Линк към страницата на приложението: <https://github.com/xhist/SVG>

**За кого е предназначена тази документация ?**

Документацията служи като документ за справка относно функционалностите на приложението и начин за усвояване на процеса на работа с него. Документацията е подходяща за всички лица, които изявяват желание да се докоснат до използването на приложението.

**Структура на документацията:**

* **Въведение** – запознава лицата, които изпитват интерес към проекта, с неговата документация, с основната идея, която се опитва да предаде на аудиторията.
* **Преглед на предметната област** – проследява дефиниции и концепции от предметната област, възникнали проблеми и ниво на трудност при минаване през различните етапи от решаване на задачата, избор на подходи и методи за преодоляване на трудности в кода.
* **Проектиране** – представя архитектурата на приложението и описва всеки един компонент от нея, както и връзките между тях. Описва се цикъла на работа на приложението.
* **Реализация, тестване** – включва избрани моменти от разработването на приложението, както и примерни тестови сценарии, подбрани за специфични случаи.

1. **Увод**
   1. **Цел и описание на проекта**

Когато работим с SVG файлове често ни се налага да помним основни ключови думи характерни за езика, на който комуникират SVG файловете с нашата машина. Това създава проблем например, когато забравим някоя ключова дума, която е съществено важна за правилната визуализация например на фигури. Когато работим с SVG фигури е необходимо да знаем какви са основните атрибути, които една фигура трябва да притежава, за да знаем, че сме я създали правилно. Проектът реализира именно това. Той предоставя интуитивен дизайн за работа както и възможността да се създават правилно SVG фигури с минимални усилия от потребителска страна, като потребителя задава само стойностите на атрибутите, които дадената фигура да заема, както и нейния тип. Основната идея, върху която стъпва проекта, е създаването на гъвкаво приложение за работа с SVG файлове или по-точно за работа с фигури записани в SVG формат. Проектът се стреми да предостави на потребителя възможност да работи пряко с SVG файлове чрез зареждане на съдържанието на тези файлове и запазване на промените в същите или други SVG файлове.

* 1. **Цел и задачи на разработка**

1. **Цел на разработка**

Целта, която сме си поставили при разработването на приложението, е да можем да зареждаме правилно фигури от SVG файлове и да можем да извършваме операции над тях. Но за постигането на тази цел е необходимо да разпределим реализацията на тази цел в отделни задачи, които да интегрираме в нашето приложение. Тези отделни задачи се обединяват под общото множество на операции, които могат да се извършват над фигурите заредени от даден SVG файл, както и задачи насочени спрямо работата с SVG файлове.

1. **Задачи на разработка**

**Работа с файлове в формат SVG** – това, което приложението трябва да притежава е способността да чете и записва в файлове от дадения формат. Тази определена задача се разпределя на отделни подзадачи свързани с отваряне на файл, записване в файл както и затваряне на файл.

**Създаване на фигура –** изискванията за създаването на една фигура е да има правилни стойности за данни и типът и да бъде поддържан от системата.

**Изтриване на фигура -** при избор на точно определен индекс, на който отговаря дадена фигура, се изтриват данните за тази фигура.

**Транслация на фигура/и –** при избор на координати по съответните оси се транслират с тези координати всички фигури или една избрана от нас фигура ако предоставим индекс на съответната фигура.

**Принадлежност към посочен регион –** при наличието на дадена фигура, създадена през командния интерфейс, се прави проверка за принадлежност към тази фигура за последователността от фигури генерирани от SVG файл.

1. **Преглед на предметната област**

А) Осмисляне на предмета

Придобиване на знания, усвояване на концепции и дефиниции за съответния предмет. Придобитите умения се прилагат в изработката на приложението. Набор от такива знания са различните типове фигури, техните атрибути, начина на изразяването им в файловете, и структурата на всички SVG файлове. На базата на тези знания построяваме архитектурата на приложението. Предпочетени алгоритми са обхождане на масив, последователно търсене в масив, разцепване на низ на отделни низове.

