ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ ПОЛІТЕХНІЧНИЙ УНІВЕРСИТЕТ
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ ПОЛІТЕХНІЧНИЙ УНІВЕРСИТЕТ
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Кваліфікаційна наукова праця на правах рукопису

**КОЛОНКО МАТТІАС ХАНС ГЕОРГ**

**УДК 004.652**

**ДИСЕРТАЦІЯ**

# МОВА КОНЦЕПТУАЛЬНОГО МОДЕЛЮВАННЯ ТА МЕТОДИ ДЛЯ РОЗРОБКИ БАЗ ДАНИХ ЗІ БАГАТОВАРІАНТНОЮ ПЕРСИСТЕНТНІСТЮ

Спеціальність 122 – Комп’ютерні науки
Галузь знань 12 – Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

М. Колонко

Наукові керівники:

**Арсірій Олена Олександрівна** доктор технічних наук, професор

**Мюлленбах Сабине** доктор філософії**,** професор

Одеса – 2020

# АНОТАЦІЯ

*Колонко М.* Мова концептуального моделювання та методи для розробки баз даних зі багатоваріантною персистентністю. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 122 - Комп'ютерні науки. – Одеський національний політехнічний університет МОНУ, Україна, Одеса, 2020.

У *вступі* обґрунтовано актуальність вирішення завдання розробки розширення мови концептуального моделювання та методів автоматизації розробки баз даних зі багатоваріантною персистентностю з метою зниження часу проектування баз даних

У *першому розділі* дисертаційної роботи проаналізовано існуючі мови концептуального моделювання, технології персистентності, методи для автоматизації процесу розробки баз зі багатоваріантною персистентністю та обґрунтовано мету та задачі дослідження.

Показано, що і сьогодні процес проектування БД згідно класичної роботи Ramez Elmasri and Shamkant B. Navathe вимагає від IT-розробника вирішення низки задач при проведенні трьохетапного моделювання даних на концептуальному, логічному та фізичному рівні (Рис. 1).

На етапі концептуального проектування IT-розробники БД спільно зі спеціалістами предметної області (ПрО) створюють концептуальну модель, яка перетворюється в логічну модель. Проведено аналіз існуючих мов концептуального моделювання, а саме ER, ORM/NIAM, IDEF1X, EXPRESS, EER, UML, Information Analysis та AGILA MOD та показано що всі вони частково не відповідають вимогам, які висувають IT-розробники БД: є складними і не зрозумілими нетехнічним спеціалістам із ПрО для узгодження функціональних вимог, які висуваються до даних; не мають достатньої кількості засобів для автоматизованого відображення концептуальної схеми в логічну схему даних, щоб забезпечити мінімальне ручне втручання; орієнтовані на проектування тільки РБД і тому не використовуються при розробці NoSQL

або Polyglot Persistence БД. Показано що саме недосконаліть існуючих мов концептуального моделювання призводить до підвищення витрат часу та трудомісткості проектування БД.

| Етапи процесу проектування БД | Технологічна підтримка |
|---|---|
| Conceptual Schema — Implementation independent level | ER, ORM/NIAM, IDEF1X, EXPRESS, EER, UML, Information Analysis, AGILA MOD |
| Derivation Algorithm; Logical Schema — DB System Type level | *Традиційні:* ієрархічні, мережеві. реляційні, об'єктно-орієнтовані об’єктно-реляційна *Новітні:* колоночні, документо-орієнтовані, графо-орієнтовані, ключ-значення, Polyglot Persistence |
| Derivation Algorithm; Physical Schema — DB System Vendor level | Microsoft Visio, Data Modeler (Oracle) |

Рисунок 1 – Взаємозв’язок етапів проектування БД та їх технологічної підтримки

З точки зору можливостей відображення концептуальної схеми в логічну схему даних на логічному рівні проведено аналіз існуючих традиційних та новітніх логічних моделей даних. Показано спадкоємність структур з одного боку ієрархічних, мережевих та реляційних логічних моделей, а з іншого реляційних та колоночних; ієрархічних, мережевих та графо-орієнтованих; об’єктно-орієнтованих та документно-орієнтованих. Обґрунтовано переваги логічних моделей даних «ключ-значення» для створення пошукових БД.

