Politechnika Świętokrzyska w Kielcach

Wydział Elektrotechniki, Automatyki i Informatyki

Technologie obiektowe

Zajęcia projektowe

Konwersja obiektów Python do baz NoSQL: kolumnowe, grafowe oraz dokumentowe

Wykonali:

Dominik Jaroszek

Sebastian Iwan

Błażej Jakubczyk

Kielce, 23.06.2025r.

Spis treści:

1. Bazy danych - Wstęp

1.1. Baza Grafowa – Neo4j

Neo4j to baza grafowa, w której dane są przechowywane jako węzły (nodes) oraz krawędzie (relationships). Każdy węzeł i krawędź mogą mieć własne właściwości (properties). Zapytania do bazy odbywają się w języku **Cypher**.

W Neo4j dane są reprezentowane jako graf skierowany, np.:

(Alice) --[FRIEND]--> (Bob)

(Bob) --[WORKS_AT]--> (CompanyX)

Neo4j obsługuje następujące typy wartości:

- **Podstawowe typy**
  - String – np. "Alice"
  - Integer – np. 42
  - Float – np. 3.14
  - Boolean – true, false
  - Null – NULL
- **Strukturalne typy**
  - List<T> – np. ["red", "green", "blue"]
  - Map<K, V> – np. {name: "Alice", age: 30}
- **Typy czasowe**
  - Date – np. 2025-03-24
  - Time – np. 12:00:00
  - LocalTime – np. 14:30:45
  - DateTime – np. 2025-03-24T12:00:00Z
  - LocalDateTime – np. 2025-03-24T14:30:45
  - Duration – np. P2DT3H4M (2 dni, 3 godziny, 4 minuty)

1.2. Kolumnowy – Cassandra

Apache Cassandra to baza kolumnowa, przechowująca dane w strukturze podobnej do tabel, ale w sposób bardziej elastyczny niż w relacyjnych bazach danych. W Cassandra dane są podzielone na **klastry**, **kolumny rodziny** i **wiersze**. Każdy wiersz może mieć różne kolumny.

Brak natywnych relacji – każda tabela działa niezależnie. Można odwzorować relacje, ale trzeba używać denormalizacji. Sprawia to, że gdy dana używana w wielu miejscach się zmieni należy zmienić ją w każdym miejscu.

Przykładowa tabela w Cassandra

```
CREATE TABLE users (
   id UUID PRIMARY KEY,
   name TEXT,
   age INT,
   emails LIST<TEXT>
);
```

W Apache Cassandra zapytania można pisać w CQL (Cassandra Query Language). Jego składnia jest inspirowana SQL, co umożliwia programistom łatwą adaptację.

Cassandra obsługuje następujące typy:

- **Podstawowe typy**
  - ascii – znak ASCII, np. 'A'
  - bigint – liczba całkowita 64-bitowa, np. 9223372036854775807
  - blob – dane binarne, np. 0x89ABCDEF
  - boolean – true, false
  - counter – licznik, używany do inkrementacji (+1, -1)
  - decimal – liczby zmiennoprzecinkowe o zmiennej precyzji
  - double – liczby zmiennoprzecinkowe podwójnej precyzji (64-bit)
  - float – liczby zmiennoprzecinkowe pojedynczej precyzji (32-bit)
  - int – liczby całkowite 32-bitowe
  - smallint – liczby całkowite 16-bitowe
  - tinyint – liczby całkowite 8-bitowe
  - text – ciąg znaków (jak string)
  - uuid – identyfikator UUID, np. 123e4567-e89b-12d3-a456-426614174000
  - timeuuid – UUID z informacją o czasie

    - inet – adresy IP (IPv4 i IPv6)
    - varint – liczby całkowite o dowolnej wielkości
- **Strukturalne typy**
    - list<T> – np. ["email1", "email2"]
    - set<T> – np. { "football", "gaming" }
    - map<K, V> – np. { "home": "123456", "work": "654321" }
    - tuple<T1, T2, ...> – np. (1, "A", true)
    - frozen<T> – pozwala przechowywać struktury w niezmienialnej postaci
- **Typy czasowe**
    - timestamp – np. 2025-03-24 14:30:00
    - date – np. 2025-03-24
    - time – np. 14:30:00
    - duration – np. P2DT3H4M

Cassandra **nie obsługuje** typów węzłów czy relacji jak Neo4j.

1.3. JSON - MongoDB

JSON (JavaScript Object Notation) to format przechowywania danych używany przez bazy dokumentowe, np. MongoDB. Dane są zapisane w postaci **obiektów**.

Przykład dokumentu użytkownika:

```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "name": "Alice",
  "age": 30,
  "emails": ["alice@email.com"],
  "address": {
    "street": "Main St",
    "city": "New York"
  }
}
```

W MongoDB zapytania można pisać w MongoDB Query Language (MQL). Dla programistów znających strukturę JavaScript i składnię JSON, ten język jest intuicyjny.

- **Podstawowe typy**
  - String – np. "Alice"
  - Number – np. 42, 3.14
  - Boolean – true, false
  - Null – null
- **Strukturalne typy**
  - Array – np. ["email1", "email2"]
  - Object – np. { "name": "Alice", "age": 30 }

JSON **nie obsługuje** typów binarnych, dat, czasów czy UUID w sposób natywny. W bazach NoSQL takich jak **MongoDB** są one przechowywane w specjalnym formacie BSON, który dodaje obsługę:

- Date – np. ISODate("2025-03-24T14:30:00Z")
- ObjectId – np. 507f1f77bcf86cd799439011 (unikalny identyfikator)
- Binary – np. BinData(0, "QkN0ZXN0")
- Timestamp – np. {t: 1624000000, i: 1}

2. Python

2.1. Typy danych

Typy numeryczne

- int - liczby całkowite
- float - liczby zmiennoprzecinkowe
- complex - liczby zespolone

Typ tekstowy

- str - ciągi znaków (stringi)

Typy sekwencyjne

- list - listy (kolekcje uporządkowane i modyfikowalne)
- tuple - krotki (kolekcje uporządkowane i niemodyfikowalne)

Typ odwzorowania

- dict - słowniki (kolekcje par klucz-wartość)

Typy zestawów

- set - zbiory (kolekcje unikalnych elementów, modyfikowalne)
- frozenset - zamrożone zbiory (kolekcje unikalnych elementów, niemodyfikowalne)

Typ logiczny

- bool - wartości logiczne (True/False)

2.2. Tabela mapowań

Python	MongoDB	Cassandra	Neo4j
int	Int64	Int	Integer
float	Double	Float	Float
complex	Object	Map<K,V>	String
str	String	Text	String
list	Array	List	Map<K,V>
tuple	Array	Tuple	Map<K,V>
dict	Object	map<K,V>	Map<K,V>
set	Array	set	Map<K,V>
frozenset	Array	frozen<T>	Map<K,V>
bool	Boolean	Boolean	Boolean

Neo4j:

Typy podstawowe takie jak liczby całkowite, zmiennoprzecinkowe, wartości logiczne i ciągi znaków są przechowywane bezpośrednio jako właściwości węzła w bazie danych Neo4j. Liczby zespolone wymagają dodatkowej konwersji - są najpierw przekształcane na format tekstowy, a następnie zapisywane jako właściwość węzła. Kolekcje uporządkowane i nieuporządkowane, takie jak listy, krotki, zbiory i zamrożone zbiory, są przetwarzane w sposób bardziej złożony - każdy element kolekcji posiadający własne atrybuty staje się osobnym węzłem w grafie i zostaje połączony z węzłem nadrzędnym odpowiednią relacją. Słowniki są obsługiwane podobnie do innych kolekcji, z tą różnicą, że tylko wartości słownika posiadające własne atrybuty są przekształcane na osobne węzły i łączone relacjami, podczas gdy klucze słownika są pomijane w procesie konwersji.

Cassandra:

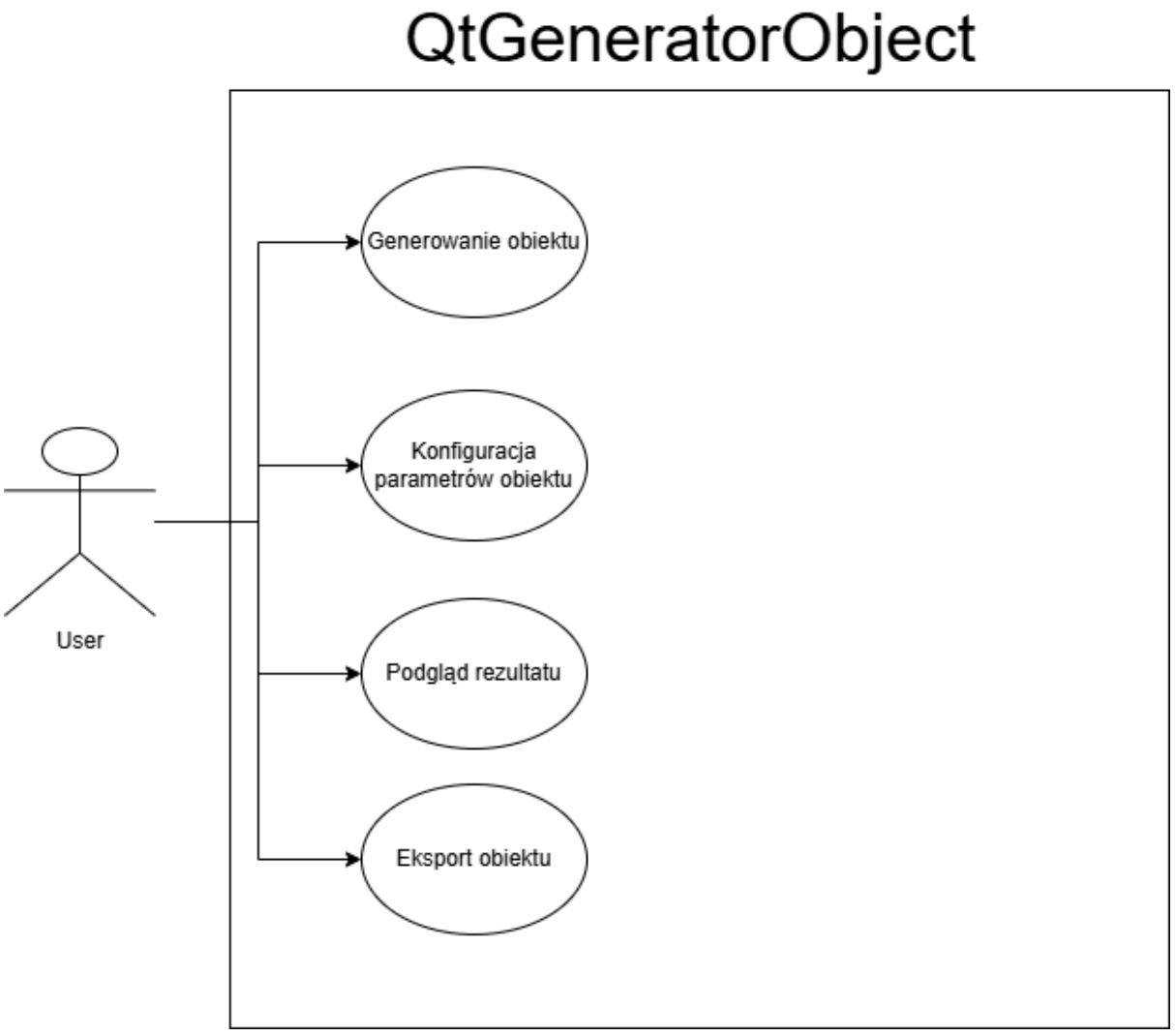
Typy podstawowe takie jak ciągi znaków, wartości logiczne, liczby całkowite i zmiennoprzecinkowe są przechowywane bezpośrednio jako kolumny w tabelach Cassandra z odpowiadającymi im typami CQL (text, boolean, int, float). Identyfikatory UUID również zachowują swoje natywne typy Cassandra. Liczby zespolone wymagają specjalnej konwersji - są przekształcane na mapę zawierającą części rzeczywistą i urojoną jako ciągi znaków wraz z oznaczeniem typu. Kolekcje takie jak listy, krotki, zbiory i zamrożone zbiory są konwertowane na listy tekstowe, gdzie każdy element jest przekształcany na reprezentację tekstową. Wszystkie te kolekcje są przechowywane jako typ list<text> w Cassandra. Słowniki są zapisywane jako typ map<text,text> w Cassandra, gdzie zarówno klucze jak i wartości są konwertowane na reprezentacje tekstowe. Obiekty posiadające atrybuty (__dict__) są serializowane do formatu JSON z dodatkowym polem _type identyfikującym typ obiektu, a następnie przechowywane jako tekst. Każdy obiekt otrzymuje automatycznie wygenerowany identyfikator UUID jako klucz główny tabeli. System automatycznie tworzy tabele na podstawie nazwy klasy obiektu lub typu danych. Gdy obiekt zawiera nowe atrybuty nieobecne w istniejącej tabeli, system dynamicznie dodaje odpowiednie kolumny za pomocą instrukcji ALTER TABLE. Wszystkie obiekty otrzymują dodatkowe pole _type przechowujące informację o oryginalnym typie Python.

MongoDB:

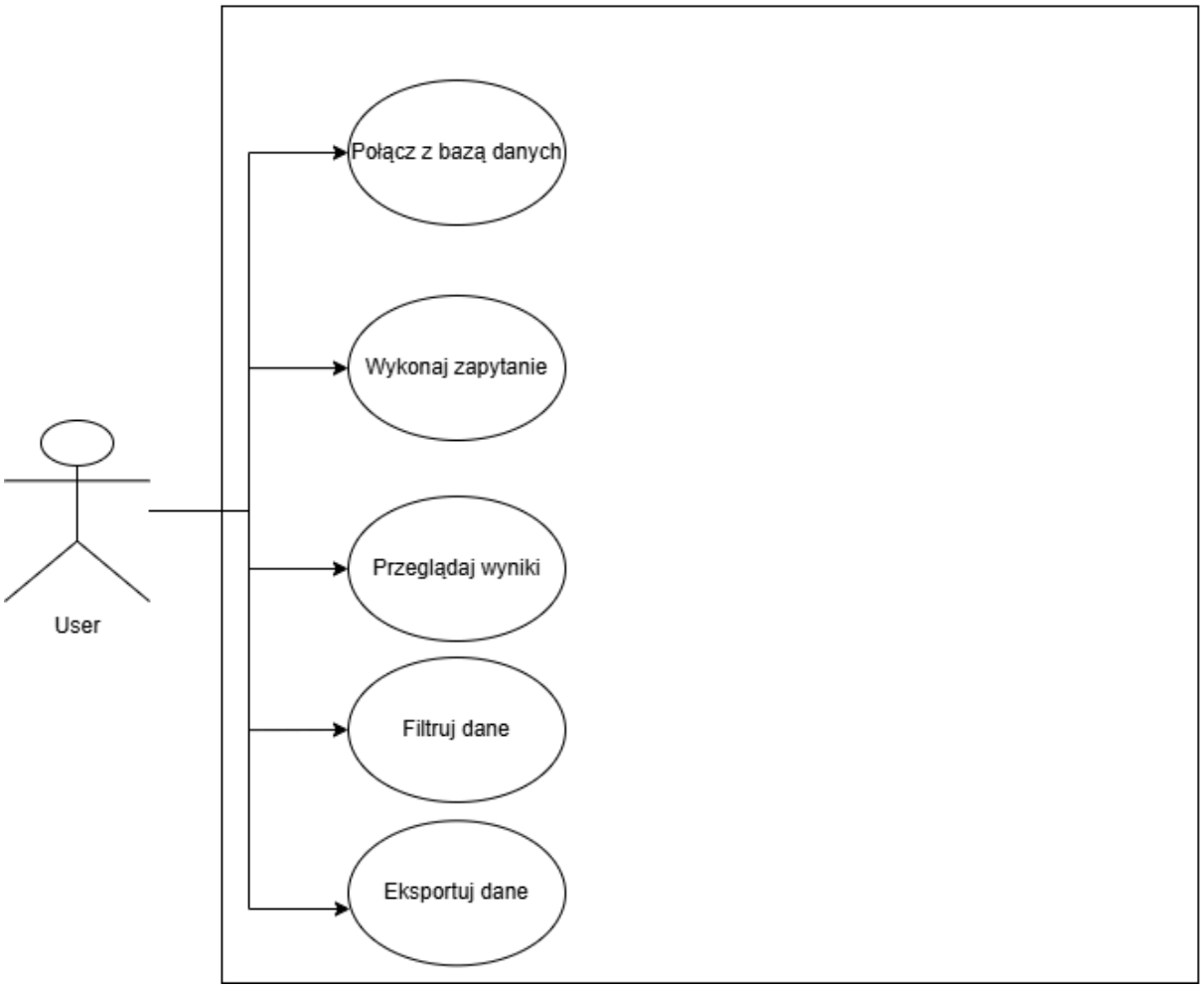
Typy podstawowe takie jak ciągi znaków, liczby całkowite, zmiennoprzecinkowe i wartości logiczne są przechowywane bezpośrednio w MongoDB z zachowaniem swoich natywnych typów BSON (String, Int64, Double,

Boolean). Wartość None jest automatycznie konwertowana na typ Null w MongoDB. Liczby zespolone są przekształcane na dokumenty BSON zawierające części rzeczywistą i urojoną wraz z polem _type identyfikującym typ jako "complex". Kolekcje uporządkowane i nieuporządkowane, takie jak listy, krotki, zbiory i zamrożone zbiory, są konwertowane na tablice BSON (Array), gdzie każdy element kolekcji jest rekurencyjnie przetwarzany zgodnie z regułami konwersji. Słowniki są zapisywane jako dokumenty BSON (Object), gdzie klucze są konwertowane na ciągi znaków, a wartości są rekurencyjnie przetwarzane. Obiekty posiadające atrybuty (__dict__) są serializowane do dokumentów BSON z dodatkowym polem _type zawierającym nazwę klasy obiektu. System automatycznie tworzy kolekcje na podstawie nazwy klasy obiektu i obsługuje wykrywanie cyklicznych referencji, zastępując je specjalnym znacznikiem.

