Uniwersytet Gdański Wydział Matematyki, Fizyki i Informatyki Instytut Informatyki

Oliver Gruba, Maciej Nasiadka

23 grudnia 2025

Imię i Nazwisko (nr indeksu)	Oliver Gruba (292583) Maciej Nasiadka (292574)
Nazwa uczelni	Uniwersytet Gdański
Kierunek	Informatyka (profil praktyczny)
Prowadzący	dr inż. Stanisław Witkowski
Specjalność	-
Nazwa ćwiczenia	Statyczna struktura systemu informatycznego poprzez tworzenia diagramu
Numer sprawozdania	3
Data zajęć	06.11.2025
Data oddania	12.11.2025
Miejscę na ocenę	

Spis treści

1. Diagram klas - notacja i semantyka

Diagram klas w języku UML służy do opisu struktury systemu poprzez przedstawienie jego klas, atrybutów, operacji oraz powiązań między nimi. Każda klasa jest reprezentowana w postaci prostokąta podzielonego na trzy części:

- **Nazwa klasy** - identyfikator klasy, zwykle pisany wielką literą.
- **Atrybuty** - cechy lub dane opisujące stan obiektu danej klasy.
- **Operacje (metody)** - działania, które klasa może wykonywać.

Diagram klas pozwala na zrozumienie, jakie obiekty występują w systemie i w jaki sposób są one ze sobą powiązane.

2. Atrybuty diagramu klas

Atrybut to właściwość klasy, która opisuje dane przechowywane przez jej obiekty. Każdy atrybut ma określony typ danych, zakres dostępu (np. publiczny, prywatny, chroniony) oraz opcjonalnie wartość domyślną.

Przykład notacji atrybutu:

```
-nazwaAtrybutu: TypDanych = wartoscDomyslna
```

gdzie:

- **-** oznacza dostęp prywatny,
- + oznacza dostęp publiczny,
- # oznacza dostęp chroniony.

3. Podstawowe relacje stosowane w diagramie klas

Diagram klas przedstawia różne rodzaje relacji pomiędzy klasami. Do najczęściej stosowanych należą:

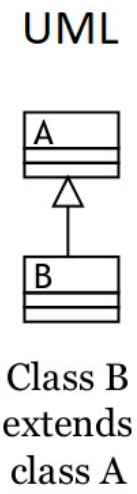
- **Asocjacja** - ogólne powiązanie między dwiema klasami.
- **Agregacja** - związek typu ”całość-część”, w którym obiekt części może istnieć niezależnie od całości.
- **Kompozycja** - silniejsza forma agregacji, w której część nie może istnieć bez całości.
- **Zależność (dependency)** - wskazuje, że jedna klasa korzysta z innej.

Każda relacja może być opatrzona dodatkowymi informacjami, takimi jak krotność (np. 1..*, 0..1) określająca liczbę wystąpień obiektów w danej relacji.

