

AKADEMIA NAUK STOSOWANYCH W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA ZAAWANSOWANE PROGRAMOWANIE

Algorytm listy dwukierunkowej z zastosowaniem GitHub

Autor:
Dawid Michura

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2025

Spis treści

1. Ogólne określenie wymagań	3
1.1. Cel pracy	3
1.2. Zakres funkcjonalny	3
1.3. Przewidywane wyniki	4
2. Analiza problemu	5
2.1. Gdzie używa się list dwukierunkowych	5
2.2. Sposób działania naszego programu	6
2.3. Przykład wykorzystania listy dwukierunkowej	6
2.4. Analiza wzorców projektowych	7
2.5. Narzędzia wspomagające projekt	8
3. Projektowanie	9
3.1. Narzędzia programistyczne	9
3.2. Struktura klas	9
3.3. Praca z kontrolą wersji	10
4. Implementacja	13
4.1. Struktura projektu	13
4.2. Ciekawe fragmenty kodu	13
4.3. Przykładowe wyniki działania	14
5. Wnioski	15
Literatura	16
Spis rysunków	16
Spis tabel	17
Spis listingów	18

1. Ogólne określenie wymagań

Celem tego projektu jest stworzenie **listy dwukierunkowej** w języku C++, która może przechowywać dane dowolnego typu i umożliwiać wygodną pracę z nimi. Lista pozwala na łatwe dodawanie, usuwanie i wstawianie elementów w dowolnym miejscu.

1.1. Cel pracy

Kluczowym celem projektu jest nie tylko stworzenie działającej listy dwukierunkowej, ale również zapewnienie wysokiej jakości kodu poprzez zastosowanie sprawdzonych wzorców projektowych – **fabryki (Factory)**, która upraszcza i ujednolica tworzenie nowych list, **iteratora (Iterator)**, umożliwiającego bezpieczne i wygodne przeglądanie elementów listy w obu kierunkach, oraz **Singleton** Loggera, zapewniającego spójne i centralne logowanie wszystkich operacji w programie.

1.2. Zakres funkcjonalny

Głównym komponentem projektu jest klasa implementująca listę dwukierunkową, która udostępnia pełen zestaw operacji do manipulacji jej zawartością. W ramach wymagań funkcjonalnych lista musi umożliwiać:

- Dodanie elementu na początek listy
- Dodanie elementu na koniec listy
- Dodanie elementu pod wskazany indeks
- Usunięcie elementu z początku listy
- Usunięcie elementu z końca listy
- Usunięcie elementu z pod wskazanego indeksu
- Wyświetlenie całej listy
- Wyświetlenie listy w odwrotnej kolejności
- Wyświetlenie następnego elementu
- Wyświetlenie poprzedniego elementu
- Czyszczenie całej listy

Wszystkie powyższe funkcjonalności zostały przetestowane w funkcji `main`, aby upewnić się, że działają poprawnie w różnych scenariuszach, w tym w przypadkach brzegowych, takich jak operacje na pustej liście lub liście zawierającej tylko jeden element.

1.3. Przewidywane wyniki

Finalnym rezultatem projektu będzie kompletny program, który łączy kilka kluczowych elementów:

- W pełni funkcjonalny program w języku C++, który spełnia wszystkie założenia projektowe i poprawnie realizuje operacje na liście dwukierunkowej
- Kompletny kod źródłowy umieszczony w repozytorium na platformie GitHub, z przejrzystą i logiczną historią zmian (commitów), co umożliwia śledzenie postępów i wprowadzonych modyfikacji
- Automatycznie wygenerowana dokumentacja w formacie PDF (przy użyciu LaTeX i Doxygen), opisująca wszystkie klasy, metody oraz strukturę projektu
- Dokumentacja projektowa, przedstawiająca proces analizy, projektowania i implementacji, a także użyte narzędzia i wzorce projektowe

Podczas tego projektu nauczymy się korzystać z takich narzędzi jak: **Git** (System kontroli wersji, który umożliwia śledzenie zmian w kodzie źródłowym w czasie), **Doxygen** (Narzędzie do automatycznego generowania czytelnej dokumentacji z komentarzy w kodzie źródłowym) oraz **LaTeX** (system składania dokumentów, który pozwala tworzyć profesjonalnie sformatowane teksty, w tym raporty, dokumentacje i prace naukowe)

