# DOKUMENTACJA TECHNICZNA PROJEKTU

16 stycznia 2025

# Spis treści

1. Struktura i komponenty projektu 4
   1. [Cel projektu 4](#_TOC_250027)
   2. [Opis ogólny 4](#_TOC_250026)
   3. [Środowisko i wymagania 5](#_TOC_250025)
   4. [Instrukcja obsługi 5](#_TOC_250024)
      1. [Instalacja i uruchomienie programu 5](#_TOC_250023)
      2. [Konfiguracja parametrów 6](#_TOC_250022)
      3. [Uruchomienie symulacji 6](#_TOC_250021)
      4. [Monitorowanie postępu 6](#_TOC_250020)
      5. [Zapis wyników 6](#_TOC_250019)
   5. [Struktura projektu 6](#_TOC_250018)
      1. [Struktura katalogów 6](#_TOC_250017)
      2. [Pliki źródłowe i ich funkcje 7](#_TOC_250016)
   6. [Architektura programu 9](#_TOC_250015)
2. Szczegóły techniczne implementacji 10
   1. [Graficzne przedstawienie przepływu danych w aplikacji 10](#_TOC_250014)
      1. [Opis etapów działania aplikacji 10](#_TOC_250013)
      2. [Schemat przepływu danych 11](#_TOC_250012)
   2. [Zarządzanie symulacjami przy użyciu puli wątków i bufora 13](#_TOC_250011)
   3. [Szczegółowy opis schematu przepływu danych i symulacji w oprogramowaniu 14](#_TOC_250010)
   4. [Omówienie interfejsu graficznego aplikacji 18](#_TOC_250009)
      1. [Przegląd interfejsu użytkownika 18](#_TOC_250008)
      2. [Podsumowanie 20](#_TOC_250007)
   5. [Opis kontenerów danych 21](#_TOC_250006)
      1. Klasa Ubezpieczyciel 21
      2. Klasa VectorSim 22

2

* + 1. Klasa VectorPozarPierwotny 24
    2. [Klasa VectorPozarRozprzestrzeniony 25](#_TOC_250005)
  1. [Opis buforów danych i mechanizmów synchronizacji 26](#_TOC_250004)
     1. Struktury danych: BuforPierwotny i BuforRozprz 27
     2. [Globalne bufory i mechanizmy synchronizacji 27](#_TOC_250003)
     3. [Funkcje dodające dane do buforów 28](#_TOC_250002)
     4. [Wątki zapisujące dane 29](#_TOC_250001)
  2. [Opis działania funkcji 32](#_TOC_250000)
     1. Funkcja testAll 32
     2. Funkcja simulateExponsureTEST 40
     3. Funkcja randZeroToOne 42
     4. Funkcja sample vec 43
     5. Funkcja randBin 43
     6. Funkcja search closest 44
     7. Funkcja percentage of loss 45
     8. Funkcja calc reas bligator 45
     9. Funkcja reasecuration build fire 47
     10. Funkcja mean spread function 53
     11. Funkcja index in ring 56
     12. Funkcja render gui() 57

**Rozdział 1**

# Struktura i komponenty projektu

## Cel projektu

Celem projektu jest stworzenie symulatora pożarów, który pozwala na analizę ryzyka wystą- pienia pożaru na danym obszarze dla podanych ubezpieczycieli. Symulator wykorzystuje dane dotyczące położenia budynków, ryzyka pożarowego oraz parametrów reasekuracyjnych do prze- prowadzenia symulacji, które umożliwiają ocenę potencjalnych strat.

## Opis ogólny

Projekt składa się z kilku kluczowych komponentów, w tym klas do przechowywania da- nych, funkcji do przetwarzania danych wejściowych, funkcji symulacyjnych oraz interfejsu użyt- kownika opartego na bibliotece **ImGui**. Klasy takie jak VectorSim, VectorPozarPierwotny, VectorPozarRozprzestrzeniony oraz Ubezpieczyciel są odpowiedzialne za organizację i przechowywanie danych związanych z symulacjami pożarów oraz informacjami o ubezpieczycie- lach. Program zawiera również funkcje, które wczytują i przetwarzają dane wejściowe z plików CSV, co umożliwia załadowanie istotnych informacji o pożarach i ubezpieczeniach. Zestaw funk- cji symulacyjnych przeprowadza obliczenia dotyczące ryzyka pożarowego oraz rozprzestrzenienia pożarów, pozwalając na symulację różnych scenariuszy i analizę wyników. Interfejs użytkow- nika, stworzony z wykorzystaniem biblioteki ImGui, zapewnia łatwy dostęp do funkcji pro- gramu, umożliwiając konfigurację parametrów wejściowych, wczytywanie danych, uruchamia- nie symulacji oraz zapis wyników. Dodatkowo, program został zaprojektowany jako aplikacja wielowątkowa, co pozwala na równoległe przetwarzanie danych, a zastosowanie odpowiednich mechanizmów zapewnia synchronizację wątków, gwarantując bezpieczeństwo danych i efektyw-

4

ność wykonywanych operacji. Całość projektu tworzy złożony system, który umożliwia analizę ryzyka pożarowego dla ubezpieczeń, wspierając użytkowników w podejmowaniu lepszych decyzji na podstawie symulacji i analiz danych.

## Środowisko i wymagania

##### Wymagania sprzętowe

Współczesny komputer z systemem operacyjnym Windows 7

Procesor wielordzeniowy

Minimum 8 GB RAM

Dysk SSD zalecany

##### Wymagania programowe

Kompilator C++ (MSVC zawarty w Visual Studio 2022)

Biblioteki: Boost, GLFW, ImGui, BS thread pool (domyślnie są załączone do projektu)

## Instrukcja obsługi

### Instalacja i uruchomienie programu

* + - 1. Skopiuj wszystkie pliki projektu, w tym plik rozwiązania .sln, na swój komputer. Upewnij się, że struktura katalogów jest zachowana.
      2. Upewnij się, że masz zainstalowane *Visual Studio 2022 Community*. Podczas instalacji wybierz opcję *„Desktop development with C++”*, aby zainstalować wszystkie niezbędne narzędzia i biblioteki.
      3. Otwórz plik .sln projektu w *Visual Studio*.
      4. Wybierz odpowiednią konfigurację (Release) oraz platformę docelową (x64) z paska na- rzędzi.
      5. Kliknij *„Build”* → *„Build Solution”* lub użyj skrótu klawiszowego Ctrl + Shift + B, aby skompilować projekt.

### Konfiguracja parametrów

Uruchom program i skonfiguruj parametry symulacji za pomocą interfejsu użytkownika. W interfejsie użytkownika dostępne są wszystkie opcje pozwalające dostosować przebieg symulacji do indywidualnych potrzeb.

### Uruchomienie symulacji

Aby rozpocząć proces symulacji, wykonaj następujące kroki:

* + - 1. Kliknij przycisk *”Wczytaj listę ubezpieczycieli”*.
      2. Następnie wybierz opcję *„Wczytaj dane”*.
      3. Na koniec kliknij *„Włącz symulację”*, aby uruchomić symulację.

### Monitorowanie postępu

Podczas symulacji program wyświetla pasek postępu, który wizualizuje, jak daleko jest w pro- cesie symulacji. Dzięki temu użytkownik może na bieżąco monitorować stan wykonywanych obliczeń.

### Zapis wyników

Po zakończeniu symulacji wyniki są automatycznie zapisywane w wybranym katalogu w formie plików CSV. Pliki te można następnie otworzyć w arkuszu kalkulacyjnym lub innym programie do analizy danych.

## Struktura projektu

### Struktura katalogów

Projekt posiada następującą strukturę katalogów:

src/ - Katalog zawierający pliki źródłowe programu.

include/ - Katalog zawierający pliki nagłówkowe.

data/ - Katalog zawierający dane wejściowe w formacie CSV.

output/ - Katalog, w którym zapisywane są wyniki symulacji.

### Pliki źródłowe i ich funkcje

Projekt wykorzystuje kilka plików źródłowych, które pełnią różne role w implementacji pro- gramu. Poniżej znajduje się ich opis:

main.cpp - Główny plik programu, zawierający:

* Funkcje do obsługi interfejsu użytkownika.
* Funkcje do przetwarzania danych wejściowych.
* Funkcje symulacyjne.
* Algorytmy rozprzestrzeniania się pożarów.
* Obliczenia związane z wpływem pożarów na ubezpieczenia.
* Obsługę parametrów symulacji.
* Wczytywaniem danych wejściowych z plików CSV.
* Przetwarzaniem danych na potrzeby symulacji.
* Zapisywaniem wyników symulacji w plikach wyjściowych.
* Obsługa elementów graficznych z wykorzystaniem biblioteki ImGui.
* Reakcje na akcje użytkownika, takie jak wczytywanie danych, konfiguracja symulacji czy uruchamianie procesu symulacyjnego.
* Deklaracje i inicjalizacje zmiennych, m.in. takich jak:
  + std::atomic<double> stanSymulacji = 0.0;
  + std::atomic<double> stanSymulacjiZapisu = 0.0;
  + std::atomic<double> stanSymulacjiZapisuSzkod = 0.0;
  + std::atomic<int> licznik sym = 0;
  + ...
* Obsługę bibliotek, w tym:

##### Standardowe biblioteki C++:

<atomic> - Używana do obsługi zmiennych atomowych, co zapewnia bez- pieczną współpracę wątków w środowisku wielowątkowym.

<chrono> - Umożliwia precyzyjne zarządzanie czasem i pomiarami czasu, np. do mierzenia czasu trwania symulacji.

<iostream> - Służy do obsługi wejścia i wyjścia, np. wypisywania komuni- katów na konsolę.

<vector> - Wykorzystywana do przechowywania i manipulacji dynamicz- nymi tablicami danych.

<string> - Służy do obsługi ciągów znaków.

<thread> - Umożliwia tworzenie i zarządzanie wątkami w programie.

<future> - Obsługuje obiekty (std::future), które są wykorzystywane do zarządzania wynikami asynchronicznych obliczeń.

<random> - Używana do generowania liczb losowych w procesie symulacji.

<filesystem> - Obsługuje operacje na systemie plików, takie jak wczyty- wanie i zapisywanie danych.

<mutex> - Zapewnia mechanizmy synchronizacji w środowisku wielowątko- wym.

<deque> - Umożliwia efektywne przechowywanie i manipulację danymi w dwukierunkowej kolejce.

<condition variable> - Służy do zarządzania współbieżnym dostępem do zasobów przez wątki.

##### \* Zewnętrzne biblioteki:

csvstream.hpp - Biblioteka do wygodnego odczytu i zapisu danych w for- macie CSV.

BS thread pool.hpp - Biblioteka implementująca pulę wątków (*thread pool* ), co pozwala na efektywne zarządzanie zadaniami w środowisku wielowątko- wym. W projekcie jest wykorzystywana do równoległego przetwarzania za- dań symulacyjnych.

boost/random/beta distribution.hpp - Moduł z biblioteki Boost, który implementuje rozkład beta. W projekcie służy do generowania danych loso- wych o określonych parametrach statystycznych.

##### \* Biblioteki graficzne:

imgui.h, imgui impl glfw.h, imgui impl opengl2.h - Biblioteki wykorzy- stywane do tworzenia interfejsu graficznego użytkownika (GUI). ImGui umoż- liwia tworzenie nowoczesnych i interaktywnych elementów interfejsu.

GLFW/glfw3.h - Biblioteka do obsługi okien i kontekstu OpenGL, używana w połączeniu z ImGui.

Pliki związane z biblioteką ImGui:

* imgui.cpp, imgui draw.cpp, imgui tables.cpp, imgui widgets.cpp - Pliki imple- mentujące podstawowe funkcje graficzne interfejsu użytkownika.
* imgui impl glfw.cpp, imgui impl opengl2.cpp, imgui impl opengl3.cpp - Pliki implementujące integrację ImGui z bibliotekami GLFW oraz OpenGL.

## Architektura programu

Program został zaprojektowany w oparciu o trzy kluczowe komponenty:

**Zarządzanie danymi** - odpowiedzialne za wczytywanie, przetwarzanie i zapisywanie danych.

**Symulacja** - wykonuje obliczenia związane z rozprzestrzenianiem się pożarów i ich wpły- wem na ubezpieczenia.

**Interfejs użytkownika** - umożliwia użytkownikowi interakcję z programem i konfigurację symulacji.

Program został zaprojektowany w sposób modularny, co ułatwia jego rozwój i modyfikacje.

**Rozdział 2**

# Szczegóły techniczne implementacji

## Graficzne przedstawienie przepływu danych w aplika- cji

Proces działania aplikacji został przedstawiony na ogólnym diagramie, który ilustruje główne etapy pracy programu oraz przepływ danych. Diagram ten obrazuje logiczną sekwencję dzia- łań, od konfiguracji danych wejściowych, przez przygotowanie i przetwarzanie danych, aż po symulację oraz zapis wyników.

### Opis etapów działania aplikacji

##### Konfiguracja parametrów danych wejściowych

Pierwszy etap działania aplikacji obejmuje ustawienia początkowe, które użytkownik musi zdefiniować przed przystąpieniem do symulacji. W ramach tego etapu użytkownik:

Podaje rok, który ma być brany pod uwagę podczas symulacji.

Wskazuje ścieżkę do folderu zawierającego dane wejściowe.

Wybiera ubezpieczycieli, których dane zostaną uwzględnione w symulacji.

Włącza lub wyłącza możliwość odniesień do dodatkowych odnowień.

Po zakończeniu konfiguracji aplikacja wczytuje dane wejściowe. W trakcie tego procesu pasek postępu informuje użytkownika o statusie wczytywania danych.

10

##### Ustawienie parametrów symulacji

Po wczytaniu danych użytkownik konfiguruje parametry symulacji. Na tym etapie defi- niowane są:

Liczba iteracji symulacji.

Liczba wątków, które mają być wykorzystywane do przetwarzania danych.

Wartości progowe szkód (minimalnej i katastroficznej).

Promień oddziaływania zdarzeń.

Lokalizacja, w której wyniki symulacji zostaną zapisane.