4. Dziedziczenie

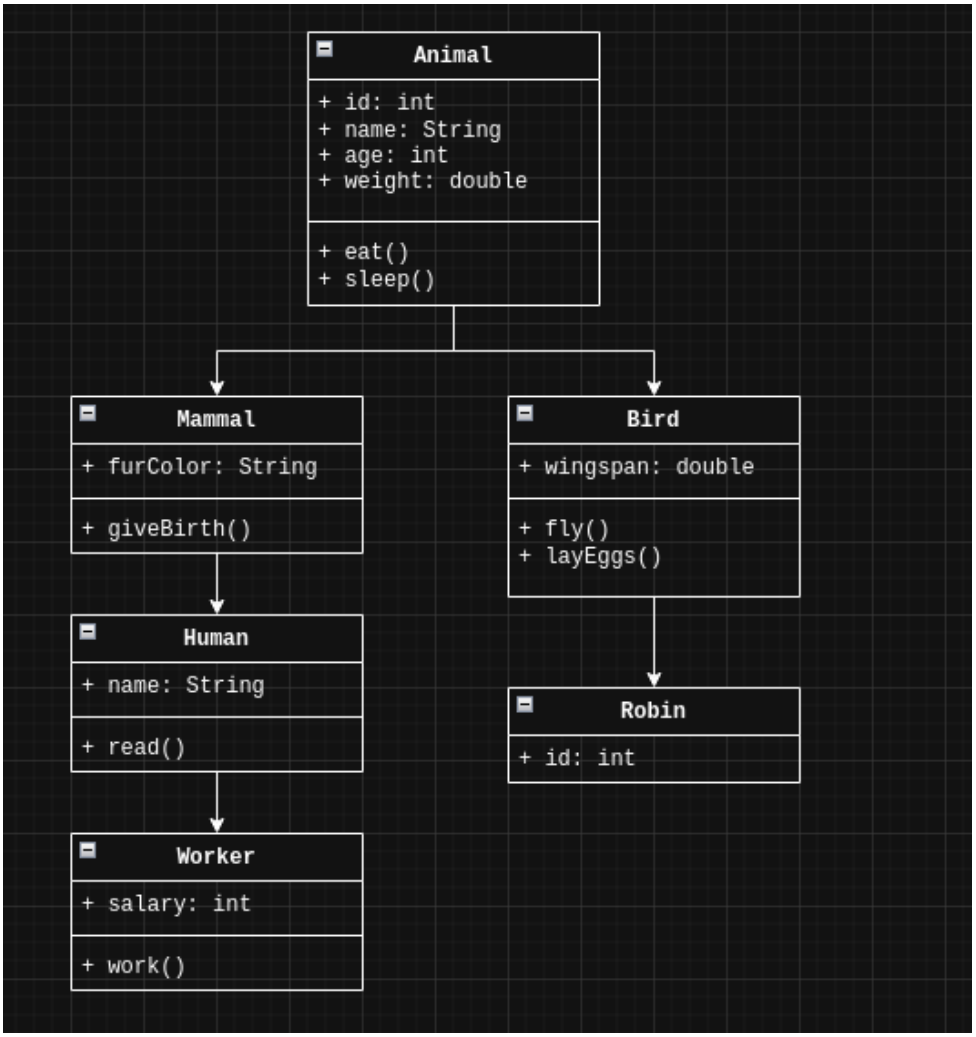
Dziedziczenie jest mechanizmem pozwalającym na tworzenie nowych klas w oparciu o już istniejące. Klasa pochodna (podklasa) dziedziczy cechy i zachowania klasy bazowej (nadklasy), co umożliwia ponowne wykorzystanie kodu i ułatwia jego utrzymanie.

Na diagramie UML dziedziczenie przedstawia się za pomocą strzałki z pustym trójkątem skierowanym 1 w stronę klasy bazowej.



Rysunek 1: Przykład wyglądu notacji dziedziczenia (klasa B dziedziczy cechy klasy A)

