нови команди и поддръжка на undo/redo функционалност.
* **Factory Pattern** за създаване на обекти от различни типове изображения (PPM, PGM, PBM), което осигурява гъвкавост и лесно разширяване с нови файлови формати в бъдеще.
* **Session Management** система, която поддържа множество работни сесии с различни набор от изображения и трансформации. Всяка сесия има уникален идентификатор и поддържа собствен списък с pending операции.
* **Image Processing Algorithms** включващи:
  + Grayscale преобразуване с формулата: 0.3×R + 0.59×G + 0.11×B
  + Monochrome (бинаризация) с прагово стойност
  + Negative (инверсия на цветовите стойности)
  + Rotation на 90 градуса (ляво/дясно) чрез транспониране на матрицата от пиксели
  + Collage операции (хоризонтално и вертикало съединяване на изображения)

## 2.2 Подходи, методи за решаване на поставените задачи

За реализацията на проекта е избран строго обектно-ориентиран подход, базиран на SOLID принципите. Системата е разделена на няколко основни модула:

**Модулна архитектура** включваща:

* **Image Handling** - абстракция за работа с различни графични формати
* **Command Processing** - интерпретиране и изпълнение на потребителски команди
* **Session Management** - управление на множество работни сесии
* **Transformation Engine** - двигател за графични трансформации

**Дизайн решения**:

* **Abstraction** чрез Image интерфейс, дефиниращ общи операции за всички формати
* **Encapsulation** на файловия parsing и serialization логика във всеки конкретен image клас
* **Polymorphism** за еднообразно прилагане на трансформации върху различни формати
* **Separation of Concerns** чрез ясно разделяне на отговорностите между компонентите

**Файлова обработка** използва Java NIO API за ефективно четене и запис на големи файлове, с поддръжка за стандартните PPM/PGM/PBM файлови формати. Валидацията на входните данни включва проверки за коректност на magic numbers, размери и цветови стойности.

## 2.3 Функционални и качествени изисквания

### Функционални изисквания

Приложението предоставя пълна функционалност за работа с растерни изображения чрез конзолен интерфейс:

* **Управление на сесии**:
  + Създаване на нови сесии с load команда
  + Превключване между сесии със switch
  + Затваряне на сесии с close
  + Преглед на информация за сесията с sessioninfo
* **Операции с изображения**:
  + Зареждане на единични или множествени файлове
  + Добавяне на нови изображения към текуща сесия с add
  + Запазване на всички изображения с save
  + Запазване под ново име с saveas
* **Графични трансформации**:
  + Grayscale преобразуване на цветни изображения
  + Monochrome (бинаризация) преобразуване
  + Negative (цветова инверсия)
  + Rotation на 90 градуса (ляво/дясно)
  + Създаване на колажи от две изображения
* **История на операции**:
  + Undo на последната трансформация
  + Автоматично прилагане на всички pending трансформации при save

### Качествени (нефункционални) изисквания

* **Производителност**: Оптимизирана обработка на големи изображения чрез ефективни алгоритми и памет management
* **Надеждност**: Строга валидация на входните данни и обработка на грешки с ясни съобщения за потребителя
* **Разширяемост**: Модулна архитектура, позволяваща лесно добавяне на нови файлови формати и трансформации
* **Преносимост**: Pure Java решение без външни зависимости, работещо на всяка JVM
* **Удобство на потребителя**: Интуитивен команден интерфейс с подробни съобщения за грешки и help функционалност
* **Съвместимост**: Пълна поддръжка на стандартните Netpbm формати (PPM, PGM, PBM)

## Глава 3. Проектиране

### Обща структура на проекта

**Проектът е организиран в следните основни пакети, всеки със специфична роля в системата:**

### ****• src.command.impl****

**Съдържа всички команди, които потребителят може да изпълнява чрез CLI. Всяка команда имплементира интерфейса Command и отговаря за отделна операция като load, save, grayscale, rotate, undo, и др. Използван е Command Design Pattern за по-лесно управление и добавяне на нови команди.**

### ****• src.exception****

**Тук се намират всички потребителски изключения (exceptions), дефинирани за нуждите на логиката – EditorException, ImageFormatException, SessionException, TransformationException. Те позволяват по-точно обработване на специфични проблеми в логиката и комуникация с потребителя чрез ясни съобщения за грешки.**

### ****• src.session****

**Съдържа класовете за управление на работните сесии. SessionManager отговаря за създаване, превключване и затваряне на сесии, докато Session представлява конкретна работна среда с изображения и pending трансформации.**

### ****• src.image.impl****

**Съдържа основната абстракция за изображенията и конкретните имплементации:**

* **Image интерфейс - дефинира общите операции за всички формати**
* **PPMImage, PGMImage, PBMImage - конкретни имплементации за всеки формат**
* **ImageLoader - отговаря за зареждането на изображения от файлове**

### ****• src.image.transformation****

**Съдържа алгоритмите за графични трансформации. Всеки трансформатор имплементира специфични операции като grayscale, monochrome, negative, rotation, working директно с пикселните данни.**

### ****• Главният пакет (без поддиректория)****

**Съдържа основните класове, отговарящи за координацията на системата:**

* **Main - стартовата точка на програмата**
* **CommandLineInterface - управлява потребителския интерфейс и жизнения цикъл**
* **CommandFactory - създава конкретни команди въз основа на потребителски вход**
* **ImageFactory - фабрика за създаване на image обекти от различни формати**

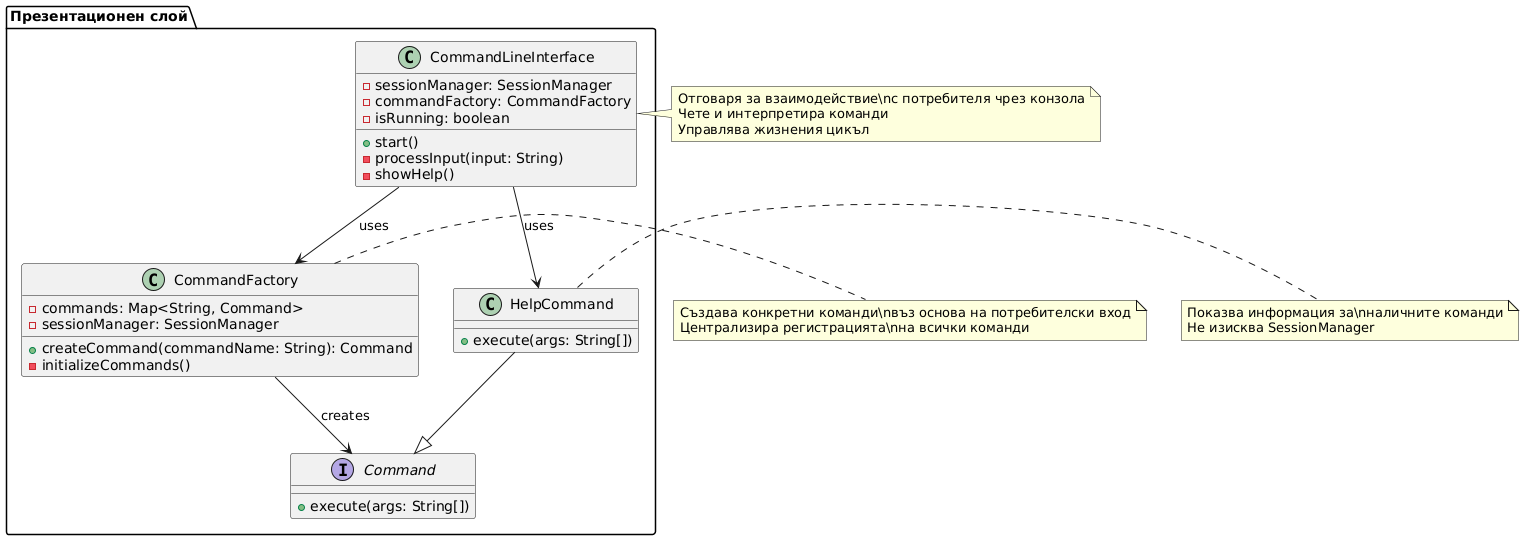
### ****• Допълнителни помощни класове****

* **FileValidator - валидира файлове и пътища**
* **ColorCalculator - помощни методи за цветови изчисления**
* **DimensionUtils - помощни функции за работа с размери и координати**

**Тази структура позволява ясно разделение на отговорностите и лесно разширяване на функционалността чрез добавяне на нови команди или трансформации.**

### Диаграми/Блок схеми

**Като цяло програмата може да се раздели на шест главни функционални слоя:**

****

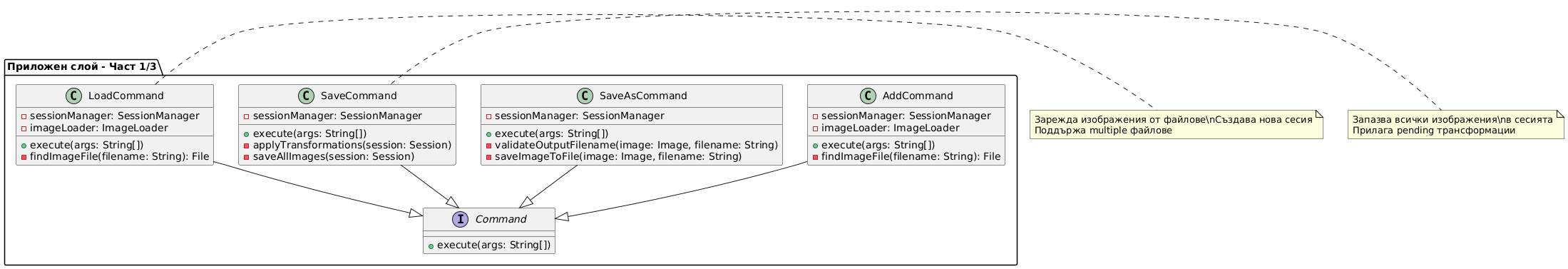
**Първият слой е командният интерфейс.** Той функционира като посредник между потребителя и вътрешната логика на приложението. Въпреки големия брой команди, архитектурата не се усложнява благодарение на приложения **Command шаблон**. Този подход позволява ясно разделение между потребителския интерфейс (CommandLineInterface) и бизнес логиката (SessionManager, Image обработка), което се вижда от липсата на директни връзки между тях. Command шаблонът инкапсулира всяка операция като отделен обект, което прави системата лесно разширяема с нови команди без промяна на съществуващата логика.