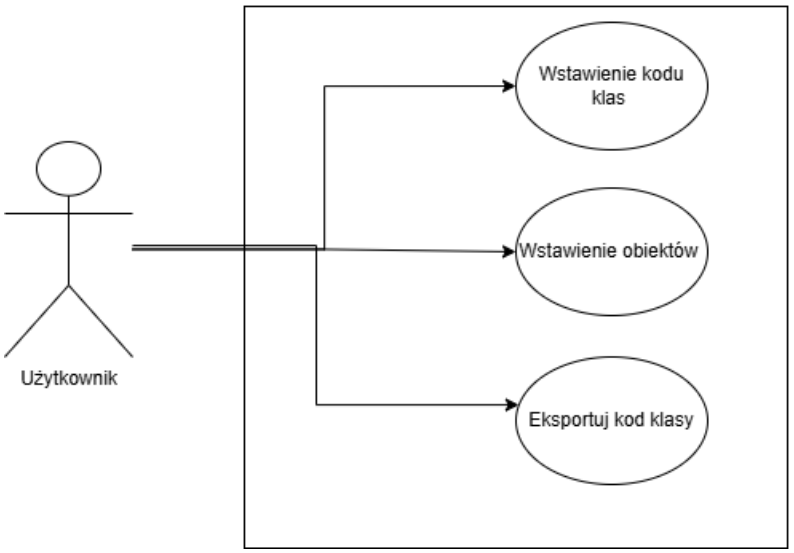
3. Diagramy przypadków użycia



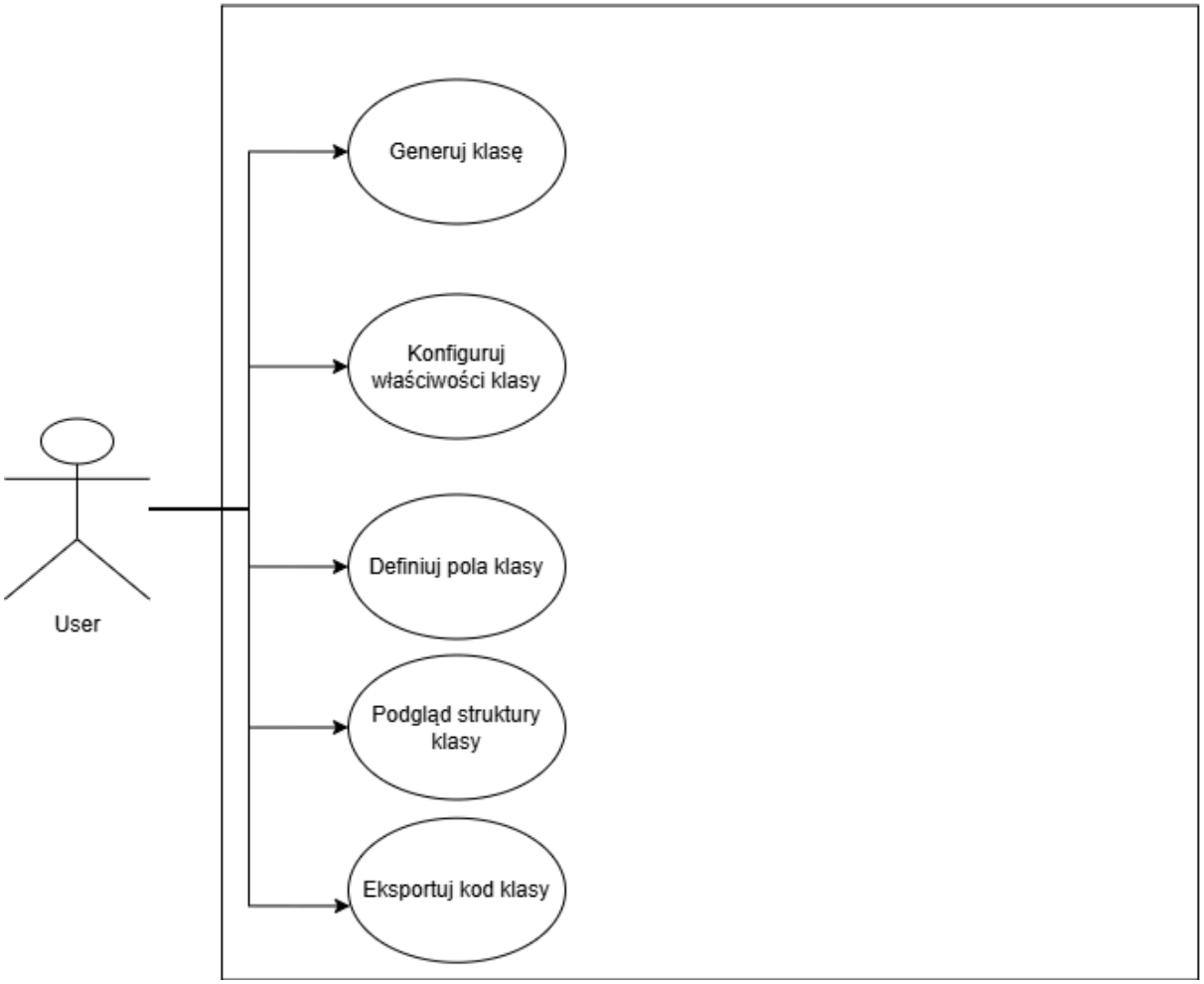
QtDataReader



QtTextEditior



QtGeneratorClass



4. Schematy blokowe

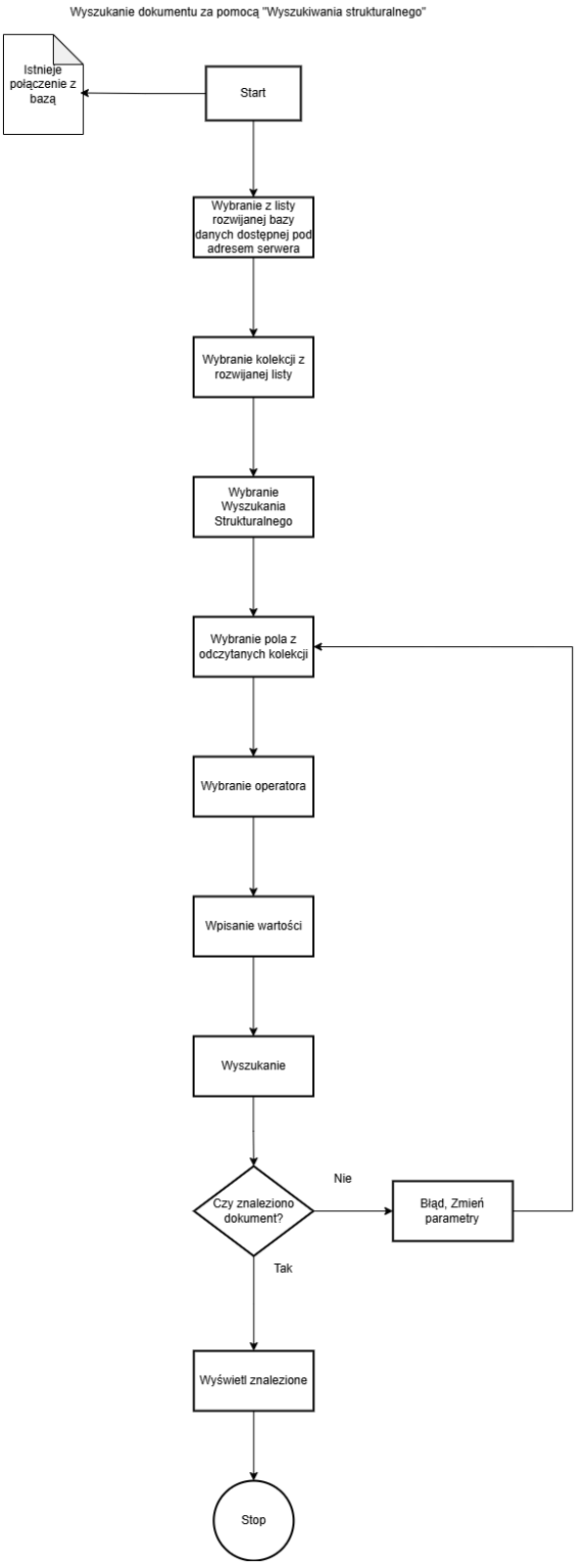


Diagram 1. Wyszukiwanie strukturalne w QtObjectReader

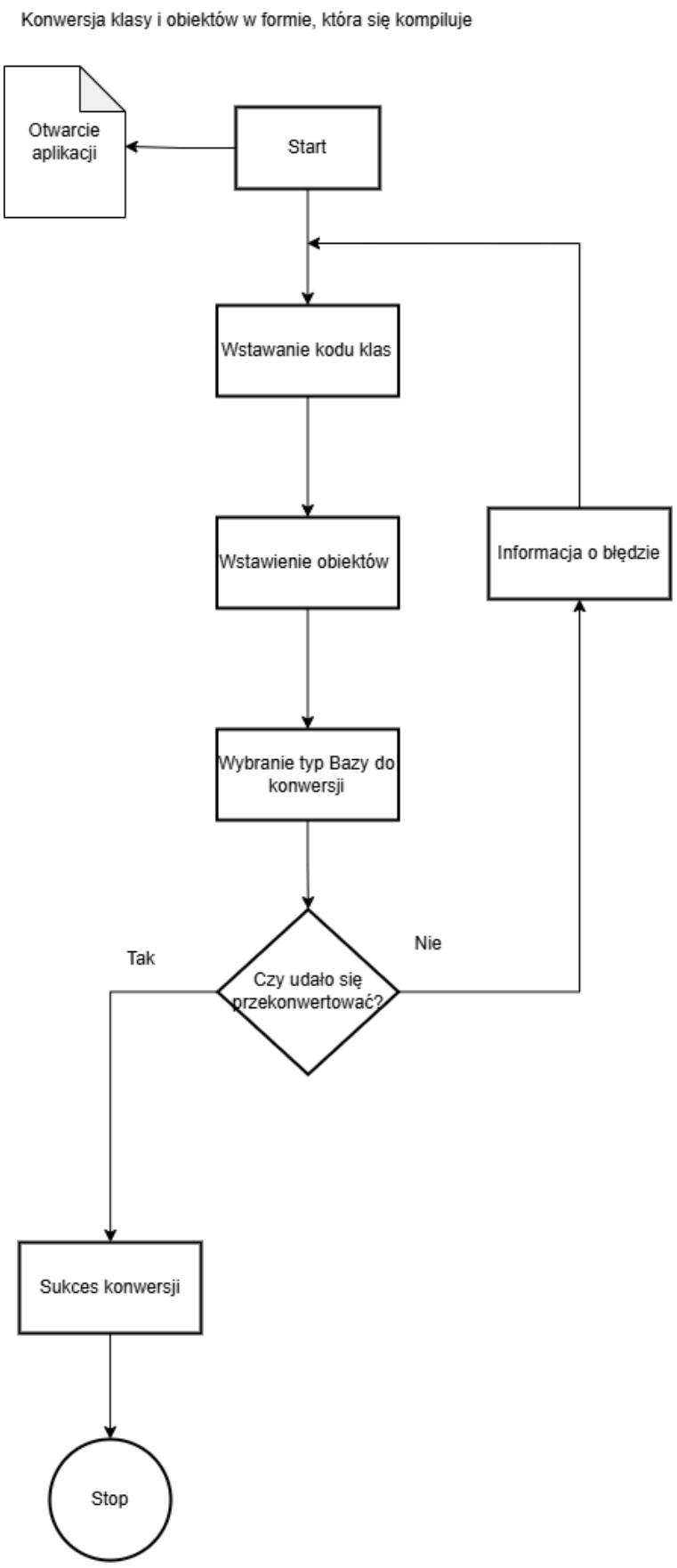


Diagram 2. Konwersja obiektów w QtTextEditior

Diagram 3. Utworzenie pełnej klasy z jednym polem w QtClassGenerator

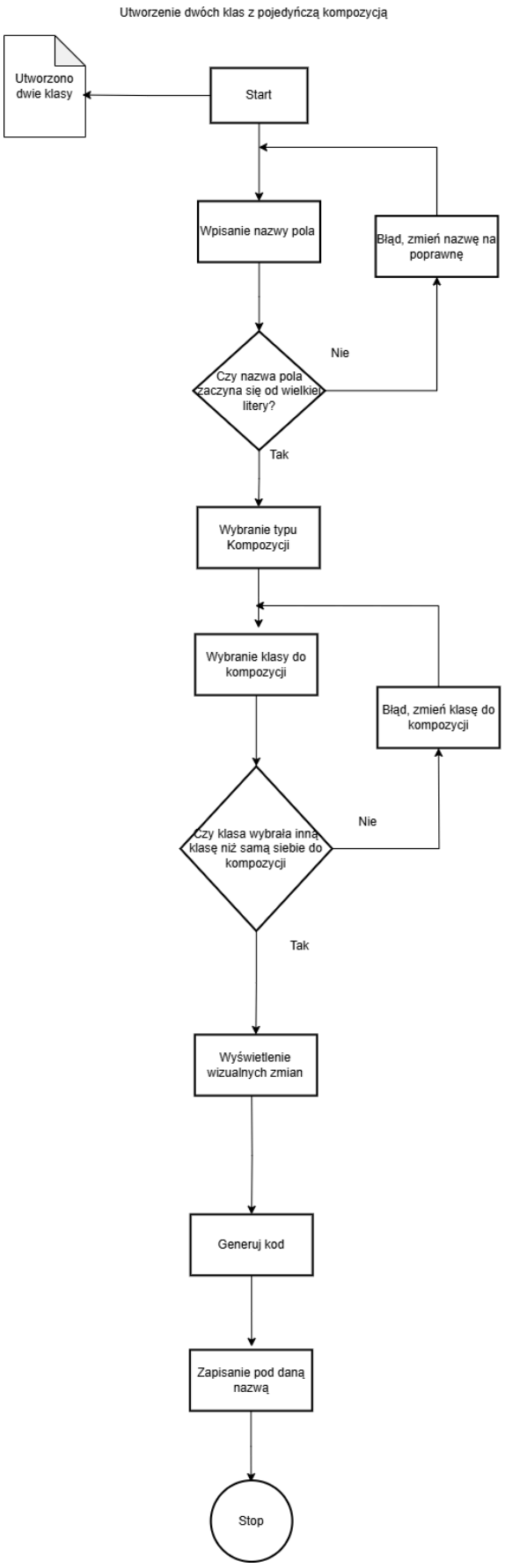


