Rozdział 1

Zaimplementowane algorytmy

W poniższych podrozdziałach zaprezentowano zasadę działania zaimplemenetowanych w oprogramowaniu algorytmów.

1.1 Działanie przycisków

W aplikacji każdy przycisk jest obiektem klasy ButtonOperator, która dziedziczy po klasie QPushButton. Dzięki czemu fizyczne obiekty przycisków mogą korzystać zarówno z metod klasy nadrzędnej np. isCheckable(), jak i klasy dziedziczącej. Właśnie dzięki nadpisaniu metod domyślnych klasy QPushButton - hoverEnter() i hoverLeave() stworzono możliwość detekcji, nad którym przyciskiem aktualnie znajduje się punkt fiksacji wzroku. Podczas wywołania obu metod zmieniana jest wartość logiczna zmiennej isHovered aktualnie obserwowanego przycisku. Prócz informacji o tym, czy dany przycisk jest aktualnie używany przechowuje się również dane o tym, czy przycisk zalicza się do tkzw. "specjalnych", czy też nie, jak i listę dostępnych dla niego tekstów do wyświetlania – wyglądów przycisku dla 6 stanów klawiatury (małe litery, duże litery, znaki specjalne karta pierwsza, znaki specjalne karta druga, polskie litery, menu kontekstowe). Wszystkie te dane wprowadza się podczas uruchamiania klawiatury i wtedy także zapisuje się przyciski kolejno do specjalnej listy. Kolejność jest znacząca w przypadku przycisków "specjalnych", gdyż ich obsługa zależna jest od wartości ich ID zapisanego w pliku ze stałymi. Określenie przycisku działającego odbywa się poprzez sprawdzenie ID przycisku, którego zmienna isHovered jest prawdziwa w klasie TimeManager. Do zarządzania stanem klawiatury, ostatnio wybranym przyciskiem specjalnym oraz ostatnio obserwowanym przyciskiem powstała klasa HoverManager. Zbiera ona na bieżąco informacje o ID przycisku ostatnio najechanego, o tym przez jaki czas dany przycisk jest już pod punktem fiksacji, o aktualnym stanie klawiatury, o ostatnio wybranym specjalnym przycisku (np. CapsLock) oraz przez jaki czas działanie tego przycisku się utrzymuje. Większość z tych danych odświeżana jest co 200ms (stała zdefiniowana w pliku ze stałymi) podczas każdego wykonywania się metody

TimerStep(). Jej działanie przedstawiono za pomocą pseudokodu 1.

Algorytm 1 Działanie funkcji TimerStep()

```
pobierz ID aktualnie fiksowanego przycisku
if pobrane ID różne jest od -1 then
  if zwykłe przyciski są w trybie wstrzymania then
    if jeśli aktualnie fiksowany przycisk jest specjalny then
      hoverState (obiekt klasy HoverManager) ustaw aktualnie aktywny
      przycisk na pobrane ID
      Wykonaj krok timera (funkcja executeTimerStep())
      Pokaż czas przez jaki przycisk jest fiksowany na pasku postępu dla
      informacji użytkownika.
    end if
  else
    Wykonaj powyższe funkcje dla wszystkich przycisków - niezależnie od
    tego, czy są specjalne, czy nie.
  end if
end if
Wywołaj funkcję verifyTimerTickCount().
```

Zadaniem wyżej wspomnianej funkcji executeTimerStep() jest sprawdzenie, czy dany przycisk był fiksowany przez odpowiednią ilość czas (zmienna, której wartość zależy od ilości pomyłek popełnionych przez użytkownika i dynamicznie zmieniana podczas korzystania z klawiatury – analizowane to jest w funkcji verifyTimerTickCount()). Jeżeli warunek ten został spełniony to dalszy przebieg działań zależy od tego, czy przycisk był specjalny, czy też nie oraz czy klawiatura nie znajduje się w trybie wstrzymania.