2. Analiza problemu

2.1. Gdzie używa się list dwukierunkowych

Listy dwukierunkowe są jedną z podstawowych struktur danych w informatyce i mają wiele praktycznych zastosowań. Ich główną zaletą jest możliwość poruszania się w obu kierunkach – zarówno od początku do końca listy, jak i w odwrotną stronę. Dzięki temu są bardziej elastyczne niż listy jednokierunkowe i nadają się do sytuacji, w których często wstawiamy lub usuwamy elementy w środku kolekcji danych. Są powszechnie stosowane:

- **w Systemach operacyjnych** – listy procesów lub wątków w kolejce gotowych zadań, gdzie konieczne jest szybkie dodawanie i usuwanie elementów oraz przemieszczanie się w obu kierunkach.
- **w Edytorach tekstu** – implementacja funkcji cofania i ponawiania operacji (undo/redo), gdzie każdy stan dokumentu może być reprezentowany jako element listy. Dzięki temu można łatwo przechodzić między poprzednim a następnym stanem.
- **w Przeglądarkach internetowych** – historia odwiedzanych stron, gdzie użytkownik może wracać do poprzednich stron lub przechodzić do następnych, co idealnie pasuje do listy dwukierunkowej.
- **w Algorytmach wymagających dynamicznej zmiany danych** – np. symulacje, kolejki priorytetowe czy grafy, gdzie elementy mogą być wstawiane i usuwane w różnych miejscach w trakcie działania programu.

2.2. Sposób działania naszego programu

Projekt składa się z kilku głównych **elementów**:

- **Element listy (`Item<Type>`)** – przechowuje wartość oraz wskaźniki na poprzedni i następny element. Dzięki temu lista może być dwukierunkowa i łatwo przeszukiwana.
- **Lista dwukierunkowa (`ListTwoway<Type>`)** – przechowuje wskaźniki do pierwszego (**head**) i ostatniego (**tail**) elementu oraz liczbę elementów (**size**). Umożliwia dodawanie, usuwanie i wstawianie elementów w dowolne miejsce.
- **Iterator (`ListIterator<Type>`)** – pozwala przechodzić po liście w obu kierunkach i wypisywać jej zawartość. Można też resetować pozycję iteratora na początek.
- **Logger (`Logger`)** – singleton, który zapisuje wszystkie operacje w konsoli, dzięki czemu można śledzić działanie programu i diagnozować ewentualne błędy.

2.3. Przykład wykorzystania listy dwukierunkowej

Aby lepiej zrozumieć działanie listy dwukierunkowej, warto przedstawić prosty przykład. Załóżmy, że tworzymy listę liczb całkowitych:

1. Tworzymy pustą listę.
2. Dodajemy elementy na koniec listy kolejno: **1024**, **2048**, **2048**
Nasza lista wygląda tak: **1024 ↔ 2048 ↔ 2048**
3. Dodajemy element na początek listy: **512**
Nasza lista wygląda tak: **512 ↔ 1024 ↔ 2048 ↔ 2048**
4. Dodajemy element na koniec listy: **8192**
Nasza lista wygląda tak: **512 ↔ 1024 ↔ 2048 ↔ 2048 ↔ 8192**
5. Wstawiamy element 4096 przed ostatnim elementem (czyli przed 8192)
Nasza lista wygląda tak: **512 ↔ 1024 ↔ 2048 ↔ 2048 ↔ 4096 ↔ 8192**
6. Usuwamy pierwszy element (**512**) i ostatni element (**8192**)
1024 ↔ 2048 ↔ 2048 ↔ 4096

2.4. Analiza wzorców projektowych

1. **Iterator**¹ - Iterator umożliwia sekwencyjne przeglądanie elementów kolekcji bez ujawniania jej wewnętrznej struktury. Oddziela sposób przechowywania danych od sposobu ich przeglądania
 - **Problem:** Bez iteratora przeglądanie listy wymagałoby ręcznego operowania wskaźnikami lub indeksami, co jest podatne na błędy.
 - **Rozwiązanie:** W projekcie klasa ListIterator pozwala bezpiecznie poruszać się po liście dwukierunkowej w obu kierunkach oraz wyświetlać jej zawartość.
 - **Efekt:** Operacje na liście są eleganckie i czytelne, użytkownik klasy nie musi znać szczegółów implementacji listy.
2. **Singleton**² - Singleton zapewnia istnienie tylko jednej instancji danej klasy w całym programie, która jest globalnie dostępna. Umożliwia centralne zarządzanie zasobami lub usługami, takimi jak logowanie.
 - **Problem:** W projekcie potrzebny jest centralny mechanizm logowania wszystkich operacji listy. Tworzenie wielu instancji loggera mogłoby prowadzić do niespójności.
 - **Rozwiązanie:** Klasa Logger implementuje Singleton, gwarantując istnienie jednej, wspólnej instancji dostępnej w całym programie.
 - **Efekt:** Wszystkie logi trafiają do jednej instancji, co ułatwia diagnostykę i testowanie programu.