Б) Проблеми на разработката

При употреба на полиморфизъм възниква проблем с сравняването на фигури за принадлежност към определен регион. Това създава допълнителни изисквания за правилното функциониране на отделните класове, където трябва да се познава типа на фигурата, с която сравняваме текущата. Този проблем е в основата на проблемите с задълбочена трудност, както и на разширеното описание на класовете. Чрез използването на командния интерфейс транслирането на фигури се превръща в проблем с натежаваща трудност. Случая, в който се заражда проблема, е, когато е неясно по кои оси извършваме транслирането. Транслирането може да става по произволна ос или по двете оси. Това налага необходимото създаване на допълнителни алгоритми за работа с низове, получени от командния интерфейс. Други проблеми добавящи препятствия към пътя на оформянето на финалния вид на приложението са създаването на фигури, за което се изисква да бъде фигурата от поддържан от системата тип, както и четенето на фигури от файл. При четенето на фигури от файл се набляга на структурата на файловете. Отчитайки, че за всеки файл всеки елемент не е еднозначно определен като отделен ред, тъй като всеки елемент, може да бъде разпределен на няколко реда, се приема идеята за допълнително разглеждане на случаите на валидност на този файл. Пример за това е, когато една фигура заедно с нейните атрибути може да се разпространява на няколко реда. Целостта на отделен елемент от файла се формира от специални изразни средства, които преведени на езика, на който е записан файла, означават начало и край на елемента. Тези изразни средства се наричат тагове и се обозначат по следния начин “<>”, “</>”.

В) Подходи при преодоляването на проблемите

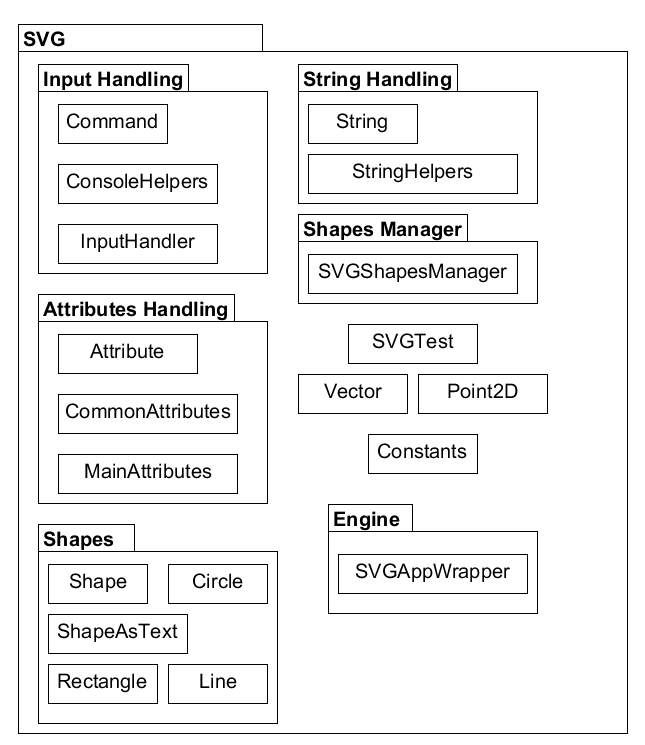
* Разбиване на проверката за принадлежност на една фигура към друга фигура на отделни случаи, които се обработват по конкретен начин. Всяка фигура съдържа собствена имплементационна схема за различните случаи на сравнения.
* За намирането на оста, задължителна за успешна транслация на фигурите, се разцепва низа, получен от командния интерфейс, на отделни низове, сред които намираме наименованието на конкретната ос и стойността по тази ос.
* Отделните елементи на текущия файл преминават през серия от валидации, за да се извлече правилно крайния резултат от прочитането на конкретния файл. Фокусът е върху намирането на началото на елемента и краят му.

Г) Функционални и нефункционални изисквания

* Всички операции, които се изпълняват над фигурите, изискват да има зареден предварително файл.
* Всяка фигура, която се създава трябва да бъде от поддържан от системата тип.
* Потребителят трябва да въвежда правилни данни за всяка една команда
* Всеки файл, който се зарежда, трябва да бъде файл с валидно съдържание и разширение
* Приложението работи чрез командния интерфейс
* Всяка фигура, която се прочита от файла, се игнорира за съществуваща в контекста на приложението и се счита просто за текст.
* Приложението трябва да позволява добавянето на нови типове фигури .