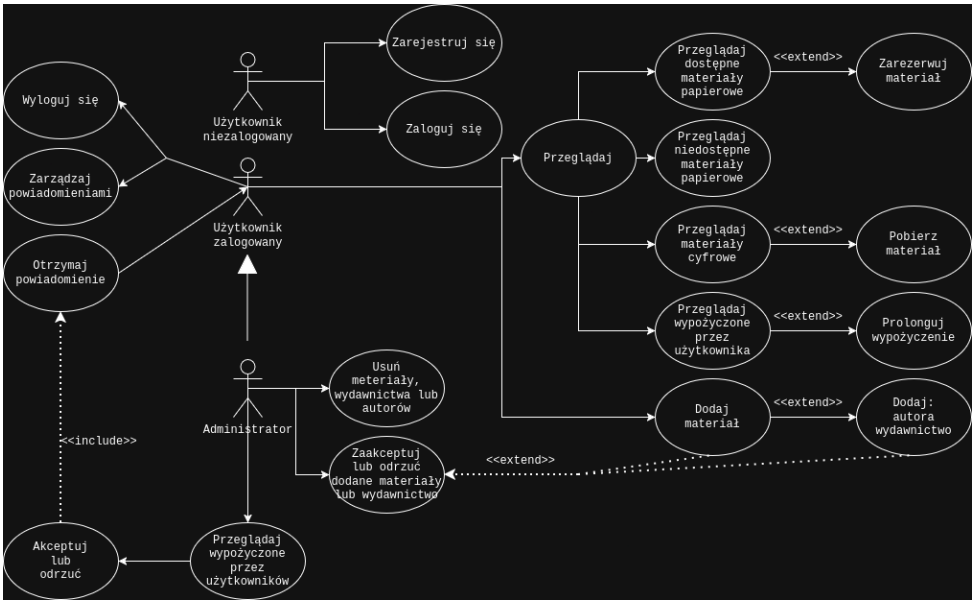
5. Przykład 1. Zwierzęta



Rysunek 2: Hierarchia dziedziczenia na przykładzie klasy Animal

Diagram przedstawia hierarchię dziedziczenia w systemie obiektowym. Klasa ‘Animal‘ jest klasą bazową, która zawiera wspólne cechy i metody dla wszystkich zwierząt, takie jak ‘eat()‘ i ‘sleep()‘. Klasy ‘Mammal‘ i ‘Bird‘ dziedziczą po klasie ‘Animal‘, a każda z nich posiada dodatkowe cechy i metody specyficzne dla swojego typu, np. ‘furColor‘ i ‘giveBirth()‘ w przypadku ssaków, oraz ‘wingspan‘ i metody ‘fly()‘ i ‘layEggs()‘ w przypadku ptaków. Klasy ‘Human‘ i ‘Robin‘ są bardziej wyspecjalizowanymi klasami, które mają wiele warstw dziediczenia‘.