Показано, що використанням різних технологій зберігання даних – Polyglot Persistence в рамках одного застосування на основі декількох баз даних з різними логічними моделями на стороні сервера має наступні переваги: хороша масштабованість застосування, ефективне управління різнорідними даними, більш швидкий час відгуку і як наслідок більш висока продуктивність. Але підводячи підсумки щодо процесу проектування БД на концептуальному та логічному рівні можна зробити висновок поширення баз даних NoSQL та концепція багатоваріантної персистентності зробили процес розробки бази даних із їх використанням ще більш довготривалим та трудомістким.

У зв'язку з перерахованим вище сформульовано мету і завдання дослідження.

У *другому розділі* дисертаційної роботи удосконалено мову концептуального моделювання AGILA MOD+ для автоматизації процесу розробки окремих концептуальних моделей даних зі багатоваріантною персистентністю. AGILA MOD (***A***utomatic ***G***enerators for ***I***nformation representation, ***L***ogical DB structures and ***A***pplications – AGILA ***Mod***eling language – MOD) – це автоматичний генератор для інформаційного представлення, логічних структур БД. Загальна ідея полягає в тому, щоб створити концептуальну модель шляхом перекладу природної мови, як речень, які семантично не спрощуються, в графічну нотацію. Основні структурні елементи є об'єкти $O$ та асоціації $A$ між ними показані на рисунку 2, а загальна концептуальна модель задається як (1).


$$MOD =< O, A > \qquad (1)$$

Рисунок 2 – Об'єкти-учасники асоціації

Введено формальний опис викладеного: нехай є множина $O$ об'єктів певних типів, потужність множини – $|O|$:

$$O = \{o(t)_1, o(t)_2, \dots o(t)_i, \dots o(t)_{|O|}\}, \qquad (2)$$

Тип $t$ об'єкта $o_i$ задається як:

$$t = \begin{Bmatrix} \text{inheritance}\{o_i \cap o_j \vee o_i \cup o_j\}, abstract, \\ base\ \{\text{enumeration}, \text{set}\}, \\ system\ \{\text{string}, \text{integer}, \text{boolean}\} \end{Bmatrix} \qquad (3)$$

де $system$ – звичайний тип даних, наприклад «String», «Integer» або «Boolean»;. $base$ – базовий тип: перелік(enumeration) це фіксований список значень та набір (set) це впорядкована колекція без дублікатів; $inheritance$ – задає успадкування між супертипом та його підтипами; $abstract$ – складний тип об'єкта

Множину $A$ асоціацій певних типів задано як:

$$A = \{a_1, a_1, \dots a_i, \dots a_{|A|}\} \qquad (4)$$

де $|A|$ – потужність множини асоціацій

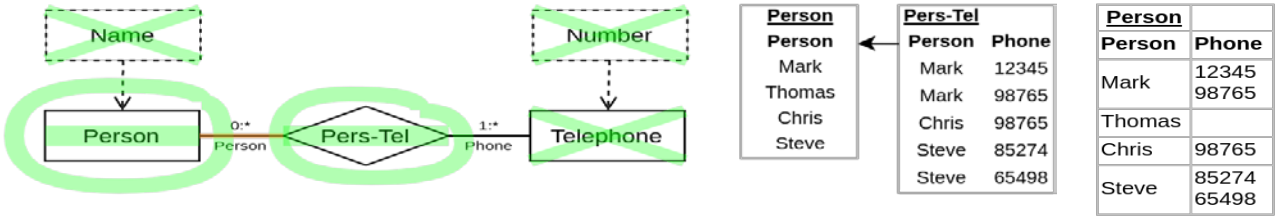
Тоді об’єкти $o_i$ та $o_j$ зв’язані за допомогою асоціації $a_i$,якщо:

$$\langle o_i, a_i, o_j \rangle = \{o_i \in O \wedge o_j \in O \wedge a_i \in A\} \quad (4)$$