1. **Проектиране**

А) Общ вид на архитектурата

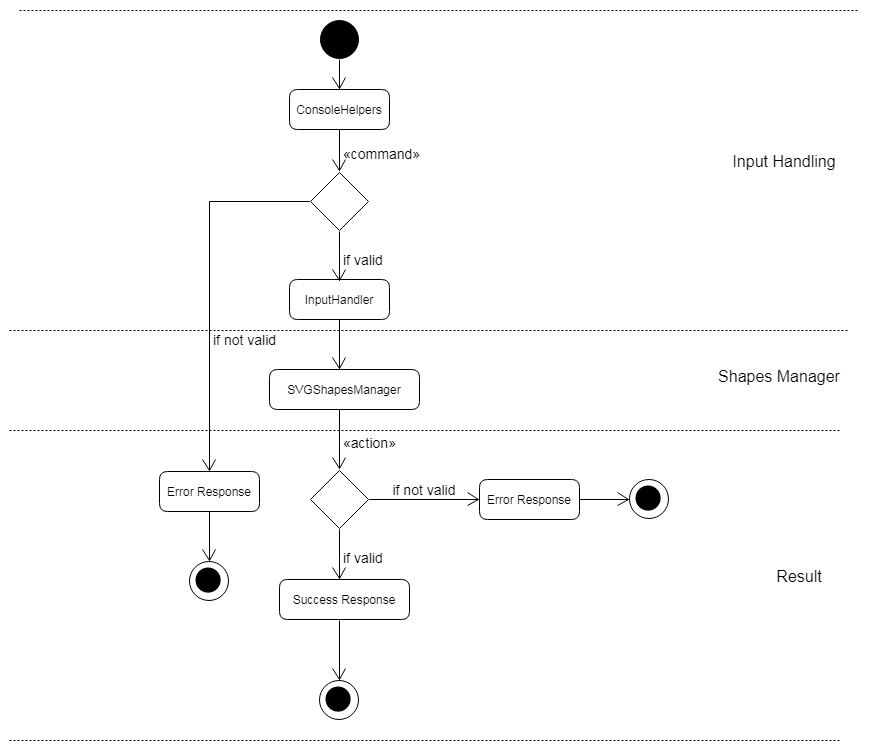


Тук е представена UML диаграма, която обозначава различните класове, групирани в отделни модули, като всеки модул означава ролята на самите класове. Обща структура на всеки един от модулите:

1. Input Handling - задачата му е да работи пряко с въведените данни в конзолата и да ги обработва.
2. String Handling – замества работата с примитивните низове вградени в езика, като добавя и допълнителни операции над тях.
3. Attributes Handling – поема задачата да управлява всички атрибути, които се подават от конзолата. Правят се валидации над атрибутите и се обработват за по-нататъшни процеси.
4. Shapes –базата от всички фигури, които се поддържат от приложението.
5. Shapes Manager – управлява заявките за работа с фигури и осъществява обратна връзка към конзолата.
6. Engine – готовото състояние на програмата. Отговаря за режима на работа на програмата и за последователността от действия, които се прилагат.
7. Подробно описание на всички класове:

* **Command –** осъществява четенето от конзолата, разпределя прочетеното съдържание от конзолата в отделни категории – име на командата и параметри.
* **ConsoleHelpers –** имплементира логиката за четенето от конзолата с помощта на класа Command и ако командата е позната за системата предаваме параметрите на тази команда напред в веригата от процеси.
* **InputHandler –** осъществява връзката между ConsoleHelpers класа и SVGShapesManager класа. Изпълнява ролята на медиатор между конзолата и класа, който обработва заявките за работа с фигури.
* **Attribute –** представя всеки атрибут с неговите име и стойност. Стойноста е възможно да бъде от специален тип – число или низ.
* **CommonAttributes –** съхранява последователност от общи за всички фигури атрибути, информация за това, кои от тях изискват специални стойности. Не всички общи атрибути са задължителни за съществуването на една фигура.
* **MainAttributes –** съхранява последователност от характерни за всяка една фигура атрибути, информация за това, кои от тях изискват специални стойности. Всяка фигура се нуждае от основните атрибути като шаблон, за да съществува.
* **Shape –** базисната структура на всяка една фигура, това какво се изисква да съдържа фигурата като функционалност и като атрибути.
* **Circle –** имплементира логиката за работа с окръжности, наследява базисната структура на класа Shape.
* **Rectangle –** имплементира логиката за работа с правоъгълници, наследява базисната структура на класа Shape.
* **Line –** имплементира логиката за работа с линии, наследява базисната структура на класа Shape.
* **ShapeAsText –** добавя логиката за работа с неправилни фигури, третира ги като текст, не се стреми да взаимодейства с операциите над фигури.
* **String –** преобразува цялата работа с примитивните низове, вградени в езика C++, в една обща структура от член-данни и член-функции. Обобщава основните операции, които се използват за взаимодействие с низове, както и допълнителни, на които приложението разчита за изпълнението си.
* **StringHelpers –** клас с всички помощни функции, които се използват в класа String за реализацията на член-функциите.
* **SVGShapesManager –** контролира действията с фигури. Съдържа информацията за тях, както и дефинира операциите над тях.
* **SVGTest –** съдържа тестовите сценарии, инструкции за работа с операциите над фигури.
* **Vector –** замества работата с последователност от данни, дефинира операции за работа с данните съхранени в контейнер, допуска разширение на контейнера с още данни, предлага премахване на данни.
* **Point2D –** допълнителна структура, която намира място в реализацията на фигурата Line. Предлага възможност за работа с точки, дефинирани в 2D пространството.
* **Constants –** представя обща информация за основните и общите атрибути, които намират употреба в рамките на приложението.
* **SVGAppWrapper –** рамка на приложението, обединява класа ConsoleHelpers с класа SVGShapesManager, за да запечата връзката между двата класа и да осъществи стартирането на приложението.