6. Przykład 2. Czytelnia

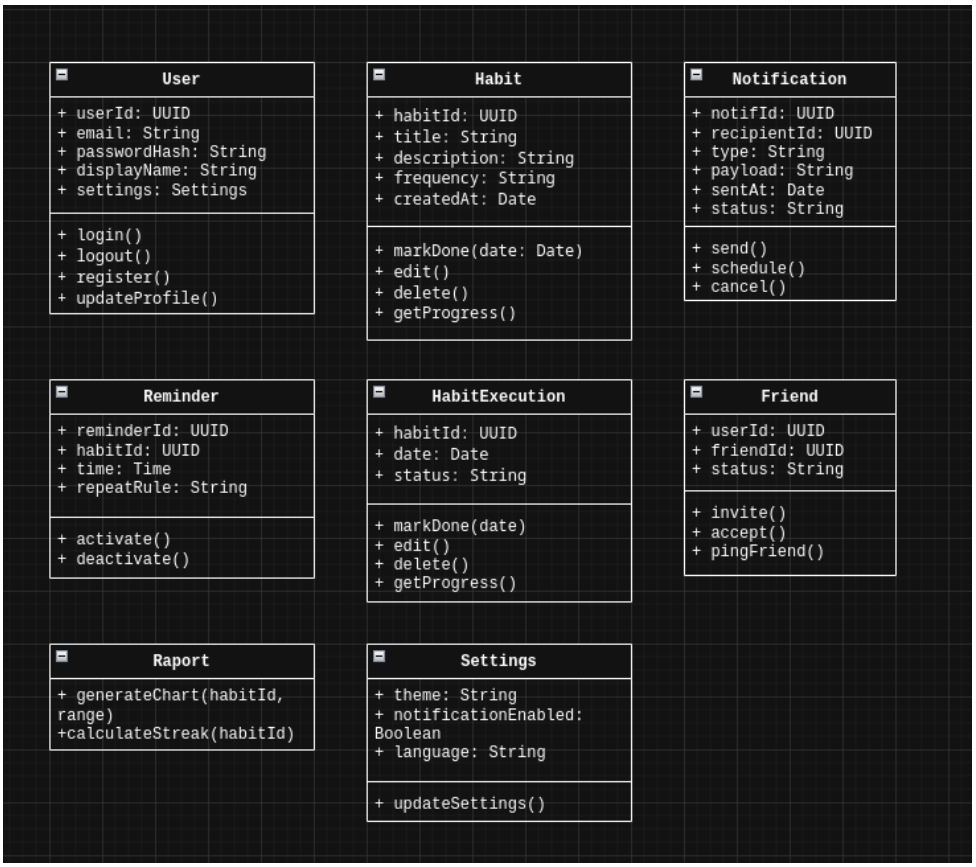


Rysunek 3: Przykład dziedziczenia ukazany w przykładzie systemu biblioteki.

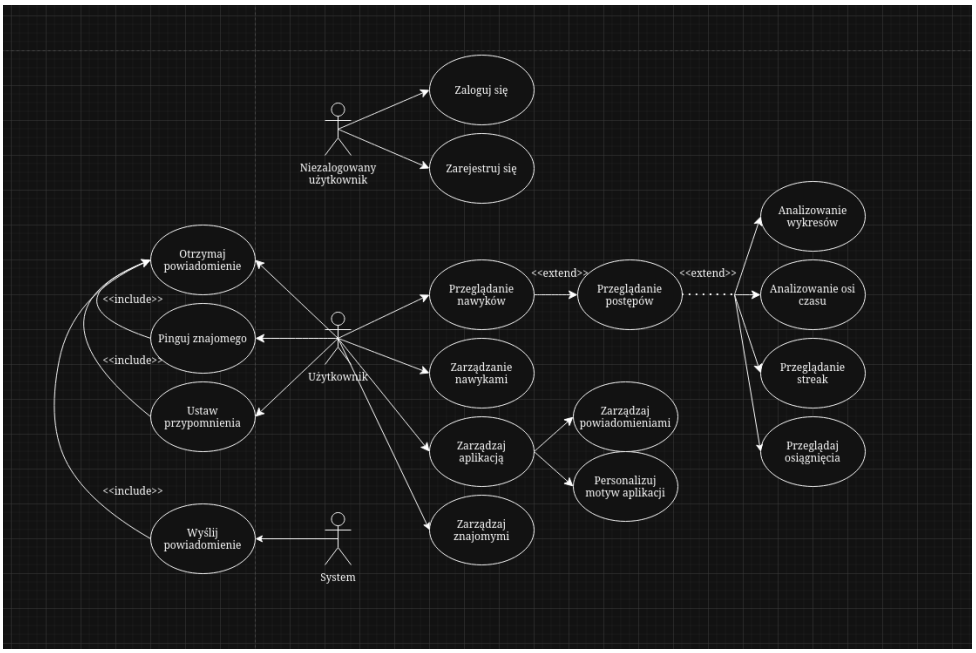
Przykład dziedziczenia ukazany w diagramie funkcjonalności czytelni. Użytkownik Administrator może zarówno korzystać z możliwości systemu tak jak zwykły użytkownik, ale jego klasa także rozbudowuje możliwości o funkcje administracyjne

7. Przykład 3. projekt zespołowy - KairoHabit

System odpowiedzialny za możliwość logowania się i ”trackowania” swoich nawyków jako użytkownik, mieszając elementy pożyczteczne dla zdrowia osby z gamifikacją poprzez elementy społeczne inspirowane podobnymi aplikacjami (pod względem socjalnym) jak międzyinnymi Duolingo.



Rysunek 4: Diagram klas aplikacji KairoHabit



Rysunek 5: Diagram przypadków użycia aplikacji KairoHabit

7.1. Aktorzy

1. Niezalogowany użytkownik

   - Reprezentuje osobę nieposiadającą sesji w systemie (gość).
   - Dostępne przypadki: Zaloguj się, Zarejestruj się.

   Rola: wejściowy punkt systemu - umożliwia przejście do konta.