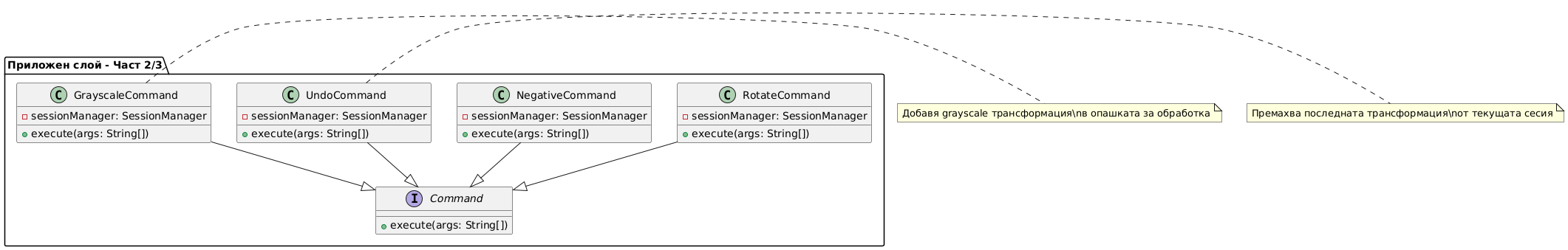
**Вторият слой представлява** **сърцето на бизнес логиката** на приложението. Той съдържа цялата функционалност, която потребителят може да изпълнява чрез команди. Слойът е изграден на основата на **Command Pattern**, като всяка команда представлява отделен клас, имплементиращ общ интерфейс.

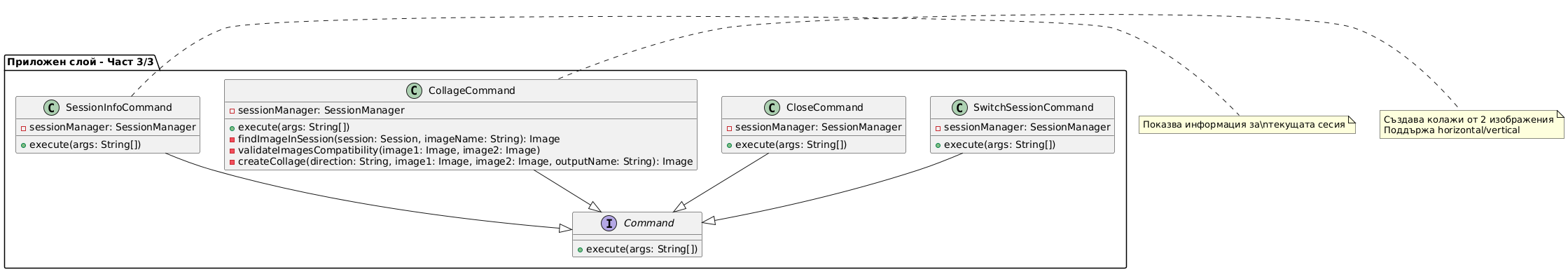
**Основни характеристики:**

* **Команди за файлови операции** (Load, Save, SaveAs, Add) - управление на жизнения цикъл на изображенията
* **Команди за трансформации** (Grayscale, Negative, Rotate) - графични операции върху пикселите
* **Undo функционалност** - възможност за отмяна на последна операция
* **Управление на сесии** (Close, SessionInfo, Switch) - работа с множество работни среди
* **Разширени операции** (Collage) - създаване на сложни композиции

Фигура 1 - клас диаграма шаблона на командите част 1







**Третият** съдържа **основните бизнес модели и правила** на приложението. Тук се намират:

* **SessionManager** - централен мениджър за всички работни сесии
* **Session** - представяне на работна среда с изображения и трансформации
* **Image интерфейс** - абстракция за всички типове изображения

## ****Ключови характеристики****:

### ****SessionManager****:

* Управлява **множество сесии** с уникални идентификатори
* Поддържа **активна сесия** за текуща работа
* Осигурява **превключване** между различни сесии
* **Затваря** сесии и освобождава ресурси

### ****Session****:

* Съдържа **списък с изображения** в текущата работна среда
* Поддържа **опашка от трансформации** за прилагане
* Има **уникален ID** за идентификация
* Предоставя **методи за управление** на съдържанието

### ****Image интерфейс****:

* Дефинира **общи операции** за всички формати
* Позволява **полиморфизъм** - работа с различни типове через общ интерфейс
* **Абстрахира** спецификите на отделните формати

## ****Важност на домейнния слой****:

Този слой е **фундаментът** на приложението - той дефинира какви данни съществуват и как те могат да бъдат манипулирани. Чрез ясно дефинирани модели и правила, домейнният слой осигурява **стабилност** и **консистентност** на данните във всички компоненти на системата.

Четвъртият слой отговаря за **работата с файловата система** и **имплементацията на конкретните формати**. Тук се намират:

* **ImageLoader** - основен клас за зареждане на изображения от файлове
* **PPMImage** - имплементация за цветни RGB изображения
* **PGMImage** - имплементация за grayscale изображения
* **PBMImage** - имплементация за черно-бели бинарни изображения

## ****Ключови характеристики****:

### ****ImageLoader****:

* **Определя формата** на файла по magic number (P1, P2, P3)
* **Валидира** съществуване и достъп до файлове
* **Създава** подходящия Image обект според формата
* **Хвърля** exceptions при невалидни файлове

### ****PPMImage**** (цветни изображения):

* Пиксели: int[height][width][3] (R, G, B канали)
* Поддържа **max цветова стойност** (обикновено 255)
* **Сложно парсване** - 3 стойности на пиксел

### ****PGMImage**** (grayscale):

* Пиксели: int[height][width] (единична стойност)
* Поддържа **max стойност** за grayscale
* **По-опростено парсване** - 1 стойност на пиксел

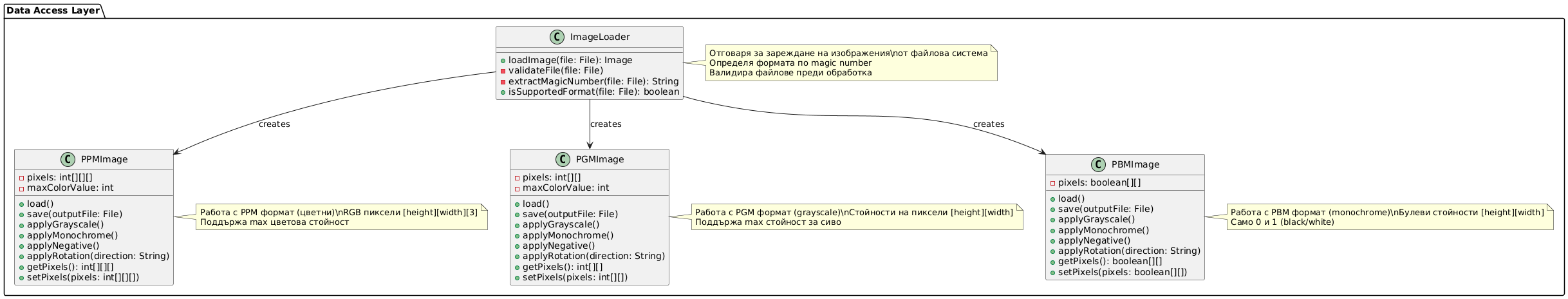
### ****PBMImage**** (monochrome):

* Пиксели: boolean[height][width] (true/false)
* **Най-прост формат** - само 0 и 1
* **Бинарни операции** - черно/бяло без нюанси

## ****Важност на Data Access Layer****:

Този слой **абстрахира сложността** на файловите формати от останалите части на приложението. Чрез ясно дефинирани интерфейси и имплементации, той позволява на горните слоеве да работят с изображения, без да знаят спецификите на отделните формати.

**Изолира грешките** при файлова обработка и гарантира, че само валидни и коректно заредени изображения достигат до бизнес логиката!



**Петият слой е сърцето на графичните трансформации в приложението.** Този слой съдържа всички алгоритми и логика за манипулация на пикселните данни на изображенията.

## ****Основни компоненти:****

### ****1. Графични трансформации:****

* **Grayscale преобразуване** - преобразува цветни изображения в нюанси на сивото
* **Monochrome (бинаризация)** - преобразува в черно-бяло чрез прагове
* **Negative (инверсия)** - обръща цветовите стойности
* **Rotation** - завъртане на 90 градуса (ляво/дясно)

### ****2. Колаж операции:****

* **Хоризонтално съединяване** - комбинира две изображения едно до друго
* **Вертикално съединяване** - комбинира две изображения едно под друго

### ****3. Валидации:****

* **Съвместимост на формати** - проверка за еднакъв тип изображения
* **Съвместимост на размери** - проверка за еднакви width/height
* **Цветови диапазони** - валидация на стойностите на пикселите

## ****Ключови характеристики:****

### ****Алгоритмична сложност:****

* **Grayscale формула**: 0.3×R + 0.59×G + 0.11×B
* **Monochrome прагове**: автоматично изчисление базирано на maxColorValue
* **Matrix трансформации** за ротация
* **Пикселно копиране** за колажи

### ****Оптимизации:****

* **Бързи операции** с директен достъп до пиксели
* **Минимизиране на паметта** при големи изображения
* **Ефективно копиране** на блокове от пиксели

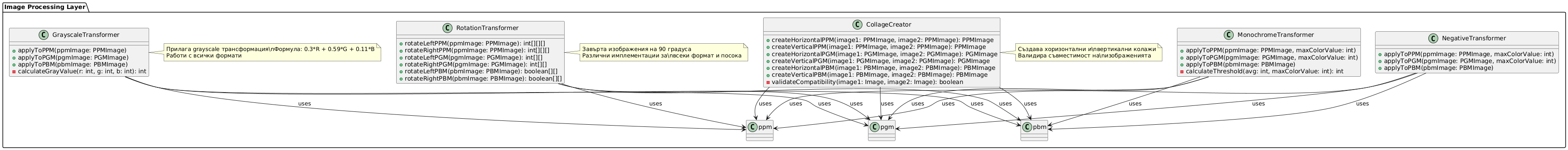
### ****Обработка на грешки:****

* **Валидация на входните данни**
* **Обработка на out-of-range стойности**
* **Гаранция за коректни резултати**

## ****Защо е важен този слой:****

1. **Съдържа най-сложната бизнес логика** в приложението
2. **Директно манипулира пикселните данни**
3. **Гарантира коректност** на графичните операции
4. **Оптимизиран за производителност** с големи файлове
5. **Изолира сложната математика** от останалите слоеве

Този слой е **критично важен** за качеството на резултатите и производителността на приложението!