Б) Общо структура на процесите, изпълнявани в контекста на програмата.



1. Общо описание на структурата на процесите

**Начална/отправна точка** (изобразена с черна точка) – показва началото на процеса

**„Работници”** (изобразени чрез правоъгълници с заоблени върхове) –създават мостове за процеса напред във веригата, потвърждават автентичността на процеса.

**Поток на действие** (изобразен чрез стрелки) – показва в какъв ред се извършват действията във веригата.

**Условие** (изобразено чрез ромб) – показва в каква посока може да протече процеса, в зависимост от състоянието на системата или предприетото действие

**Крайна точка** (изобразена чрез черна точка, вградена в друга бяла точка) – показва крайното състояние на процеса.

1. Подробно разглеждане на структурата на процесите

Когато говорим за процеси в контекста на приложението говорим за операциите, които се извършват както над файлове, така и над фигури. Началната точка маркира мястото, от което започва всеки процес - стартиране на приложението. Процесът зависи от въвеждане на данни в конзолата, което се случва в следващата стъпка от веригата от действия – прочитане на команда от конзолата с помощта на класа ConsoleHelpers. Ако командата е грешна се преминава към извеждане на резултат от процеса. В противен случай, командата се праща към следващия „работник” във веригата – класа InputHandler, който преглежда командата и нейните параметри и оформя крайния вид на информацията. Информацията се получава от следващия „работник” във веригата и това е класа SVGShapesManager, който отговаря за прилагането на операции над фигурите. Сред информацията стои името на командата, която ни праща към дадена операция от приложението, която да се извърши. Параметрите търпят проверка в рамките на текущия „работник” и се определя в коя посока ще продължи действието, като ще се върне накрая резултат от процеса.

1. **Реализация и тестове**

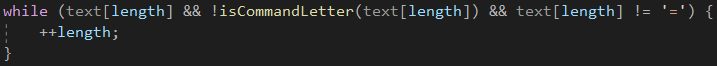
При построяването на приложението се набляга на ключови моменти, които позволяват на приложението да се разширява и без които приложението попада в дупки при реализацията. Когато разглеждаме тези моменти като например член-функции, чрез които заобикаляме даден проблем или член-данни, които притежават свойството да намалят риска от проблем в някои член-функции, ние всъщност наблягаме на ползата от тези член-функции или член-данни, и на техните откъси от код, които ни дават по-голяма видимост върху реализационния план, неговите подобрения, както и какво е значението им за цялото приложение.

А) Описание на реализационната част на ключови член-функции и член-данни на класовете

1. **Command**

Класа Command включва няколко важни момента, съставляващи целостта на приложението. Те са четенето от конзолата и подредбата на прочетените данни. Това спомага както за улеснената работа с прочетените данни, така и за пълноценното разпределяне на ролите на класовете.