2. Użytkownik

   - Zarejestrowany i zalogowany użytkownik aplikacji.
   - Główne przypadki: przeglądanie i zarządzanie nawykami, przeglądanie postępów, konfiguracja powiadomień, zarządzanie znajomymi, personalizacja aplikacji.

   Rola: podstawowy aktor realizujący większość funkcjonalności.

3. System

   - Reprezantacja strony serwera / mechanizmu powiadomień.
   - Realizuje automatyczne akcje: Wyślij powiadomienie

   Rola: wykonawca zadań bez interakcji bezpośrednio inicjowanych przez człowieka (automatyzacja).

7.2. Główne przypadki użycia

1. Zaloguj się

   - Cel: umożliwić niezalogowanemu użytkownikowi autoryzację i uzyskanie dostępu do funkcji zalogowanego użytkownika.
   - Prewarunek: konto użytkownika istnieje (użytkownik podaje poprawne dane) (login/hasło).
   - Scenariusz główny: użytkownik wypełnia formularz -> system weryfikuje dane -> sesja jest tworzona -> użytkownik zostaje przekierowany do pulpitu.
   - Postcondition: użytkownik z uprawnieniami ”Użytkownik”.
   - Alternatywy/wyjątki: błędne hasło (komunikat o błędzie), konto zablokowane, wymuszona zmiana hasła.

2. Zarejestruj się

- Cel: utworzyć nowe konto użytkownika.
- Prewarunek: brak konta z tym samym identyfikatorem (email)
- Scenariusz główny: formularz rejestracji -> walidacja (email, hasło, captcha) -> zapis w bazie -> opcjonalne wysłanie maila aktywacyjnego.
- Postcondition: utworzone konto (użytkownik może się zalogować)
- Wyjątki: e-mail już istnieje, walidacja nie przeszła.

3. Przeglądanie nawyków

   - Cel: użytkownik może zobaczyć listę swoich nawyków, ich statusy i skrócone statystyki.
   - Prewarunek: użytkownik zalogowany.
   - Scenariusz: użytkownik przechodzi do ekranu "Nawyki" -> system odczytuje listę nawyków -> wyświetla karty/wiersze z informacją o ostatnim wykonaniu, streakach itp.
   - Postcondition: widok z danymi; użytkownik może podjąć akcję (edytuj/usuń/oznacz jako zrobione).

4. Zarządzanie nawykami

   - Cel: tworzenie/edycja/usuwanie nawyków.
   - Prewarunek: użytkownik zalogowany.
   - Scenariusz: użytkownik tworzy nowy nawyk (nazwa, częstotliwość, przypomnienia) -> system zapisuje obiekt Nawyk.
   - Postcondition: zaktualizowana lista nawyków - ewentualne powiadomienia ustawione.

5. Przeglądanie postępów

   - Cel: przeglądanie szczegółowych statystyk i historii wykonania nawyków.
   - Prewarunek: użytkownik zalogowany (istnieją dane wykonania)
   - Scenariusz: użytkownik otwiera ekran "Postępy" –> system agreguje dane (czas, liczba wykonania, streak) -> wyświetla opcjonalne analizy (wykresy, oś czasu, osiągnięcia).
   - Postcondition: użytkownik widzi wskaźniki i może przejść do szczegółowych analiz.

- Relacje diagramu: Przeglądanie postępów rozszerza («extend») Przeglądanie nawyków - oznacza, że postępy nie są konieczne przy każdorazowym przeglądaniu nawyków, ale są logicznie powiązane (rozwinięcie).

6. Analiza

   - Cel: różne sposoby analizy postępów.
   - Opis: te przypadki są przedstawione jako rozszerzenia do Przeglądania postępów. Użytkownik może żądać konkretnej analizy:
     - Analizowanie wykresów - wykresy liniowe/słupkowe trendów.
     - Analizowanie osi czasu - przegląd aktywności w osi chronologicznej.
     - Przeglądanie streak - analiza serii kolejnych dni spełnienia nawyku.
     - Przeglądaj osiągnięcia - lista badge'ów/osiągnięć zdobytych za regularność.
   - Prewarunek: istnieją dane historyczne.