¹Opis wzorca Iterator [link](#) (term. wiz. 25. 10. 2025)

²Opis wzorca Singleton [link](#) (term. wiz. 25. 10. 2025)

3. **Factory (Fabryka)**³ - Wzorzec Factory pozwala na tworzenie obiektów bez ujawniania dokładnej klasy. Umożliwia odseparowanie logiki tworzenia obiektu od jego użycia, co zwiększa modularność kodu.

- **Problem:** Bezpośrednie wywoływanie konstruktorów w różnych miejscach programu może prowadzić do duplikacji kodu, błędów i utrudnia testowanie.
- **Rozwiązanie:** W projekcie ListTwowayFactory centralizuje tworzenie instancji listy dwukierunkowej, eliminując potrzebę bezpośredniego wywoływania konstruktora.
- **Efekt:** Tworzenie nowych list jest jednolite, bezpieczne i łatwiejsze do zarządzania.

2.5. Narzędzia wspomagające projekt

1. **Git**⁴ – system kontroli wersji, który umożliwia śledzenie wszystkich zmian w kodzie źródłowym. Pozwala na tworzenie gałęzi, testowanie nowych funkcjonalności i bezpieczny powrót do wcześniejszych wersji. Dzięki Git możliwe jest również wygodne współdzielenie projektu w zespole.
2. **GitHub**⁵ – platforma do hostowania repozytoriów Git. Umożliwia przechowywanie kodu w chmurze, śledzenie historii commitów, tworzenie pull requestów i zarządzanie zadaniami projektowymi.
3. **Doxygen** – narzędzie do automatycznego generowania dokumentacji z komentarzy w kodzie źródłowym. Pozwala tworzyć czytelną dokumentację w formacie HTML, PDF lub LaTeX, co ułatwia utrzymanie i zrozumienie kodu.

³Opis wzorca Factory [link](#) (term. wiz. 25. 10. 2025)

⁴Dokumentacja Git'a [link](#) (term. wiz. 25. 10. 2025)

⁵Informacje o Github [link](#) (term. wiz. 25. 10. 2025)

3. Projektowanie

3.1. Narzędzia programistyczne

- **Język:** C++17
- **Biblioteki:** iostream, string
- **Kompilator:** g++ 12.2 lub nowszy
- **System kontroli wersji:** Git
- **Edytor tekstowy:** Visual Studio Code⁶
- **System operacyjny:** Windows/Linux/macOS

3.2. Struktura klas

1. Klasa **Item**

- Przechowuje dane i wskaźniki na elementy poprzedni i następny.
- Konstruktor inicjalizuje element z danymi i wskaźnikami ustawionymi na nullptr.

2. Klasa **ListTwoWay**

- Właściwości: head, tail, size
- Metody: push, unshift, insertAt, removeAt, pop, shift, clear, getSize, getHead, getTail
- Obsługuje dynamiczne dodawanie i usuwanie elementów.

3. Klasa **ListIterator**

- Właściwości: current, list
- Metody: hasNext(), next(), reset(), printAll(), printAllReverse()