1.1.1 Działanie przycisków specjalnych

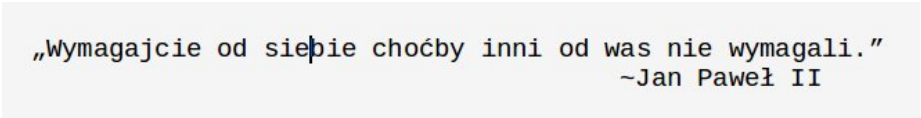
W tabeli 1.1

Nazwa przycisku	Działanie
CapsLock	11C .
Shift	9C
Znaki specjalne	10C
Backspace	.
Wyślij	.
Strzałka w lewo i prawo	.
Menu	.
Przyciski podpowiedzi	.
Wyjdź	.
Stop i start	.
Zmiana trybu wysyłania	.
Wyszukaj na podanej platformie	.
Przesuń się o jedno słow w prawo lub w lewo	.
Czyść	.
Przesuń się w kierunku początku tekstu (home) lub na jego koniec (end)	.

Tab. 1.1 – Lista specjalnych przycisków w raz z ich działaniem.

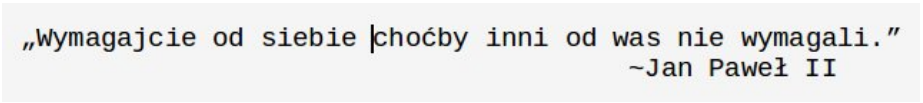
1.2 Praca z tekstem

Wpisywanie tekstu w aplikacji odbywa się poprzez specjalny algorytm kontrolujący zawartość aktualnie pisanego słowa oraz całego tekstu. Podczas każdej chwili działania programu w pamięci przechowywany jest currentWord, czyli aktualne słowo tj. ciąg znaków, liter lub cyfr, które zaczynają się na początku wpisywanego tekstu lub po spacji, a kończą się w pozycji kursora. Dodanie spacji po ciągu znaków kończy słowo i usuwa je ze zmiennej currentWord, a dodaje do zmiennej zwanej wholeText, która przechowuje dotychczas wpisany tekst. Przykładowo jeśli mamy tekst jak na rysunku 1.1 to w zmiennej wholeText przechowywujemy ""Wymagajcie od bie choćby inni od was nie wymagali." Jan Paweł II ", a w currentWord "sie". W ten sposób podpowiedzi generowane będą jedynie dla cząstki "sie", a tekst wpisywany będzie w pozycji kursora, która również monitorowana jest przez zmienną currentPosition. Scalenie wholeText i currenWord następuję, gdy zmieniamy pozycję kursora strzałkami, lub jeśli do tekstu dodana zostanie spacja, a za kursorem nie znajduje się żaden znak przykładowo w takiej sytuacji znajdziemy się na rysunku 1.2.



Rys. 1.1 – Przykład zapisywanie tekstu do zmiennych w zależności od pozycji kursora.

Zmienna currentWord została zespolona z wholeText poprzez wklenie jej na pozycji zapisanej jako currentWordStart. W celu dynamicznego ustala-



Rys. 1.2 – Przykład zapisywanie tekstu do zmiennych w zależności od pozycji kursora.

nia pozycji currentWordStart oraz zawartości currentWord powstały funkcje: getCurrentWordStart() oraz getCurrentWord(). Działanie pierwszej zademonstrowano za pomocą pseudokodu 2. Działanie drugiej sprowadza się do wycięcia

Algorytm 2 Działanie funkcji getCurrentWordStart()