7. ustaw przypomnienia

   - Ustaw przypomnienia - użytkownik konfiguruje powiadomienia dla nawyku (czas, powtarzalność).
   - Pinguj znajomego - użytkownik może wywołać akcję powiadamiającą znajomego (np. poprosić o motywację). Z diagramu widać, że Pinguj znajomego include -> Otrzymaj powiadomienie (tj. ping initjuje powiadomienie).
   - Otrzymaj powiadomienie - przypadek odebrania powiadomienia po stronie użytkownika (może być wynik działania Wyślij powiadomienie).
   - Wyślij powiadomienie - wykonywane przez aktora System (automatyczne lub poprzez akcję innego użytkownika). Na diagramie Ustaw przypomnienia oraz Pinguj znajomego include -> Wyślij powiadomienie / Otrzymaj powiadomienie (strzałki i etykiety «include»).

8. Zarządzaj aplikacją

   - Zarządzaj aplikacją - ustawienia konta/ustawienia globalne. Z diagramu widzimy powiązania (może prowadzić do zarządzania powiadomieniami i personalizacji).
   - Zarządzaj powiadomieniami - ustawienia typu push, e-mail, dźwięki.
   - Personalizuj motyw aplikacji - UI theme: ciemny/jasny itp.
   - Zarządzaj znajomymi - dodawanie/usuwanie znajomych, zaproszenia, lista kontaktów.

7.3. Relacje diagramu między przypadkami (semantyka diagramu)

- «include» (w diagramie widoczne między ”Ustaw przypomnienia”, ”Pinguj znajomego”, ”Otrzymaj powiadomienie” a ”Wyślij powiadomienie”) - oznacza wspólne, powtarzalne zachowanie. Użyj w opisie: ”Funkcja A korzysta zawsze z funkcji B (B jest wymagane).”
- «extend» (między ”Przeglądanie nawyków” a ”Przeglądanie postępów”, oraz między ”Przeglądanie postępów” a poszczególnymi analizami) - oznacza, że rozszerzenia są opcjonalne i uruchamiane w warunkach określonych przez podstawowy przypadek. Użyj: ”Przypadek B rozszerza A - B jest wykonywane tylko w określonych sytuacjach (np. użytkownik wybierze opcję analizy).”
- Strzałki kierunkowe do / od aktorów - wskazują, kto inicjuje przypadek (aktor -> przypadek) lub kto reaguje (system -> przypadek).

7.4. Mapowanie na statyczną strukturę

7.4.1 Klasy, atrybuty i metody

Aplikacja jest ciągle rozwijana, więc niektóre dane tutaj są tylko sugestią, lub mogą zostać zmienione wraz z oficjalnym wydaniem aplikacji.

1. Użytkownik
    - Atrybuty: userId: UUID, email: String, passwordHash: String, displayName: String, settings: Settings
    - Metody: login(), logout(), register(), updateProfile()
2. Nawyk
    - Atrybuty: habitId, title, description, frequency, createdAt
    - Metody: markDone(date), edit(), delete(), getProgress()
3. Wykonanie nawyku
    - Atrybuty: habitId, date, status
    - Metody: markDone(date), edit(), delete(), getProgress()
4. Powiadomienie
    - Atrybuty: notifId, recipientId, type, payload, sentAt, status
    - Metody: send(), schedule(), cancel()

5. Przypomnienie
    - Atrybuty: reminderId, habitId, time, repeatRule
    - Metody: activate(), deactivate() - mapuje ”Ustaw przypomnienia”.
6. Znajomy
    - Atrybuty: userId, friendId, status
    - Metody: invite(), accept(), pingFriend()
7. Raport
    - Atrybuty: brak
    - klasa usługowa: generateChart(habitId, range), calculateStreak(habitId)
8. Ustawienia aplikacji
    - Atrybuty: theme, notificationsEnabled, language
    - Metody: updateSettings()

7.4.2 Relacje

- User 1..* - 0..* Habit (użytkownik może mieć wiele nawyków) - asocjacja.
- Habit 1 - * HabitExecution (kompozycja: wykonania należą do nawyku).
- User 1..* - * User (znajomi) - związek wieloma do wielu przez Friendship.
- Habit 1 - * Reminder (agregacja/kompozycja w zależności od modelu: przypomnienie zazwyczaj nie istnieje poza nawykiem -> kompozycja).
- Notification powiązane z User (recipient) i ewentualnie Habit.