Diagram 4. Utworzenie dwóch klas z kompozycją w QtClassGenerator

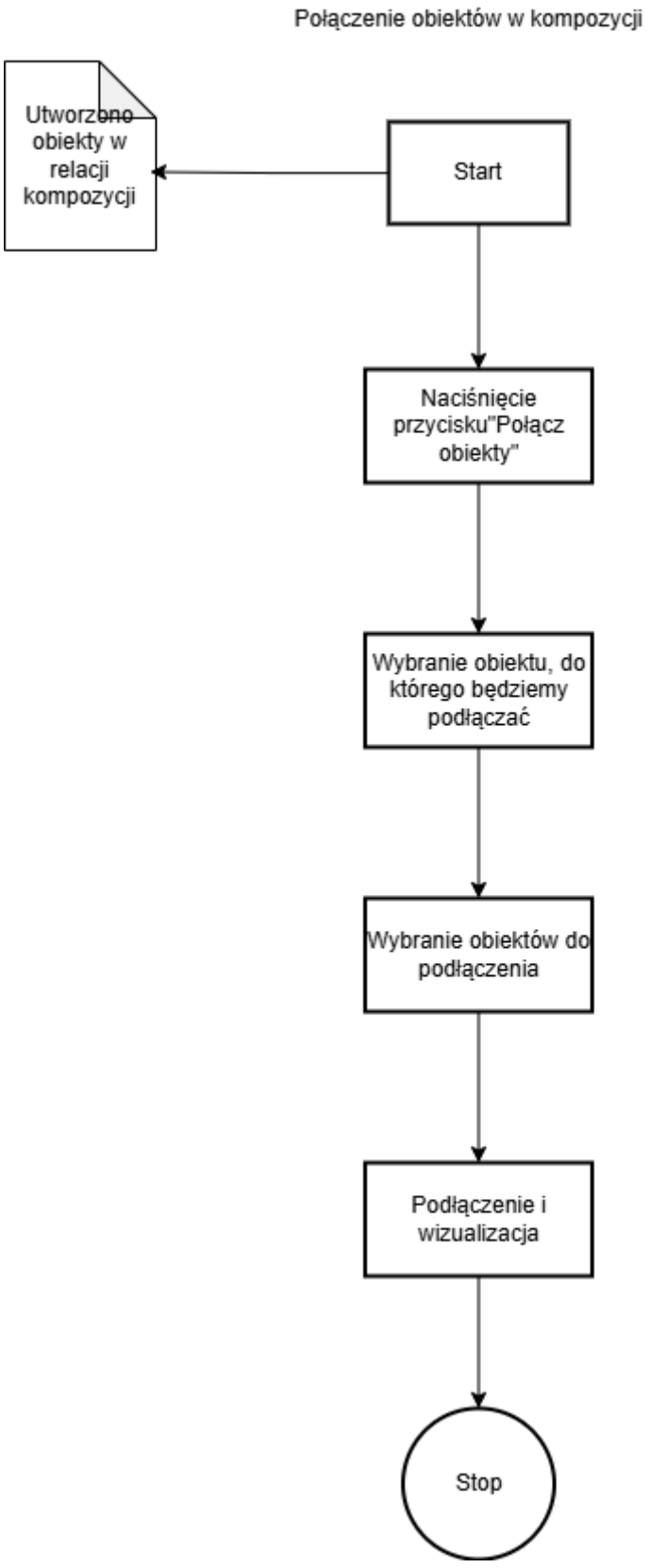


Diagram 5. Połączenie obiektów w kompozycji w QtGeneratorObject

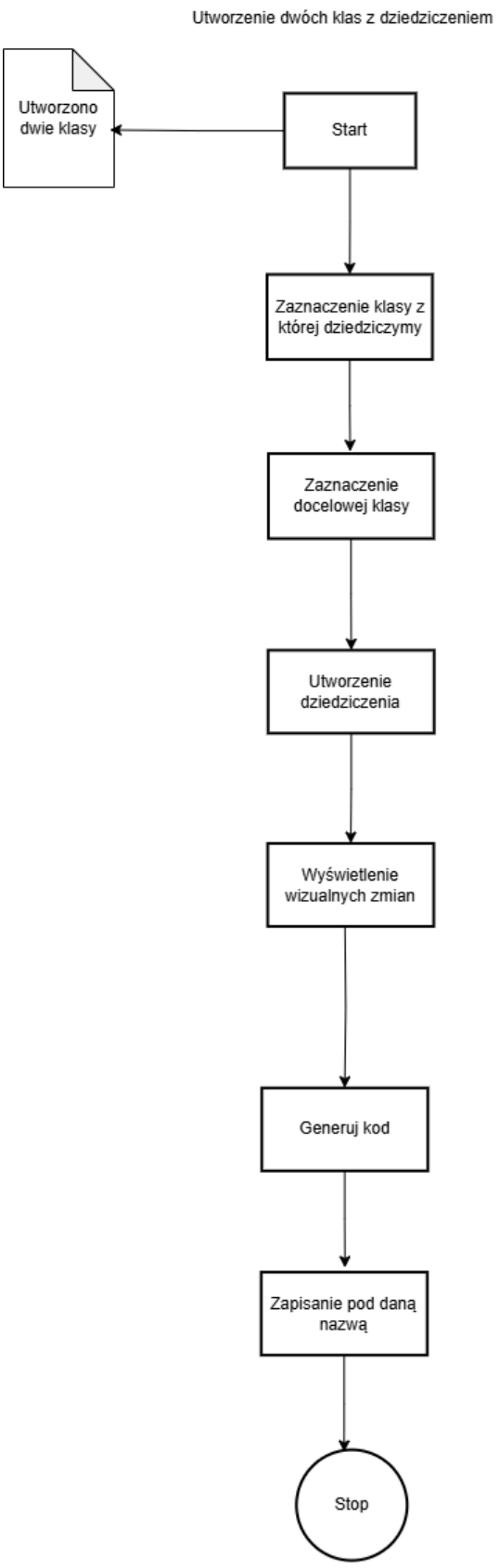


Diagram 6. Utworzenie dwóch klas z dziedziczeniem w QtGeneratorObject

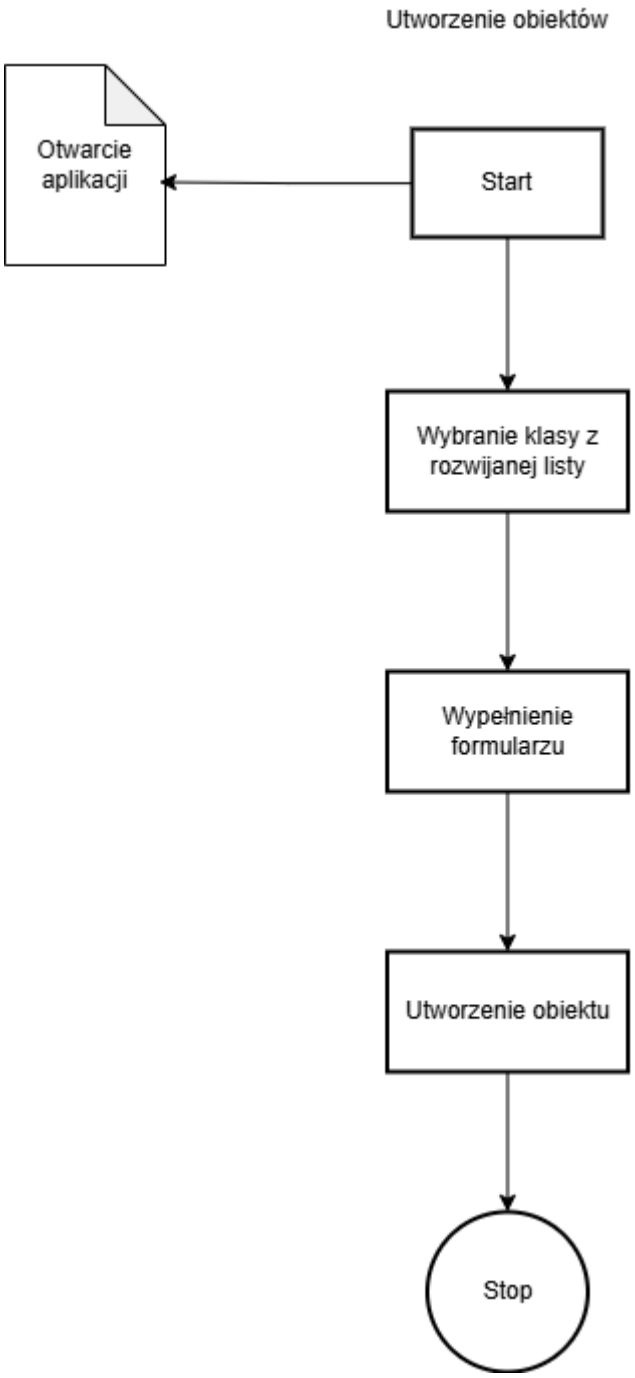
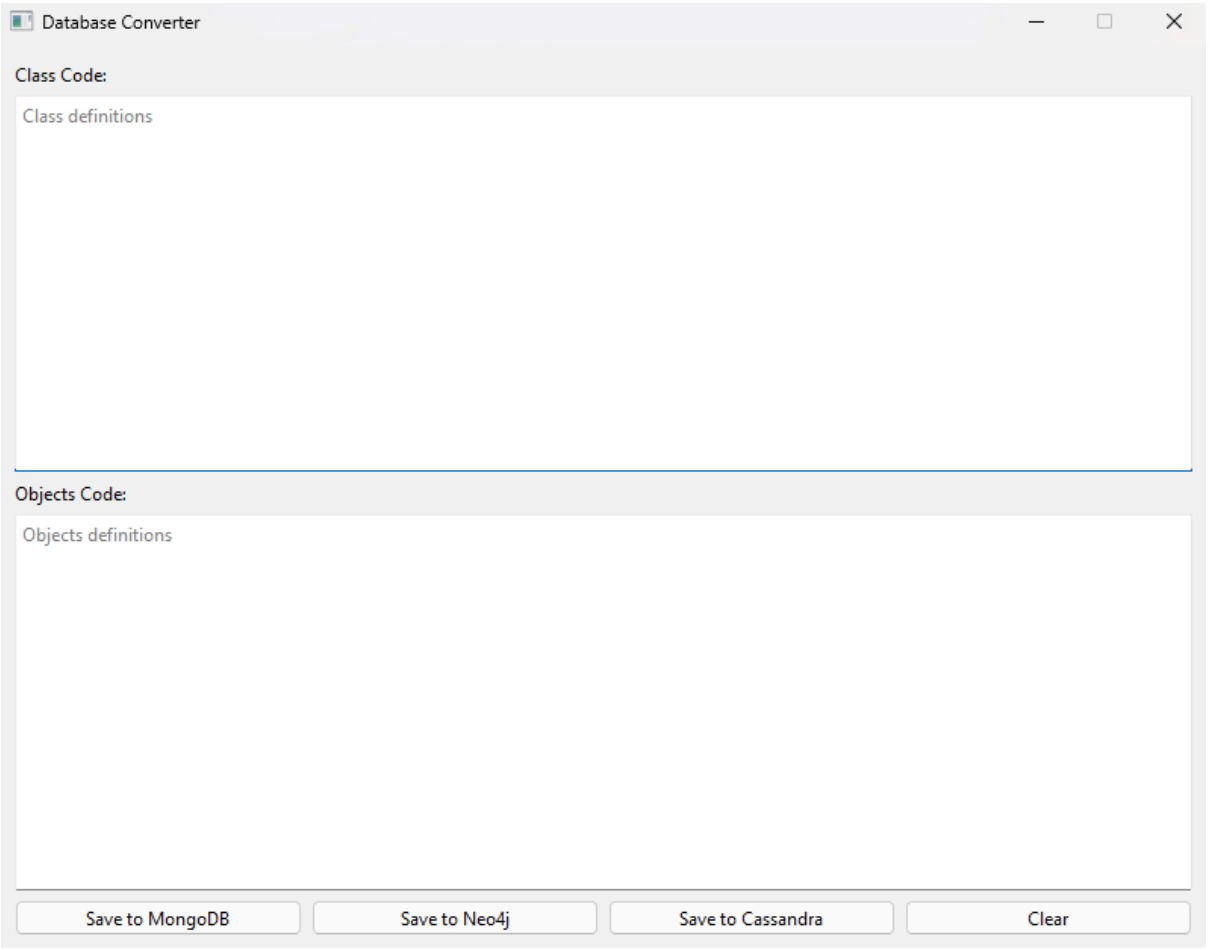


Diagram 7. Utworzenie obiektów w QtGeneratorObject

5. Opis programów

5.1. QtTextEditior