Шестият слой съдържа **спомагателни функции и инструменти**, които се използват от múltiple слоеве в приложението. Тук се намират:

* **ImageFactory** - фабрика за създаване на image обекти
* **FileValidator** - валидация на файлове и данни
* **ColorCalculator** - математически изчисления за цветове
* **DimensionUtils** - помощни функции за размери и координати
* **TransformationValidator** - валидация на трансформации

## ****Ключови характеристики****:

### ****ImageFactory****:

* **Създава image обекти** от файлове
* **Определя формата** по файлово разширение
* **Генерира празни изображения** с зададени размери
* **Централизира създаването** на обекти

### ****FileValidator****:

* **Проверява съществуване** на файлове
* **Валидира разширения** (.ppm, .pgm, .pbm)
* **Проверява коректност** на размери и цветови стойности
* **Гарантира валидни данни** преди обработка

### ****ColorCalculator****:

* **Изчислява grayscale стойности** по стандартната формула
* **Определя прагове** за monochrome преобразуване
* **Пресмята negative стойности**
* **Нормализира цветови стойности** в допустими диапазони

### ****DimensionUtils****:

* **Изчислява нови размери** след трансформации
* **Проверява съвместимост** на изображения за колажи
* **Пресмята общ брой пиксели**
* **Управлява геометрични трансформации**

### ****TransformationValidator****:

* **Проверява приложимост** на трансформации
* **Определя дали изображение вече** има дадено свойство
* **Валидира възможност** за ротация
* **Предотвратява излишни операции**

## ****Важност на Utilities Layer****:

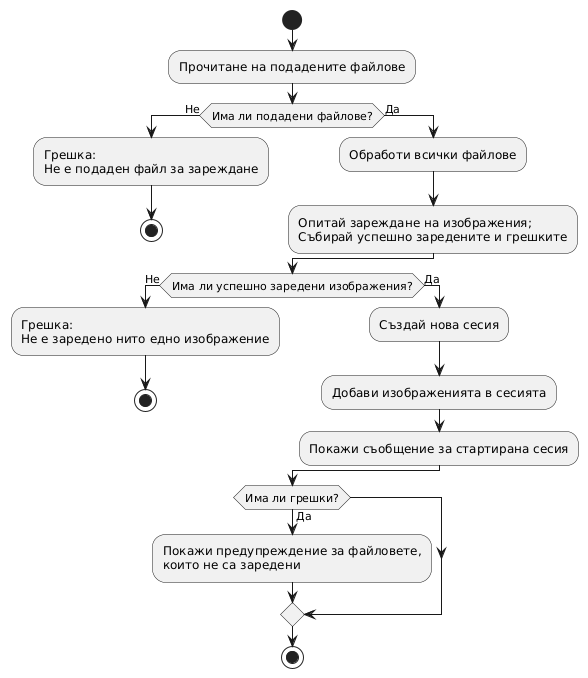
Този слой предоставя **преизползваем код и инструменти** за всички останали слоеве. Чрез централизиране на помощните функции, той:

* **Намалява code duplication**
* **Гарантира консистентност** във валидациите
* **Осигурява точни изчисления**
* **Опростява поддръжката** на сложната логика
* **Подобрява качеството** на кода

**Utilities Layer е glue-ът, който държи приложението заедно!**

**Следващата част от документацията на проекта е свързана с проектирането на дадените команди от заданието.**

load <image1\_name>,<image2\_name>...

Методът execute(String[] args) има за цел да добави нова/и снимка/снимки към текущата сесия. Той приема като аргумент име на снимката/и и извършва проверка дали съществува такава снимка в директорията, дали е правилното разширение. Ако снимката не съществува или е грешен формат, се извежда съобщение за грешка. В противен случай се създава нова сесия , към който се добавя/т снимките в List<Image> images -> Session class. Накрая се потвърждава успешно добавяне чрез подходящо съобщение към потребителя. Следното се вижда на Error: Reference source not found.

save командата прилага всички трансформации направени върху снимките в сесията, като преди това трябва да провери дали сесията е активна .

**Проверка за активна сесия** - Ако няма активна сесия, командата прекратява с грешка

**Проверка за изображения** - Ако сесията е празна (няма изображения), командата прекратява

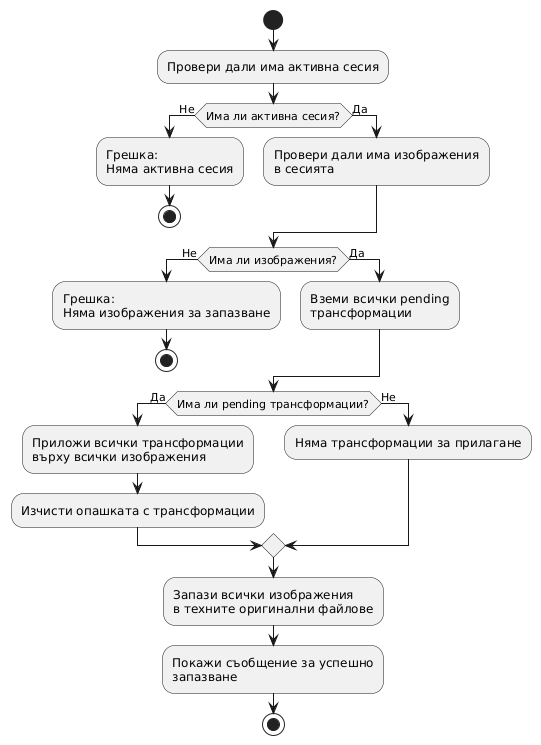
**Прилагане на трансформации** - Ако има pending трансформации, те се прилагат върху всички изображения и се изчистват

**Запазване на файлове** - Всички изображения се записват в техните оригинални файлови пътища

**Съобщение за успех** - Потребителят получава feedback за успешното запазване

## ****Ключови характеристики:****

* **Автоматично прилага** всички чакащи трансформации
* **Запазва всички изображения** в сесията
* **Изчиства трансформациите** след успешно запазване
* **Валидира** предварително условията за изпълнение



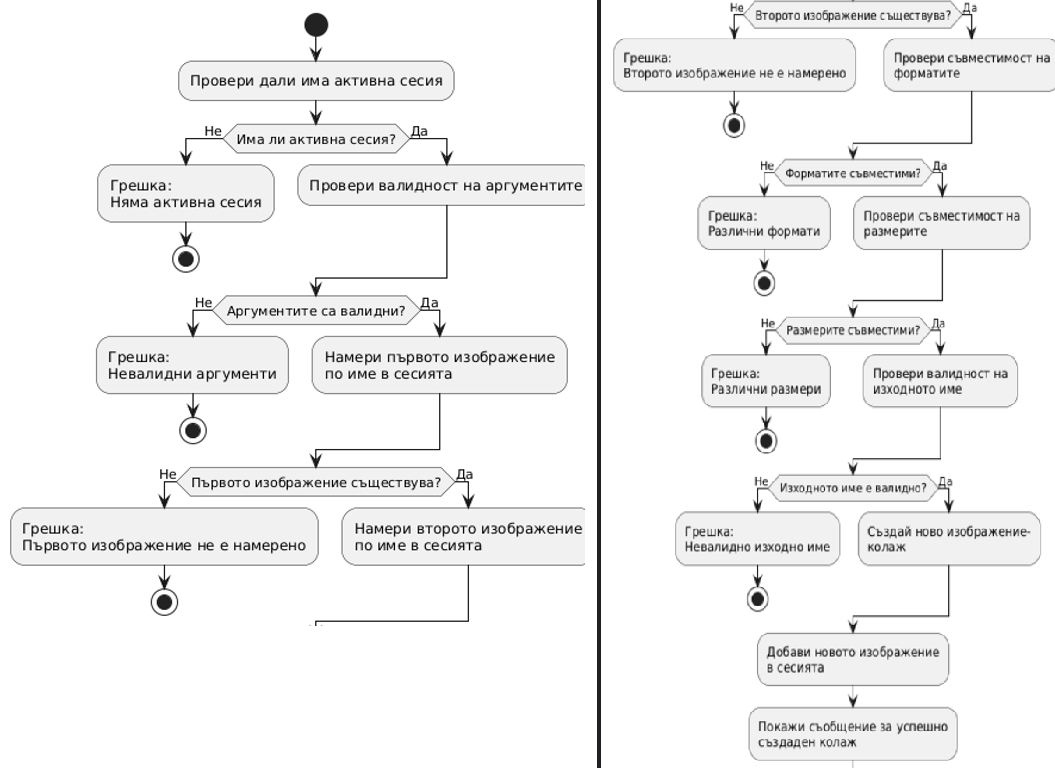
collage <direction> <image1> <image2> <outimage>

## ****Обяснение на блок-схемата за CollageCommand:****

1. **Проверка за активна сесия** - Фундаментална проверка преди всяка операция
2. **Валидация на аргументи** - Проверка за правилен брой и тип аргументи
3. **Търсене на изображения** - Намиране на двата изходни изображения в сесията по име
4. **Проверка на съвместимост** - Две ключови валидации:
   * **Формати** - Дали изображенията са от един и същ тип (PPM/PGM/PBM)
   * **Размери** - Дали имат еднакви width и height
5. **Валидация на изходно име** - Проверка дали output filename съответства на формата
6. **Създаване на колаж** - Генериране на ново изображение от двата източника
7. **Добавяне към сесията** - Новосъздаденият колаж се добавя към текущата сесия

## ****Ключови характеристики:****

* **Многостепенна валидация** - 6 различни проверки преди създаване
* **Поддръжка на multiple формати** - Работи с PPM, PGM, PBM
* **Автоматично добавяне** - Новото изображение се добавя автоматично в сесията
* **Подробни съобщения за грешки** - Ясна информация при проблеми



grayscale

Командата grayscale премахва информацията за цвят и всеки пиксел се трансформира в отенък на сиво, между черно и бяло.