Однією з найпотужніших, але простих конструкцій AGILA MOD є концепція *об'єктивізації асоціації*. Таким чином, створюється структурований об'єкт. Об'єктивізація асоціації дозволяє розробнику моделі повторно використовувати асоціацію як об’єкт для моделювання подальших речень, яке семантично не спрощується. Об'єктивізація асоціації має вигляд:

$$\langle (o_i, a_i, o_j) \vee ((o_i, a_i, o_j), a_j, o_k) \rangle = \{o_i, o_j, o_k \in O \wedge (o_i, a_i, o_j) \in O \wedge a_i, a_j \in A\} \quad (5)$$

На прикладі концептуальної моделі телефонної книги (рис. 2,a) побудованої за допомогою існуючих структурних елементів AGILA MOD розглянуті можливості фреймворку AGILA щодо автоматизованої деривації логічних моделей даних (рис. 3, b,c).




(a) Концептуальна модель із знаками деривації

(b) Реляційна модель

(c) Об'єктно-реляційна модель

Рисунок 3 – Приклади можливої та неможливої деривації концептуальної моделі

Показано, існуючий синтаксис мови AGILA MOD має недоліки щодо визначення та відображення автономних (Standalone) об'єктів, каскадного видалення та оновлення (Cascading), а також вкладених структури (Nested Structures). З врахуванням знайдених недоліків запропоновано удосконалити синтаксис мови AGILA MOD (структуру концептуальної моделі) за рахунок введення двох структурних елементів: незалежний об’єкт (*autonomous*) та «сильного» типу звязку об’єкта-учасника асоціації (Strong Association Participants)

Незалежний об’єкт виражає, що екземпляри описаного елемента ПрО існують незалежно від його відношень іншими елементами, позначається

прямокутником із вертикальними подвійними лініями (Рис.3,a). Його введення дозволяє скорегувати існуюче в фрейморку AGILA правило про необхідність скасування об’єктів-учасником асоціації з нижньою межею «1» та які не є частиною компонента узгодженості. Отримана реляційна модель (рис. 4,b) на відміну моделі (рис. 3, b) показує, що тепер незалежний об’єкт «Telephone» отримує своє власне відношення, на яке посилається відношення «Pers-Tel»

Сильний тип зв’язку об’єкта-учасника асоціації означає що асоціація сильніше пов’язана із підключеним об’єктом з межами участі в асоціації «x: 1» ніж з рештою. Тобто в асоціації «Pers-Tel» «Person» володіє «Telephone». У AGILA MOD + такий об’єкт-учасник підключається до асоціації за допомогою товстої лінії (рис 5,a). Для деривації із концептуальної моделі логічної моделі в фрейморку AGILA використовують наступний сценарій:

1. один тип об’єкта всередині компонента узгодженості позначається як ім’я відношення;

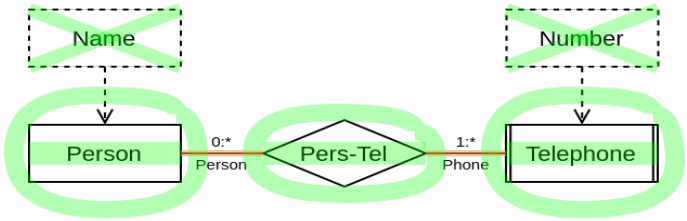
2. усі інші типи об’єктів усередині відношення стають ідентифікуючими атрибутами;

3. усі ролі типу асоціації, що лежать усередині компонента узгодженості, які вказують на тип об’єкта поза компонентом узгодженості, стають атрибутом відношення;

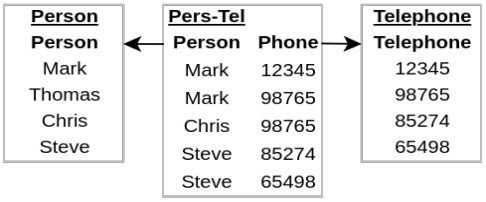
4. ролі цього типу, які є результатом позначеної сильної участі з верхньою межею, більшою за 1, інкапсулюються як вкладені атрибути.