Program QtTextEditior służy do przekonwertowania gotowego kodu i obiektów na obiekty w poszczególnych typach baz danych. Nasz program obsługuje Neo4j, Cassandra oraz MongoDB. Na Rysunku 1. zaprezentowano wygląd tej aplikacji.



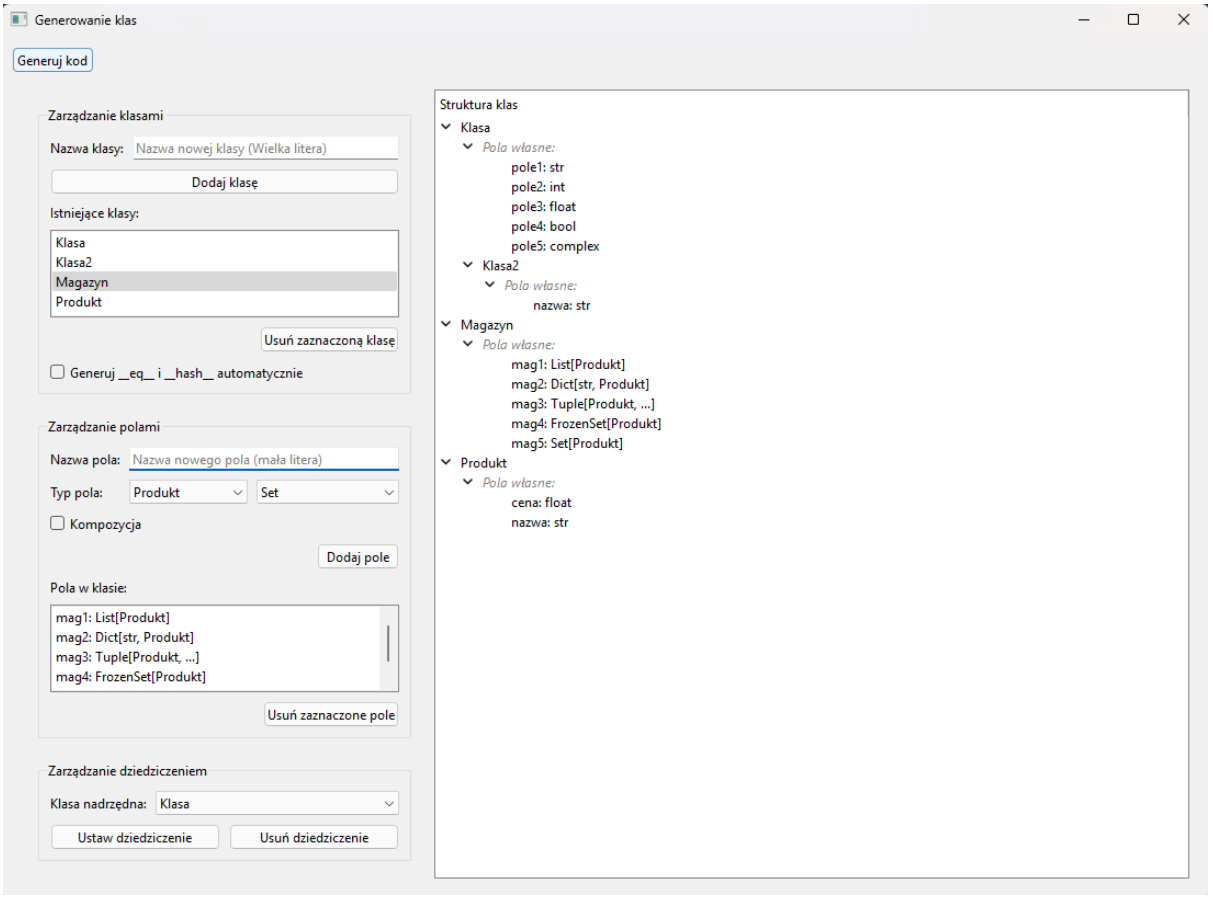
Rysunek 1. Interfejs QtTextEditior

Instrukcja obsługi:

1. Dodanie w polu “Class Code” kodu klass,
2. Dodanie w polu Objects Code: kodu obiektów,
3. Naciśnięcie “Save …” co wywoła konwersje na dany typ.