4. Klasa **Logger**

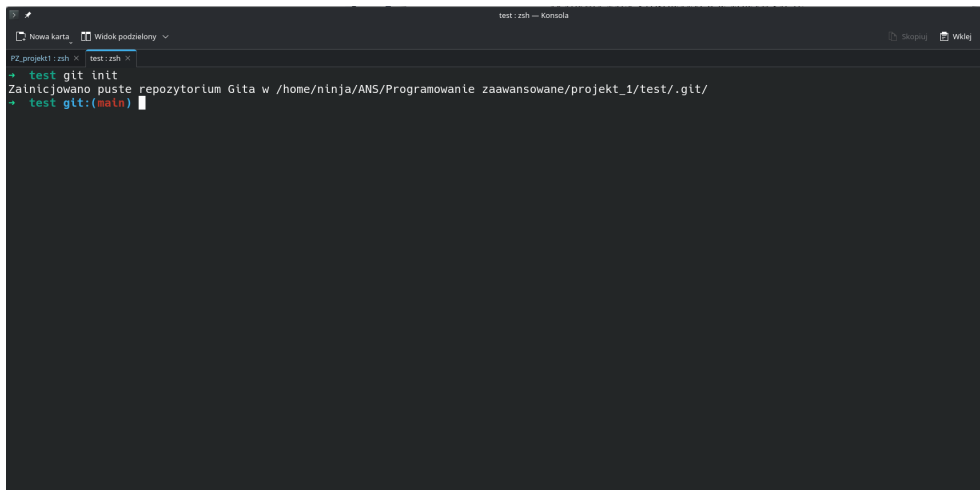
- Singleton, zapewnia logowanie w konsoli.
- Metoda log(string msg) wypisuje

⁶Oficjalna strona Visual Studio Code [link](#)

3.3. Praca z kontrolą wersji

Etap 1: Tworzenie repozytorium

`git init` – tworzy nowe repozytorium Git w bieżącym katalogu, rozpoczynając śledzenie zmian w projekcie.

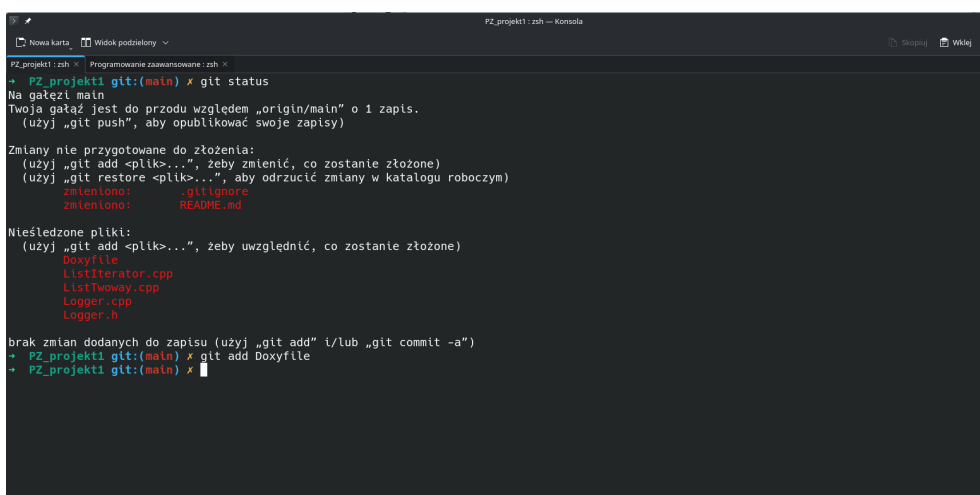


```
test:zsh ~  
+ test git init  
Zainicjowano puste repozytorium Gita w /home/ninja/ANS/Programowanie zaawansowane/projekt_1/test/.git/  
+ test git:(main)
```

Rys. 3.1. Wykonanie komendy `git init`

Etap 2: Dodawanie zmian

`git add <plik>` – dodaje wybrane pliki do obszaru przechowywania zmian (staging area), przygotowując je do zapisania w repozytorium.