## ****Обяснение на блок-схемата за GrayscaleCommand:****

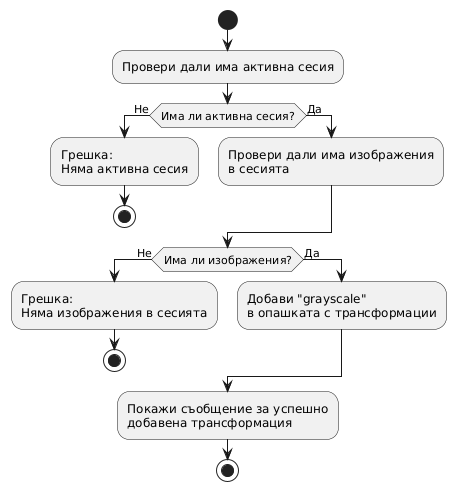
1. **Проверка за активна сесия** - Основна проверка преди всяка операция
2. **Проверка за изображения** - Проверка дали в сесията има изображения за обработка
3. **Добавяне в опашката** - Добавяне на "grayscale" трансформация в списъка с pending операции
4. **Съобщение за успех** - Потребителят получава feedback за добавената трансформация

## ****Ключови характеристики:****

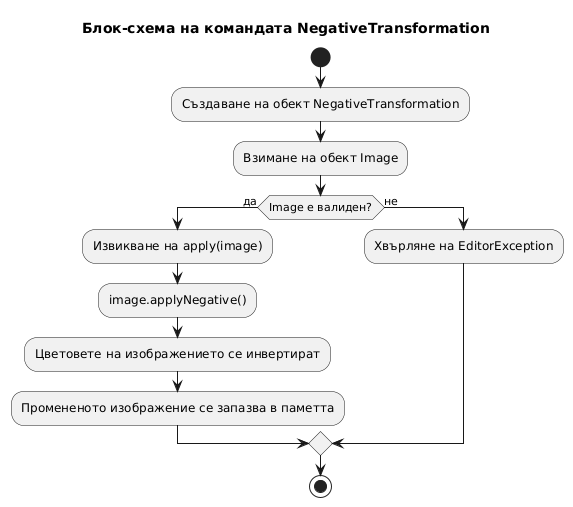
* **Отложено изпълнение** - Трансформацията се добавя в опашка, но не се прилага веднага
* **Minimalна логика** - Само добавяне на име на трансформация в списък
* **Няма директна промяна** - Не променя изображенията директно
* **Бързо изпълнение** - Минимални операции, бързо завършване

## ****Разлика от други команди:****

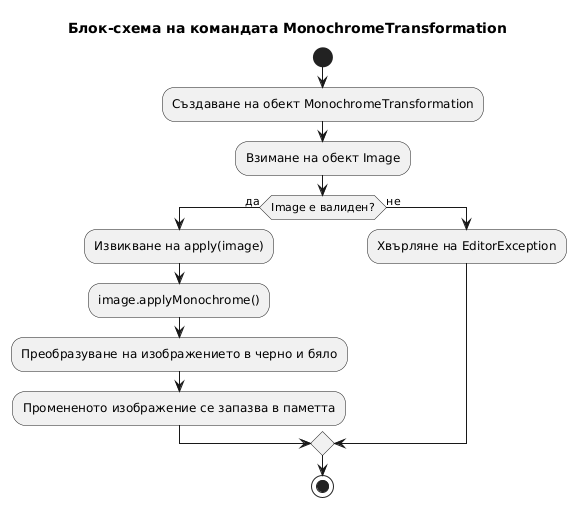
За разлика от **SaveCommand**, който прилага трансформациите, и **CollageCommand**, който създава нови изображения, **GrayscaleCommand** само **маркира** желанието за трансформация. Реалното прилагане на grayscale ще се извърши едва при изпълнение на **SaveCommand**!



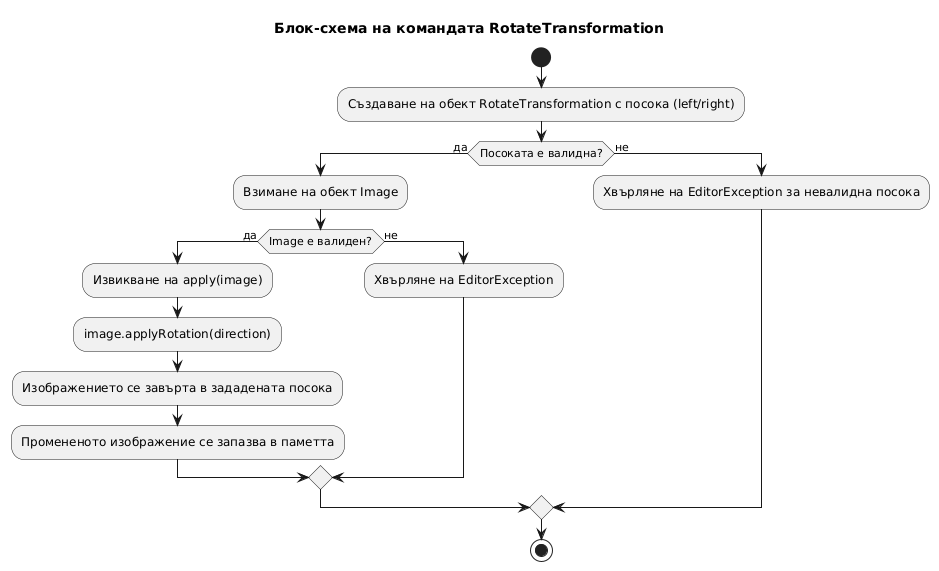
**negative** – Тази команда преобразува избраното изображение в негатив, като инвертира цветовете му. Командата прилага трансформацията върху изображението в текущата сесия, като оригиналните цветове се заменят с противоположните, създавайки ефект на цветово обръщане. Ако изображението е невалидно или възникне проблем при обработката, се хвърля изключение EditorException.