5.2. QtGeneratorClass

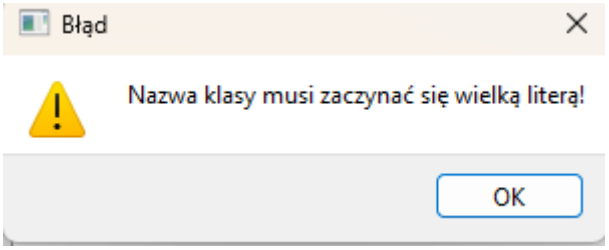
Program QtGeneratorClass służy do stworzenia kodu klasy za pomocą graficznego gui. Obsługuje on dziedziczenie, kompozycje oraz prawie wszystkie typy danych w python. Na Rysunku 2. widać budowę tego programu oraz wszystkie opcję.



Rysunek 2. Interfejs QtGeneratorClass

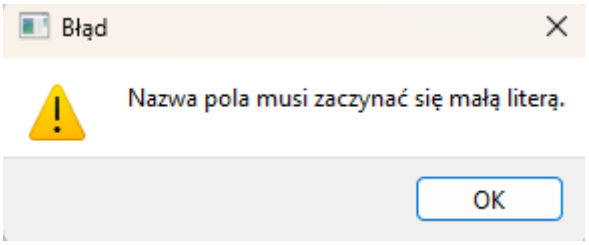
Instrukcja obsługi:

1. Na początku wstawiamy nazwę klasy koniecznie z wielkiej litery, w przeciwnym przypadku otrzymamy błąd taki jak na Rysunku 3.



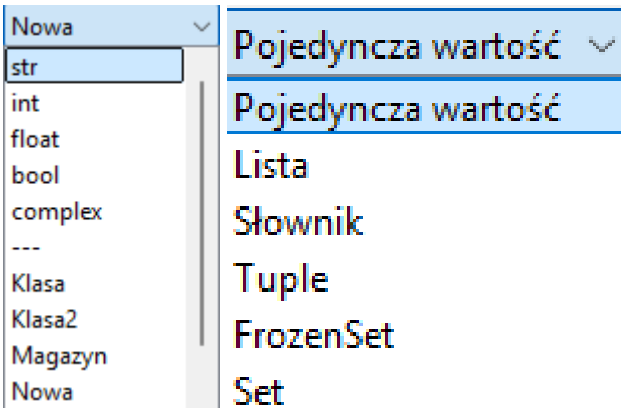
Rysunek 3. Błąd wielka litera klasy

2. W kolejnym kroku możemy dodać pola do klasy i tutaj też musimy pamiętać o konwencji małej litery, bo inaczej otrzymamy błąd jak na Rysunku 4.



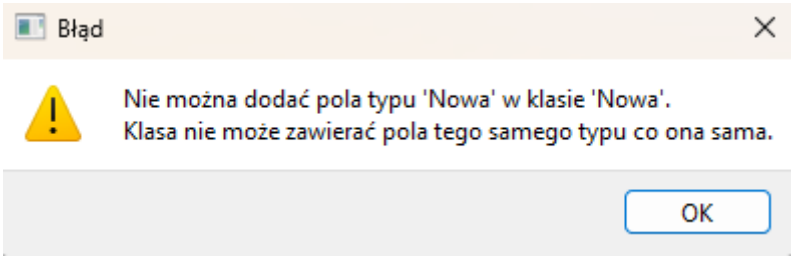
Rysunek 4. Błąd mała litera pola

3. Przy tworzeniu pola możemy wybrać jaki chcemy typ danych przedstawiony na Rysunku 5.



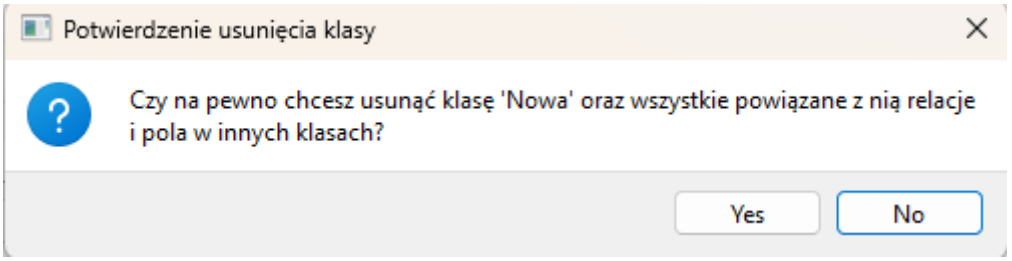
Rysunek 5. Dostępne typy danych

4. Na dole lewej listy z Rysunku 5 widzimy nazwy klas. W ten sposób możemy stworzyć kompozycje, ale musimy pamiętać o wybraniu do kompozycji klasy innej niż ta do której dodajemy. W przypadku gdy zapomnimy o tym otrzymujemy powiadomienie widoczne na Rysunku 6.