7.4.3 Podsumowanie Przykład 3. projekt zespołowy - KairoHabit

KairoHabit - system śledzenia nawyków (model funkcjonalny) Prezentowany diagram przypadków użycia ilustruje funkcjonalności aplikacji typu ”habit tracker” oraz relacje między trzema aktorami: Niezalogowany użytkownik, Użytkownik oraz System.

Aktor Niezalogowany użytkownik posiada dostęp do działań umożliwiających przejście do konta: ”Zaloguj się” i ”Zarejestruj się”. Po zalogowaniu aktorem staje się Użytkownik, który ma dostęp do głównych funkcji systemu: ”Przeglądanie nawyków”, ”Zarządzanie nawykami”, ”Przeglądanie postępów”, ”Zarządzaj znajomymi”, ”Zarządzaj aplikacją” (w

tym ”Zarządzaj powiadomieniami” i ”Personalizuj motyw aplikacji”).

Sekcja ”Powiadomienia obejmuje przypadki ”Ustaw przypomnienia”, ”Pinguj znajomego”, ”Otrzymaj powiadomienie” oraz mechanizm ”Wyślij powiadomienie”, który jest wykonywany po stronie Systemu. Relacje typu «include» wskazują, że niektóre czynności zawsze wywołują fragmenty zachowania (np. ”Pinguj znajomego” obejmuje wysłanie powiadomienia), natomiast relacje «extend» wskazują opcjonalne rozszerzenia funkcjonalności (np. ”Przeglądanie postępów” rozszerza ”Przeglądanie nawyków”; do postępów dołączone są analizy szczegółowe: wykresy, oś czasu, streaki oraz osiągnięcia).

Z punktu widzenia projektowania systemu diagram ten pełni rolę wymagań funkcjonalnych, które można odwzorować w modelu klasowym. Przykładowe klasy wynikające z diagramu to: User, Habit, HabitExecution, Notification, Reminder, Friendship oraz Analytics. Relacje między tymi klasami (kompozycja Habit -> HabitExecution, agregacja/kompozycja Habit -> Reminder, asocjacja User <-> Friendship) pozwalają zaprojektować strukturę bazy danych oraz API serwera obsługującego powyższe przypadki użycia.

Diagram jasno oddziela interakcje inicjowane przez użytkownika od działań automatycznych wykonywanych przez system (np. wysyłka przypomnień), co ułatwia podział pracy w zespole: frontend implementuje interfejsy przypadków użycia, backend realizuje mechanizmy wysyłki powiadomień i agregacji statystyk, natomiast warstwa analityczna generuje wykresy i oblicza metryki.

8. Wiele diagramów klas w złożonych projektach - przykładowa struktura

W przypadku rozbudowanych systemów informatycznych pojedynczy diagram klas nie jest w stanie przejrzyście przedstawić wszystkich elementów i zależności występujących w projekcie. W takich sytuacjach stosuje się podział systemu na logiczne moduły lub pakiety, dla których tworzy się osobne diagramy klas. Każdy z diagramów opisuje wybrany fragment struktury systemu, natomiast pełny obraz powstaje dopiero po połączeniu ich w jedną, spójną całość.

8.1. Powody stosowania wielu diagramów

- **Złożoność projektu** – duża liczba klas, powiązań i relacji powoduje, że pojedynczy diagram staje się nieczytelny i trudny w utrzymaniu.