* Член-данни:
  + String name – доставя информация за името на командата или общо казано за операцията, която ще се извършва. Полето използва класа String, за да се възползва от предимствата на неговите функционалности
  + Vector<String> parameters – представлява последователност от отделни низове, компилирани в едно общо множество, на което сме задали име parameters. Целта на това поле е да пази всички параметри за дадена команда/операция. Изборът на класа Vector е съобразен с мисълта за задълбочената работа с последователност от данни с разнообразна дължина.
* Член-функции:
* int extractCommand(const String& text, Vector<String>& words) – функцията връща като резултат броя на думите, които са били прочетени от конзолата. Целта на тази функция е да може да разцепи целия текст, въведен от конзолата на отделни думи, които намират място в полетата на класа Command. Функцията използва правило при четене от конзолата като например какви символи се допускат във всяка дума. Подтика за написването на тази функция е дългия текст, идващ от конзолата, от който трябва да отбележим, коя дума е името на командата и кои думи са параметрите.



Цикълът на изображението преглежда низът, прочетен от конзолата и нарежда на функцията да премине напред към следващия символ от низът, ако текущия символ не принадлежи към допустимите за програмата или ако е знака „=”.

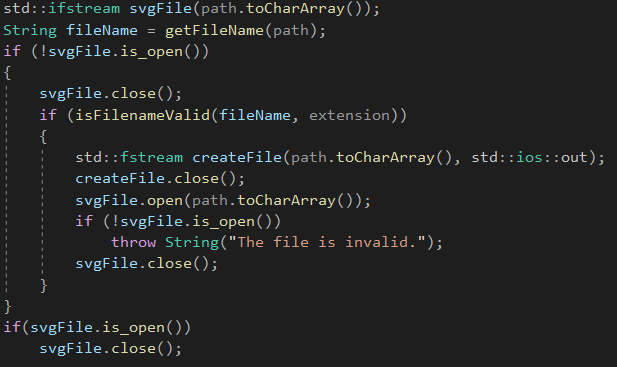


Горната снимка онагледява изчерпателно процеса на извличане на допустимите символи както и знака „=”. Това, което се открива на изображението е условието, с което започва кодът. Условието ни информира, че символ от низа е бил отбелязан за допустим и чрез циклите появяващи се по-надолу се преминава през низът символ по символ и за всеки символ, който е годен за записване в низ, го броим.

1. **InputHandler**

Освен свързващо звено между конзолата и мениджъра, който управлява фигурите, ролята на класа InputHandler далеч не се измества от това да филтрира данните, пристигащи от конзолата. Филтрирайки ги и потвърждавайки, че са годни за обработка и употреба, ги изпраща към класа SVGShapesManager, който се заема с тяхното разглеждане или обработка. Част от член-функциите и член-данните тук спомагат за по-интерактивното взаимодействие с потребителя.

* Член-данни:
  + String currentOpenFile – предназначението на тази член-данна е да запази името на файлът, който е бил отворен през конзолата. По този начин, програмата има знанието дали има текущо отворен файл, за да отвори друг. Това предотвратява злоупотребата с командата за отваряне на нов файл, за да може да се стигне до затваряне на текущия файл преди да се отвори нов файл.
* Член-функции:
  + bool isPathValid(const String& path, const String& extension) – анализира пътят към целевия файл, който се опитваме да прочетем. Прилагаме валидация над разширение на името на файла, като не се допуска друго разширение от зададеното в функцията като параметър. Прави се и проверка на автентичността на файла – дали съществува или дали пътят е правилно построен. Важна забележка тук е използването на потоци за манипулиране на файл с цел прочитане на файла, ако е правилно зададен или създаването му, ако разширението е правилно настроено. Малък фрагмент от кода, съставляващ функцията:

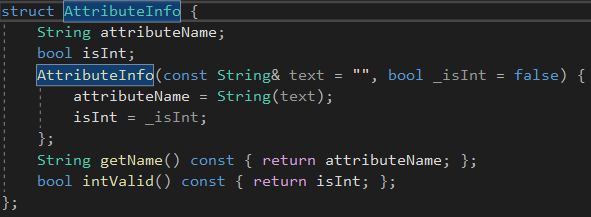


Поглеждайки към кода, необходимо е да вперим поглед към метода toCharArray(), който е част от класа String и макар и странно той връща const char\*. Главния фактор за създаването на такава функция е изискването на потоците за манипулация на файлове аргументите да бъдат от примитивен за езика C++ тип. Други особени елементи на кода са getFileName(path), като действието на този метод се развива около възможността да можем да работим пряко с пълното име на файла. Типът на върнатия резултат от метода getFileName(path) е String, както може би се очаква от внимателно преглеждане на кода. Функцията намира своето приложение малко по-надолу в метода isFilenameValid(filename, extension), като под extension тук се има предвид разширение, спрямо което трябва да извършим проверка. Този метод постига отличен резултат при отделянето на разширението на файла от името му и прокарвайки сравнения между него и разширението, което имаме като параметър на функцията.