##### Uruchamianie symulacji i zapis wyników

Ostatni etap to uruchomienie symulacji, monitorowanie jej postępu oraz zapis wyników. W trakcie tego procesu:

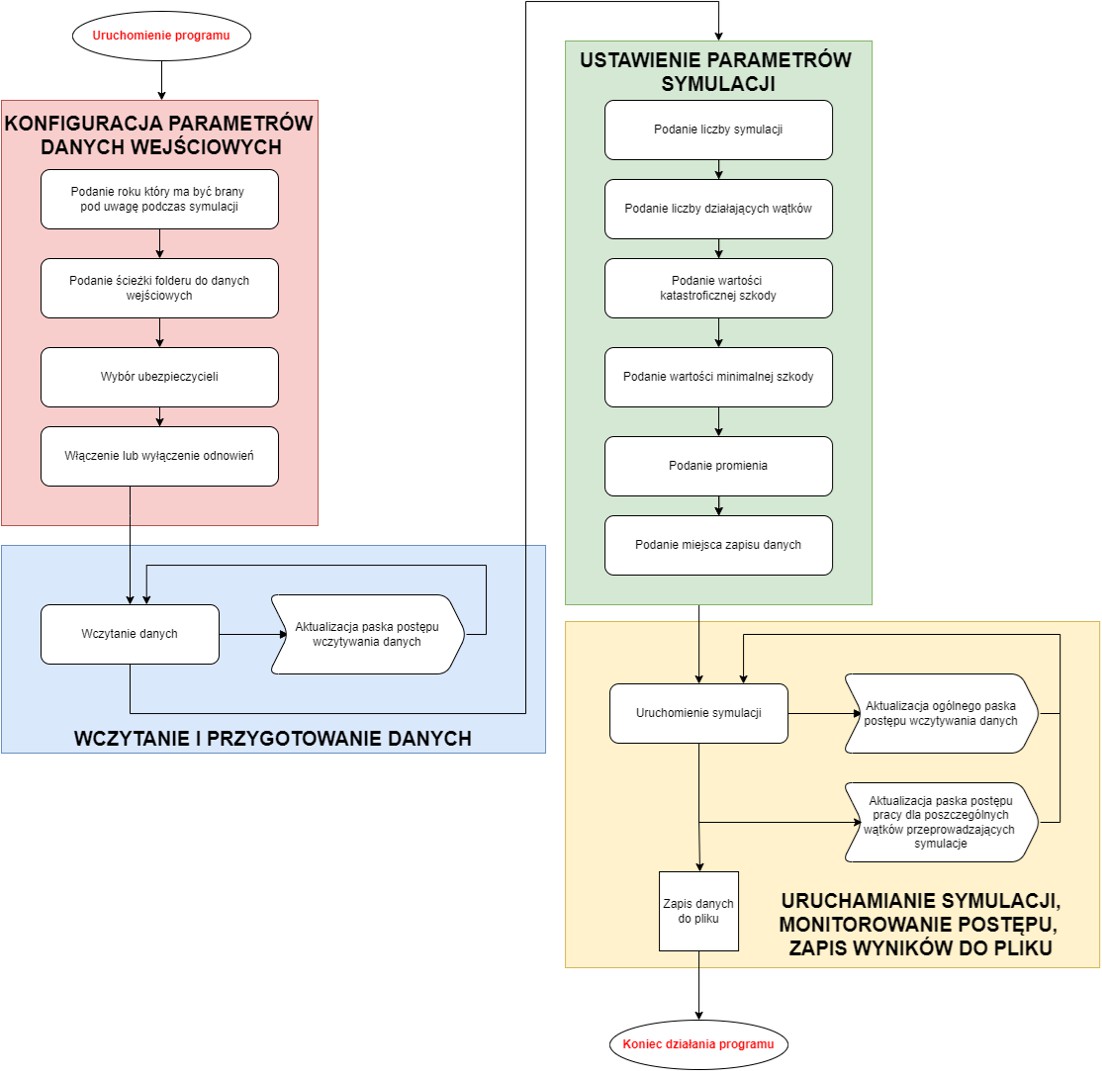
Aplikacja aktualizuje ogólny pasek postępu symulacji.

Aktualizowane są paski postępu dla poszczególnych wątków odpowiedzialnych za symulację.

Po zakończeniu działania wyniki są zapisywane w pliku CSV, a aplikacja kończy swoje działanie.

### Schemat przepływu danych

Rycina 2.1 przedstawia graficzny schemat pracy aplikacji, podzielony na opisane trzy etapy.



Rysunek 2.1: Schemat przepływu danych i symulacji

## Zarządzanie symulacjami przy użyciu puli wątków i bufora

W aplikacji zastosowano efektywny mechanizm zarządzania zadaniami w środowisku wielowąt- kowym. Mechanizm ten opiera się na wykorzystaniu puli wątków oraz bufora, co pozwala na równoległe przetwarzanie wielu symulacji oraz optymalizację zapisu wyników. Proces ten został przedstawiony na poniższym diagramie (Rycina 2.2).

##### Opis procesu zarządzania symulacjami

1. **Kolejka symulacji**

Na początku każda symulacja trafia do kolejki zadań. Kolejka ta przechowuje symulacje oczekujące na przetworzenie. Jest to struktura danych typu FIFO (ang. *First In, First Out* ), co oznacza, że symulacje są przetwarzane w kolejności ich dodania. Kolejka gwa- rantuje, że żadne zadanie nie zostanie pominięte.

##### Pula wątków

Głównym elementem zarządzania symulacjami jest pula wątków. Każdy wątek w puli działa niezależnie i jest odpowiedzialny za przeprowadzenie jednej symulacji. Liczba wąt- ków w puli jest konfigurowalna i zależy od dostępnych zasobów sprzętowych (np. liczby rdzeni procesora). Dzięki równoległemu przetwarzaniu:

Możliwe jest jednoczesne wykonywanie wielu symulacji.

Optymalizowane jest wykorzystanie mocy obliczeniowej systemu.

Minimalizowany jest czas przetwarzania dużych zbiorów danych.

Każda symulacja po wybraniu z kolejki jest przypisywana do jednego z wątków. Wątek przeprowadza symulację, a po jej zakończeniu zwalnia się, aby obsłużyć kolejne zadanie.

##### Bufor wyników

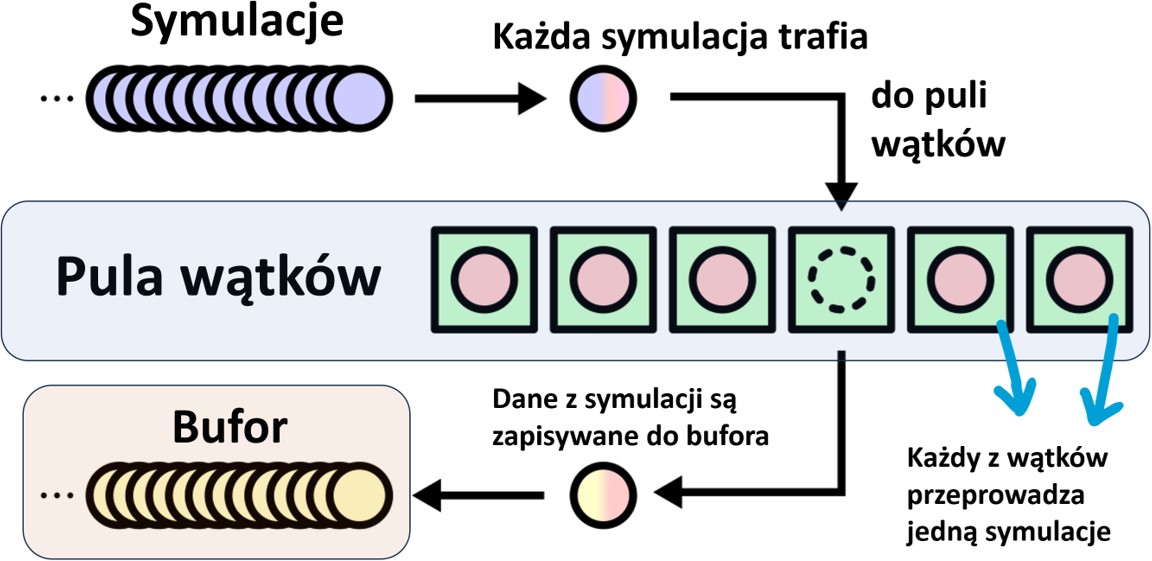
Po zakończeniu symulacji dane są zapisywane do bufora. Bufor pełni funkcję tymczaso- wego magazynu wyników, które później są zapisywane w trwałej pamięci. Takie podejście pozwala na:

Zminimalizowanie opóźnień związanych z zapisem wyników w czasie rzeczywistym.

Grupowanie danych przed ich ostatecznym zapisaniem, co zwiększa efektywność ope- racji wejścia/wyjścia (I/O).

##### Schemat zarządzania symulacjami

Rycina 2.2 przedstawia graficzny schemat zarządzania symulacjami, w którym wykorzystano kolejkę zadań, pulę wątków oraz bufor wyników.



Rysunek 2.2: Schemat zarządzania symulacjami przy użyciu puli wątków i bufora

## Szczegółowy opis schematu przepływu danych i sy- mulacji w oprogramowaniu

Diagram przedstawiony na rysunku 2.3 ilustruje architekturę i przepływ danych w systemie wielowątkowym, który zarządza symulacjami w sposób równoległy. System składa się z kilku kluczowych modułów, które współpracują w celu efektywnego przetwarzania dużej liczby sy- mulacji. Poniżej znajduje się szczegółowy opis każdego elementu.

##### Interfejs użytkownika

Interfejs użytkownika jest punktem wejścia do systemu. Umożliwia użytkownikowi dodawanie nowych symulacji do kolejki zadań. Każda nowa symulacja (np. S8) jest przekazywana do sys- temu za pomocą zaprojektowanego mechanizmu, który zapewnia integralność i poprawność danych wejściowych. Interfejs posiada mechanizm wizualny (pasek postępu), który informuje o stanie dodawania symulacji.

##### Kolejka zadań

Kolejka zadań pełni funkcję bufora pośredniego, w którym przechowywane są symulacje ocze- kujące na przetworzenie. Jak wspomniano wyżej, jest to kolejka FIFO, co oznacza, że symulacje są przetwarzane w kolejności ich dodania. Na diagramie widoczne są symulacje S7, S6, ..., S3, które czekają na przydzielenie do wątków. Mechanizm kolejki zapewnia:

**Seryjność:** Zachowanie poprawnej kolejności przetwarzania.

**Bezpieczeństwo wątkowe:** Synchronizacja dostępu do kolejki w środowisku wielowąt- kowym.

##### Pula wątków

Pula wątków jest kluczowym elementem systemu, który umożliwia równoległe przetwarzanie symulacji. Na diagramie przedstawiono dwa wątki, które równocześnie wykonują symulacje S1 i S2. Każdy wątek działa niezależnie, co pozwala na znaczną oszczędność czasu obliczeń. Wątek po zakończeniu pracy nad jedną symulacją automatycznie pobiera kolejną z kolejki. Mechanizm puli wątków zapewnia:

**Dynamiczne zarządzanie zasobami:** Liczba wątków może być dostosowana do do- stępnych zasobów sprzętowych.

**Izolację:** Każdy wątek działa na niezależnym zestawie danych, co minimalizuje ryzyko konfliktów.

##### Symulacja

Symulacja jest procesem, który obejmuje szereg obliczeń statystycznych, geograficznych i rease- kuracyjnych. Każda symulacja wykonuje następujące operacje (zgodnie z diagramem):

##### Losowanie i próbkowanie:

randBin - losuje liczbę pożarów na podstawie rozkładu dwumianowego, co pozwala na modelowanie zdarzeń losowych.

sample vec - wybiera losowe próbki z wektora, co znajduje zastosowanie w analizie statystycznej.

##### Obliczenia statystyczne:

percentage of loss - oblicza procentową wielkość straty na podstawie danych wej- ściowych, co jest kluczowe w analizie ryzyka.

##### Obliczenia geograficzne:

haversine loop cpp vec - oblicza odległości geograficzne między punktami, co jest istotne w modelowaniu zdarzeń przestrzennych.

##### Reasekuracja:

reasecuration build fire - oblicza kwotę reasekuracji dla budynku.

reassurance risk - oblicza ryzyko związane z reasekuracją.

calc reas obliga event - oblicza zdarzenia reasekuracyjne.

calc brutto ring - oblicza wartości brutto związane z ubezpieczeniami i reaseku- racją.

##### Bufor wyników

Bufor wyników jest miejscem tymczasowego przechowywania danych wygenerowanych podczas działania symulacji. Buforowanie wyników zmniejsza obciążenie operacjami wejścia/wyjścia (I/O), co poprawia wydajność systemu. Dane przechowywane w buforze są następnie prze- kazywane do trwałej pamięci lub wykorzystywane w kolejnych etapach przetwarzania.

##### Przepływ danych

Przepływ danych w systemie można opisać w następujących krokach:

* + 1. Użytkownik dodaje nową symulację (np. S8) za pomocą interfejsu użytkownika.
    2. Symulacja trafia do kolejki zadań, gdzie oczekuje na przetworzenie.
    3. Dostępne wątki w puli wątków pobierają symulacje z kolejki i rozpoczynają ich przetwa- rzanie.
    4. Podczas działania symulacji generowane są dane, które są zapisywane w buforze.
    5. Bufor przechowuje dane tymczasowo przed ich ostatecznym zapisaniem lub dalszym prze- twarzaniem.

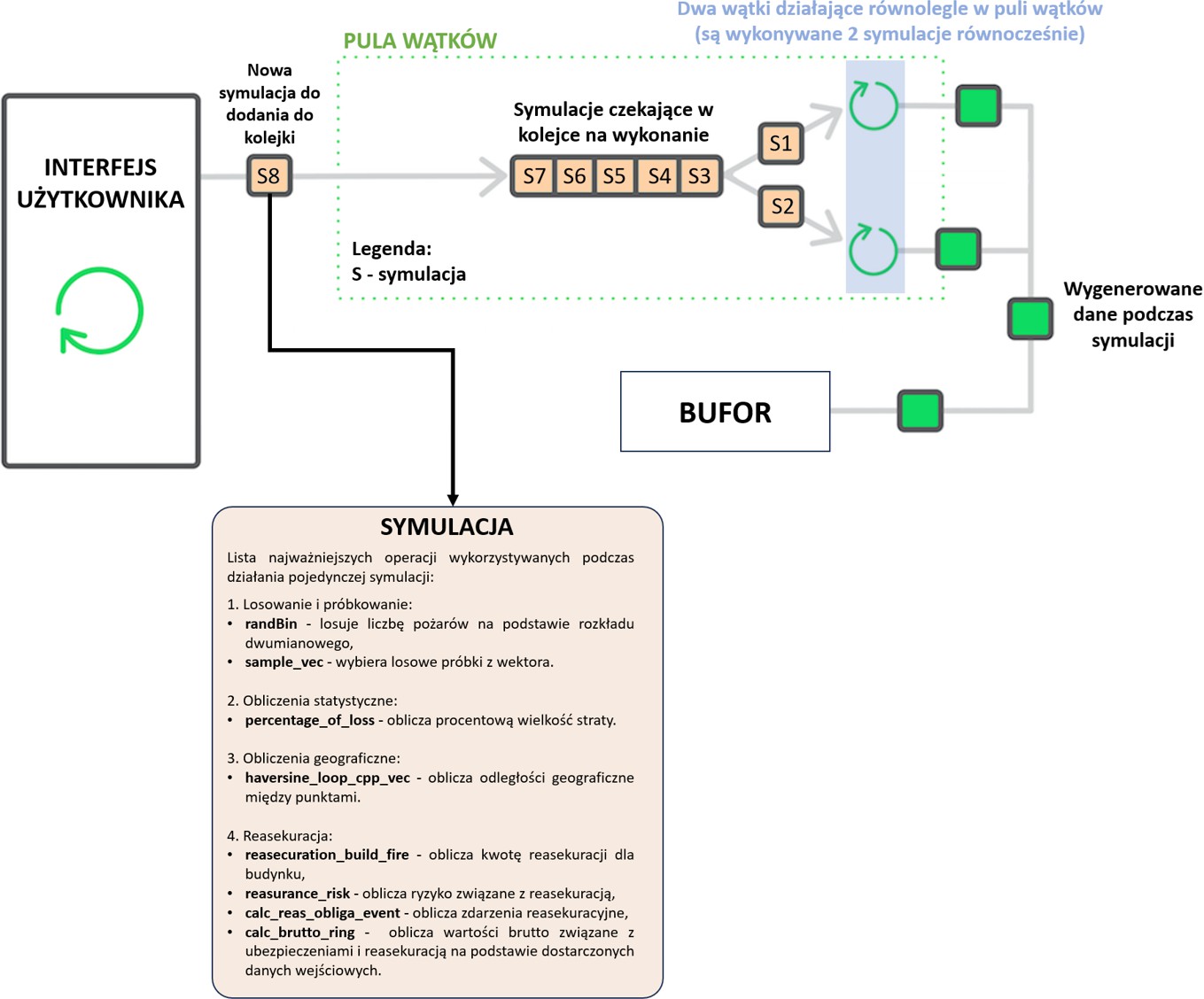
##### Zalety zastosowanego rozwiązania

**Efektywność przetwarzania:** Dzięki równoległemu wykonywaniu symulacji możliwe jest znaczące skrócenie czasu obliczeń. Mechanizm puli wątków pozwala na dynamiczne przydzielanie zasobów w zależności od obciążenia.