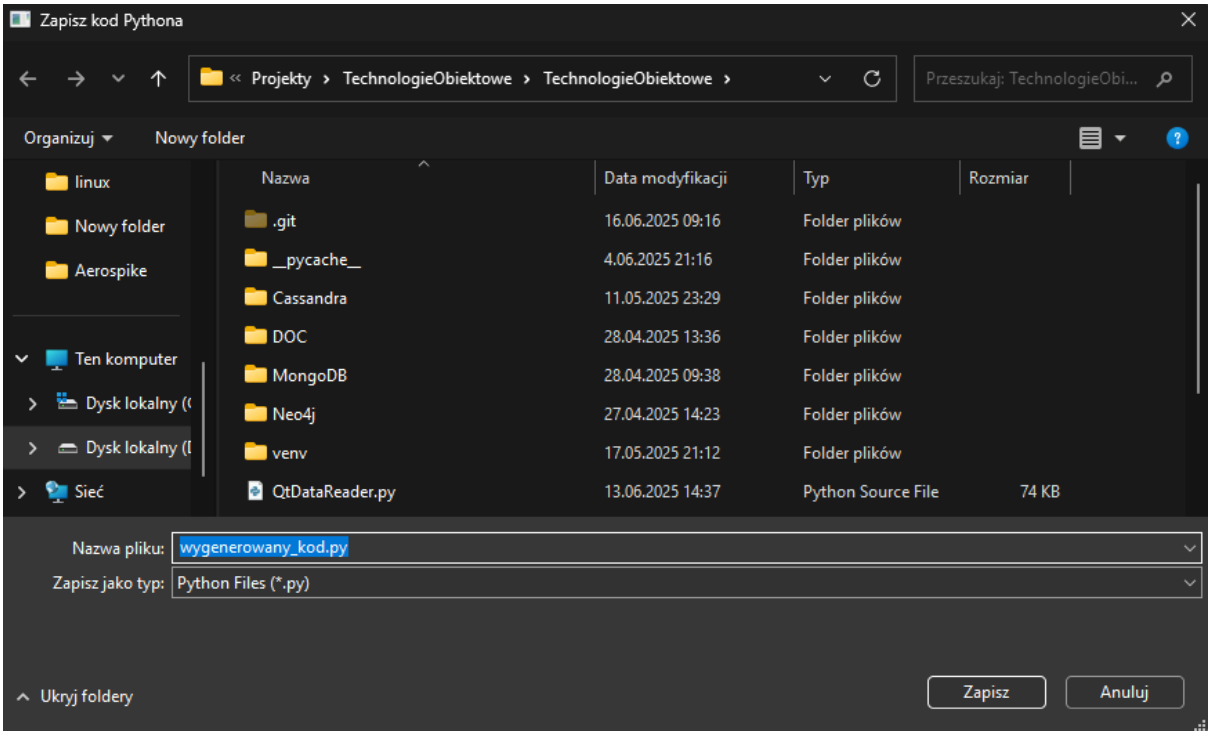
Rysunek 6. Błąd kompozycji

5. Pola oraz klasy możemy bez problemu usuwać.



Rysunek 7. Usunięcie klasy

6. Gdy skończymy tworzyć swoją klasę graficznie możemy ją wygenerować zapisując do pliku tak jak na Rysunku 8.



Rysunek 8. Zapisywanie klas

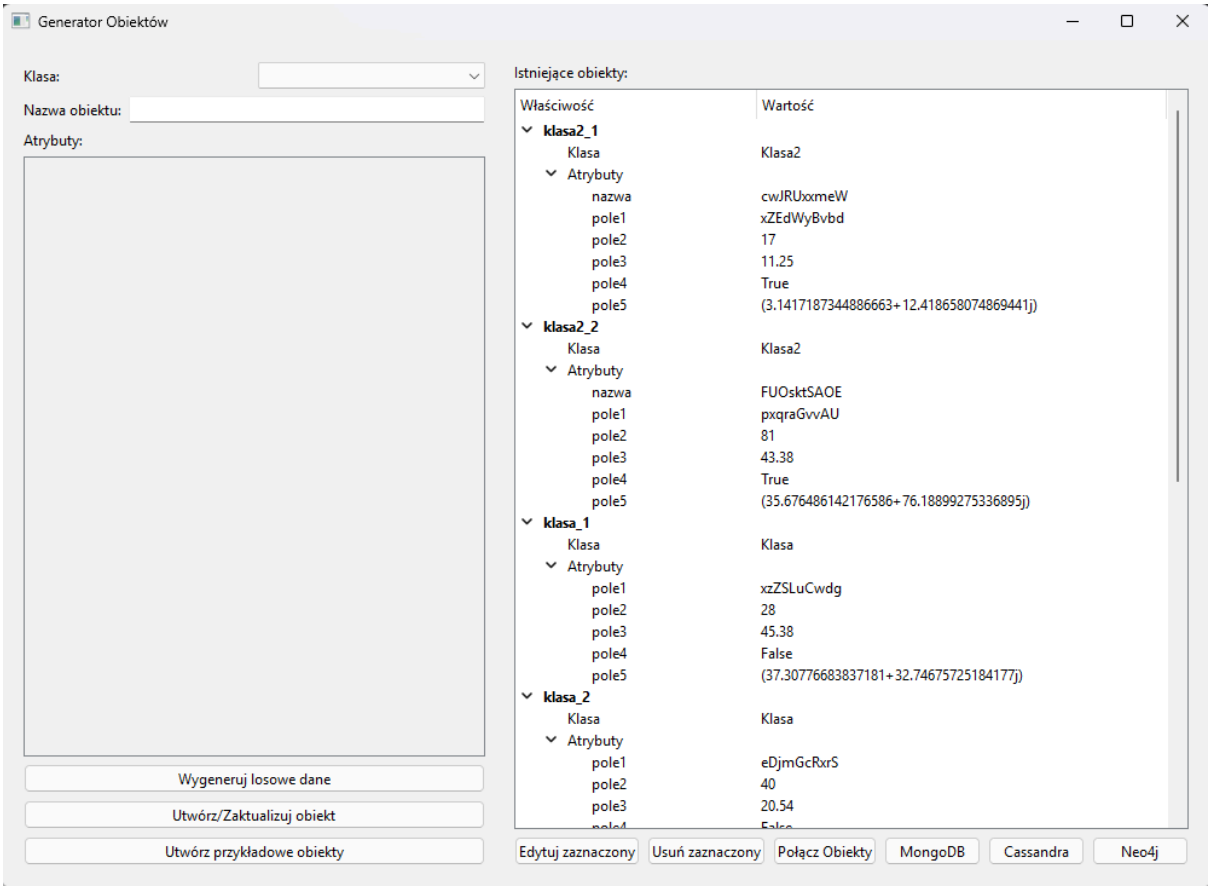
7. Na podstawie schematu z Rysunku 2. wygenerowany został poniższy kod.

```
from __future__ import annotations
from typing import Dict, List, Set, Tuple, FrozenSet


class Klasa:
    def __init__(self, pole1: str, pole2: int, pole3: float,
pole4: bool, pole5: complex):
        self.pole1: str = pole1
        self.pole2: int = pole2
        self.pole3: float = pole3
        self.pole4: bool = pole4
        self.pole5: complex = pole5
```

```
class Klasa2(Klasa):
    def __init__(self, pole1: str, pole2: int, pole3: float,
pole4: bool, pole5: complex, nazwa: str):
        super().__init__(pole1, pole2, pole3, pole4, pole5)
        self.nazwa: str = nazwa


class Magazyn:
    def __init__(self, mag1: List[Produkt] = None, mag2:
Dict[str, Produkt] = None, mag3: Tuple[Produkt, ...] = None,
mag4: FrozenSet[Produkt] = None, mag5: Set[Produkt] = None):
        self.mag1 = mag1 if mag1 is not None else []
        self.mag2 = mag2 if mag2 is not None else {}
        self.mag3 = mag3 if mag3 is not None else tuple()
        self.mag4 = mag4 if mag4 is not None else frozenset()
        self.mag5 = mag5 if mag5 is not None else set()


class Produkt:
    def __init__(self, cena: float, nazwa: str):
        self.cena: float = cena
        self.nazwa: str = nazwa


    def __eq__(self, other):
        if not isinstance(other, Produkt):
            return NotImplemented
        return (self.cena, self.nazwa,) == (other.cena,
other.nazwa,)


    def __hash__(self):
        return hash((self.cena, self.nazwa,))
```

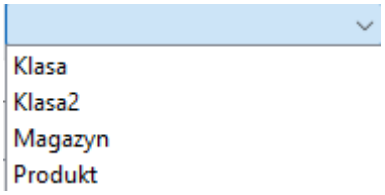
5.3. QtGeneratorObject

Program QtGeneratorObject służy do stworzenia obiektów w sposób graficzny z wcześniej wygenerowanego kodu klas. Oprócz obsługi obiektów klas ze standardowymi typami danych takich jak string czy int, służy także do łączenia w kompozycje obiektów. Interfejs programu jest widoczny na Rysunku 9.



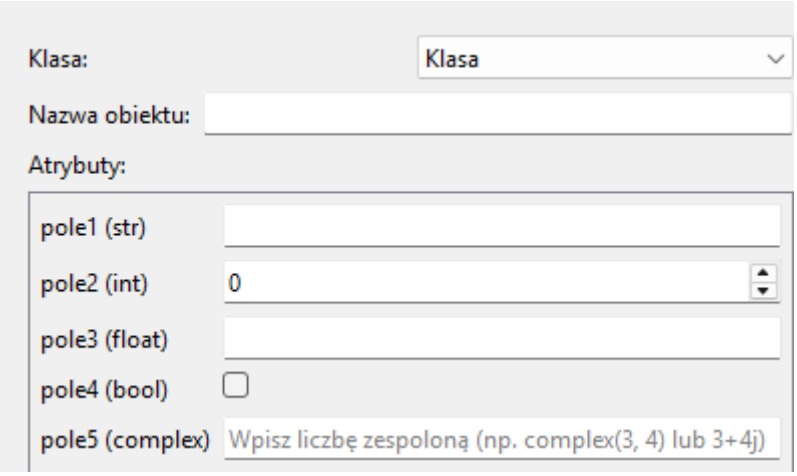
Rysunek 9. Interfejs QtGeneratorObject

1. W programie jesteśmy w stanie w kilka sposobów wygenerować obiekty. Pierwszym najprostszym jest naciśnięcie przycisku “Utwórz przykładowe obiekty.” Generuje to po 2 obiekty do każdej odczytanej przez program klasy. Efekt ten jest widoczny po prawej stronie na Rysunku 9. Lista klas odczytanych z pliku “wygenerowany_kod.py” jest widoczona na Rysunku 10.



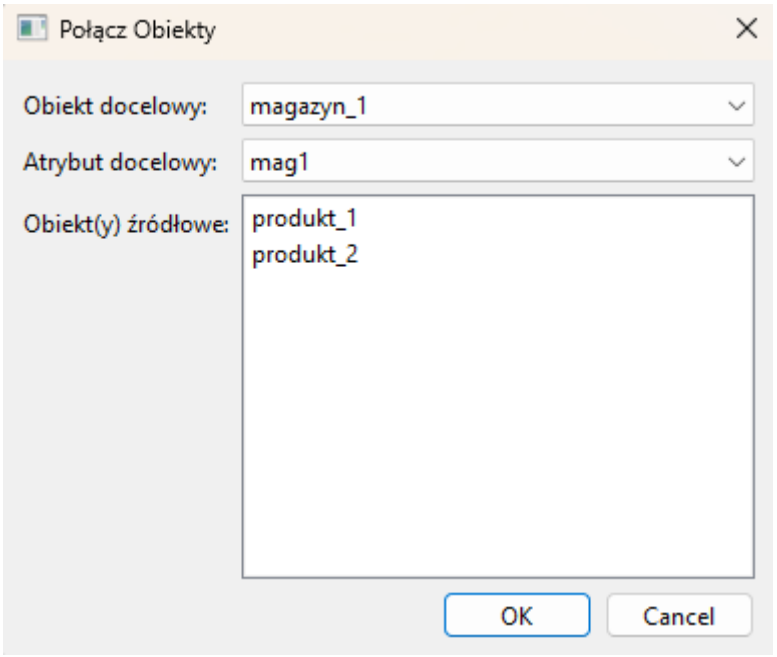
Rysunek 10. Przeczytane klasy.

2. Kolejnym sposobem utworzenia obiektu jest wybranie konkretnej klasy, dla której chcemy dodać obiekt, a następnie ręczne wpisanie danych do formularza jak na Rysunku 11.



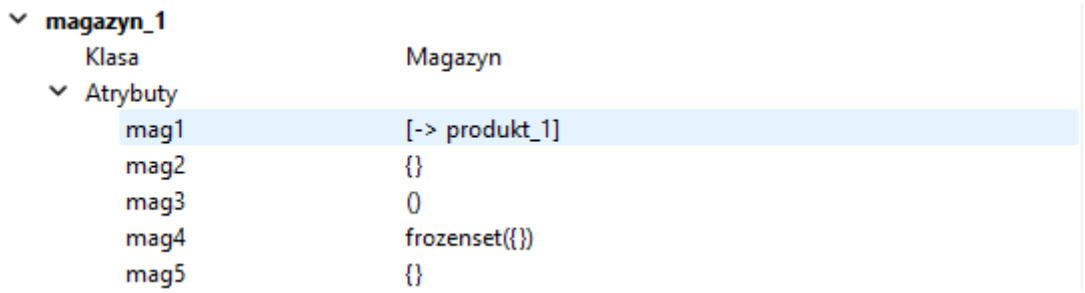
Rysunek 11. Formularz obiektu

3. Następnym sposobem jest przejście do formularza obiektu, ale zamiast ręcznie generować te dane możemy wygenerować losowe dla tego obiektu.
4. Gdy utworzyliśmy potrzebne nam obiekty możemy je połączyć. W przypadku wystąpienia kompozycji za pomocą opcji “Połączenia obiektów” wybieramy w zależności czy jest to kompozycja pojedyncza czy wielokrotna, potrzebne nam obiekty. Menu jest widoczne na Rysunku 12.



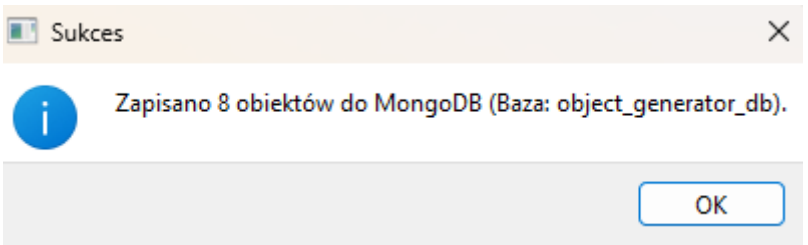
Rysunek 12. Łączenie obiektów

5. Po połączeniu obiektów wygląda to jak na Rysunku 13.



Rysunek 13. Połączone obiekty

6. Po stworzeniu obiektów, możemy użyć konwersji na MongoDB, Neo4j lub Cassandre. Jeśli wszystko zrobiliśmy dobrze otrzymamy komunikat jak na Rysunku 14.