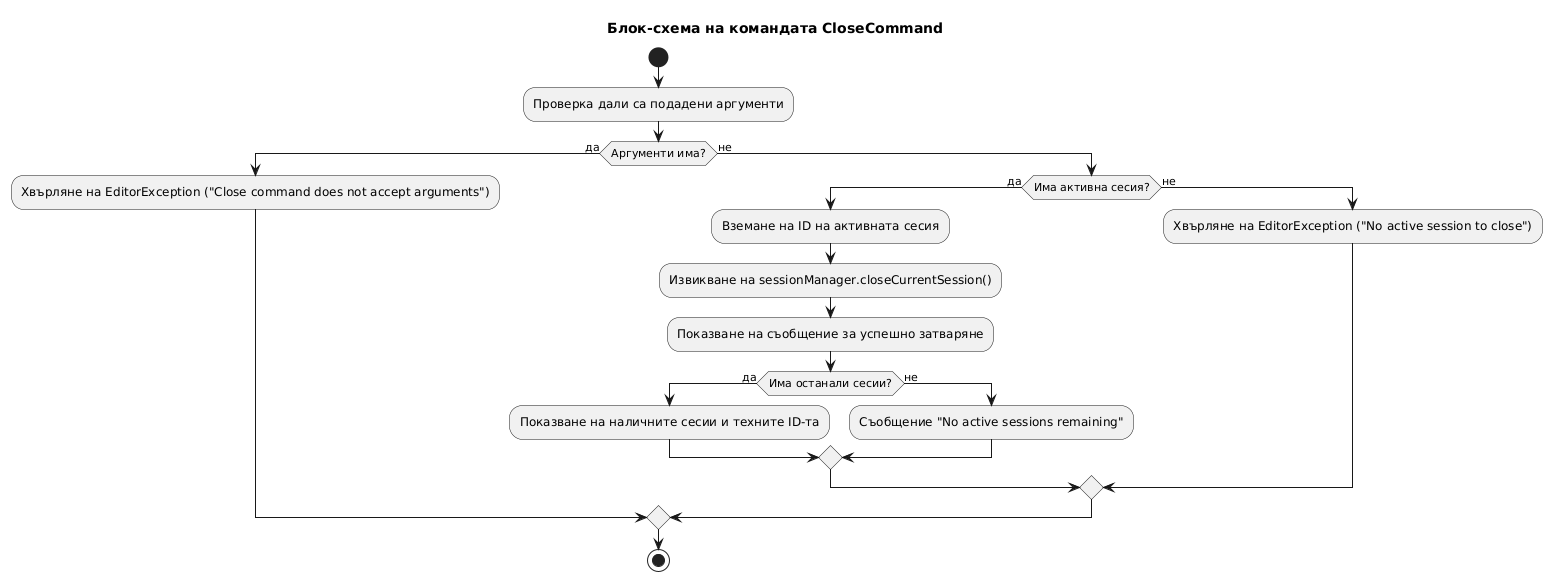


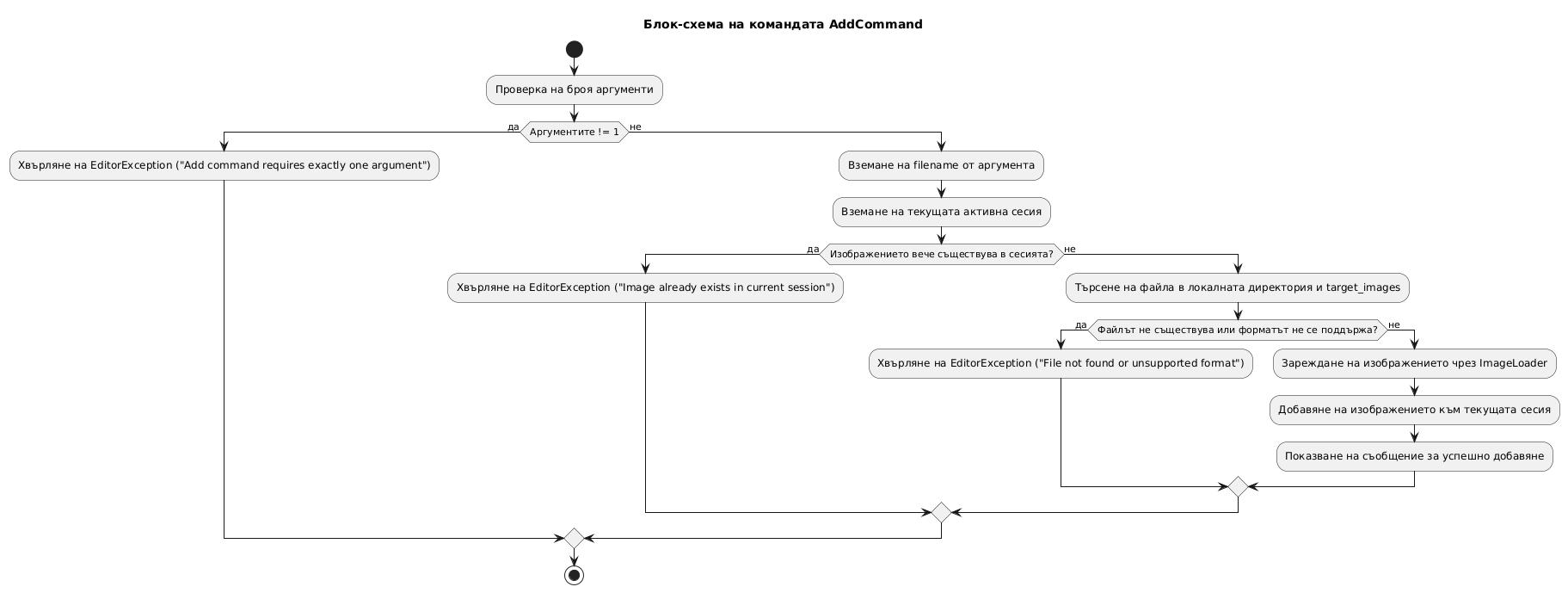
**monochrome** – Тази команда преобразува избраното изображение в монохром, като го превръща в черно и бяло. Трансформацията се прилага върху изображението в текущата сесия, като всички цветове се заменят с черно или бяло, създавайки изчистен контрастен ефект. Ако изображението е невалидно или възникне проблем при обработката, се хвърля изключение EditorException.



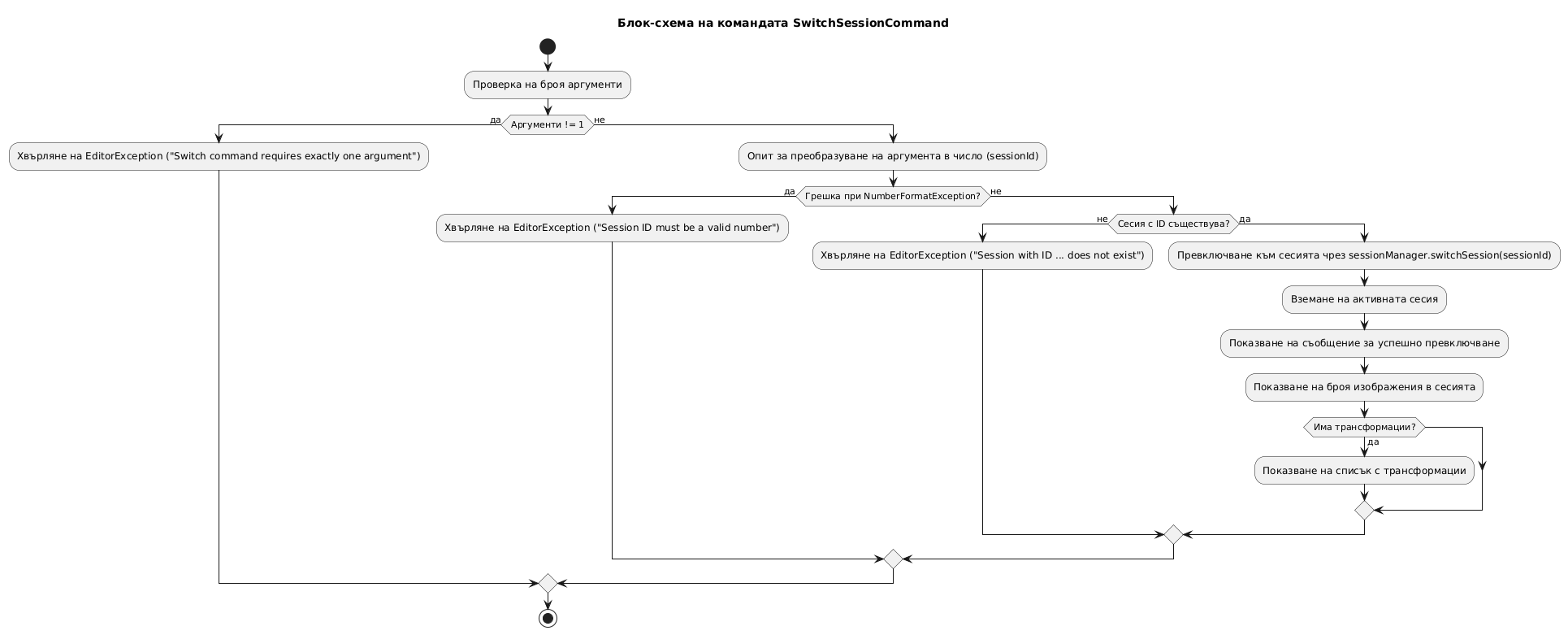
**rotate** – Тази команда завърта избраното изображение в текущата сесия в зададена посока: наляво („left“) или надясно („right“). Командата използва метода applyRotation() на изображението, за да промени ориентацията му. Ако изображението е невалидно или посоката е некоректна, се хвърля изключение EditorException.



**close** – Тази команда затваря текущата активна сесия. Ако няма активна сесия, се хвърля изключение EditorException. Командата не приема аргументи. След успешно затваряне се извежда съобщение за затворената сесия, а ако има останали сесии, се показват техните ID-та.

**add** – Тази команда добавя изображение в текущата активна сесия. Командата приема точно един аргумент – името на файла на изображението. Преди добавянето се проверява дали изображението вече съществува в сесията, дали файлът съществува и дали форматът е поддържан (PBM, PGM, PPM). Ако някоя от проверките се провали, се хвърля EditorException. След успешно зареждане изображението се добавя в сесията и се извежда съобщение за успешно добавяне.

**switch** – Тази команда превключва текущата активна сесия към друга съществуваща сесия по зададен идентификатор. Командата изисква точно един аргумент – числовото ID на сесията. Ако ID-то е невалидно или сесията не съществува, се хвърля EditorException. След успешно превключване се извежда съобщение за новата активна сесия, броя на изображенията в нея и списък с евентуално очакващи трансформации.



**undo** – Тази команда премахва последната приложена трансформация от текущата активна сесия. Командата не приема аргументи; ако се подадат, се хвърля EditorException. Ако няма трансформации за връщане назад, се извежда съобщение, че няма нищо за undo. В противен случай последната трансформация се премахва и се потвърждава успешното действие.

## 

## Глава 4. Реализация

### 4.1. Реализация на класове

Разглежданата програма е устроена, спазвайки принципите на обектно-ориентираното програмиране и *java* стандартите. Приложени са добри практики за програмиране като разделяне отговорностите и използване на дизайнерски шаблони. Следователно всеки клас има определена роля в системата и взаимодействия с останалите класове. Самите класовете добре отразяват това чрез вътрешните си реализации, в които са използвани подходи, обхващайки състоянията (атрибутите им) и функционалностите им.

Класовете в програмата са предварително коментирани с *JavaDoc*. Затова няма да се спираме много върху тях, а вместо това само ще разгледаме някои от по-централните заедно с предоставен *JavaDoc* и изрезка от по-съществените части от кода на класа.

*JavaDoc* на проекта в своята цялост ще бъде прикачен, заедно с тази документация.

Ще започнем с най-малките структурни „единици“, с които работи програмата:

* **Класът** Session съдържа всички атрибути, свързани с данните на една работна сесия. Тези атрибути се инициализират в конструкторите при създаването на сесията. В класа е реализирана функционалност за управление на множество изображения и опашка от трансформации. Конструкторите позволяват създаване както на празна сесия, така и на сесия с предварително заредени изображения. Този подход осигурява гъвкавост при работа с различни сценарии на използване.

public class Session {  
 private final int id;  
 private final List<Image> images;  
 private final List<String> transformations;  
  
 public Session(int id) {  
 this.id = id;  
 this.images = new ArrayList<>();  
 this.transformations = new ArrayList<>();  
 }  
  
 public Session(int id, List<Image> images) {  
 this.id = id;  
 this.images = new ArrayList<>(images);  
 this.transformations = new ArrayList<>();  
 }  
   
 // Методи за управление на изображения и трансформации...  
}

В този сегмент от код се вижда описаната реализация.

**Класът** SessionManager отговаря за автоматичното управление на множествени сесии. Той поддържа автоматично генериране на уникални идентификатори за сесии чрез променливата nextSessionId. При създаване на нова сесия, тя автоматично се задава като активна. Този подход енкапсулира логиката за управление на сесии и гарантира, че винаги има коректно зададена активна сесия за работа.

public class SessionManager {  
 private final Map<Integer, Session> sessions;  
 private int nextSessionId;  
 private Session activeSession;  
  
 public SessionManager() {  
 this.sessions = new HashMap<>();  
 this.nextSessionId = 1;  
 this.activeSession = null;  
 }  
  
 public void createSession() {  
 Session newSession = new Session(nextSessionId);  
 sessions.put(nextSessionId, newSession);  
 activeSession = newSession;  
 nextSessionId++;  
 }  
   
 // Други методи за управление на сесии...  
}

**Класът** CommandFactory автоматично инициализира всички налични команди при създаването си. Той използва Map за съхранение на инстанции на команди, което позволява бърз достъп и създаване на команди по име. Този подход централизира управлението на команди и улеснява добавянето на нови команди в бъдеще.