**Minimalizacja opóźnień:** Buforowanie wyników zmniejsza liczbę operacji zapisu na dysku, co jest szczególnie istotne w przypadku dużych symulacji generujących wiele da- nych.

**Skalowalność:** Mechanizm puli wątków można łatwo skalować, dostosowując liczbę wąt- ków do dostępnych zasobów sprzętowych.

**Modularność:** Każdy moduł systemu działa niezależnie, co ułatwia jego rozwój i utrzy- manie.



Rysunek 2.3: Schemat przepływu danych i symulacji w systemie wielowątkowym

## Omówienie interfejsu graficznego aplikacji

Interfejs graficzny aplikacji został zaprojektowany w sposób intuicyjny i zorganizowany, co uła- twia użytkownikom konfigurację i monitorowanie symulacji. Składa się z kilku sekcji, które odpowiadają różnym etapom pracy z aplikacją. Poniżej opisano szczegółowo każdą sekcję inter- fejsu.

### Przegląd interfejsu użytkownika

##### Sekcja przygotowania danych

Sekcja ta umożliwia wprowadzenie podstawowych informacji niezbędnych do przeprowadzenia symulacji:

**Pole tekstowe ”Rok”** – pozwala użytkownikowi wprowadzić rok, który ma być uwzględ- niony w analizie.

**Pole tekstowe ”Ścieżka do folderu input”** – umożliwia podanie ścieżki do folderu zawierającego dane wejściowe.

**Przycisk ”Wczytaj listę ubezpieczycieli”** – służy do załadowania listy ubezpieczy- cieli, którzy będą brali udział w symulacji.

**Tabela wyboru ubezpieczycieli** – wyświetla listę ubezpieczycieli, z której użytkownik może dokonać wyboru.

**Przycisk ”Wybierz wszystkich”** – umożliwia zaznaczenie wszystkich ubezpieczycieli na liście.

##### Sekcja odnowienia

Ta sekcja pozwala na konfigurację opcji odnowień:

**Przełącznik** – umożliwia włączenie lub wyłączenie odnowień w symulacji.

##### Sekcja uruchamiania i śledzenia wczytywania danych

Sekcja ta służy do inicjowania procesu wczytywania danych oraz monitorowania jego postępu:

**Przycisk ”Wczytaj dane”** – rozpoczyna proces wczytywania danych wejściowych.

**Pasek postępu** – wizualizuje aktualny postęp procesu wczytywania danych.

##### Sekcja parametrów symulacji

W tej sekcji użytkownik może skonfigurować parametry symulacji:

**Pola liczby** – pozwalają ustawić takie parametry jak:

* + liczba symulacji,
  + liczba wątków do obliczeń i zapisu,
  + wartości szkód (katastroficznych i minimalnych),
  + liczba budynków do zapisania,
  + promień analizy.

**Pole tekstowe ”Ścieżka zapisu”** – umożliwia podanie ścieżki do miejsca, w którym zostaną zapisane wyniki symulacji.

##### Sekcja wyboru zapisu budynków

W tej sekcji użytkownik może zdecydować, które dane dotyczące budynków mają zostać zapi- sane:

**Opcje:**

* + **”Wszystkie budynki”** – zapisuje dane wszystkich budynków.
  + **”Wybrane budynki”** – zapisuje dane tylko wybranych budynków.

##### Sekcja uruchamiania i śledzenia symulacji

Sekcja ta pozwala na rozpoczęcie symulacji oraz monitorowanie jej postępu:

**Przycisk ”Włącz symulację”** – inicjuje proces symulacji.

##### Paski postępu:

* **Ogólny pasek postępu** – pokazuje ogólny postęp symulacji.
* **Pasek postępu zapisu** – wizualizuje postęp zapisywania wyników.

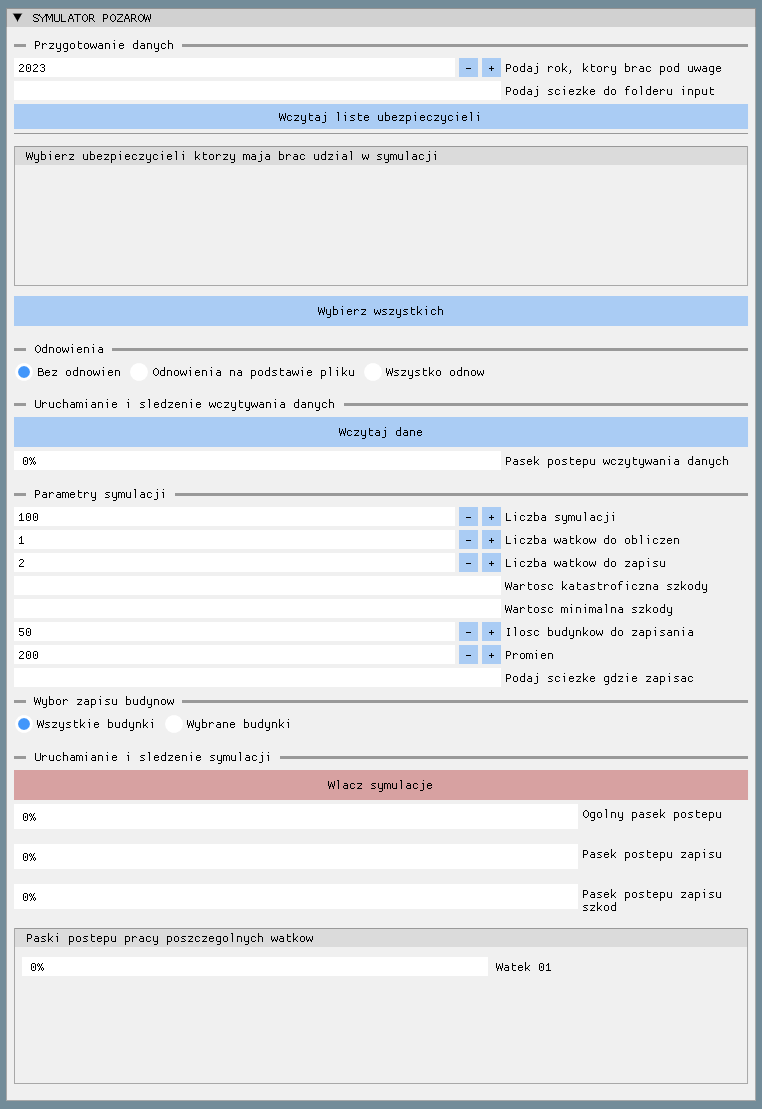
##### Sekcja pasków postępu pracy poszczególnych wątków

W tej sekcji można monitorować postęp pracy każdego wątku indywidualnie:

**Paski postępu** – wyświetlają postęp generowania symulacji dla każdego wątku.

### Podsumowanie

Interfejs graficzny aplikacji został zaprojektowany w sposób intuicyjny i przejrzysty. Podział na sekcje ułatwia użytkownikowi konfigurację symulacji, monitorowanie procesów oraz dosto- sowanie parametrów do indywidualnych potrzeb. Dzięki zastosowaniu wizualnych elementów, takich jak paski postępu, użytkownik może na bieżąco śledzić przebieg symulacji i procesów wczytywania danych.



Rysunek 2.4: Widok interfejsu graficznego aplikacji

## Opis kontenerów danych

Kod implementuje symulację strat ubezpieczeniowych związanych z pożarami. Dane są przecho- wywane i przetwarzane za pomocą klas takich jak Ubezpieczyciel, VectorSim, VectorPozarPierwotny i VectorPozarRozprzestrzeniony. Kluczowym elementem jest klasa Ubezpieczyciel, która zarządza szczegółowymi danymi dotyczącymi strat ubezpieczycieli, zarówno brutto, netto, jak

i katastroficznych.

* + 1. **Klasa** Ubezpieczyciel

Klasa Ubezpieczyciel reprezentuje pojedynczego ubezpieczyciela i przechowuje dane doty- czące strat w różnych kategoriach.

##### Składowe klasy

**Pożary pierwotne:**

* buildPierwotny brutto vec: Wektor obiektów klasy VectorPozarPierwotny, za- wierający dane o stratach brutto.
* buildPierwotny netto vec: Wektor obiektów klasy VectorPozarPierwotny, zawie- rający dane o stratach netto.

##### Pożary rozprzestrzenione:

* buildRozprzestrzeniony brutto vec: Wektor obiektów klasy VectorPozarRozprzestrzeniony, zawierający dane o stratach brutto.
* buildRozprzestrzeniony netto vec: Wektor obiektów klasy VectorPozarRozprzestrzeniony, zawierający dane o stratach netto.

##### Sumaryczne wartości strat:

* sum vec out vec: Wektor wartości sumarycznych strat brutto.
* sum vec netto out vec: Wektor wartości sumarycznych strat netto.

##### Straty katastroficzne:

* buildPierwotny brutto kat vec: Wektor danych o stratach katastroficznych brutto (pożary pierwotne).
* buildPierwotny netto kat vec: Wektor danych o stratach katastroficznych netto (pożary pierwotne).
* buildRozprzestrzeniony brutto kat vec: Wektor danych o stratach katastroficz- nych brutto (pożary rozprzestrzenione).
* buildRozprzestrzeniony netto kat vec: Wektor danych o stratach katastroficz- nych netto (pożary rozprzestrzenione).
* sum vec kat out vec: Wektor sumarycznych wartości strat katastroficznych brutto.
* sum vec netto kat out vec: Wektor sumarycznych wartości strat katastroficznych netto.

##### Opis przechowywanych danych

Każdy ubezpieczyciel przechowuje dane w postaci wektorów obiektów klas VectorPozarPierwotny i VectorPozarRozprzestrzeniony. Dane te są podzielone na:

Straty brutto (przed uwzględnieniem reasekuracji).

Straty netto (po uwzględnieniu reasekuracji).

Straty katastroficzne (przekraczające określony próg wartości).

* + 1. **Klasa** VectorSim

Klasa VectorSim jest odpowiedzialna za przechowywanie i zarządzanie danymi liczbowymi w postaci wektorów dwuwymiarowych:

std::vector*<*std::vector*<*double*>>*

##### Funkcjonalność klasy

**Dodawanie danych:**

* addDataVec(int insurance, double value): Dodaje wartość do określonego ubez- pieczyciela.
* addDataVec out(int insurance, double value, int sim miejsce): Dodaje war- tość do określonej symulacji.

##### Czyszczenie danych:

* clearVector(int num vec): Czyści dane dla określonego ubezpieczyciela.
* clear(): Czyści wszystkie dane.

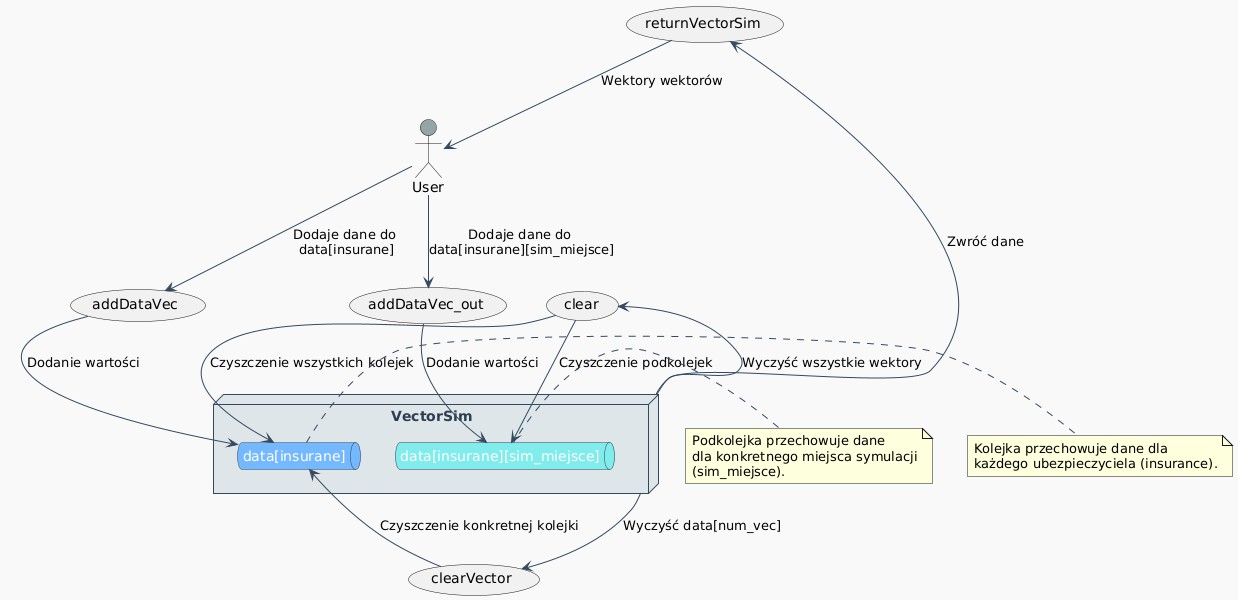
##### Zwracanie danych:

* returnVectorSim(): Zwraca wektor danych.