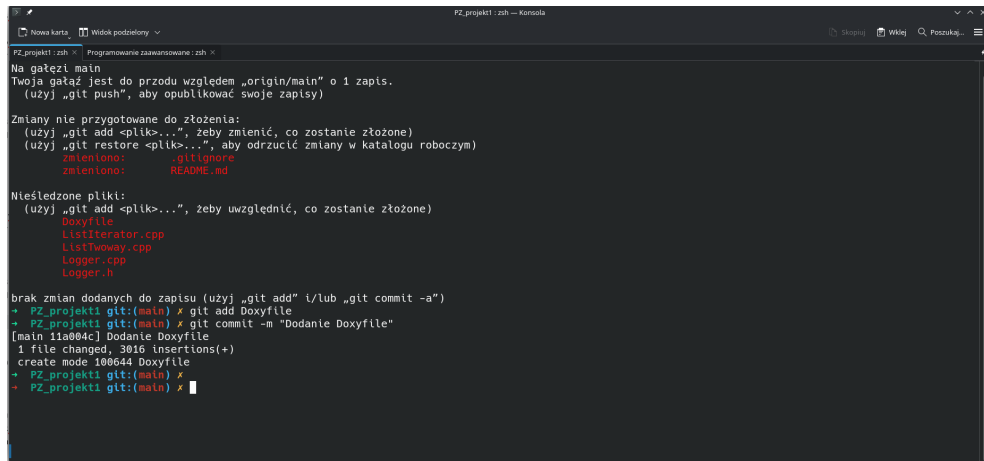


```
PZ_projekt1:zsh ~  
+ PZ_projekt1 git:(main) git status  
Na gałęzi main  
Twoja gałąź jest do przodu względem „origin/main” o 1 zapis.  
(użyj „git push”, aby opublikować swoje zapisy)  
  
Zmiany nie przygotowane do złożenia:  
(użyj „git add <plik>...”, żeby zmienić, co zostanie złożone)  
(użyj „git restore <plik>..”, aby odrzucić zmiany w katalogu roboczym)  
zmieniono: .gitignore  
zmieniono: README.md  
  
Niesledzone pliki:  
(użyj „git add <plik>...”, żeby uwzględnić, co zostanie złożone)  
Doxyfile  
ListIterator.cpp  
ListTwoWay.cpp  
Logger.cpp  
Logger.h  
  
brak zmian dodanych do zapisu (użyj „git add” i/lub „git commit -a”)  
+ PZ_projekt1 git:(main) git add Doxyfile  
+ PZ_projekt1 git:(main)
```

Rys. 3.2. Wykonanie komendy `git add`

Etap 3: Zapisywanie zmian (commit)

`git commit -m "komentarz"` – zapisuje zmiany w repozytorium z opisem wprowadzonych modyfikacji.



```
PZ_projekt1 : zsh — Konsola
PZ_projekt1 : zsh x Programowanie zaawansowane : zsh x
Na gałęzi main
Twoja gałąź jest do przodu względem „origin/main” o 1 zapis.
(użyj „git push”, aby opublikować swoje zapisy)

Zmiany nie przygotowane do złożenia:
(użyj „git add <plik>...”, żeby zmienić, co zostanie złożone)
(użyj „git restore <plik>...”, aby odrzucić zmiany w katalogu roboczym)
zmieniono: .gitignore
zmieniono: README.md

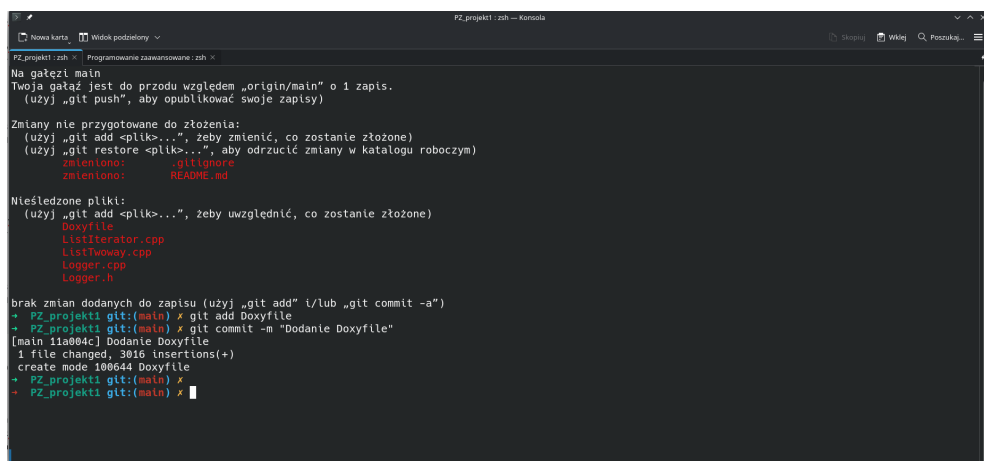
Nieśledzone pliki:
(użyj „git add <plik>...”, żeby uwzględnić, co zostanie złożone)
Doxyfile
ListIterator.cpp
ListTwoWay.cpp
Logger.cpp
Logger.h

brak zmian dodanych do zapisu (użyj „git add” i/lub „git commit -a”)
+ PZ_projekt1 git:(main) x git add Doxyfile
+ PZ_projekt1 git:(main) x git commit -m "Dodanie Doxyfile"
[main 11a084c] Dodanie Doxyfile
1 file changed, 3016 insertions(+)
create mode 100644 Doxyfile
+ PZ_projekt1 git:(main) x
+ PZ_projekt1 git:(main) x
```