При цьому пункт 4 зазначеного сценарію виконується тільки в разі деривації концептуальної моделі (рис. 5,a) в об’єктно-реляційну модель даних на логічному рівні (рис. 5,b).

Таким чином, виконане друге завдання дисертаційного дослідження та отримано перший пункт наукової новизни, а саме удосконалено структуру концептуальної моделі даних за рахунок розробки незалежного типу об'єктів та введення «сильного» типу зв’язку для об’єкта-учасника асоціації, що дало можливість вдосконалити правила деривації логічних моделей з урахуванням багатоваріантної персистентності даних.




(a) Концептуальна модель із знаками деривації, які враховують незалежний об’єкт



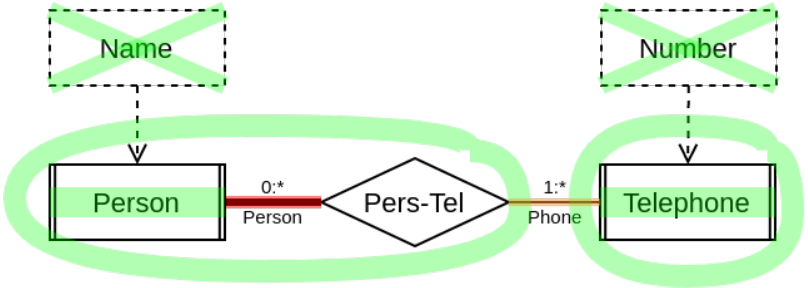
| Person |
|---|
| Person |
| Mark |
| Thomas |
| Chris |
| Steve |

| Pers-Tel | |
|---|---|
| Person | Phone |
| Mark | 12345 |
| Mark | 98765 |
| Chris | 98765 |
| Steve | 85274 |
| Steve | 65498 |

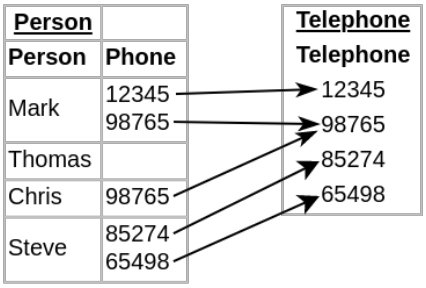
| Telephone |
|---|
| Telephone |
| 12345 |
| 98765 |
| 85274 |
| 65498 |

(b) Реляційна модель

Рисунок 4 – Результат деривації з використанням незалежного об’єкта.




(a) Концептуальна модель із знаками деривації, які враховують незалежні об’єкти та сильний тип звязку учасників асоціації



| Person | |
|---|---|
| Person | Phone |
| Mark | 12345<br>98765 |
| Thomas | |
| Chris | 98765 |
| Steve | 85274<br>65498 |

| Telephone |
|---|
| Telephone |
| 12345 |
| 98765 |
| 85274 |
| 65498 |

(b) Об'єктно-реляційна модель

Рисунок 5 – Результат деривації з використанням незалежного об’єкта та сильної участі в асоціації

У *третьому розділі* дисертаційної роботи створено методи автоматизації розробки логічних моделей баз даних на основі концептуальних субмоделей зі багатоваріантною персистентністю.

Адаптований до класичного (див. рис. 1) процес проектування БД представлено наступним чином (рис.6):

1. Створення комплексної концептуальної моделі ПрО відбувається за допомого засобів розширеної мови AGILA MOD+. Далі виконується – розділення комплексної концептуальної моделі на незалежні концептуальні субмоделі, які враховують особливості персистентності даних на логічному рівні.

2. Послідовне перетворення за допомогою алгоритму деривації отриманих концептуальних субмоделей у відповідні дато логічні моделі – «логічні схеми», вигляд яких залежить від типу персистентності даних, який підтримується засобами конкретної СУБД.

3. Послідовне перетворення відповідних дато логічних моделей у відповідні фізичні моделі (схеми) – внутрішні структури персистентності конкретної СУБД.