### Opis diagramu kolejek zadań dla klasy VectorSim

Diagram przedstawia przepływ zadań i operacji wykonywanych w klasie VectorSim. Klasa ta za-

rządza hierarchiczną strukturą danych w postaci wektorów wektorów (std::vector<std::vector<double>>). Główne elementy diagramu to:



Rysunek 2.5: Diagram kolejek zadań dla klasy VectorSim.

### Elementy diagramu

**Aktor** User: Reprezentuje system, który korzysta z klasy VectorSim. Aktor inicjuje operacje na danych.

##### Kolejki danych:

* data[insurane]: Główna kolejka przechowująca dane dla poszczególnych ubezpie- czycieli (insurance). Każdy ubezpieczyciel ma przypisaną swoją kolejkę.
* data[insurane][sim miejsce]: Podkolejka przechowująca dane dla konkretnego miejsca symulacji (sim miejsce). Jest to struktura zagnieżdżona w głównej kolejce.

**Klasa** VectorSim: Zarządza kolejkami danych, implementując operacje dodawania, czysz- czenia i zwracania danych.

### Operacje na danych

Diagram przedstawia następujące operacje wykonywane na kolejkach:

addDataVec(int insurane, double value): Dodaje wartość value do głównej kolejki

data[insurane] dla danego ubezpieczyciela (insurance).

addDataVec out(int insurane, double value, int sim miejsce): Dodaje wartość value do podkolejki data[insurane][sim miejsce] dla konkretnego miejsca symulacji.

clearVector(int num vec): Czyści dane w konkretnej kolejce data[num vec].

clear(): Czyści wszystkie kolejki i podkolejki, usuwając wszystkie dane.

returnVectorSim(): Zwraca pełną strukturę danych w postaci wektora wektorów.

### Podsumowanie

Diagram wizualizuje sposób, w jaki klasa VectorSim zarządza strukturą danych i operacjami na kolejkach. Dzięki hierarchicznej organizacji danych i metodom umożliwiającym ich modyfikację, klasa może być używana w systemach wymagających zarządzania dużymi zbiorami danych w symulacji lub innych złożonych operacji.

* + 1. **Klasa** VectorPozarPierwotny

Klasa VectorPozarPierwotny przechowuje dane dotyczące pożarów pierwotnych w postaci ma- cierzy:

std::vector*<*std::vector*<*long double*>>.*

##### Struktura danych

Każda kolumna macierzy reprezentuje różne informacje o pożarze pierwotnym:

[0]: Numer ubezpieczyciela.

[1]: Długość geograficzna budynku.

[2]: Szerokość geograficzna budynku.

[3]: Województwo.

[4]: Miesiąc wystąpienia pożaru.

[5]: Wartość sumaryczna budynku.

[6]: Indeks tabeli.

[7]: Wielkość strat w złotówkach.

[8]: Powód pożaru.

##### Funkcjonalność klasy

**Dodawanie danych:**

* addPozarPierwotny(...): Dodaje dane o nowym pożarze pierwotnym.

##### Zwracanie danych:

* returnPozarPierwotny(): Zwraca przechowywaną macierz danych.

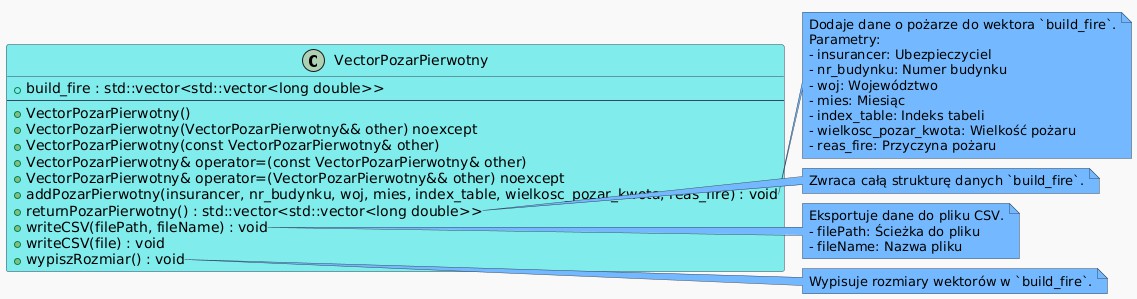
##### Zapis do pliku CSV:

* writeCSV(...): Zapisuje dane do pliku CSV.

##### Debugowanie:

* wypiszRozmiar(): Wyświetla rozmiary przechowywanych danych.

W poniższym diagramie przedstawiono strukturę klasy VectorPozarPierwotny, która za- rządza danymi o pożarach.



Rysunek 2.6: Diagram klasy VectorPozarPierwotny

#### **Klasa** VectorPozarRozprzestrzeniony

Klasa VectorPozarRozprzestrzeniony przechowuje dane dotyczące pożarów rozprzestrzenio- nych. Struktura i funkcjonalność są podobne do klasy VectorPozarPierwotny, z dodatkowymi polami dotyczącymi promienia rozprzestrzenienia pożaru oraz danych o reasekuracji.

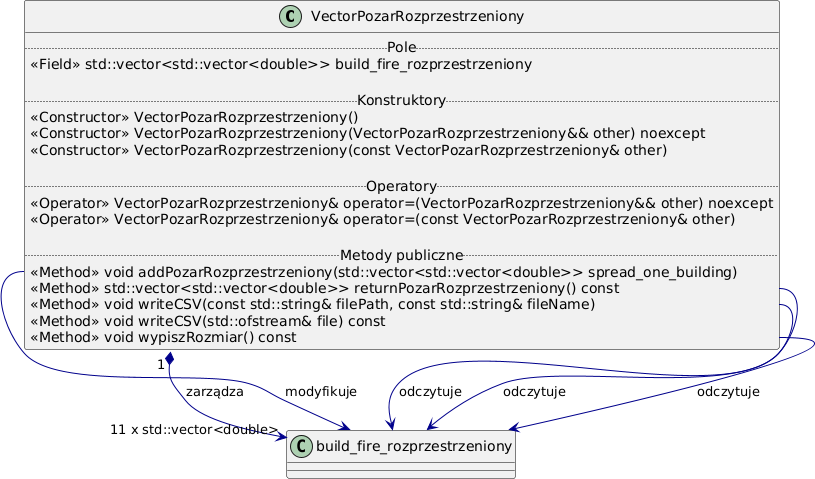
Diagram poniżej przedstawia strukturę klasy VectorPozarRozprzestrzeniony w języku C++. Klasa zarządza dwuwymiarowym wektorem build fire rozprzestrzeniony, który prze- chowuje dane o rozprzestrzenianiu się pożarów. Diagram uwzględnia:

Pole build fire rozprzestrzeniony, które jest głównym elementem danych klasy.

Konstruktory (domyślny, kopiujący, przenoszący) oraz operatory przypisania.

Metody publiczne, takie jak:

* addPozarRozprzestrzeniony – dodaje dane o rozprzestrzenianiu się pożaru do ist- niejącego wektora.
* returnPozarRozprzestrzeniony – zwraca kopię danych.
* writeCSV – zapisuje dane do pliku CSV.
* wypiszRozmiar – wypisuje rozmiary poszczególnych wierszy wektora.



Rysunek 2.7: Diagram klasy VectorPozarRozprzestrzeniony.

Diagram pokazuje również przepływ danych między metodami a polem build fire rozprzestrzeniony, wskazując, które metody modyfikują, a które odczytują dane.

## Opis buforów danych i mechanizmów synchronizacji

Kod implementuje wielowątkowe przetwarzanie danych z użyciem buforów i mechanizmów syn- chronizacji. Jego głównym celem jest zarządzanie zapisami danych o pożarach pierwotnych i rozprzestrzenionych do plików CSV, przy jednoczesnym zapewnieniu bezpieczeństwa wątków i synchronizacji.

* + 1. **Struktury danych:** BuforPierwotny **i** BuforRozprz

Struktury te przechowują dane związane z pożarami pierwotnymi i rozprzestrzenionymi. Każda struktura zawiera:

Obiekt klasy przechowującej dane (VectorPozarPierwotny lub VectorPozarRozprzestrzeniony).

Ścieżkę do pliku, gdzie dane zostaną zapisane.

Nazwę pliku.

Kod struktur:

Listing 2.1: Struktury danych

struct BuforPierwotny

{

VectorPozarPierwotny vpp ; std :: string filePath ; std :: string fileName;

BuforPierwotny ( VectorPozarPierwotny && vpp , std :: string filePath , std :: string

*‹→* fileName)

: vpp ( std :: move( vpp )), filePath ( filePath ), fileName( fileName) {}

};

struct BuforRozprz

{

VectorPozar Rozprzestrzeniony vpr; std :: string filePath ;

std :: string fileName;

BuforRozprz ( VectorPozar Rozprzestrzeniony && vpr , std :: string filePath , std ::

*‹→* string fileName)

: vpr( std :: move( vpr)), filePath ( filePath ), fileName( fileName) {}

};

### Globalne bufory i mechanizmy synchronizacji

Kod wykorzystuje globalne bufory i mutexy do synchronizacji:

Bufory:

* global buffer pierwotny: Przechowuje dane o pożarach pierwotnych.
* global buffer rozprz: Przechowuje dane o pożarach rozprzestrzenionych.

Mutexy:

* mtx pierwotny i mtx rozprz: Chronią dostęp do buforów.
* mtxx i mtxy: Chronią dostęp do warunkowych zmiennych.

Zmienne warunkowe:

* cv i cy: Powiadamiają wątki o dostępności nowych elementów w buforach.

Kod globalnych zmiennych:

Listing 2.2: Globalne bufory i mutexy std :: deque < BuforPierwotny > global\_buffer\_pierwotny ; std :: deque < BuforRozprz > global\_buffer\_rozprz ;

std :: mutex mtx\_pierwotny ; std :: mutex mtx\_rozprz ;

std :: mutex mtxx ; std :: mutex mtxy ;

std :: condition\_variable cv; std :: condition\_variable cy;

### Funkcje dodające dane do buforów

Funkcje przeniesDoBuforPierwotny i przeniesDoBuforRozprz:

Dodają dane do odpowiednich buforów.

Używają mutexów, aby zapewnić bezpieczeństwo dostępu.

Powiadamiają wątki konsumentów o nowym elemencie za pomocą zmiennych warunko- wych.

Kod funkcji:

Listing 2.3: Funkcje dodające dane do buforów

void przeniesDo BuforPierwotny ( VectorPozarPierwotny && vpp , std :: string filePath ,

*‹→* std :: string fileName)

{

{

std :: lock\_guard < std :: mutex > lock ( mtxx ); global\_buffer\_pierwotny . emplace\_back ( std :: move( vpp ), std :: move( filePath )

*‹→* , std :: move( fileName));

}

}

void przeniesDo BuforRozprz ( Vector Pozar Rozprzestrzeniony && vpr , std :: string

*‹→* filePath , std :: string fileName)

{

{

std :: lock\_guard < std :: mutex > lock ( mtxy ); global\_buffer\_rozprz . emplace\_back ( std :: move( vpr), std :: move( filePath ),

*‹→* std :: move( fileName));

}

}

### Wątki zapisujące dane

Wątki watekZapisPierwotny i watekZapisRozprz:

Pobierają dane z buforów.

Zapisują dane do plików CSV.

Synchronizują dostęp do buforów za pomocą mutexów i zmiennych warunkowych.

Kończą swoją pracę, gdy licznik symulacji (licznik sym) osiągnie 0 i bufory są puste. Kod wątków:

Listing 2.4: Wątki zapisujące dane

void watek ZapisPierwotny () { while ( true) {

std :: unique\_lock < std :: mutex > lock ( mtxx );

cv. wait( lock , []() { return ! global\_buffer\_pierwotny . empty () ||

*‹→* licznik\_sym == 0; });

if ( licznik\_sym == 0 && global\_buffer\_pierwotny . empty ()) { break ;

}

if (! global\_buffer\_pierwotny . empty ()) {

BuforPierwotny data = std :: move( global\_buffer\_pierwotny . front ()); global\_buffer\_pierwotny . pop\_front ();

licznik\_sym --; lock . unlock ();

data. vpp . writeCSV ( data. filePath , data. fileName);

}

}

}

void watek ZapisRozprz () {

while ( true) {

std :: unique\_lock < std :: mutex > lock ( mtxy );

cy. wait( lock , []() { return ! global\_buffer\_rozprz . empty () || licznik\_sym

*‹→* == 0; });

if ( licznik\_sym == 0 && global\_buffer\_rozprz . empty ()) { break ;

}

if (! global\_buffer\_rozprz . empty ()) {

BuforRozprz data = std :: move( global\_buffer\_rozprz . front ()); global\_buffer\_rozprz . pop\_front ();

licznik\_sym --; lock . unlock ();

data. vpr. writeCSV ( data. filePath , data. fileName);

}

}

}

## Opis Diagramu Systemu Buforów

Diagram przedstawia architekturę systemu zarządzania danymi pożarowymi w środowisku wie- lowątkowym. W systemie wyróżniono następujące komponenty:

##### Bufory (System Buforów):

* **BuforPierwotny (BP)**: Przechowuje dane dotyczące pierwotnych informacji o po- żarach.
* **BuforRozprzestrzeniony (BR)**: Przechowuje dane dotyczące rozprzestrzeniania się pożarów.

##### Globalne Bufory:

* **GlobalPierwotny (GP)**: Kolejka danych pierwotnych, z której korzystają wątki.
* **GlobalRozprzestrzeniony (GR)**: Kolejka danych dotyczących rozprzestrzeniania się pożarów.

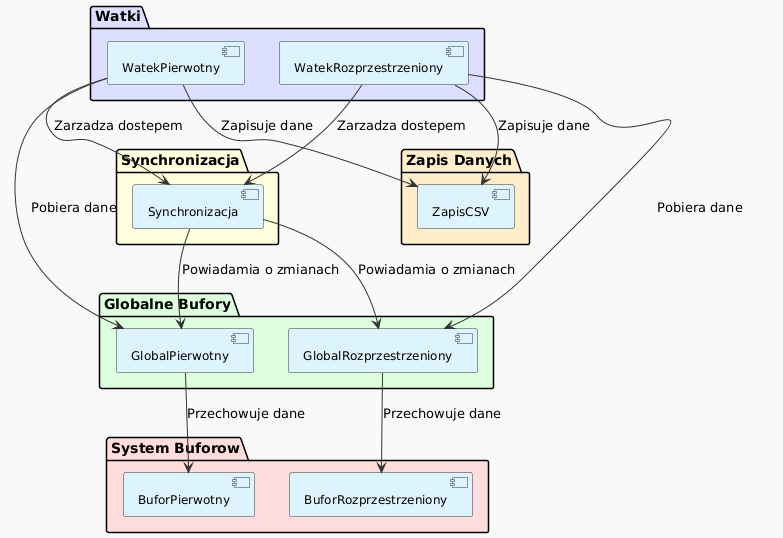
##### Wątki (Watki):

* **WatekPierwotny (WP)**: Pobiera dane z kolejki **GP**, przetwarza je i zapisuje do pliku CSV.