Rys. 3.3. Wykonanie komendy `git commit -m "Dodanie Doxyfile"`

Etap 4: Sprawdzanie stanu repozytorium

`git status` – pokazuje aktualny stan repozytorium, m.in. zmodyfikowane i nieśledzone pliki oraz pliki w staging area.



```
PZ_projekt1 : zsh — Konsola
PZ_projekt1 : zsh x Programowanie zaawansowane : zsh x
Na gałęzi main
Twoja gałąź jest do przodu względem „origin/main” o 1 zapis.
(użyj „git push”, aby opublikować swoje zapisy)

Zmiany nie przygotowane do złożenia:
(użyj „git add <plik>...”, żeby zmienić, co zostanie złożone)
(użyj „git restore <plik>...”, aby odrzucić zmiany w katalogu roboczym)
zmieniono: .gitignore
zmieniono: README.md

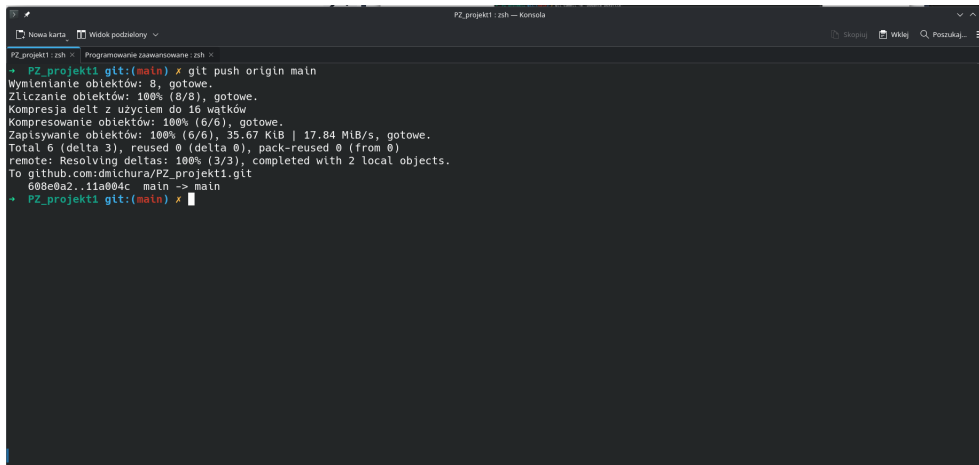
Nieśledzone pliki:
(użyj „git add <plik>...”, żeby uwzględnić, co zostanie złożone)
Doxyfile
ListIterator.cpp
ListTwoWay.cpp
Logger.cpp
Logger.h

brak zmian dodanych do zapisu (użyj „git add” i/lub „git commit -a”)
+ PZ_projekt1 git:(main) x git add Doxyfile
+ PZ_projekt1 git:(main) x git commit -m "Dodanie Doxyfile"
[main 11a084c] Dodanie Doxyfile
1 file changed, 3016 insertions(+)
create mode 100644 Doxyfile
+ PZ_projekt1 git:(main) x
+ PZ_projekt1 git:(main) x
```

Rys. 3.4. Wykonanie komendy `git status`

Etap 5: Wysyłanie zmian na zdalne repozytorium

`git push origin main` – wysyła lokalne commity na platformę GitHub, udostępniając zmiany innym członkom zespołu i zachowując kopię w chmurze.



```
PZ_projekt1: zsh % git push origin main
PZ_projekt1 git:(main) * git push origin main
Wymienianie obiektów: 100% (8/8), gotowe.
Zliczanie obiektów: 100% (8/8), gotowe.
Kompresja delt z użyciem do 16 wątków
Kompresowanie obiektów: 100% (6/6), gotowe.
Zapisywanie obiektów: 100% (6/6), 35.67 KiB | 17.84 MiB/s, gotowe.
Total 6 (delta 3), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (3/3), completed with 2 local objects.
To github.com:dmichura/PZ_projekt1.git
608e8a2..11a004c main -> main
PZ_projekt1 git:(main) *
```