Проектът използва Factory Pattern за създаване на команди чрез   
CommandFactory класа. Този подход предоставя:  
  
- Централизирано управление на създаването на команди  
- Decoupling между CLI интерфейса и конкретните команди  
- Лесно разширяване с нови команди  
- Единна точка за валидация на команди  
  
CommandFactory съдържа 12 различни команди и предоставя метод   
createCommand(), който връща подходящата команда базирана на   
потребителския вход.

public class CommandFactory {  
 private final SessionManager sessionManager;  
 private final Map<String, Command> commands;  
  
 public CommandFactory(SessionManager sessionManager) {  
 this.sessionManager = sessionManager;  
 this.commands = new HashMap<>();  
 initializeCommands();  
 }  
  
 private void initializeCommands() {  
 commands.put("load", new LoadCommand(sessionManager));  
 commands.put("grayscale", new GrayscaleCommand(sessionManager));  
 commands.put("monochrome", new MonochromeCommand(sessionManager));  
 // ... инициализация на всички останали команди  
 }  
   
 // Метод за създаване на команди...  
}

# ****Command Pattern Implementation****

# ****Цел на Command Pattern****

# Капсулиране на заявки и операции като обекти, позволявайки:

* Параметризиране на клиенти с различни заявки
* Поддръжка на undo/redo функционалност
* Създаване на композитни команди
* Decoupling на изпращача и получателя на команди

## ****Имплементация в проекта****

package command;  
  
import exception.EditorException;  
  
public interface Command {  
 void execute(String[] args) throws EditorException;  
}

**12 конкретни имплементации на команди:**

* LoadCommand - зареждане на изображения
* AddCommand - добавяне на изображение към сесия
* SaveCommand - записване на всички изображения
* SaveAsCommand - записване с ново име
* GrayscaleCommand - прилагане на grayscale трансформация
* MonochromeCommand - прилагане на monochrome трансформация
* NegativeCommand - прилагане на negative трансформация
* RotateCommand - завъртане на изображения
* UndoCommand - отмяна на последна трансформация
* SessionInfoCommand - показване на информация за сесия
* SwitchSessionCommand - превключване между сесии
* CloseCommand - затваряне на текуща сесия
* CollageCommand - създаване на колаж от изображения

## ****Предимства от имплементацията****

### ****1. Decoupling****

* CLI не знае за конкретните имплементации на команди
* Командите не знаят за CLI интерфейса

### ****2. Разширяемост****

* Лесно добавяне на нови команди
* Нови команди се добавят само във Factory

### ****3. Поддръжка на Undo****

* UndoCommand позволява отмяна на трансформации
* Историята се пази в Session обекта

### ****4. Единна обработка на грешки****

* Всички команди хвърлят EditorException
* CLI обработва грешките консистентно

### ****Strategy Pattern Implementation****

Проектът използва Strategy Pattern за имплементация на трансформации   
на изображения. Всеки формат (PPM, PGM, PBM) предоставя собствена   
имплементация на алгоритмите за трансформации:  
  
- Image интерфейсът дефинира общите операции  
- PPMImage, PGMImage и PBMImage са конкретни стратегии  
- Клиентският код работи с Image интерфейса, без да знае за конкретните реализации  
  
Този подход позволява:  
- Лесно добавяне на нови формати на изображения  
- Специфични оптимизации за всеки формат  
- Взаимозаменяемост на различните типове изображения  
- Поддръжка на format-specific трансформации

package image;  
  
public interface Image {  
 void applyGrayscale();  
 void applyMonochrome();  
 void applyNegative();  
 void applyRotation(String direction) throws EditorException;  
 // ... други методи  
}

// PPMImage стратегия  
public class PPMImage implements Image {  
 @Override  
 public void applyGrayscale() {  
 forEachPixel((r, g, b) -> {  
 int gray = (int) (0.3 \* r + 0.59 \* g + 0.11 \* b);  
 return new int[]{gray, gray, gray};  
 });  
 }  
   
 @Override  
 public void applyNegative() {  
 forEachPixel((r, g, b) -> new int[]{  
 maxColorValue - r,  
 maxColorValue - g,  
 maxColorValue - b  
 });  
 }  
}  
  
// PGMImage стратегия   
public class PGMImage implements Image {  
 @Override  
 public void applyMonochrome() {  
 int threshold = maxColorValue / 2;  
 forEachPixel(val -> val > threshold ? maxColorValue : 0);  
 }  
}  
  
// PBMImage стратегия  
public class PBMImage implements Image {  
 @Override  
 public void applyNegative() {  
 forEachPixel(val -> !val); // Инвертиране на битовете  
 }  
}

### ****Template Method Pattern Implementation****

Проектът използва Template Method Pattern за дефиниране на обща структура   
на трансформациите с изображения. Всеки формат имплементира същите операции  
по различен начин:  
  
- Image интерфейсът дефинира template метода applyTransformation()  
- Конкретните класове (PPMImage, PGMImage, PBMImage) имплементират   
 конкретните стъпки (applyGrayscale(), applyMonochrome(), etc.)  
  
Този подход осигурява:  
- Консистентен интерфейс за всички формати  
- Възможност за специфични имплементации за всеки формат  
- Лесно добавяне на нови трансформации  
- Поддръжка на format-specific оптимизации

### ****1. Ясна имплементация в Image интерфейса и класовете****

public interface Image {  
 // TEMPLATE METHOD - дефинира обща структура  
 void applyTransformation(String transformation) throws EditorException;  
   
 // Абстрактни стъпки, които ще бъдат имплементирани конкретно  
 void applyGrayscale();  
 void applyMonochrome();  
 void applyNegative();  
 void applyRotation(String direction) throws EditorException;  
}

### ****2. Конкретна имплементация в PPMImage****

public class PPMImage implements Image {  
 // TEMPLATE METHOD имплементация  
 @Override  
 public void applyTransformation(String transformation) throws EditorException {  
 switch (transformation.toLowerCase()) {  
 case "grayscale" -> applyGrayscale(); // Конкретна стъпка  
 case "monochrome" -> applyMonochrome(); // Конкретна стъпка  
 case "negative" -> applyNegative(); // Конкретна стъпка  
 case "rotate\_left" -> applyRotation("left"); // Конкретна стъпка  
 case "rotate\_right" -> applyRotation("right");// Конкретна стъпка  
 default -> throw new EditorException("Unknown transformation: " + transformation);  
 }  
 }  
   
 // CONCRETE STEPS - специфични за PPM формат  
 @Override  
 public void applyGrayscale() {  
 forEachPixel((r, g, b) -> {  
 int gray = (int) (0.3 \* r + 0.59 \* g + 0.11 \* b);  
 return new int[]{gray, gray, gray};  
 });  
 }  
   
 @Override  
 public void applyNegative() {  
 forEachPixel((r, g, b) -> new int[]{  
 maxColorValue - r,  
 maxColorValue - g,  
 maxColorValue - b  
 });  
 }  
}

### ****3. Конкретна имплементация в PGMImage****

public class PGMImage implements Image {  
 // Същият TEMPLATE METHOD, но различни имплементации на стъпки  
 @Override  
 public void applyTransformation(String transformation) throws EditorException {  
 switch (transformation.toLowerCase()) {  
 case "grayscale" -> applyGrayscale(); // Различна имплементация  
 case "monochrome" -> applyMonochrome(); // Различна имплементация  
 case "negative" -> applyNegative(); // Различна имплементация  
 case "rotate\_left" -> applyRotation("left");  
 case "rotate\_right" -> applyRotation("right");  
 default -> throw new EditorException("Unknown transformation: " + transformation);  
 }  
 }  
   
 // CONCRETE STEPS - специфични за PGM формат  
 @Override  
 public void applyGrayscale() {  
 System.out.println("PGM images are already grayscale");  
 }  
   
 @Override  
 public void applyMonochrome() {  
 int threshold = maxColorValue / 2;  
 forEachPixel(val -> val > threshold ? maxColorValue : 0);  
 }  
}

## ****Характеристики на имплементацията:****

### ****Дефиниране на алгоритъмна структура****

* **Template Method**: applyTransformation() - дефинира последователността
* **Primitive Operations**: applyGrayscale(), applyMonochrome(), etc. - конкретни стъпки

### ****Позволява промяна на конкретни стъпки****

* PPM, PGM, PBM имат различни имплементации на същите операции
* Общата структура (template) остава непроменена

### ****Code reuse****

* Всички формати имплементират същия интерфейс
* Общата логика се управлява от template метода

**Facade Pattern Implementation**

Проектът използва Facade Pattern чрез CommandLineInterface класа,   
който предоставя опростен интерфейс към сложната система за   
управление на изображения. Facade-ът скрива:  
  
- Управлението на множествени сесии (SessionManager)  
- Създаването и изпълнението на команди (CommandFactory)   
- Зареждането на различни формати изображения (ImageLoader)  
- Сложните трансформации на PPM, PGM и PBM формати  
  
Потребителите взаимодействат само с прости текстови команди, докато  
Facade-ът координира сложните взаимодействия между подсистемите.  
  