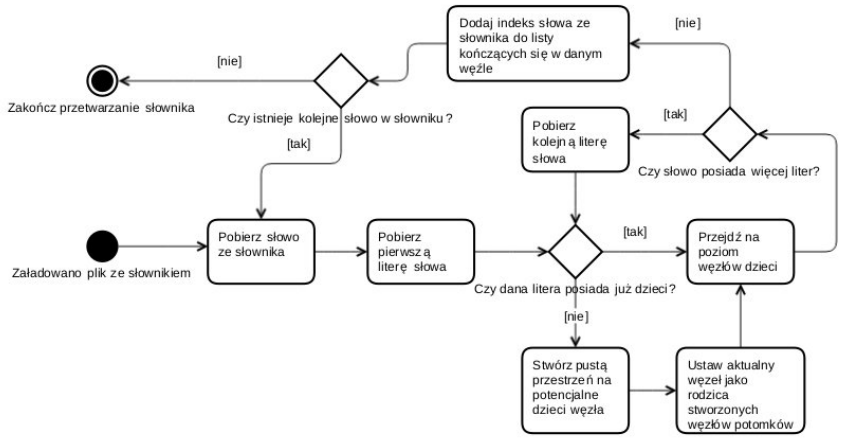
```
if currentWord nie jest pusty && currentPosition nie jest na początku tekstu
then
  for każda pozycja aż do początku tekstu do
    pobierz literę z wholeText w danej pozycji
    if pobrana wartość nie jest liczbą ani literą then
      currentWordStart = danapozycja + 1
    end if
  end for
end if
```

faragmentu tekstu między currentWordStart, zaimplementowanym w wyniku działania poprzedniej funkcji, a currentPosition.

1.3 Trie tree

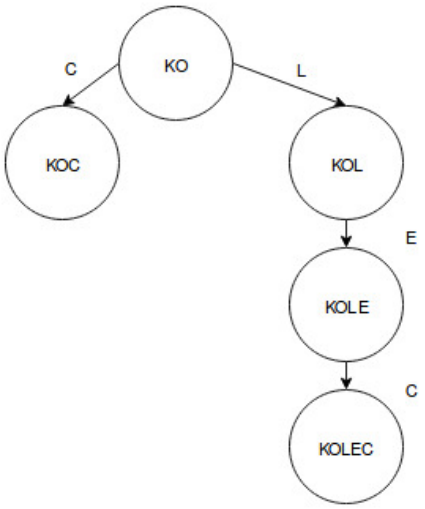
Jak wyżej **??** wspomniano, w pracy wykorzystano drzewo typu Trie w celu pracy z rozległym słownikiem. Słowa z pliku w formacie TXT wczytywane są do programu podczas uruchamiania klawiatury- każda z linii dostarczanego pliku powinna stanowić pojedynczy wyraz. Słowa te, za pomocą sztucznie wygenerowanych list kodujących oraz dekodujących polski alfabet, wprowadzane są do drzewa typu Trie. Drzewo takie powstaje poprzez stworzenie pustego węzła typu node - struktury zadeklarowanej w pliku nagłówkowym klasy obsługującej współpracę ze słownikiem – Dictionary. Struktura node przechowuje informacje o rodzicu bieżącego węzła, o jego potomkach – czyli węzłach następujących oraz wektor zawierający informację o przynależności danego węzła literowego do słowa. Po zainicjowaniu węzła zerowego, po kolei, analizowane są słowa ze słownika w funkcji insertWord().Każe jest rozpatrywane jako tablica liter (char) i iteracyjnie następuje najpierw kodowanie litery na odpowiadający jej indeks (za pomocą mapy alfabet), potem, sprawdzane jest, czy drzewie nie istnieje już węzeł odpowiadający danej wartości. Gdy nie znaleziono gałęzi drzewa pasujących do poszukiwanej wartości tworzy się za pomocą funkcji calloc przestrzeń w pamięci na przyszłe dzieci tego węzła,a jako ich rodzica podaje się aktualnie przeglądany węzeł z literą. Kolejno, niezależnie od wyniku uprzednio rozpatrywanego warunku, o ile słowo wciąż posiada litery do przeglądu, przenosi się o poziom niżej w drzewie (na poziom dziecka uprzednio rozpatrywanej litery) i proces zachodzi od początku. Gdy przetworzono wszystkie litery słowa do

listy wystąpień ostatniego odwiedzonego węzła dopisany zostaje indeks słowa w słowniku – tym samym oznaczając jego koniec. W sposób graficzny przedstawiono działanie danej funkcji na rysunku 1.3