**– WatekRozprzestrzeniony (WR)**: Pobiera dane z kolejki **GR**, przetwarza je i zapisuje do pliku CSV.

**Synchronizacja (Sync)**: Zarządza dostępem do globalnych buforów, zapewniając bez- pieczeństwo w środowisku wielowątkowym.

**Zapis Danych (ZapisCSV)**: Odpowiada za zapis przetworzonych danych do plików CSV.



Rysunek 2.8: Diagram architektury systemu buforów

Strzałki na diagramie przedstawiają przepływ danych oraz interakcje między komponentami:

Wątki (**WP** i **WR**) pobierają dane z globalnych kolejek (**GP** i **GR**).

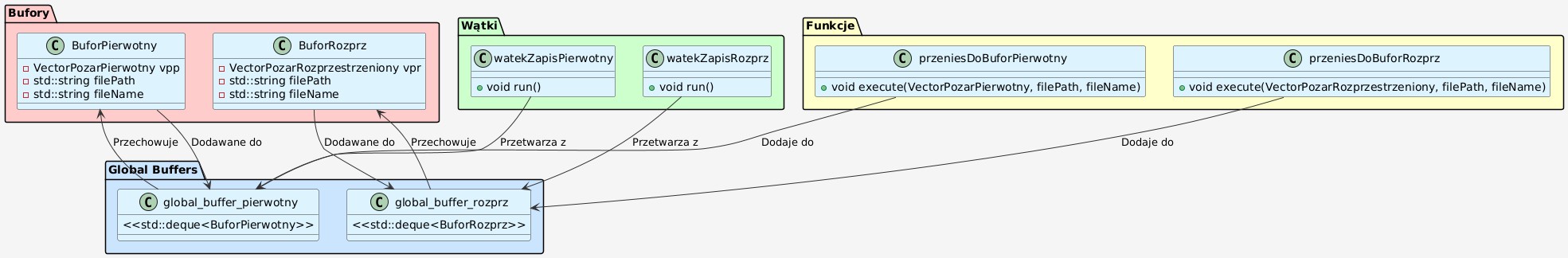
Dane są przechowywane w odpowiednich buforach (**BP** i **BR**).

Synchronizacja (**Sync**) zapewnia kolejność przetwarzania i dostęp do zasobów.

Dane są ostatecznie zapisywane do plików CSV przez komponent **ZapisCSV**.

## Diagram systemu zarządzania buforami

Poniżej przedstawiono diagram ilustrujący relacje między komponentami systemu zarządzania buforami:



### Opis Diagramu

Diagram przedstawia system zarządzania buforami dla danych pożarów pierwotnych i rozprze- strzenionych. Składa się z następujących komponentów:

**Bufory**:

* *BuforPierwotny* — przechowuje dane pożarów pierwotnych.
* *BuforRozprz* — przechowuje dane pożarów rozprzestrzenionych.

##### Globalne Bufory:

* *global buffer pierwotny* — kolejka przechowująca obiekty *BuforPierwotny*.
* *global buffer rozprz* — kolejka przechowująca obiekty *BuforRozprz*.

**Funkcje**:

* *przeniesDoBuforPierwotny* — dodaje dane do kolejki *global buffer pierwotny*.
* *przeniesDoBuforRozprz* — dodaje dane do kolejki *global buffer rozprz*.

**Wątki**:

* *watekZapisPierwotny* — przetwarza dane z kolejki *global buffer pierwotny*.
* *watekZapisRozprz* — przetwarza dane z kolejki *global buffer rozprz*.

## Opis działania funkcji

* + 1. **Funkcja** testAll

Funkcja testALL jest rozbudowaną procedurą w języku C++, której celem jest zarządzanie procesem:

wczytywania i przetwarzania danych wejściowych,

uruchamiania symulacji,

zapisywania wyników do odpowiednich struktur i plików.

Funkcja działa w dwóch trybach, zależnie od wartości parametru wejściowego choice:

choice = 1: Wczytywanie danych wejściowych i ich przetwarzanie.

choice = 2: Uruchamianie symulacji oraz zapisywanie wyników.

## Opis szczegółowy funkcji

### Parametr wejściowy

int choice - parametr określający tryb działania funkcji:

* choice = 1 - wczytywanie i przygotowanie danych wejściowych,
* choice = 2 - uruchamianie symulacji i zapis wyników.

### Główne etapy działania funkcji

##### Resetowanie pul wątków

pool. reset( liczba\_dzialajacych\_watkow ); poolFiles. reset (4);

Pule wątków są resetowane:

pool - pula wątków odpowiedzialnych za symulacje. Liczba wątków jest określona przez zmienną liczba dzialajacych watkow.

poolFiles - pula wątków do obsługi operacji na plikach. Liczba wątków w tej puli jest ustawiona na 4.

##### Filtrowanie nazw ubezpieczycieli

for ( int i = 0; i < ubezp\_nazwy . size (); i++) { if ( flagi[ i] != 0) {

testVec. push\_back ( ubezp\_nazwy [ i]);

}

}

Z listy nazw ubezpieczycieli (ubezp nazwy) wybierane są te, które mają ustawiony odpo- wiedni znacznik w tablicy flagi. Wybrane nazwy są zapisywane w wektorze testVec.

##### Wyświetlanie wybranych ubezpieczycieli

for ( int i = 0; i < testVec. size (); i++) { std :: cout << testVec[ i] << std :: endl;

}

Wszystkie nazwy ubezpieczycieli, które zostały zapisane w testVec, są wypisywane w kon- soli.

##### Ustawienia regionalne i kopiowanie nazw

std :: setlocale( LC\_ALL , " nb\_NO . UTF -8 "); std :: vector < std :: string > fileNames = testVec;

Ustawiana jest lokalizacja na nb NO.UTF-8 (norweska), co jest istotne dla przetwarzania danych w lokalnych formatach liczbowych i tekstowych.

Kopiowana jest zawartość testVec do wektora fileNames, który będzie używany w dalszej części funkcji.

**Tryb** choice = 1**: Wczytywanie danych wejściowych**

##### Inicjalizacja struktur danych

for ( int woj = 0; woj < 17; ++ woj) { exponsure\_longitude [ woj]. resize (12); exponsure\_latitude [ woj]. resize (12); exponsure\_insurance [ woj]. resize (12); exponsure\_reassurance [ woj]. resize (12); exponsure\_sum\_value [ woj]. resize (12);

}

Przygotowywane są struktury danych do przechowywania informacji o ekspozycji w 17 wo- jewództwach. Każda z nich ma 12 elementów, co może odpowiadać np. miesiącom.

##### Ścieżki i parametry wejściowe

std :: string dane\_wejsciowe = std :: string ( sciezka\_input );

std :: string odnowienia = ( czy\_wlaczyc\_odnowienia == 1) ? " tak " : " nie"; std :: string year = std :: to\_string ( wybrany\_rok );

dane wejsciowe - ścieżka do katalogu z danymi wejściowymi.

odnowienia - informacja, czy włączyć odnowienia ("tak" lub "nie").

year - rok wybrany do analizy.

##### Liczenie wierszy w plikach CSV

count\_rows += count\_csv\_rows\_ 1 (...); count\_rows += count\_csv\_rows\_ 2 (...); count\_rows += count\_csv\_rows\_ 3 (...); count\_rows += count\_csv\_rows\_ 4 (...);

Funkcje count csv rows X obliczają liczbę wierszy w różnych plikach CSV. Wynik jest wykorzystywany do ustalenia postępu wczytywania danych.

##### Przetwarzanie danych wejściowych

processReas (...);

processOblig (...); processBudynki (...);

Dane wejściowe są przetwarzane przez różne funkcje:

processReas - przetwarza dane reasekuracji,

processOblig - przetwarza dane dotyczące zobowiązań,

processBudynki - przetwarza dane dotyczące budynków.

**Tryb** choice = 2**: Uruchamianie symulacji**

##### Wczytywanie wartości katastroficznych i minimalnych szkód

std :: stringstream ss( inputString ); while ( std :: getline( ss , token , ’,’)) {

double value = std :: stod ( token ); wartosci\_katastrof\_szk . push\_back ( value);

}

Z ciągu znaków rozdzielonego przecinkami wczytywane są wartości katastroficznych szkód (wartosci katastrof szk) oraz minimalnych szkód (wartosci minimal szk).

##### Uruchamianie symulacji

for ( int sim\_num = 0; sim\_num < sim ; sim\_num ++) { pool. detach\_task ([ nazwakatalogu , sim , ...]( BS :: concurrency\_t idx ) {

simulateExponsureTEST (...);

});

}

pool. wait ();

Symulacje są uruchamiane w puli wątków pool. Każda symulacja jest realizowana jako osobne zadanie.

##### Zapisywanie wyników

zapiszDo CSV (...); create\_custom\_directory ( full\_path );

create\_csv\_files ( full\_path , subfolders [0], insurerIndex , ...);

Wyniki symulacji są zapisywane do plików CSV w odpowiednich katalogach.

##### Czyszczenie danych

out\_brutto\_final . clear (); out\_brutto\_kat\_final . clear (); out\_netto\_final . clear (); out\_netto\_kat\_final . clear ();

Po zapisaniu wyników wektory wynikowe są czyszczone, aby przygotować je na kolejne symulacje.

## Opis diagramu sekwencji funkcji testALL

Diagram sekwencji przedstawia szczegółowy przebieg działania funkcji testALL(int choice), która realizuje różne zadania w zależności od wartości parametru choice. Funkcja ta obsługuje dwa główne scenariusze: wczytywanie i przetwarzanie danych ubezpieczeniowych (choice == 1) oraz wykonywanie symulacji szkód katastroficznych (choice == 2). Poniżej opisano szcze- gółowo poszczególne etapy działania funkcji.

### Inicjalizacja

Na początku funkcja wykonuje operacje inicjalizacyjne:

Resetuje dwie pule wątków:

* + pool – główna pula wątków odpowiedzialna za przetwarzanie równoległe,
  + poolFiles – osobna pula wątków przeznaczona do operacji na plikach.

Tworzy pusty wektor testVec, który będzie przechowywał nazwy ubezpieczycieli speł- niających określone kryteria.

### Filtrowanie nazw ubezpieczycieli

W tej części funkcja iteruje po wektorze ubezp nazwy, zawierającym nazwy ubezpieczycieli, i sprawdza, które elementy spełniają warunek flagi[i] != 0. Nazwy spełniające ten warunek są dodawane do wektora testVec. Po zakończeniu filtrowania zawartość testVec jest wyświetlana w konsoli.

### Decyzja na podstawie parametru choice

Funkcja podejmuje różne działania w zależności od wartości parametru choice:

**Gdy** choice == 1 **(wczytywanie danych ubezpieczeniowych):**

* + Kopiuje zawartość testVec do wektora fileNames.
  + Inicjalizuje wektory ekspozycji (exponsure longitude, exponsure latitude, itp.) dla 17 województw.
  + Wczytuje dane wejściowe, takie jak parametry reasekuracji, dane o budynkach, praw- dopodobieństwa pożaru, itp.
  + Wyświetla pasek postępu wczytywania danych, obliczając krok postępu na podstawie liczby wierszy w przetwarzanych plikach CSV.
* Na końcu tej sekcji wyświetla czas wczytywania danych oraz informację o poprawnym zakończeniu wczytywania.

**Gdy** choice == 2 **(symulacja szkód katastroficznych):**

* Funkcja parsuje wartości katastroficznych i minimalnych szkód dla ubezpieczycieli, przekształcając ciągi znaków na liczby zmiennoprzecinkowe.
* Wyświetla te wartości w konsoli, w zależności od liczby ubezpieczycieli.
* Inicjalizuje parametry symulacji, takie jak liczba iteracji, krok zapisu wyników oraz liczba ubezpieczycieli.
* Tworzy folder wyjściowy, w którym będą zapisywane wyniki symulacji.
* Uruchamia wątki odpowiedzialne za zapis danych w tle (watekZapisPierwotny i

watekZapisRozprz).

* Dla każdej iteracji symulacji uruchamia zadania równoległe w puli wątków pool, które wykonują symulację ekspozycji.
* Po zakończeniu symulacji czeka na zakończenie wszystkich zadań w puli wątków i zapisuje wyniki do plików CSV.
* Wyświetla czas trwania symulacji oraz informację o poprawnym zapisaniu wyników.

### Zakończenie

Na końcu funkcja wyświetla status końcowy, informując o poprawnym zakończeniu wczytywa- nia danych lub zapisaniu wyników symulacji do plików. W przypadku choice == 2, funkcja dodatkowo tworzy strukturę katalogów i zapisuje wyniki w odpowiednich podfolderach, takich jak Brutto, Brutto Kat, Netto i Netto Kat.