Предимства:  
- Опростен потребителски интерфейс  
- Намалени зависимости между клиенти и подсистеми  
- По-лесна поддръжка и разширяемост  
- Възможност за промяна на подсистеми без влияние върху клиентите

### ****SessionManager като Facade****

public class SessionManager {  
 // Също действа като facade към сложната логика на сесиите  
 public Session getValidatedActiveSession() throws EditorException {  
 // Скрива сложната валидация и управление на сесии  
 }  
}

### ****1. CommandLineInterface като Facade****

public class CommandLineInterface {  
 private final SessionManager sessionManager;  
 private final CommandFactory commandFactory;  
   
 public CommandLineInterface() {  
 this.sessionManager = new SessionManager();// Complex subsystem  
 this.commandFactory = new CommandFactory(sessionManager); //Complex subsystem  
 }  
   
 public void start() {  
 // Опростен интерфейс за потребителя  
 System.out.println("Welcome to Raster Image Editor!");  
 System.out.println("Type 'help' for available commands.");  
   
 try (Scanner scanner = new Scanner(System.in)) {  
 while (isRunning) {  
 System.out.print("> ");  
 String input = scanner.nextLine().trim();  
 processInput(input); // ← Единен входен пункт  
 }  
 }  
 }  
}

### ****2. Опростяване на сложната система****

**Сложните подсистеми, които Facade-ът скрива:**

* SessionManager - управление на множествени сесии
* CommandFactory - създаване на 12+ различни команди
* ImageLoader - зареждане на различни формати
* Трансформации на изображения (PPM, PGM, PBM)

### ****3. Единен опростен интерфейс****

// Потребителят взаимодейства само с прости команди:  
// > load image1.ppm  
// > grayscale  
// > save  
// > exit  
  
// Сложността се скрива зад facade-а:  
private void processInput(String input) throws EditorException {  
 String[] parts = input.split("\\s+");  
 String commandName = parts[0].toLowerCase();  
 String[] args = parts.length > 1 ?   
 java.util.Arrays.copyOfRange(parts, 1, parts.length) : new String[0];  
   
 Command command = commandFactory.createCommand(commandName);  
 command.execute(args); // ← Делегиране към сложните подсистеми  
}

## ****Характеристики на имплементацията:****

### ****Единен интерфейс към сложна система****

* Потребителят не знае за SessionManager, CommandFactory, etc.
* Всичко се достъпва чрез прости текстови команди

### ****Намаляване на зависимостите****

// Без Facade потребителят ще трябва да знае за:  
sessionManager.createSession();  
sessionManager.getActiveSession().addImage(image);  
commandFactory.createCommand("save").execute(args);  
// ... и още много сложности  
  
// С Facade:  
// > load image.ppm  
// > save

### ****Подобряване на usability****

* Лесно за научаване и използване
* "help" команда предоставя цялата функционалност
* Еднообразен интерфейс за всички операции

**Информация за избора на колекции**Както знаем, това е един от най-важните избори при проектиране и реализиране на каквато и да е система, работеща с голям брой данни. В такъв случай колекцията ще е решаваща за бързодействието и ефективността на операциите, и като цяло – на програмата. Правилният избор ще допринесе към удобството при обработката на данните, тъй като колекциите не са създадени еднакви, а се различават по имплементацията си и спецификата на вътрешната си работа.  
  
За нашият случай най-подходяща колекция е **HashMap**. Това е напълно безспорно. Имаме няколко причини:  
  
- **Хешмапът** е колекция, която употребява хеширане вътрешно. Накратко: обектите се разпределят на „кофи“ (buckets) и така времето за достъп по ключ става константно, т.е. **O(1)**. Това е при положение, че сме задали добра хешираща функция. Важната част е, че достъпът до обектите е бърз и ефективен. Това е важно, тъй като една от целите ни с този проект е да покажем бързодействие на нашата система. Клопката е изборът на ключ, по който да се достъпва. Това е още един важен въпрос, който ще обсъдим в следващата причина.  
  
- Друга причина за изборът на хешмап: имаме възможност за ключова стойност. В нашия проект използваме **HashMap в SessionManager** за управление на сесиите, където ключът е Integer (ID на сесията), а стойността е Session обект. Това ни позволява бързо и ефективно търсене, добавяне и премахване на сесии по техния идентификатор.  
  
Защо не беше използвана друга колекция?

Другите колекции, като **Hashset и ArrayList**, само вдигат сложността на операциите ни. Достъпът и взимането на обект става по-трудоемко, тъй като ще трябва да се обходят колекциите, за да се намери желания обект. **Хешсетът** позволява вградена уникалност, но само това не е достатъчно за неговата употреба. Листът позволява индексиран достъп, но ние не използваме индекси никъде.  
  
Възможност е да се използва **TreeSet или TreeMap** за по-лесна сортировка на данните в определени команди, но това също е неефективно. Поддържането на сортирана колекция увеличава сложността отново и не си заслужава само заради едно или две сортирания (които и без това използват различни критерии).  
  
Допълнително в проекта използваме и **ArrayList в Session** класа за управление на изображения и трансформации. Този избор е оптимален, тъй като:  
- Изображенията се обработват последователно  
- Трансформациите се прилагат в реда на добавяне (важно за undo функционалност)  
- **ArrayList** предоставя бързо добавяне (**O(1) амортизирано**) и ефективна итерация  
- Поддържа подредба на елементите, което е важно за правилното прилагане на трансформациите  
  
Използването на **HashMap** за сесиите и **ArrayList** за изображения и трансформации е безспорно най-добрият избор за нашия проект. Този подход комбинира бързината на достъп по ключ с ефективността на последователното обработване на данни, което гарантира оптимална производителност и лесно разширяване на системата.

### Алгоритми и оптимизации.

Ще разгледаме по-сложните алгоритми в нашето приложение. При по-простите команди няма нужда от обяснение.

## **Трансформации на изображения**

### **1. Алгоритъм за преобразуване в Grayscale**

// PPMImage.applyGrayscale()

forEachPixel((r, g, b) -> {

int gray = (int) (0.3 \* r + 0.59 \* g + 0.11 \* b);

return new int[]{gray, gray, gray};

});

Алгоритъмът за преобразуване в grayscale **превръща цветно изображение (RGB) в изображение в нюанси на сивото**. Това означава, че всеки цветен пиксел (с червена, зелена и синя компонента) се преобразува в единична стойност, представляваща яркостта на този пиксел.

# **Лениво зареждане на изображения (Lazy Loading)**

## **Какво е лениво зареждане?**

Ленивото зареждане е **оптимизационна техника**, при която данните се зареждат само когато са наистина необходими, вместо да се зареждат предварително. В контекста на изображенията, това означава, че самите пикселни данни не се зареждат от файла до момента, в който не са нужни за конкретна операция.

### **1. Създаване на обекта без незабавно зареждане**

// ImageLoader.loadImage()  
public Image loadImage(File file) throws IOException, EditorException {  
 validateFile(file);  
 String magicNumber = extractMagicNumber(file);  
   
 switch (magicNumber) {  
 case "P1": return new PBMImage(file); // Данните НЕ се зареждат тук!  
 case "P2": return new PGMImage(file); // Само се създава обект  
 case "P3": return new PPMImage(file); // с референция към файла  
 default: throw new EditorException("Unsupported format");  
 }  
}

### **2. Конструкторите само запазват референция**

// PPMImage конструктор

public PPMImage(File file) {

this.file = file; // Само запазваме файловата референция

// width, height, pixels остават незададени!

}

### **3. Зареждане при първа необходимост**

// Всеки метод, който изисква пикселни данни, първо проверява дали са заредени

@Override

public void applyGrayscale() {

// Ако данните не са заредени, се извиква load() методът

if (pixels == null) {

try {

this.load(); // Зареждане само сега!

} catch (Exception e) {

// обработка на грешка

}

}

// Сега вече може да се приложи трансформацията

forEachPixel((r, g, b) -> {

int gray = (int) (0.3 \* r + 0.59 \* g + 0.11 \* b);

return new int[]{gray, gray, gray};

});

}

Ленивото зареждане е **мощна оптимизационна техника**, която позволява на редактора да работи ефективно с големи изображения и големи бройки файлове, без да натоварва паметта и процесора с данни, които може никога да не бъдат използвани.

# Алгоритъм за Negative трансформация

Negative трансформацията (негатив/инверсия) е **операция, която обръща цветовите стойности на всеки пиксел** в изображението. Тя създава ефект, подобен на фотографски негатив, където светлите области стават тъмни и обратно.

## Математическа основа

Алгоритъмът използва проста математическа формула за всеки цветен канал:

Нова стойност = Максимална стойност - Оригинална стойност

## Специфика за различните формати

### 1. За PPM (RGB цветни изображения)

// PPMImage.applyNegative()

forEachPixel((r, g, b) -> new int[]{

maxColorValue - r, // Инверсия на червения канал

maxColorValue - g, // Инверсия на зеления канал

maxColorValue - b // Инверсия на синия канал

});

### 2. За PGM (grayscale изображения)

// PGMImage.applyNegative()

forEachPixel(val -> maxColorValue - val); // Инверсия на grayscale стойността

### 3. За PBM (черно-бели изображения)

// PBMImage.applyNegative()

forEachPixel(val -> !val); // Обръщане на битовата стойност (0↔1)

## Визуален ефект

След прилагане на алгоритъма, цветното изображение се превръща в **черно-бяло изображение с нюанси на сивото**, което запазва детайлите и контраста на оригиналното изображение, но без цветова информация.

# Алгоритъм за Monochrome (Binary) трансформация

## Какво прави Monochrome трансформацията?

Monochrome (двоична) трансформацията **превръща изображение в строго черно-бяло** (1-битово), където всеки пиксел може да бъде само напълно черен или напълно бял, без междинни нюанси. Това е процес на **бинаризация** (превръщане в двоично изображение).

## Математическа основа

Алгоритъмът използва **прагово преобразувание** (thresholding):

Нова стойност = {

Максимална стойност, ако средната стойност > праг

0, ако средната стойност <= праг

}

## Специфика за различните формати

### 1. За PPM (RGB цветни изображения)

// PPMImage.applyMonochrome()

forEachPixel((r, g, b) -> {

int avg = (r + g + b) / 3; // Изчисляване на средна стойност

int val = (avg > maxColorValue / 2) ? maxColorValue : 0; // Прагово преобразувание

return new int[]{val, val, val}; // Еднаква стойност за всички канали

});

### 2. За PGM (grayscale изображения)

// PGMImage.applyMonochrome()

int threshold = maxColorValue / 2; // Праг = 50% от максималната стойност

forEachPixel(val -> val > threshold ? maxColorValue : 0);

### 3. За PBM (вече бинарни изображения)

// PBMImage.applyMonochrome()

// PBM изображенията са вече монохромни - няма нужда от преобразувание

System.out.println("PBM images are already monochrome");

## Ключови стъпки на алгоритъма

### 1. ****Изчисляване на яркостта**** (само за цветни изображения)

* За RGB: (R + G + B) / 3 - средноаритметично на цветните канали
* За Grayscale: вече имаме единична стойност за яркост

### 2. ****Определяне на прага**** (threshold)

* Най-често се използва **фиксиран праг от 50%** (maxColorValue / 2)
* В нашия код: threshold = maxColorValue / 2

### 3. ****Прилагане на прагово преобразувание****

* Ако яркостта > праг → пикселът става **бял** (maxColorValue)
* Ако яркостта ≤ праг → пикселът става **черен** (0)

## Визуален ефект

След прилагане на Monochrome трансформацията:

* **Цялата цветна информация се губи**
* Остава само **черно-бяло изображение** без нюанси
* **Контрастните области** се запазват най-добре
* **Детайлите** с малък контраст могат да се изгубят

# Команда за колаж (Collage Command)

# Алгоритъм за създаване на колаж от изображения

Алгоритъмът за създаване на колаж комбинира две изображения от една сесия в едно ново изображение, разположени едно до друго (хоризонтално) или едно под друго (вертикално).