1. **Constants**

Предназначението на този файл е да се държи като хранилище за основната информация за всеки тип фигура, както и за непрекъснатостта на изпълнението на приложението. Тук се описват основните компоненти влизащи в употреба в приложението. Добавени са допълнителни структури за облекчение на работата на приложението. Важни акценти, които трябва да поставим върху съдържанието на конкретния файл са:

* struct AttributeInfo – семантиката на тази допълнителна структура ни диктува, че всеки атрибут, който принадлежи на дадена фигура приема стойности от разнообразен тип. Тези типове се делят на число и низ. Това, което структура казва е дали един атрибут приема стойности от тип число или низ.



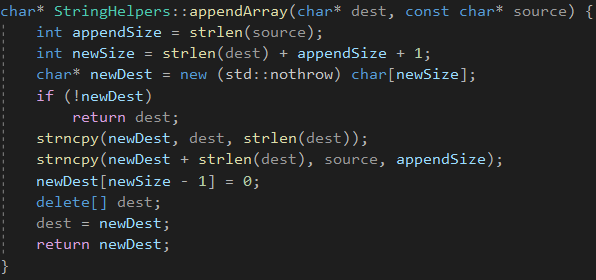
От вида на кода заключаваме, че структурата съдържа като полета за член-данни името на атрибута и булева стойността, носеща за информация дали атрибутът е число или низ. Конструктора използва проста логика за инициализиране на полетата. Останалите член-функции са средства, чрез които външните класове говорят с структурата. Те са залепени за полетата на структурата, за да изнесат информацията за тях към външни източници.

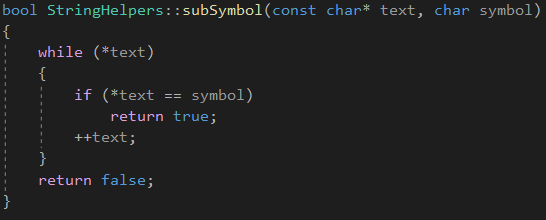
Б) Управление на паметта и алгоритми.

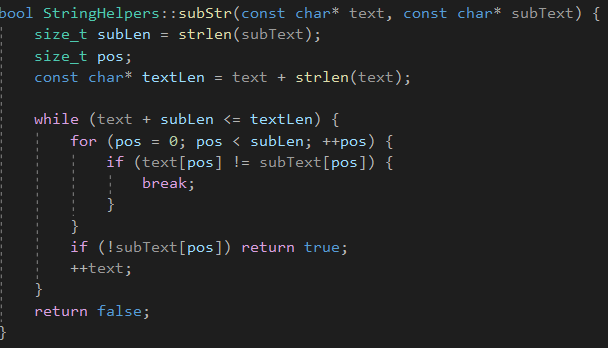
В хода на разработката на приложението, когато работим с памет, често опираме до нейното управление, като в повечето случаи ние трябва да се грижим за нея заради продуктивността на програмата, защото може да се стигне до изтичане на данни и това може да укаже забавящо въздействие върху програмата. Надолу са изброени ключови места, в които имаме организирани действия с паметта, но и алгоритми от изключително значение за приложението.

1. **String**

В областта на класа String управлението на памет често излиза наяве, защото имаме операции с низове, които могат да бъдат от произволна дължина. Затова прибягваме до употребата на динамична памет, като след заделяне на определен блок от памет трябва да се освободи паметта чрез изрично нареждане. От съвкупността на алгоритми, оформящи приложението, голяма част са концентрирани именно тук. Причината за това са операциите с низове, които изискват пълно проследяване на съдържанието им. Като примери за такива са търсене на низ в друг низ, търсене на символ в низ, както и намирането на броя срещания на низ в друг низ или на символ в низ. Цялата информация за алгоритмите с техните декларация и дефиниция са отделени в общ клас, пригоден специално за класа String, наречен StringHelpers. Основни алгоритми за класа String са слепянето на два низа, търсенето на низ в друг низ или на символ в низ. Надолу са илюстрирани кратки фрагменти от реализациите на алгоритмите.