Rys. 3.5. Wykonanie komendy `git push origin main`

Przydatne komendy

- Cofnięcie się o dwa commity
`git reset --soft HEAD 2`
- Pobranie repozytorium:
`git clone https://github.com/dmichura/PZ_projekt1.git`
- Otwarcie 4 ostatnich commitów, umożliwia usunięcie wybranego commita (zamieniając „pick” na „drop”)
`git rebase -i HEAD 4`
- Ustawienie głównej gałęzi repozytorium na „main”
`git branch -M main`
- Pobranie najnowszych zmian z GitHuba do lokalnego projektu
`git pull`
- Przywraca konkretny plik z ostatniej wersji zdalnej
`git checkout origin/main -- plik`

Podsumowanie

Dzięki temu prostemu workflow można efektywnie śledzić historię projektu, testować różne rozwiązania i współpracować z innymi programistami w bezpieczny sposób.

4. Implementacja

4.1. Struktura projektu

Projekt został zaimplementowany w języku C++ i składa się z kilku kluczowych klas:

- **Item** – klasa szablonowa reprezentująca pojedynczy węzeł listy dwukierunkowej. Przechowuje dane oraz wskaźniki na poprzedni i następny element.
- **ListTwoway** – główna klasa listy dwukierunkowej, umożliwiająca dodawanie, usuwanie i wstawianie elementów w dowolne miejsce.
- **ListIterator** – implementacja wzorca Iterator, pozwalająca wygodnie przeglądać listę w obu kierunkach oraz wypisywać jej zawartość.
- **ListTwowayFactory** – fabryka (wzorzec Factory) do tworzenia nowych instancji listy.
- **Logger** – singleton odpowiedzialny za logowanie wszystkich operacji na liście.

Kluczowym celem implementacji było połączenie funkcjonalności listy z zastosowaniem wzorców projektowych, co pozwala na czytelny i bezpieczny kod.

4.2. Ciekawe fragmenty kodu

```
1 template<typename Type>
2 void ListTwoway<Type>::insertAt(int index, Type data)
3 {
4     if (index < 0 || index > size) {
5         Logger::getInstance()->log("Index jest niepoprawny!\nIndex
6 musi byc >= 0 i nie moze byc > size!\n");
7         return;
8     }
9     if (index == 0) {
10         unshift(data);
11         return;
12     }
13
14     if (index == size) {
15         push(data);
16         return;
```

```
17     }
18
19     Item<Type>* newItem = new Item<Type>(data);
20     Item<Type>* current = head;
21
22     for (int i = 0; i < index - 1; ++i) {
23         current = current->next;
24     }
25
26     newItem->next = current->next;
27     if (current->next) {
28         current->next->prev = newItem;
29     }
30
31     current->next = newItem;
32     newItem->prev = current;
33
34     size++;
35 }
```

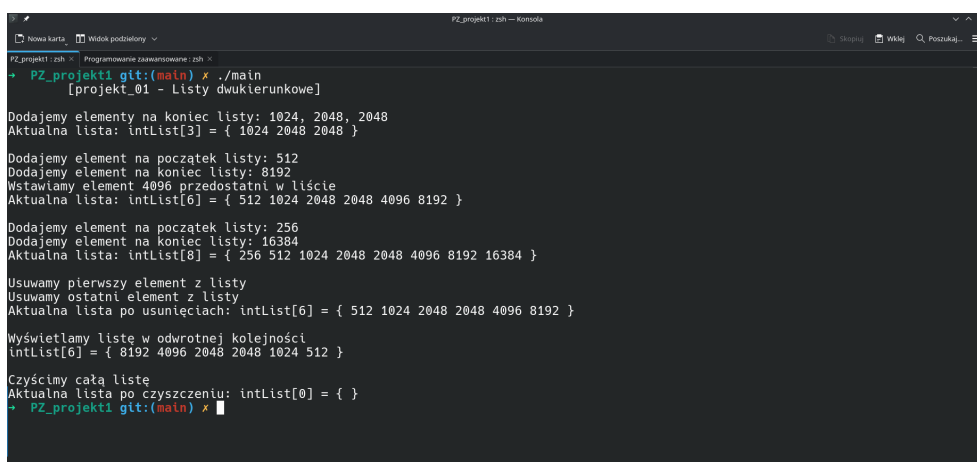
Listing 1. Metoda odpowiadająca za dodanie Itemu w danym indeksie

Metoda `insertAt` umożliwia wstawienie nowego elementu w dowolne miejsce listy dwukierunkowej, zgodnie z podanym indeksem. Funkcja najpierw sprawdza poprawność indeksu – jeśli jest nieprawidłowy, generowany jest komunikat przez singleton `Logger`.