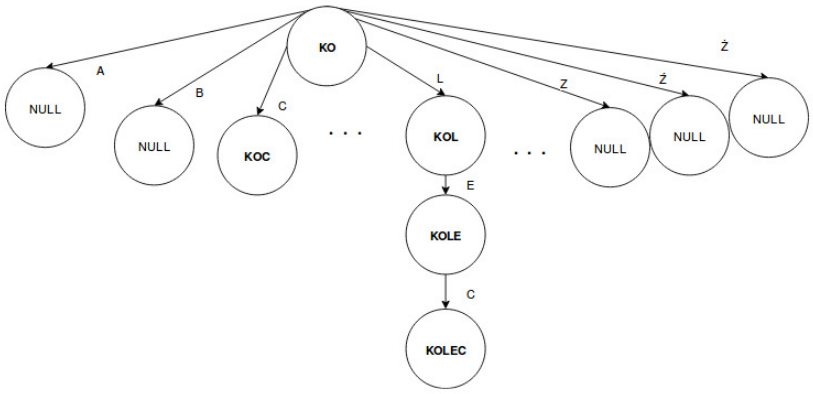
Rys. 1.3 – Diagram przepływu dla funkcji insertWord() wprowadzającej słowa do drzewa typu Trie.

Po uzupełnieniu słownika można przejść do jego wykorzystywania - auto uzupełniania wpisywanego tekstu. Każde wpisanie litery powoduje wywołanie metody updateHints(), która jest odpowiedzialna za tworzenie oraz wyświetlanie podpowiedzi, zgodnie z wprowadzonym do tej pory słowem. Jako poszukiwaną frazę traktujemy ciąg znaków, które użytkownik wpisał do pozycji kursora od ostatniej spacji, bądź początku tekstu. W wypadku, gdy ten ciąg znaków jest dłuższy niż dwa, wykorzystywana jest funkcja komunikująca się ze stworzonym słownikiem – searchWord(). Przekazywane jest drzewo Trie słownika oraz aktualnie poszukiwana fraza (bez formatowania). Wpisywana fraza, również traktowana jest jako zbiór liter, które przeglądane są jedna po drugiej. Dla każdej następuje zmiana dzięki mapie kodującej alfabet na odpowiadający indeks, który umożliwia przeszukiwanie słownika. Sprawdzane jest, czy istnieje potomek drzewa Trie o danym indeksie – jeśli tak, to następuje zmiana węzła na węzeł dziecka, tak, że przy przeszukiwaniu drzewa brane pod uwagę będą jedynie węzły z rodzicem będącym pierwszą literą poszukiwanej frazy. Gdy przeszuka się wszystkie litery, bądź w trakcie tego procesu zabraknie węzłów potomków dla danej kombinacji liter to zwracany jest odpowiednio ostatni węzeł wspólny dla danej frazy lub też węzeł pusty. Przykładowo - załóżmy, że mamy drzewo takie jak na rysunku **??**. Jeśli wyszukamy frazę ”ko” to funkcja wróci nam węzeł i jego dzieci - czyli możliwe auto uzupełnienia wyglądają tak jak na rysunku 1.4. Złożenie końcówek wyrazów zachodzi w funkcji getSimilarEndings(), której przekazywany jest znaleziony węzeł, pusty wektor, do którego mają zostać zapisane wynikowe wyrazy oraz węzeł znaków niezbędny do dekodowania. W pierwszym kroku sprawdzane jest, czy w danym węźle kończą się jakieś słowa, jeśli tak (wektor wystąpień węzła jest różny od zera), to każdą literę zapisaną w wyżej wymienionym wektorze znaków zapisuje się do jednego słowa (tworząc końcówkę do auto uzupełniania). Gotowy ciąg znaków zapisywany