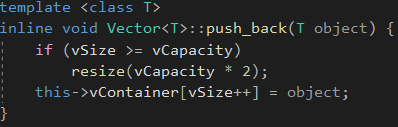


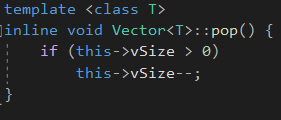


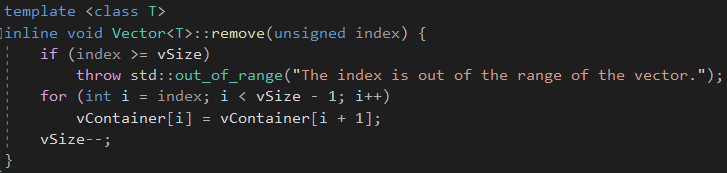
Горните примерни изображения подчертават реализацията на функциите за слепяне на два низа, търсенето на низ в низ или на символ в низ, като са показани в последователност.

1. **Vector**

От предназначението на класа Vector, който се въплъщава в ролята на заместник на последователност от данни от еднакъв тип, се разбира, че ще имаме размер на това колко данни могат да се добавят, както и колко е текущата им дължина. Това обаче е с променливо поведение, тъй като винаги можем да добавим нови данни. Това също от своя страна показва отношението на класа към управление на паметта и как паметта успява да се намести в центъра на реализацията на класа. Когато работим с последователност от данни с произволна дължина, ние пращаме отпратка към низовете с произволна дължина, защото също както тях имаме пряко взаимодействие с динамична памет – заделяне и освобождаване. Основни алгоритми за приложението също пазят своето място тук като например изтриване на елемент от последователността от данни, добавяне на нов елемент или премахване на последния елемент от опашката с данни. Надолу са показани алгоритмите с код, като са показани в последотелност от изтриване на елемент на определен индекс, на който стои елемента, добавяне на нов елемент, премахване на последния елемент от опашката с данни.







В) Тестове и тълкуване на смисълът им

След добавяне на финалната щриха към приложението се губи смисълът от непрестранното писане и поправяне на код, продължителното влагане на мисъл върху конструирането на финалния резултат, ако не вкараме приложението в употреба. Тогава се стига до така наречените тестове. Чрез тестовете се усвоява идеята за работа с приложението, дават отговор дали една функция работи според конкретните очаквания, като потенциалните отговори могат да бъдат тази функция да съдържа дупка, която може да бъде попълнена с бъг (преведено на български – грешка в основната идея на функцията), или да оправдае очакванията. Като примери за тестове са подготвени няколко тестови сценария за различни случаи при работа с някои функции. Всяка една от функциите няма тип на връщане, защото резултатът от работата с нея са съобщения, които се изпращат към конзолата. Тестовите сценарии са поместени в класа SVGTest.

1. void testPrintShapes() – тества функционалността за извеждане на информация за всички фигури.
2. void testCreateValidShape() – тества създаването на фигура чрез валидни данни.
3. void testCreateInvalidShapeWithInvalidName() – тества умението на функцията, грижеща се за създаването на фигури, дали може да улови грешка при подаването на грешен тип на фигура.
4. void testCreateInvalidShapeWithInvalidAttributes() – тества функционалността за създаване на фигура, ако атрибутите на фигурата са невалидни.
5. void testEraseValidShape() – тества изтриването на фигура с правилен индекс.
6. void testEraseInvalidShape() – тества изтриването на фигура с грешен индекс.
7. void testTranslateAll() – тества транслирането на цялото множество от фигури с някакви координати.
8. void testTranslateValidIndex() – тества транслирането на фигура с коректно зададен индекс с някакви координати.
9. void testTranslateInvalidIndex() – тества транслирането на фигура с невалиден индекс с някакви координати.
10. void testWithinShape() – тества проверката за принадлежност на фигура към даден регион/фигура.
11. void testAllFunctions() – обединение на всички тестове.

Приложението спазва коректността на всяка една функционалност, както и улавя успешно всяка една грешка при изпълнение на функциите. Целите, които се поставени в началото на документацията, са постигнати и приложението е готово за експлоатация. На места в файловете, съставящи приложението са оставени следи, които разкриват потенциала за бъдещо разрастване на приложението и разширение на функционалностите му. Като задача за реализиране в бъдеще с цел разширение на проекта, с нагласата да провокира възможните допринасяния за развитие на приложението е задачата за добавяне на още един тип атрибути – допълнителни атрибути, специфични за конкретна фигура, на английски - extra attributes, които не влияят върху съществуването на фигурата, а са като опции за обогатяване на съдържанието на фигурата.