### Kluczowe elementy diagramu

Diagram sekwencji przedstawia interakcje pomiędzy następującymi elementami:

**Użytkownik** – inicjuje funkcję testALL, przekazując jej parametr choice.

**Funkcja** testALL – główny element, który zarządza całym procesem wczytywania danych lub wykonywania symulacji.

**Pule wątków (**pool **i** poolFiles**)** – umożliwiają równoległe przetwarzanie danych oraz operacje na plikach.

**Wektory danych (**testVec **i** fileNames**)** – przechowują przefiltrowane nazwy ubezpie- czycieli.

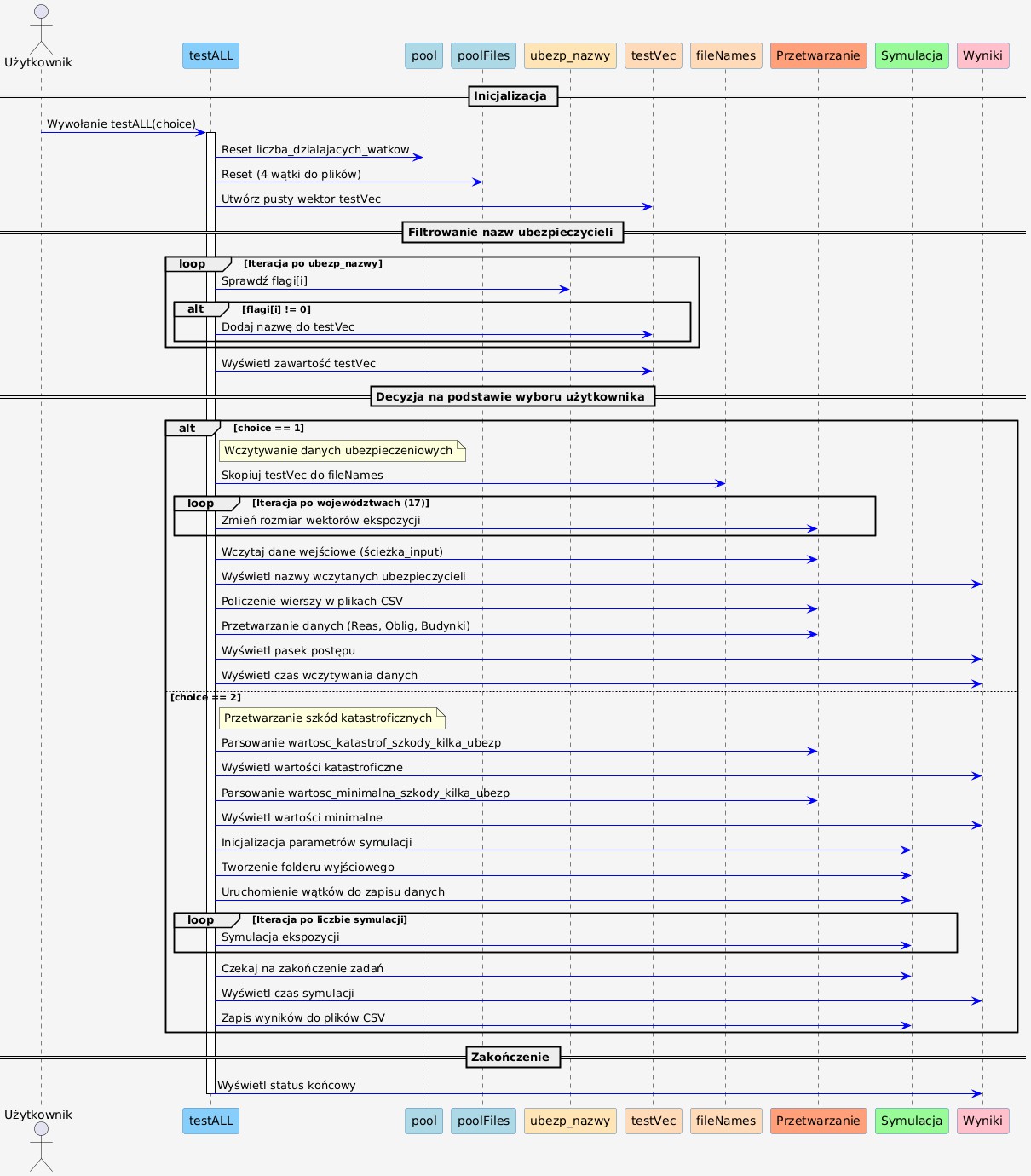
**Przetwarzanie danych (**Processing**)** – obejmuje wczytywanie i przetwarzanie plików wejściowych.

**Symulacja (**Simulation**)** – odpowiada za wykonywanie równoległych symulacji szkód katastroficznych.

**Wyniki (**Output**)** – odpowiada za wyświetlanie informacji o postępie, czasie przetwarza- nia oraz zapis wyników do plików.

### Diagram sekwencji

Diagram sekwencji przedstawiający przebieg działania funkcji testALL znajduje się na rycinie 2.9.



Rysunek 2.9: Diagram sekwencji funkcji testALL

### Podsumowanie

Diagram w pełni odzwierciedla logikę funkcji testALL, podzielonej na etapy inicjalizacji, fil- trowania danych, decyzji na podstawie parametru choice, przetwarzania danych lub symulacji oraz zakończenia. Dzięki zastosowaniu diagramu sekwencji, procesy równoległe oraz interakcje pomiędzy poszczególnymi komponentami zostały przedstawione w sposób czytelny i uporząd- kowany.

* + 1. **Funkcja** simulateExponsureTEST

Funkcja simulateExponsureTEST symuluje skutki pożarów w różnych lokalizacjach w zależności od parametrów takich jak:

liczba ubezpieczycieli,

wielkość strat,

reasekuracja,

rozprzestrzenianie się pożarów.

Wyniki symulacji są zapisywane w strukturach danych i mogą być eksportowane do plików CSV.

##### Rozmiar kroku symulacji:

step size = 1*.*0

17 · 12 + ilosc ubezpieczycieli

Oznacza wielkość kroku postępu dla każdego wątku. Liczba 17 · 12 odpowiada liczbie województw (17) i miesięcy (12).

##### Rozmiar kroku paska postępu:

bar step = 1*.*0

(17 + ilosc ubezpieczycieli) · sim

Oznacza wielkość kroku dla paska postępu, uwzględniając liczbę województw, ubezpie- czycieli i symulacji.

##### Wielkość pożaru w procentach:

wielkosc pozar kwota = wielkosc pozar procent·exponsure sum value[*woj*][*mies*][*nr budynku*] Oznacza wartość strat w wyniku pożaru dla konkretnego budynku.

##### Minimalna wartość strat:

wielkosc pozar kwota − 500 Wartość strat nie może być mniejsza niż 500.

##### Opis działania funkcji

**Inicjalizacja zmiennych**:

Funkcja definiuje zmienne takie jak rozmiar kroku, liczba budynków i wartości strat.

##### Iteracja przez województwa i miesiące:

Symulacja jest wykonywana dla każdego województwa i miesiąca.

##### Losowanie liczby pożarów:

Liczba pożarów jest losowana na podstawie rozkładu dwumianowego.

##### Obliczanie strat:

Dla każdego budynku obliczana jest wielkość strat w procentach i kwotach.

##### Uwzględnienie reasekuracji:

Straty są dzielone pomiędzy ubezpieczycieli z uwzględnieniem reasekuracji.

##### Zapisywanie wyników:

Wyniki są zapisywane w strukturach danych i mogą być eksportowane do plików CSV.

##### Opis diagramu sekwencji

Diagram przedstawia przepływ danych i wywołań funkcji w programie symulacyjnym ‘simulate- ExponsureTEST‘. Funkcja ta jest kluczowym elementem systemu symulacji ryzyka pożarowego i reasekuracyjnego. Główne elementy diagramu to:

**simulateExponsureTEST** – główna funkcja odpowiedzialna za inicjalizację i zarządza- nie przepływem symulacji.

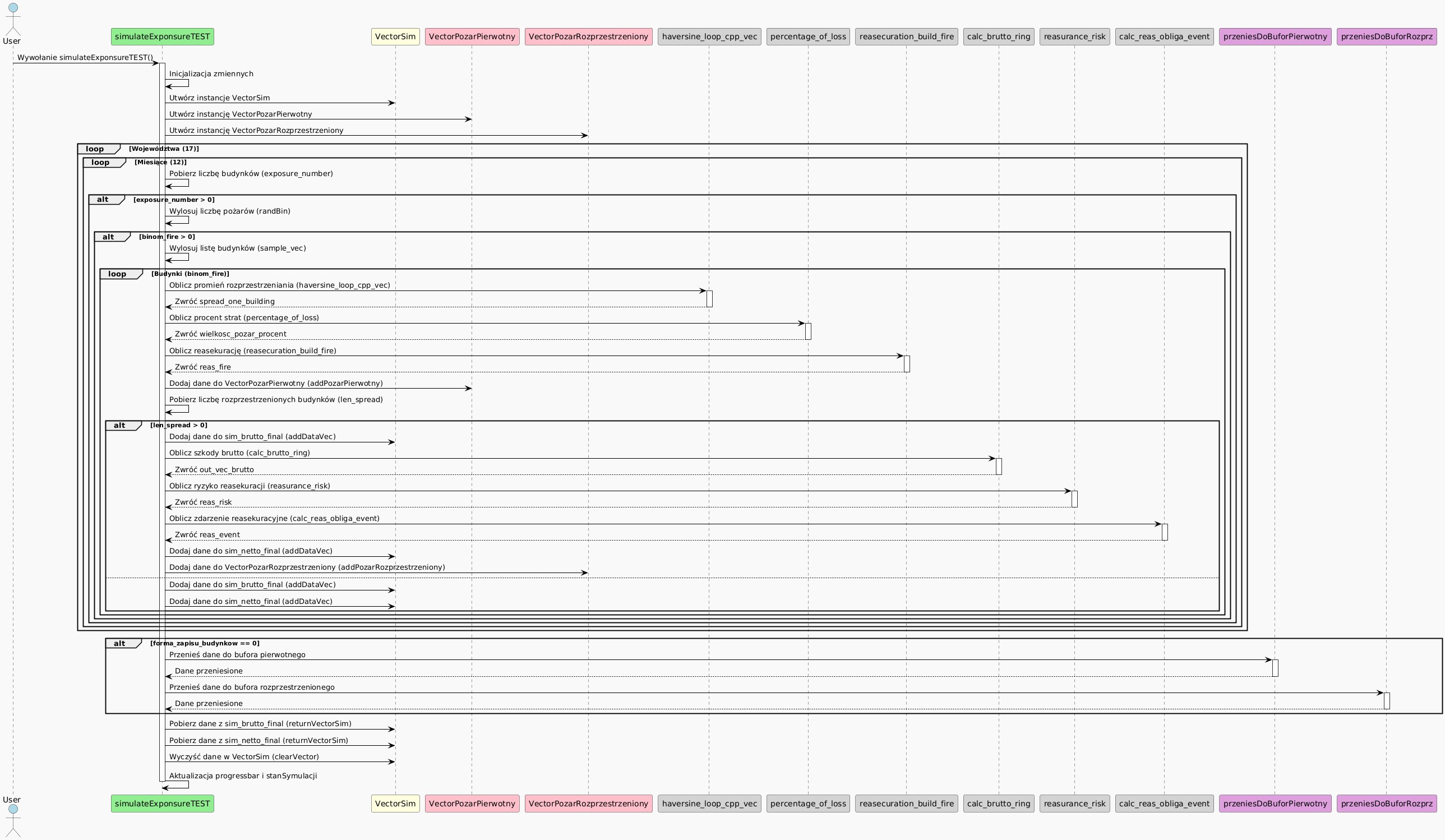
**VectorSim, VectorPozarPierwotny, VectorPozarRozprzestrzeniony** – klasy prze- chowujące dane dotyczące budynków, pożarów pierwotnych i rozprzestrzenionych.

**Funkcje pomocnicze** – takie jak *haversine loop cpp vec*, *percentage of loss*, czy *calc brutto ring*, które realizują obliczenia związane z promieniem rozprzestrzeniania, stratami i ryzykiem reasekuracyjnym.

**Pętle i warunki** – diagram zawiera pętle dla województw, miesięcy i budynków oraz warunki decyzyjne, które określają, czy dane są przetwarzane.

**Bufory** – dane są przenoszone do buforów (‘przeniesDoBuforPierwotny‘, ‘przeniesDoBu- forRozprz‘) przed zapisaniem.

Poniżej znajduje się diagram sekwencji ilustrujący ten proces:



Rysunek 2.10: Diagram sekwencji funkcji simulateExponsureTEST.

* + 1. **Funkcja** randZeroToOne

Funkcja randZeroToOne generuje losową liczbę rzeczywistą z przedziału [*a, b*] przy użyciu roz- kładu jednostajnego. Matematycznie, jeśli *X* ∼ *U* (*a, b*), to *X* jest zmienną losową o rozkładzie jednostajnym na przedziale [*a, b*], gdzie wszystkie wartości z tego przedziału mają jednakowe prawdopodobieństwo.



*P* (*X* = *x*) =

1

*b−a*

*,* dla *x* ∈ [*a, b*]*,*

0*,* dla *x* ∈*/* [*a, b*]*.*

Listing 2.5: Funkcja randZeroToOne

double rand Zero To One ( int a, int b)

{

std :: random\_device rd; std :: mt19937 gen ( rd ());

std :: uniform\_real\_distribution < double > distribution ; distribution . param ( std :: uniform\_real\_distribution < double >:: param\_type (a, b))

*‹→* ;

return distribution ( gen );

}

* + 1. **Funkcja** sample vec

Funkcja sample vec losowo wybiera próbkę o rozmiarze *n* (sampleSize) z populacji population.

Wybór odbywa się poprzez losowanie indeksów elementów z populacji za pomocą funkcji randZeroToOne.

Dla populacji *P* = {*p*1*, p*2*, . . . , pN* } i próbki *S* = {*s*1*, s*2*, . . . , sn*}:

*S* = {*P* [*i*] | *i* ∼ *U* (0*, N* − 1)}*, i* = 1*, . . . , n.*

Listing 2.6: Funkcja sample vec

std :: vector <int > sample\_vec ( std :: vector <int >& population , int sampleSize )

{

std :: vector <int > sampleData ( sampleSize ); for ( int i = 0; i < sampleSize ; i++)

{

int random Index = rand Zero To One (0.0 , population . size () - 1); sampleData [ i] = population [ random Index ];

}

return sampleData ;

}

* + 1. **Funkcja** randBin

Funkcja randBin generuje losową liczbę całkowitą na podstawie rozkładu dwumianowego *B*(*n, p*), gdzie *n* to liczba prób (size exp), a *p* to prawdopodobieństwo sukcesu (prob size).

*P* (*X* = *k*) = 3*n*4*pk*(1 − *p*)*n−k, k* = 0*,* 1*, . . . , n.*

*k*

Listing 2.7: Funkcja randBin

int rand Bin ( int size\_exp , double prob\_size )

{

std :: random\_device rd; std :: mt19937 gen ( rd ());

std :: binomial\_distribution <> distrib ( size\_exp , prob\_size );

return distrib ( gen );

}

* + 1. **Funkcja** search closest

Funkcja search closest wyszukuje indeks najbliższego elementu *x* w posortowanej tablicy *A* = {*a*1*, a*2*, . . . , an*}. Używa algorytmu wyszukiwania binarnego (std::lower bound) w celu optymalizacji.

closest(*x*) = arg min *ai* − *x, i* = 1*, . . . , n.*

*i*

Listing 2.8: Funkcja search closest

int search\_closest ( const std :: vector < double >& sorted\_array , double x)

{

auto iter\_geq = std :: lower\_bound ( sorted\_array . begin (), sorted\_array . end (),

x);

if ( iter\_geq == sorted\_array . begin ())

{

return 0;

}

else if ( iter\_geq == sorted\_array . end ())

{

return sorted\_array . size () - 1;

}

double a = \*( iter\_geq - 1); double b = \*( iter\_geq );

if ( fabs( x - a) < fabs( x - b))

{

return iter\_geq - sorted\_array . begin () - 1;

}

return iter\_geq - sorted\_array . begin ();

}

* + 1. **Funkcja** percentage of loss

Funkcja percentage of loss wyznacza procent strat w zależności od wielkości pożaru i jego prawdopodobieństwa. Wykorzystuje funkcję randZeroToOne do generowania losowego praw- dopodobieństwa oraz funkcję search closest do znalezienia najbliższej wartości w wektorze prawdopodobieństw.

percentage of loss = exponsure sensitiv[search closest(probability*,* randZeroToOne(0*,* 1))]*.*

Listing 2.9: Funkcja percentage of loss

double percentage\_of\_loss ( std :: vector < std :: vector < double >> wielkosc\_pozaru )

{

int ind\_prob ; double exp\_sen ; double val\_dist;

val\_dist = rand Zero To One (0 , 1); std :: vector < double > probability ; probability = wielkosc\_pozaru [1]; std :: vector < double > exponsure\_sensitiv ; exponsure\_sensitiv = wielkosc\_pozaru [0];

ind\_prob = search\_closest ( probability , val\_dist); exp\_sen = exponsure\_sensitiv [ ind\_prob ];

return ( exp\_sen );

}

* + 1. **Funkcja** calc reas bligator

Funkcja calc reas bligator oblicza wysokość ryzyka przypisanego do reasekuracji obowiąz- kowej w zależności od sumy składek ubezpieczeniowych (sum prem) oraz parametrów ryzyka obowiązkowego (vec obligat insur risk).