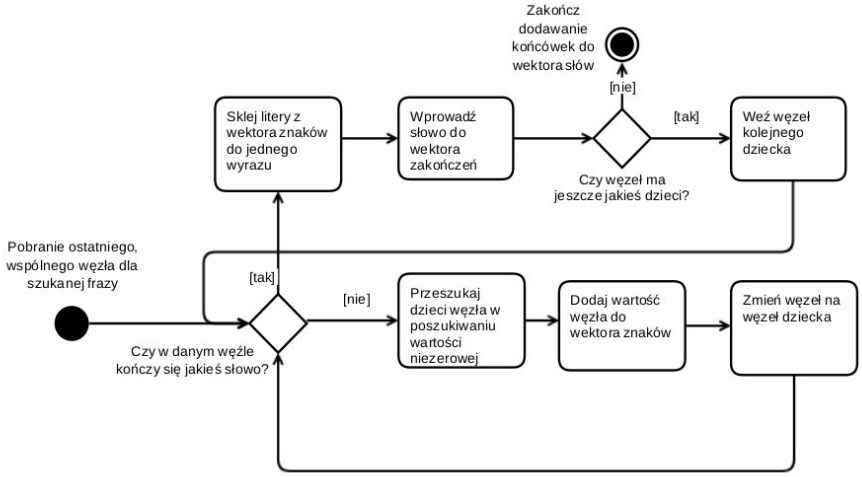


Rys. 1.4 – Przykładowy węzeł do autouzupełniania słów po wpisaniu frazy ”ko”.

jest do wektora z końcówkami. W wypadku pierwszego wykonania się funkcji mamy do czynienia z pustym wektorem znaków- toteż nie zostanie stworzona żadna końcówka. Nawet jeśli ‘’ko” było pełną formą słowa nie powinna ona się wyświetlać w proponowanych opcjach użytkownika. Aby uzupełnić wektor znaków należy przejrzeć przesłany węzeł (ten z rysunku 1.4) – w tym celu sprawdza się, czy dzieci węzła odpowiadającej każdej literze alfabetu nie są puste, gdyż programowe drzewo ma, prócz wcześniej przedstawionych gałęzi, jeszcze 33 (zakładając, że zaimplementowany alfabet posiada 35 liter) nieobdarzone wartością gałęzie przedstawione na rysunku 1.5. Gdy znaleziono element o niezerowej wartości pobierana jest za pomocą mapy symetrycznej (reverseAlfabet) wartość literowa węzła i wprowadzana jest do wektora znaków. Węzeł z ‘’ko” zmieniany jest węzeł pierwszego pierwszego dziecka – w tym wypadku ‘’koc”. Następnie przez rekurencję ponownie rozpoczyna się sprawdzanie, czy dane słowo kończy się w tym węźle, jeśli tak to proces zachodzi według powyżej opisanych kroków, jeśli nie to znów poszukiwany jest węzeł potomny z kolejną literą końcówki słowa do auto uzupełniania. Algorytm przedstawiono w sposób graficzny na rysunku 1.6 Ostatnim krokiem w stworzeniu podpowiedzi jest skrócenie listy końcówek do liczby przycisków przeznaczonych na podpowiedzi oraz zespolenie ich z dotychczas wpisanym słowem. Taki ciąg znaków można przedstawić użytkowniku jako napis na przycisku, który po wybraniu wpisuje reprezentowany tekst do pola tekstowego zamieniając dotychczasową frazę na wybraną oraz dodając znak spacji na końcu nowowybranego słowa.



Rys. 1.5 – Reprezentacja przykładowego węzła widzianego w programie.



Rys. 1.6 – Diagram przepływu dla funkcji getSmiliarEndings().

1.4 Google Api