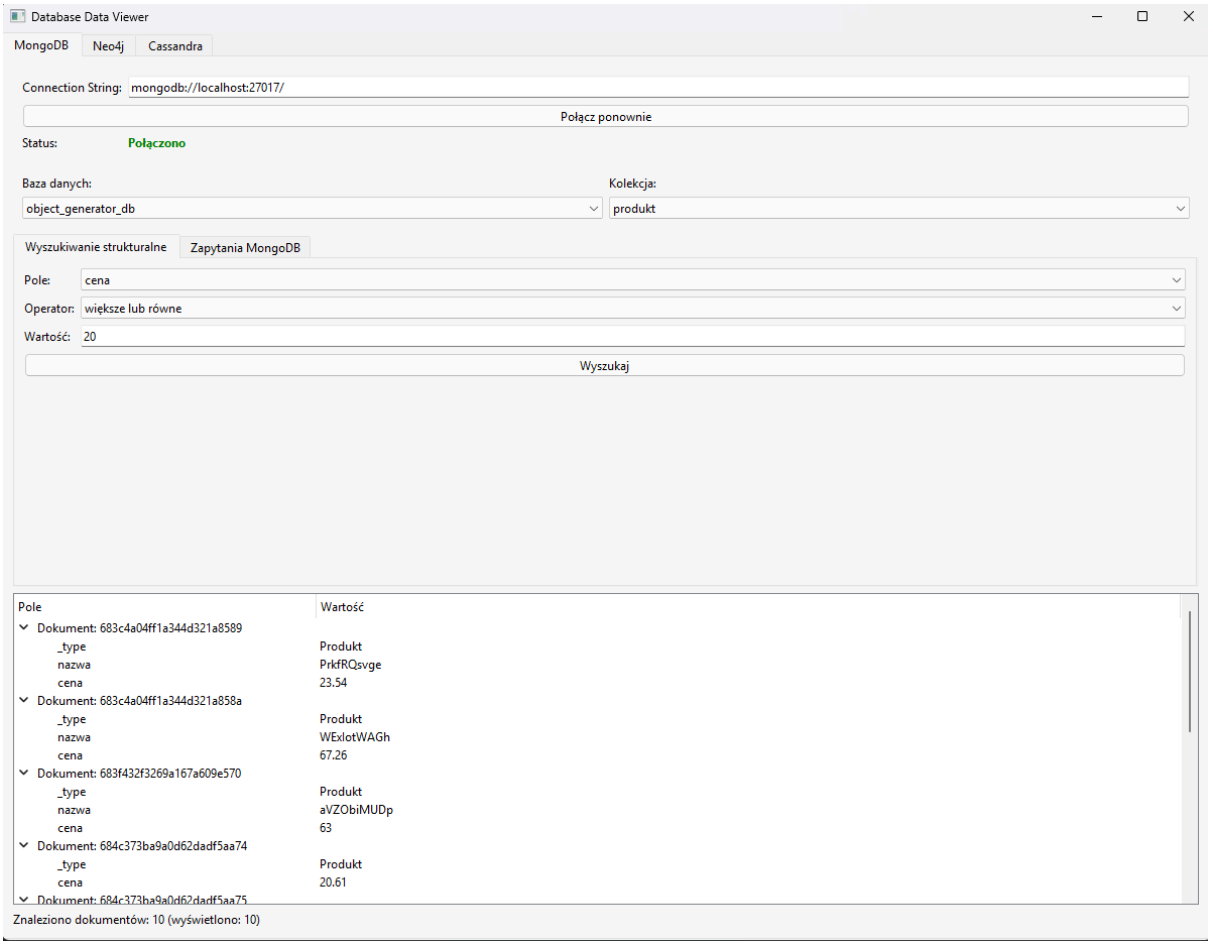
Rysunek 14. Połączone obiekty

7. Na Rysunku 15. widzimy przekonwertowany z aplikacji do MongoDB.

Rysunek 15. Dokument w MongoDB

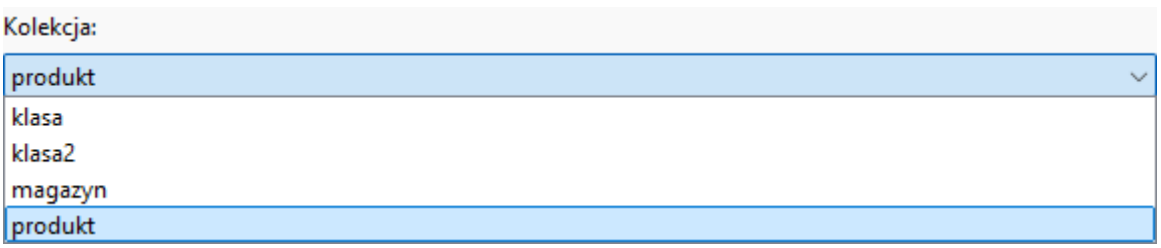
5.4. QtReaderData

Program QtReaderData służy do odczytywania danych z MongoDB, Cassandry oraz Neo4j. Pozwala on w miarę możliwości zaprezentować znalezione rekordy. Największe problemy istnieją z Neo4j, z powodu jej grafowego charakteru. Aplikacja umożliwia wybranie konkretnej bazy, kolekcji, keyspace, a następnie istnieje możliwość wybrania danego pola i wyszukanie obiektów według kryterium.



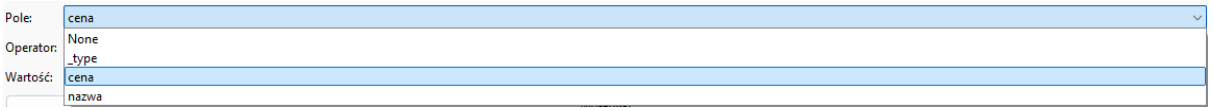
Rysunek 16. Interfejs QtReaderData

1. Po wybraniu jednej z trzech baz i wpisaniu adresu do podłączenia do serwera otrzymujemy komunikat “Połączono”.
2. W sekcji baza danych wybieramy jedną z dostępnych na serwerze.
3. W sekcji kolekcja wybieramy jedną z rozwijanej listy.
4. Kolejną opcją do wyboru jest rodzaj wyszukiwania obiektów, dokumentów, węzłów. Wybór przedstawiono na Rysunku 17.

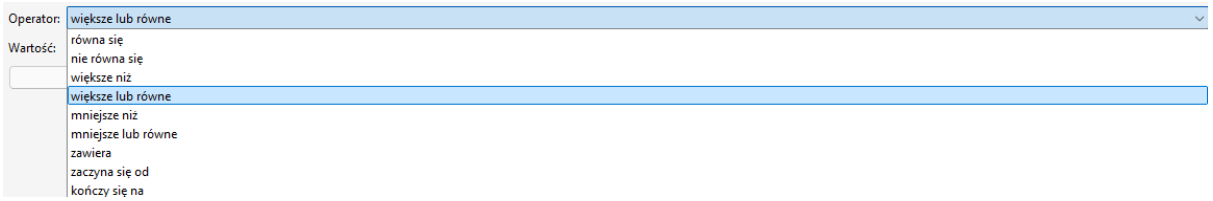


Rysunek 17. Wybór kolekcji

5. W wyszukiwaniu strukturalnym osoba może wybrać po jakim polu szuka, następnie operator i wartość. Natomiast w wyszukiwaniu zwykłym mamy możliwość wykonania zapytania w języku natywnym. Przykładowe wybory znajdują się na Rysunkach 18. oraz 19.

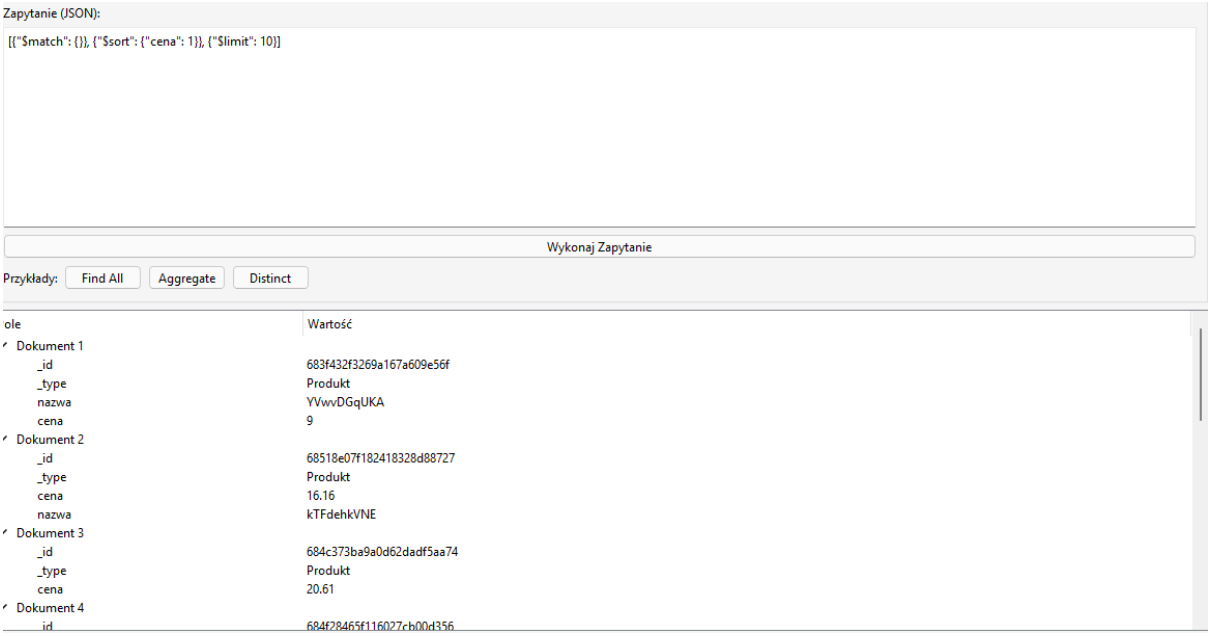


Rysunek 18. Wybór pola



Rysunek 19. Wybór operatora

6. Po wywołaniu wyszukiwania w oknie na samym dole aplikacji pokażą się znalezione dokumenty. Na Rysunku 20. znajduje się wynik zapytania w języku natywnym MongoDB.



Rysunek 20. Wynik wyszukiwania