Matematycznie funkcja realizuje następujące operacje:



vec obligat insur risk[3] · sum prem*,*

dla sum prem *<* vec obligat insur risk[0]*,*



out obl = vec obligat insur risk[2] · vec obligat insur risk[0]*,*

dla vec obligat insur risk[0] “ sum prem *<* vec obligat insur risk[1]*,*



sum prem − (vec obligat insur risk[1] − vec obligat insur risk[0])*,*



dla sum prem − vec obligat insur risk[1]*.*

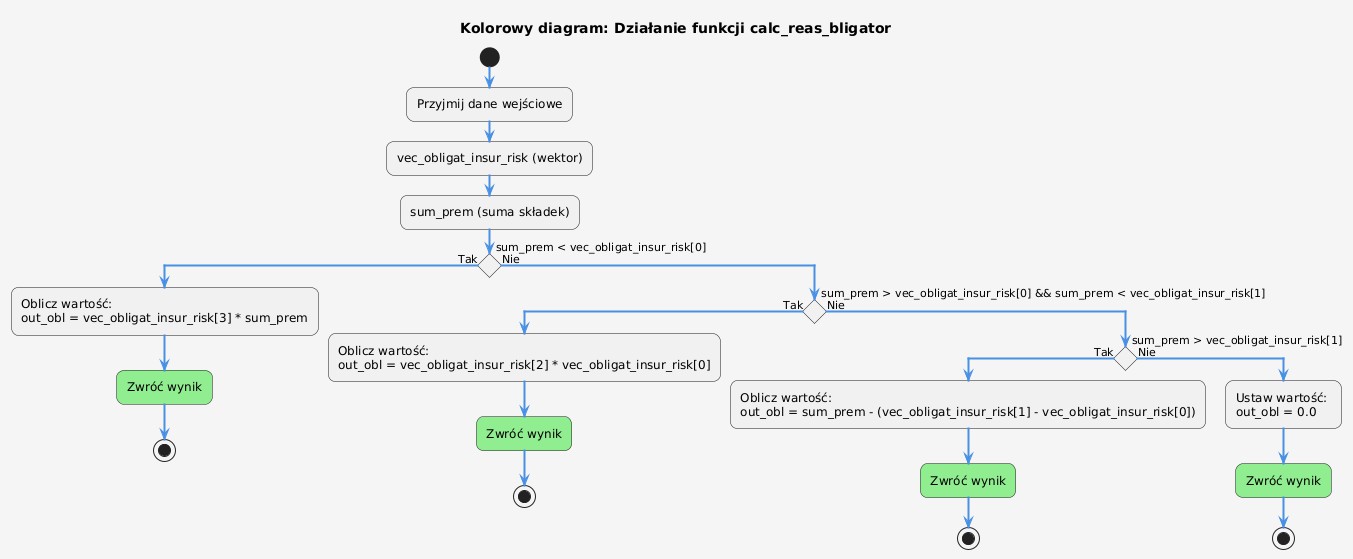
vec obligat insur risk - wektor zawierający dane ryzyka ubezpieczeniowego,

sum prem - suma składek.

Logika działania funkcji opiera się na sprawdzeniu kilku warunków i odpowiednim obliczeniu wartości wyjściowej out obl. Aby lepiej zrozumieć proces przepływu danych w funkcji, poniżej przedstawiono diagram kolejki.

##### Diagram kolejki funkcji

Diagram kolejki ilustruje przepływ danych i procesy zachodzące w funkcji calc reas bligator.



Rysunek 2.11: Diagram kolejki funkcji calc reas bligator

##### Opis działania funkcji

Funkcja działa w następujący sposób:

* + - 1. Użytkownik wywołuje funkcję calc reas bligator, przekazując dwa argumenty: vec obligat insur ris

oraz sum prem.

* + - 1. Funkcja pobiera dane wejściowe:

vec obligat insur risk - wektor zawierający wartości graniczne i współczynniki ryzyka,

sum prem - suma składek.

* + - 1. Funkcja sprawdza kolejne warunki:

Jeśli sum prem < vec obligat insur risk[0], obliczana jest wartość:

out obl = vec obligat insur risk[3] \* sum prem.

Jeśli sum prem > vec obligat insur risk[0] oraz sum prem < vec obligat insur risk[1], obliczana jest wartość:

out obl = vec obligat insur risk[2] \* vec obligat insur risk[0].

Jeśli sum prem > vec obligat insur risk[1], obliczana jest wartość:

out obl = sum prem - (vec obligat insur risk[1] - vec obligat insur risk[0]).

W przeciwnym przypadku out obl jest ustawiane na 0.0.

* + - 1. Po wykonaniu obliczeń funkcja zwraca wynik out obl.

##### Wnioski

Diagram kolejki funkcji calc reas bligator pozwala lepiej zrozumieć przepływ danych oraz logikę działania funkcji. Dzięki wizualizacji można łatwiej przeanalizować warunki oraz sposób obliczania wartości wyjściowej.

* + 1. **Funkcja** reasecuration build fire

Funkcja reasecuration build fire oblicza wartość ryzyka związanego z reasekuracją w przy- padku pożaru dla danego budynku. Wykorzystuje parametry reasekuracji obowiązkowej i fa- kultatywnej oraz wcześniejsze obliczenia dotyczące ekspozycji na ryzyko pożaru.

Matematycznie:

Jeśli budynek jest objęty reasekuracją fakultatywną:

reas fakultat = exp fire pre·*bf* +max !0*,* (1−*bf* )·exp fire pre−vec fakul insur val[1]"*.*

W przeciwnym przypadku:

reas fakultat = min exp fire pre*,* vec fakul insur val[0] +

! "

max !0*,* exp fire pre − vec fakul insur val[0] − vec fakul insur val[1]"*.* (2.1)

Wartość reasekuracji obowiązkowej jest obliczana za pomocą funkcji calc reas bligator.

## Opis logiki funkcji

Na rycinie 2.12 przedstawiono diagram obrazujący proces obliczania wartości zmiennej reas oblig na podstawie danych wejściowych dotyczących ekspozycji na ryzyko pożaru (exp fire pre), re- gionu (woj, mies) oraz numeru budynku (nr budynku).

Logika funkcji składa się z następujących kroków:

**Krok 1: Odczytanie danych wejściowych**: Funkcja rozpoczyna od odczytania danych wejściowych:

exp fire pre – wartość ekspozycji na ryzyko pożaru,

woj, mies – identyfikatory regionu (województwo i miesiąc),

nr budynku – numer budynku.

**Krok 2: Inicjalizacja zmiennych**: Następnie inicjalizowane są zmienne:

reas – wartość reasekuracji,

vec fakul insur num, vec fakul insur val – wektory dotyczące reasekura- cji fakultatywnej,

vec obligat insur risk – wektor dotyczący reasekuracji obligatoryjnej,

reas fakultat i reas oblig – początkowo ustawione na wartość exp fire pre.

**Krok 3: Obliczenie wartości** reas: Wartość reas jest obliczana na podstawie tablicy

exponsure reassurance, z uwzględnieniem danych wejściowych (woj, mies, nr budynku).

**Krok 4: Sprawdzenie warunku** reas < 999: Jeśli reas jest mniejsze niż 999, wykony- wane są kolejne operacje:

1. **Sprawdzenie obecności** reas **w** vec fakul insur num: Jeśli reas znajduje się w wektorze vec fakul insur num, wykonywane są obliczenia reasekuracji fakultatywnej:

Wartość b f jest pobierana z vec fakul insur val.

Obliczana jest wartość reas fakultat według wzoru:

reas fakultat = exp fire pre · b f+

1 2

max 0*,* (1 − b f) · exp fire pre − vec fakul insur val[reas][1]

(2.2)

Wartość reas oblig jest aktualizowana na podstawie reas fakultat.

1. **Obliczenia alternatywne dla reasekuracji fakultatywnej**: Jeśli reas nie znajduje się w vec fakul insur num, obliczana jest wartość reas fakultat według wzoru:

reas fakultat = min 1exp fire pre*,* vec fakul insur val[reas][0]2+

1 2

max 0*,* exp fire pre−vec fakul insur val[reas][0]−vec fakul insur val[reas][1]

(2.

Wartość reas oblig jest również aktualizowana na podstawie reas fakultat.

**Krok 5: Sprawdzenie warunku obligatoryjnego**: Jeśli wartość floor(vec obligat insur risk[0])

jest większa lub równa 0, obliczana jest wartość reas oblig za pomocą funkcji

calc reas bligator. Funkcja ta uwzględnia:

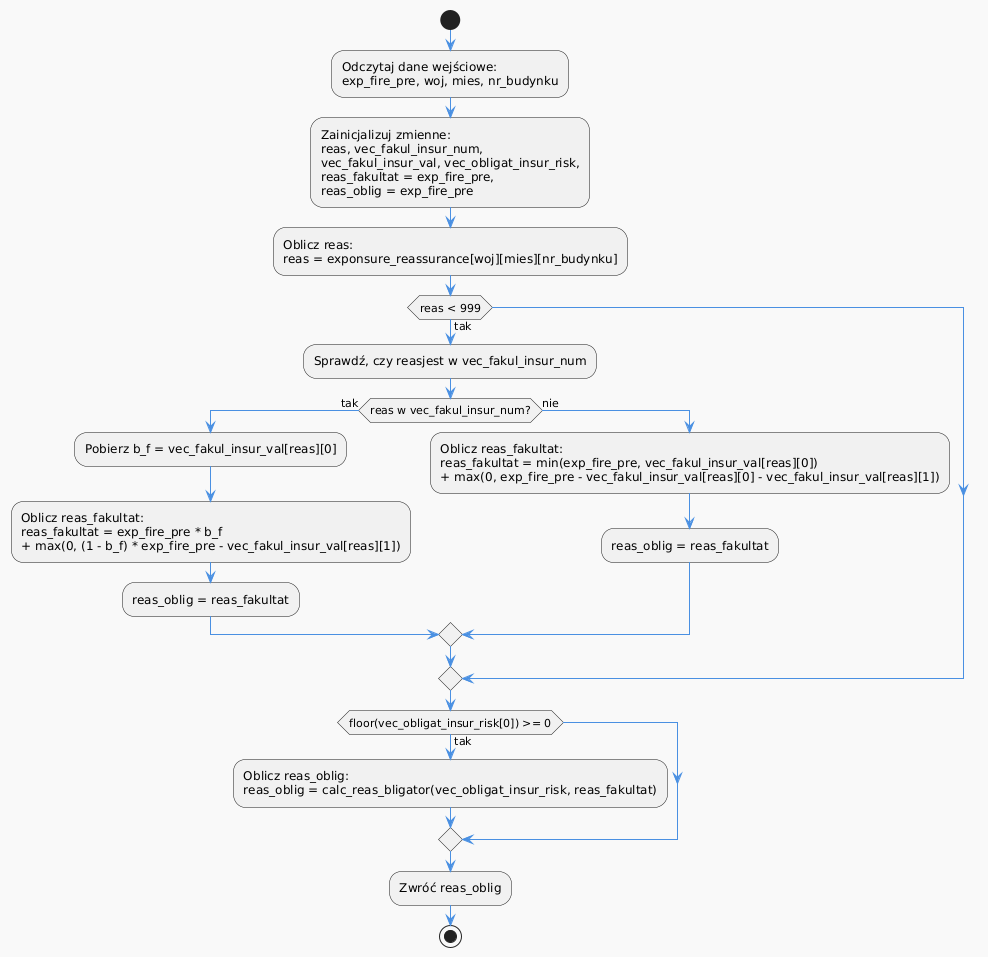
wektor ryzyk obligatoryjnych (vec obligat insur risk),

wartość reas fakultat.

**Krok 6: Zwrócenie wyniku**: Na końcu funkcja zwraca wartość reas oblig, która repre- zentuje końcową wartość zabezpieczenia reasekuracyjnego.

## Uwagi

Diagram przedstawia szczegółową logikę funkcji, uwzględniając zarówno przypadki reasekuracji fakultatywnej (indywidualnej), jak i obligatoryjnej (grupowej). Każdy krok jest realizowany w sposób warunkowy, zależny od danych wejściowych i wartości pośrednich.



Rysunek 2.12: Przebieg logiki funkcji reasecuration build fire

**Opis funkcji** index spread build

Funkcja index spread build służy do modelowania rozprzestrzeniania się pożarów w oparciu o dane wejściowe dotyczące położenia geograficznego, odległości, ekspozycji oraz parametrów

ubezpieczeniowych. Wynik działania funkcji to tablica danych (std::vector<std::vector<double>>), która zawiera szczegółowe informacje o rozprzestrzenionych pożarach.

### Wejście funkcji

Funkcja przyjmuje następujące parametry:

lat center, lon center – współrzędne geograficzne centrum obszaru.

distance res – macierz odległości dla pierścieni.

lat ring, lon ring – współrzędne geograficzne dla kolejnych pierścieni.

insu ring, reas ring – dane ubezpieczeniowe i przyczyny pożarów dla pierścieni.

exposure sum ring – suma ekspozycji w pierścieniach.

### Działanie funkcji

Funkcja wykonuje następujące kroki:

* + - 1. **Inicjalizacja zmiennych pomocniczych**: Tworzone są wektory przechowujące dane tymczasowe, takie jak odległości, współrzędne, ubezpieczenia, przyczyny pożarów oraz wyniki.
      2. **Iteracja przez pierścienie (0–8)**: Dla każdego pierścienia funkcja:

Sprawdza, czy pierścień zawiera ekspozycje.

Oblicza prawdopodobieństwo rozprzestrzenienia pożaru w oparciu o dane wejściowe oraz parametry rozkładu beta.

Losuje liczbę ekspozycji, na które pożar się rozprzestrzenił.

Dodaje dane o pożarach do wyniku, uwzględniając minimalną wartość strat (500).

* + - 1. **Zwrócenie wyników**: Wyniki są zwracane w formie tablicy out data.

### Wyjście funkcji

Funkcja zwraca tablicę out data, która zawiera następujące dane:

Odległości do ekspozycji.

Współrzędne geograficzne ekspozycji.