- **Podział funkcjonalny systemu** – poszczególne moduły (np. autoryzacja, zarządzanie użytkownikami, obsługa powiadomień) mogą być rozwijane niezależnie, dlatego logiczne jest tworzenie osobnych diagramów dla każdej z tych części.
- **Ułatwienie pracy zespołowej** – w projektach zespołowych różni członkowie zespołu mogą odpowiadać za inne obszary systemu, więc rozdzielenie diagramów pozwala im skupić się na swoim zakresie odpowiedzialności.
- **Łatwiejsze utrzymanie i rozszerzanie systemu** – zmiany w jednym module nie wymagają edycji całego diagramu, co poprawia skalowalność projektu.

8.2. Przykładowa struktura podziału diagramów

Dla aplikacji zespołowej przedstawionej w Przykładzie 3 (referencja 7) można wyróżnić kilka kluczowych modułów, z których każdy może mieć własny diagram klas:

1. **Moduł autoryzacji użytkowników** – zawiera klasy odpowiadające za logowanie, rejestrację, weryfikację danych oraz obsługę sesji użytkownika.
2. **Moduł zarządzania nawykami** – klasy reprezentujące nawyki, przypomnienia, historię aktywności oraz mechanizmy śledzenia postępów.
3. **Moduł powiadomień i komunikacji** – klasy odpowiedzialne za wysyłanie, odbieranie i przechowywanie powiadomień, w tym komunikację między użytkownikami (np. „ping znajomego").
4. **Moduł personalizacji interfejsu** – klasy związane z ustawieniami motywu aplikacji, preferencjami użytkownika i przechowywaniem danych konfiguracyjnych.
5. **Moduł analizy postępów** – klasy realizujące przetwarzanie danych, generowanie wykresów, analizę osi czasu oraz prezentację wyników użytkownika.

Każdy z tych modułów może zostać przedstawiony jako osobny diagram klas, zawierający:

- zestaw klas powiązanych funkcjonalnie,
- atrybuty i metody istotne tylko dla danej części systemu,
- relacje z innymi modułami poprzez wyraźnie zdefiniowane interfejsy (np. klasa `User` jako centralny punkt powiązania między modułami).

8.3. Przykładowy sposób organizacji

Praktycznym rozwiązaniem jest tworzenie hierarchicznej struktury pakietów, poniżej znajduje się przykład:

```
system
 ├── auth
 |    ├── User
 |    ├── LoginManager
 |    └── Session
 ├── habits
 |    ├── Habit
 |    ├── Reminder
 |    └── ProgressTracker
 ├── notifications
 |    ├── Notification
 |    ├── NotificationManager
 |    └── Ping
 └── analytics
      ├── Graph
      ├── TimeAxis
      └── Statistics
```

Taki podział ułatwia zarówno implementację, jak i wizualizację systemu na poziomie UML. Zachowanie spójności między poszczególnymi diagramami (np. wspólne klasy bazowe, interfejsy) pozwala utrzymać jednolitą architekturę projektu i umożliwia jego dalszy rozwój w sposób uporządkowany.

9. Wnioski

9.1. Zalety

Diagram klas pozwala na wczesne zrozumienie struktury systemu przed jego implementacją. Ułatwia komunikację między członkami zespołu oraz dokumentuje zależności pomiędzy komponentami. Dzięki graficznej reprezentacji relacji można łatwiej wykryć błędy projektowe i uniknąć powielania kodu.

9.2. Zastosowania

Diagramy klas są stosowane na etapie analizy i projektowania systemu informatycznego, zwłaszcza w metodach obiektowych. Wykorzystywane są w modelowaniu systemów biz-

nesowych, aplikacji webowych oraz systemów bazodanowych. Służą również jako dokumentacja techniczna ułatwiająca utrzymanie i rozwój oprogramowania.

9.3. Implementacje

Na podstawie diagramu klas można generować szkielet kodu w językach obiektowych (np. Java, Python, C++). Diagram może stanowić punkt wyjścia dla dalszego modelowania zachowań systemu poprzez diagramy sekwencji, przypadków użycia czy stanów.