W przypadku indeksu równego 0 element jest dodawany na początek listy przy użyciu metody `unshift`, natomiast indeks równy `size` powoduje dodanie elementu na końcu listy metodą `push`.

Dla indeksów wewnątrz listy tworzony jest nowy węzeł, a wskaźniki `prev` i `next` są odpowiednio aktualizowane, aby zachować spójność struktury dwukierunkowej.

4.3. Przykładowe wyniki działania



```
PZ_projekt1:zh -- Konsola
PZ_projekt1:zh x Programowanie zaawansowane: zh x
+ PZ_projekt1 git:(main) x ./main
[projekt_01 - Listy dwukierunkowe]

Dodajemy elementy na koniec listy: 1024, 2048, 2048
Aktualna lista: intList[3] = { 1024 2048 2048 }

Dodajemy element na początek listy: 512
Dodajemy element na koniec listy: 8192
Wstawiamy element 4096 przedostatni w liście
Aktualna lista: intList[6] = { 512 1024 2048 2048 4096 8192 }

Dodajemy element na początek listy: 256
Dodajemy element na koniec listy: 16384
Aktualna lista: intList[8] = { 256 512 1024 2048 2048 4096 8192 16384 }

Usuujemy pierwszy element z listy
Usuujemy ostatni element z listy
Aktualna lista po usunięciach: intList[6] = { 512 1024 2048 2048 4096 8192 }

Wyświetlamy listę w odwrotnej kolejności
intList[6] = { 8192 4096 2048 2048 1024 512 }

Czyścimy całą listę
Aktualna lista po czyszczeniu: intList[0] = { }
+ PZ_projekt1 git:(main) x
```

Rys. 4.1. Przykładowa lista dwukierunkowa

5. Wnioski

Projekt realizujący listę dwukierunkową w języku C++ pozwolił na zdobycie praktycznej wiedzy zarówno z zakresu struktur danych, jak i wzorców projektowych. Dzięki implementacji listy z użyciem Iteratora, Fabryki oraz Singletona udało się uzyskać kod, który jest jednocześnie funkcjonalny, bezpieczny i czytelny. Główne wnioski z przeprowadzonego projektu to: Zrozumienie działania list dwukierunkowych – implementacja własnej listy pozwoliła poznać szczegóły zarządzania wskaźnikami, dodawania, usuwania i wstawiania elementów w różnych miejscach listy. Znaczenie wzorców projektowych – zastosowanie Fabryki uprościło tworzenie obiektów, Iterator umożliwił bezpieczne przeglądanie listy, a Singleton zagwarantował centralne logowanie wszystkich operacji. Znaczenie narzędzi wspierających projekt – Git i GitHub ułatwiły śledzenie zmian i współpracę, Doxygen pozwolił na wygenerowanie przejrzystej dokumentacji kodu, a LaTeX umożliwił przygotowanie profesjonalnego raportu. Testowanie i stabilność kodu – przetestowanie wszystkich funkcji listy w różnych scenariuszach, w tym na liście pustej i z jednym elementem, potwierdziło poprawność implementacji i odporność na błędy. Podsumowując, projekt nie tylko spełnił wszystkie założenia funkcjonalne, ale również pokazał, jak istotne jest stosowanie dobrych praktyk programistycznych i wzorców projektowych w tworzeniu skalowalnego, łatwego w utrzymaniu kodu.

Spis rysunków

3.1. Wykonanie komendy <code>git init</code>	10
3.2. Wykonanie komendy <code>git add</code>	10
3.3. Wykonanie komendy <code>git commit -m "Dodanie Doxyfile"</code>	11
3.4. Wykonanie komendy <code>git status</code>	11
3.5. Wykonanie komendy <code>git push origin main</code>	12
4.1. Przykładowa lista dwukierunkowa	15

Spis tabel

Spis listingów

1. Metoda odpowiadająca za dodanie Itemu w danym indeksie 13