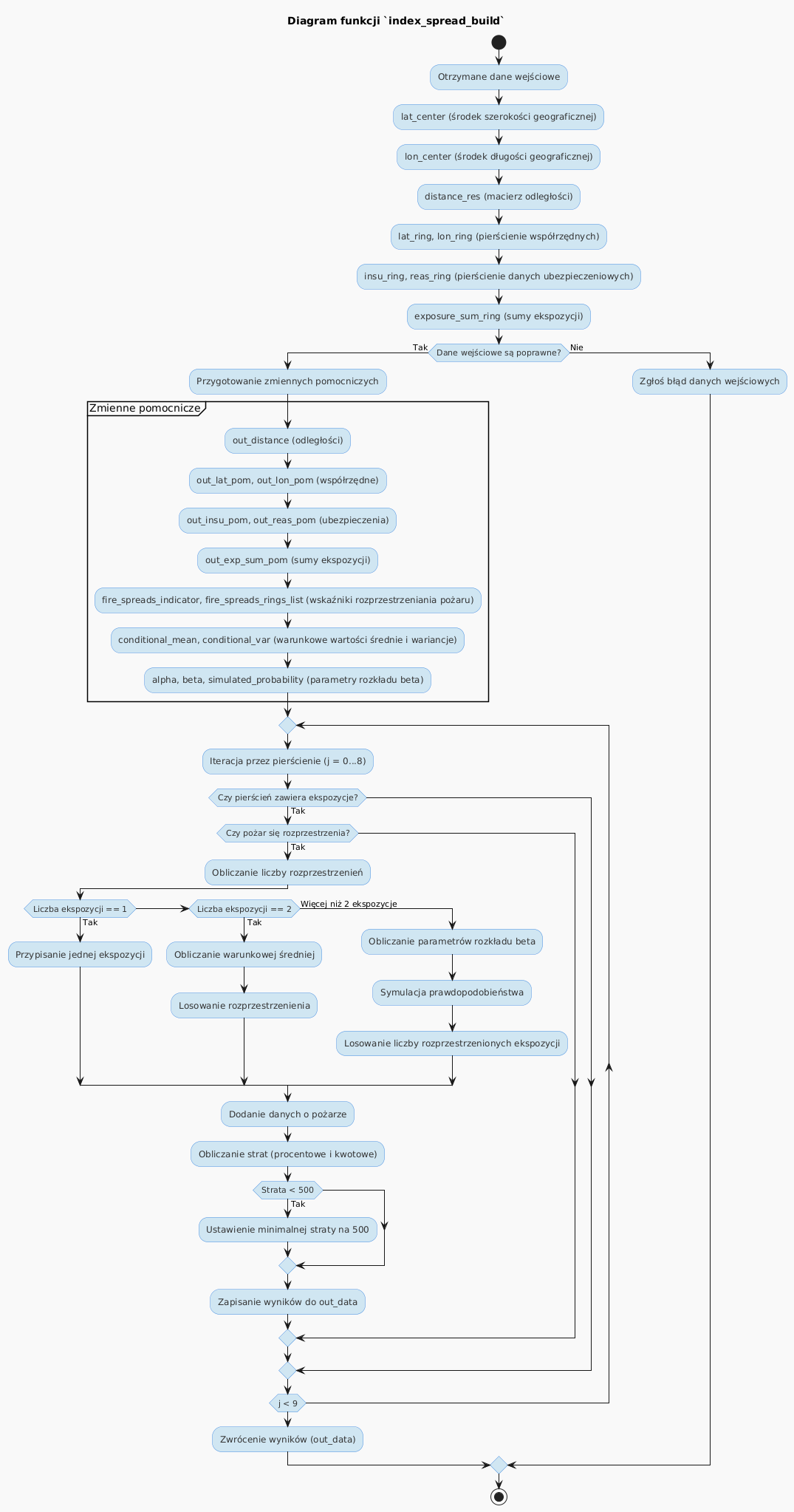
Dane ubezpieczeniowe i przyczyny pożarów.

Suma ekspozycji.

Wielkość strat w wyniku pożaru.

### Diagram przepływu funkcji

Na rycinie 2.13 przedstawiono diagram przepływu funkcji index spread build, który ilustruje jej działanie krok po kroku.



Rysunek 2.13: Diagram przepływu funkcji index spread build

* + 1. **Funkcja** mean spread function

Funkcja mean spread function oblicza średnią wartość rozprzestrzeniania się pożaru w za- leżności od pierścienia (ring), ekspozycji (exposure) oraz parametrów *µ*. Matematycznie jest zdefiniowana jako:

exposure

*µ*spread = 1 + exp(−*µ*0) · ring*−µ*1 · exposure*−µ*2 *,*

gdzie:

*µ*0*, µ*1*, µ*2 są parametrami z wektora mu spread parameters,

exp(−*µ*0) jest funkcją wykładniczą,

ring to numer pierścienia (odległość od centrum),

exposure to liczba obiektów w danym pierścieniu.

**Funkcja** var spread function

Funkcja var spread function oblicza wariancję rozprzestrzeniania się pożaru w pierścieniu, uwzględniając średnią wartość rozprzestrzeniania (*µ*spread) oraz parametry wariancji (*σ*):

*σ*2 = *µ*spread · 31 −  *µ*spread 4 · 31 + exposure − 1 4 *,*

spread

gdzie:

exposure

1 + exp(−*σ*0) · ring*−σ*1 · exposure*−σ*2

*µ*spread jest obliczane za pomocą funkcji mean spread function,

*σ*0*, σ*1*, σ*2 są parametrami z wektora sigma spread parameters.

**Funkcja** haversine cpp

Funkcja haversine cpp oblicza odległość między dwoma punktami na powierzchni Ziemi za pomocą wzoru:

*d* = 2*r* · arcsin Aósin2 3 ∆*φ* 4 + cos(*φ*1) · cos(*φ*2) · sin2 3 ∆*λ* 4B *,*

2 2

gdzie:

*r* to promień Ziemi (domyślnie 6378137 metrów),

*φ*1*, φ*2 to szerokości geograficzne (w radianach),

*λ*1*, λ*2 to długości geograficzne (w radianach),

∆*φ* = *φ*2 − *φ*1,

∆*λ* = *λ*2 − *λ*1.

**Diagram przepływu funkcji** haversine loop cpp vec

Na rycinie 2.14 przedstawiono diagram przepływu funkcji haversine loop cpp vec, który ilu- struje główne kroki i logikę działania tej funkcji.

### Opis diagramu

##### Inicjalizacja zmiennych:

Funkcja rozpoczyna się od inicjalizacji zmiennych oraz obliczenia współrzędnych cen- tralnych (lat center, lon center) i granic obszaru (south lat, north lat, west lon, east lon).

##### Iteracja po danych:

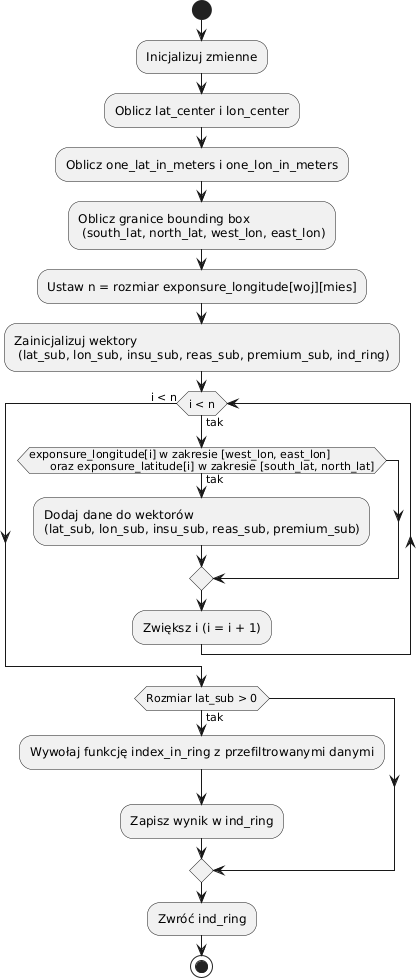
Pętla while iteruje przez wszystkie dane w zadanym województwie (woj) i miesiącu (mies). Dla każdego punktu sprawdzane jest, czy znajduje się w granicach obszaru. Jeśli tak, dane są dodawane do odpowiednich wektorów.

**Wywołanie funkcji** index in ring:

Jeśli istnieją dane w wektorach, są one przetwarzane przez funkcję index in ring, a wynik jest zapisywany w zmiennej ind ring.

##### Zwrócenie wyniku:

Na końcu funkcja zwraca wynik w postaci dwuwymiarowego wektora ind ring.



Rysunek 2.14: Diagram przepływu funkcji haversine loop cpp vec.

* + 1. **Funkcja** index in ring

### Cel funkcji

Funkcja index in ring jest kluczowym elementem modelowania przestrzennego rozprzestrze- niania się pożarów. Jej celem jest przypisanie obiektów (np. budynków, pól ubezpieczeniowych) do odpowiednich pierścieni wokół centrum pożaru, na podstawie odległości geograficznej. Pier- ścienie są definiowane jako zbiory obiektów znajdujących się w określonych przedziałach odle- głości od centrum.

### Opis działania

Funkcja wykorzystuje współrzędne geograficzne (lat center, lon center) centrum pożaru oraz współrzędne obiektów (lat sub, lon sub), aby obliczyć odległość każdego obiektu od centrum. Odległość ta jest obliczana za pomocą funkcji haversine cpp, która implementuje wzór haver- sine na sferyczną odległość między dwoma punktami na powierzchni Ziemi:

*d* = 2*r* · arcsin Aósin2 3 ∆*φ* 4 + cos(*φ*1) · cos(*φ*2) · sin2 3 ∆*λ* 4B *,*

2 2

gdzie:

*r* to promień Ziemi (domyślnie 6378137 metrów),

*φ*1*, φ*2 to szerokości geograficzne (w radianach),

*λ*1*, λ*2 to długości geograficzne (w radianach),

∆*φ* = *φ*2 − *φ*1,

∆*λ* = *λ*2 − *λ*1.

Obiekty są następnie przypisywane do jednego z dziewięciu pierścieni, zdefiniowanych jako przedziały odległości:

Pierścień 1: 0 “ *d <* 0*.*005 km,

Pierścień 2: 0*.*005 “ *d <* 25 km,

Pierścień 3: 25 “ *d <* 50 km,

Pierścień 4: 50 “ *d <* 75 km,

Pierścień 5: 75 “ *d <* 100 km,

Pierścień 6: 100 “ *d <* 125 km,

Pierścień 7: 125 “ *d <* 150 km,

Pierścień 8: 150 “ *d <* 175 km,

Pierścień 9: 175 “ *d <* 200 km.

Dla każdego pierścienia przechowywane są następujące informacje:

Odległości obiektów od centrum (distance res),

Współrzędne geograficzne obiektów (lat ring, lon ring),

Informacje o ubezpieczeniu (insu ring),

Informacje o reasekuracji (reas ring),

Suma ekspozycji (exponsure sum ring).

Na końcu funkcja wywołuje index spread build, która modeluje dalsze rozprzestrzenianie się pożaru w każdym pierścieniu.

### Znaczenie funkcji

Funkcja index in ring umożliwia przestrzenną segmentację danych, co jest kluczowe w mode- lowaniu rozprzestrzeniania się pożarów i analizie ryzyka ubezpieczeniowego. Dzięki niej możliwe jest dokładne określenie, które obiekty znajdują się w zasięgu pożaru i w jakim stopniu mogą być zagrożone.

* + 1. **Funkcja** render gui()

Funkcja render gui() odpowiada za rysowanie graficznego interfejsu użytkownika (GUI) w symulatorze pożarów. Wykorzystuje bibliotekę ImGui do tworzenia interaktywnych elementów, takich jak pola tekstowe, przyciski, tabele i paski postępu. Główne zadania funkcji to:

Pobieranie danych wejściowych od użytkownika (np. rok, ścieżka do folderu z danymi, parametry symulacji).

Wczytywanie listy ubezpieczycieli z plików w folderze.

Umożliwienie wyboru ubezpieczycieli do symulacji.

Konfiguracja parametrów symulacji (np. liczba wątków, liczba symulacji).

Uruchamianie symulacji w osobnych wątkach.

Wyświetlanie pasków postępu dla procesu wczytywania danych i symulacji.

##### Diagram przepływu funkcji

Na poniższym diagramie przedstawiono przepływ danych i interakcji w funkcji render gui(). Diagram przedstawia główne komponenty oraz ich interakcje.



Rysunek 2.15: Diagram przepływu funkcji render gui()

##### Szczegóły diagramu

Diagram zawiera następujące elementy:

**User (Użytkownik)**: Reprezentuje użytkownika, który wchodzi w interakcję z interfej- sem graficznym.

##### Komponenty GUI:

* ImGui::InputInt/InputText: Odpowiada za wprowadzanie danych wejściowych, takich jak liczby całkowite i tekst.
* ImGui::Button: Obsługuje przyciski, które wywołują różne akcje.
  + ImGui::ProgressBar: Wyświetla pasek postępu podczas wczytywania danych lub symulacji.
  + ImGui::BeginChild/BeginTable: Wyświetla sekcje GUI, takie jak listy ubezpieczy- cieli.

##### Backendowe komponenty:

* + std::thread: Reprezentuje wątki uruchamiane w tle (np. do wczytywania danych lub uruchamiania symulacji).
  + getFiles(): Funkcja odpowiedzialna za wczytywanie listy plików z folderu.
  + std::vector: Przechowuje dane, takie jak lista ubezpieczycieli.

##### Interpretacja diagramu

Przepływ działania funkcji można podzielić na następujące etapy:

* + - 1. Użytkownik wprowadza dane wejściowe (rok, ścieżka do folderu) za pomocą pól tekstowych (InputText).
      2. Po kliknięciu przycisku "Wczytaj listę ubezpieczycieli" wywoływana jest funkcja getFiles(), która wczytuje listę plików z folderu. Dane są przechowywane w wektorze (std::vector).
      3. Użytkownik może wybrać ubezpieczycieli lub zaznaczyć wszystkich przy użyciu przycisku

"Wybierz wszystkich".

* + - 1. Parametry symulacji (liczba symulacji, liczba wątków itp.) są konfigurowane za pomocą pól wejściowych.
      2. Kliknięcie przycisku "Wczytaj dane" uruchamia wątek (std::thread) odpowiedzialny za wczytywanie danych.
      3. Pasek postępu (ProgressBar) aktualizuje się w czasie rzeczywistym w trakcie wczytywa- nia danych.
      4. Kliknięcie przycisku "Włącz symulację" uruchamia wątek symulacji, a pasek postępu monitoruje jej postęp.

##### Wnioski

Diagram przepływu funkcji render gui() pokazuje, jak interfejs graficzny i backend współpra- cują w celu realizacji zadań symulatora pożarów. Dzięki zastosowaniu wątków (std::thread) i dynamicznej aktualizacji GUI (np. paski postępu), aplikacja może efektywnie obsługiwać zło- żone procesy w tle.