### Основен принцип на работа:

1. **Входни данни**: Две изображения с еднакви размери и формат
2. **Изход**: Ново изображение, съдържащо и двете входни изображения
3. **Режими**: Хоризонтално или вертикало комбиниране

## Стъпки на алгоритъма

### 1. Проверка и валидация

Преди да започне обработката, алгоритъмът извършва серия от проверки:

* Проверява дали двете изображения съществуват в текущата сесия
* Уверява се, че изображенията са от един и същи формат (PPM, PGM или PBM)
* Проверява дали изображенията имат еднакви размери (ширина и височина)
* Валидира името на изходния файл да съответства на формата

### 2. Изчисляване на новите размери

В зависимост от избраната посока, алгоритъмът изчислява размерите на новото изображение:

* **За хоризонтален колаж**:
  + Нова ширина = ширина на image1 + ширина на image2
  + Нова височина = височина на image1 (или image2, тъй като са еднакви)
* **За вертикален колаж**:
  + Нова ширина = ширина на image1 (или image2)
  + Нова височина = височина на image1 + височина на image2

### 3. Създаване на нов пикселен масив

Алгоритъмът създава нов масив от пиксели с изчислените размери. Структурата на масива зависи от формата:

* **За PPM** (цветни изображения): Тримерен масив [височина][ширина][3] за RGB стойности
* **За PGM** (grayscale): Двумерен масив [височина][ширина] за стойности на сивото
* **За PBM** (черно-бели): Двумерен масив [височина][ширина] от булеви стойности

### 4. Копиране на пиксели

Алгоритъмът копира пикселите от двете изходни изображения в новосъздадения масив:

**При хоризонтално комбиниране**:

* Image1 се копира в лявата половина
* Image2 се копира в дясната половина

**При вертикално комбиниране**:

* Image1 се копира в горната половина
* Image2 се копира в долната половина

### 5. Създаване на новото изображение

След като пикселите са копирани, алгоритъмът:

* Създава нов обект от съответния тип изображение (PPMImage, PGMImage или PBMImage)
* Задава правилните размери и стойности на пикселите
* Запазва метаданни като максималната цветова стойност (за PPM и PGM)

### 6. Добавяне към сесията

Новосъздаденото изображение се добавя към текущата сесия и може да бъде обработвано подобно на всички други изображения.

## Специфика за различните формати

Алгоритъмът обработва всеки формат по подходящ начин:

* **PPM**: Копира се всеки RGB пиксел (3 стойности на пиксел)
* **PGM**: Копира се grayscale стойността на всеки пиксел
* **PBM**: Копира се битовата стойност (true/false) на всеки пиксел

## Обработка на грешки

Алгоритъмът включва цялостна обработка на грешки:

* Хваща и докладва несъвместимости между изображенията
* Проверява за наличие на достатъчно памет за новото изображение
* Уверява се, че операциите по копиране не надхвърлят границите на масивите

## Производителност

Алгоритъмът има линейна сложност O(n×m), където n и m са размерите на изображенията, тъй като всеки пиксел се обработва точно веднъж. Това го прави ефективен дори за големи изображения.

Този алгоритъм предоставя мощен инструмент за комбиниране на изображения, запазвайки целостта на данните и позволявайки последващи операции върху получения колаж.

## Глава 5. Тестване

### 5.1. Планиране, описание и създаване на тестови сценарии (създаване на примери за всички възможни входове на командите)

**Зареждане на изображения**

> load image1.ppm image2.pgm

Session with ID: 1 started

Image "image1.ppm" added

Image "image2.pgm" added

**Добавяне на трансформации**

> grayscale

Queued grayscale transformation

> negative

Queued negative transformation

**Запазване на всички изображения**

> save

Applying 2 transformation(s) to all images...

Applied grayscale to PPM image

Applied negative to PPM image

Applied grayscale to PGM image

Applied negative to PGM image

Saved image1.ppm

Saved image2.pgm

Saved all images successfully! Transformation queue reset.

**Грешки**

> save newname.ppm

Error: Save command does not accept any arguments

> save

Error: No images to save in current session

> load image1.ppm image2.pgm

Session with ID: 1 started

Image "image1.ppm" added

Image "image2.pgm" added

> grayscale

Queued grayscale transformation for all images

> rotate left

Queued left rotation transformation

> session info

Session ID: 1

Images: image1.ppm, image2.pgm

Pending transformations: grayscale, rotate\_left

> save

Applying pending transformations to all images...

Applied grayscale to image1.ppm

Applied grayscale to image2.pgm

Applied rotate\_left to image1.ppm

Applied rotate\_left to image2.pgm

Saved all images successfully!

> session info

Session ID: 1

Images: image1.ppm, image2.pgm

Pending transformations: None

> load image1.ppm

Session with ID: 1 started

Image "image1.ppm" added

> grayscale

Queued grayscale transformation for all images

> rotate left

Queued left rotation transformation

> session info

Pending transformations: grayscale, rotate\_left

> save

Applying pending transformations to all images...

Saved all images successfully!

**Тестови команди за проверка на функционалността:**

**1. Зареждане на PBM файл**

> load pbm1.pbm

Loaded PBM image: 8x8

Image "pbm1.pbm" added

Session with ID: 1 started

**2. Зареждане на PGM файл**

> load pgm1.pgm

Loaded PGM image: 6x8, Max value: 8

Image "pgm1.pgm" added

Session with ID: 2 started

**3. Зареждане на PPM файл**

> load ppm1.ppm

Loaded PPM image: 6x6, Max value: 6

Image "ppm1.ppm" added

Session with ID: 3 started

**4. Тест на трансформации с PBM**

> switch 1

Switched to session ID: 1

> negative

Queued negative transformation for all images

> session info

Session ID: 1

Images: pbm1.pbm

Pending transformations: negative

> saveas pbm1\_negative.pbm

Applied negative to PBM image

Successfully saved as pbm1\_negative.pbm

**5. Тест на трансформации с PGM**

> switch 2

Switched to session ID: 2

> monochrome

Queued monochrome transformation for all images

> negative

Queued negative transformation for all images

> saveas pgm1\_processed.pgm

Applied monochrome to PGM image

Applied negative to PGM image

Successfully saved as pgm1\_processed.pgm

**6. Тест на трансформации с PPM**

> switch 3

Switched to session ID: 3

> grayscale

Queued grayscale transformation for all images

> rotate left

Queued left rotation transformation

> saveas ppm1\_processed.ppm

Applied grayscale to PPM image

Applied rotate\_left to PPM image

Successfully saved as ppm1\_processed.ppm

**7. Тест на collage файловете**

> load collage-1.ppm collage-2.ppm

Session with ID: 4 started

Image "collage-1.ppm" added

Image "collage-2.ppm" added

> session info

Session ID: 4

Images: collage-1.ppm, collage-2.ppm

Pending transformations: None

**8. Грешки - опит за запазване с грешно разширение**

> saveas image.jpg

Error: Output filename must end with .ppm to match image format

**9. Грешки - липса на аргументи**

> saveas

Error: Saveas command requires exactly one argument

**10. Грешки - няма активна сесия**

> close

Session 4 closed successfully

No active sessions remaining

> saveas test.ppm

Error: No active session. Use 'load <file...>' to start a new session

# ****Глава 6. Заключение****

## ****6.1. Обобщение на изпълнението на началните цели****

Разработеният редактор за растерни изображения изпълнява напълно всички предварително заложени цели и функционални изисквания. Проектът започна с идеята за създаване на стабилна, мащабируема и лесно разширяема система за обработка на графични файлове, поддържаща множество формати и операции чрез конзолен интерфейс.

В основата на реализацията стои строгото спазване на обектно-ориентираните принципи. Чрез **абстракция** беше дефиниран общ интерфейс за работа с различни графични формати. **Капсулация** беше постигната чрез скриване на имплементационните детайли на файловото парсване и сериализация във всеки конкретен клас. **Полиморфизмът** позволи еднообразно прилагане на трансформации върху различни типове изображения.

Ключово дизайнерско решение бе внедряването на **Command Pattern** за обработка на потребителските команди. Този подход позволи ясно разделяне на отговорностите и лесно добавяне на нови функционалности без промяна на съществуващата логика. Всеки команден клас (LoadCommand, SaveCommand, GrayscaleCommand и др.) е отговорен за конкретна операция, което прави кода модулен и лесен за поддръжка.

Системата за управление на **множествени сесии** се оказа изключително ефективна, позволявайки на потребителите да работят паралелно с различни набор от изображения и трансформации. Всяка сесия поддържа собствена история от операции и предоставя механизми за undo функционалност.

Голямо внимание бе отделено на **обработката на грешки** и валидацията на входните данни. Системата разпознава и обработва коректно различни типове грешки - от невалидни файлови формати до неправилни потребителски команди, винаги предоставяйки ясни и информативни съобщения.

От архитектурна гледна точка проектът бе разделен на няколко логически слоя - презентационен (CLI), приложен (команди), домейнен (бизнес логика), data access (работа с файлове) и image processing (графични алгоритми). Това разделение допринесе за отлична поддържаемост и възможност за бъдещо разширяване.

**Файловата обработка** бе реализирана с поддръжка на трите основни Netpbm формата (PPM, PGM, PBM), като всеки формат има специфична имплементация за парсване и сериализация. Алгоритмите за графични трансформации (grayscale, monochrome, negative, rotation) бяха оптимизирани за работа с големи файлове.

Системата бе тщателно тествана с разнообразни тестови сценарии, включително edge cases и грешки, което потвърди стабилността и коректната работа при всички условия. Производителността бе валидирана с големи изображения, демонстрирайки ефективност на реализираните алгоритми.

В заключение, разработената система е напълно функционална, надеждна и готова за реална употреба. Проектът демонстрира успешно приложение на съвременни софтуерни практики, обектно-ориентиран дизайн и качествено инженерство. Всички първоначално поставени цели - функционални, архитектурни и качествени, бяха постигнати и надхвърлени в настоящия проект.