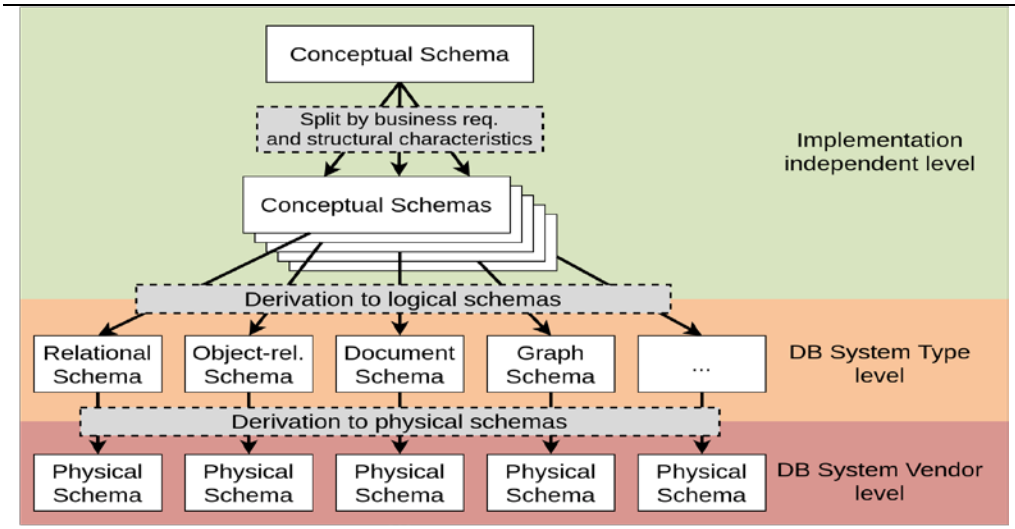
| Етапи адаптованого процесу проектування БД | Методична підтримка |
|---|---|
|  | **AGILA MOD +**<br><br>**Метод розділення** комплексної концептуальної моделі на субмоделі<br><br>**Метод створення** концептуальних субмоделей<br><br>**API** |

Рисунок 6 – Взаємозв’язок етапів адаптованого процесу проектування БД з урахуванням багатоваріантної персистентності та розробленої їх методичної підтримки

Для методичної підтримки дій першого етапу розроблено метод розділення комплексної концептуальної моделі ПрО на незалежні концептуальні субмоделі для подальшої їх деривації у відповідні логічні моделі. Метод включає такі етапи:

1. Створення копії існуючої комплексної концептуальної моделі.
2. Оточення замкненою, суцільною напівпрозорою оболонкою елементів комплексної концептуальної моделі об’єктів, асоціацій та ліній зв’язків між ними, із яких створюється субмодель.
3. Видалення усіх асоціацій та підключених до них ліній зв’язків, що лежать зовні оболонки оточення.
4. Видалення усіх об'єктів системних та базових типів, що лежать зовні оболонки оточення, і не мають з'єднання з об'єктами, які належать до внутрішньої частини оточення.
5. Залишення усіх об'єктів системних та базових типів, що лежать зовні оболонки оточення, та мають з'єднання з об'єктами що належать до внутрішньої частини оточення.
6. Заміна типу усіх залишених об'єктів, що лежать зовні оболонки оточення і мають з'єднання всередині на тип «посилання» (referencing) на субмодель, до якої належить оригінальний об'єкт.

Таким чином, в результаті розділення комплексної концептуальної моделі маємо три множині елементів (об’єктів, асоціацій та ліній зв’язків між ними): «inside», «outside» та «remainder». До множини «inside» належать елементи оточені оболонкою на другому кроці, до «outside» – елементи які було видалено на третьому та четвертому кроці. Множина «remainder» складається із об'єктів що мають тип «посилання», лежать зовні оболонки оточення, і мають з'єднання з об'єктами, що належать множини «inside».

Для побудови оболонки – оточення елементів комплексної концептуальної моделі запропоновано метод створення концептуальних субмоделей, які враховують особливості персистентності даних. Метод передбачає проведення аналізу структурних особливостей множині елементів комплексної концептуальної моделі, а саме об’єктів, асоціацій та ліній зв’язків між ними:

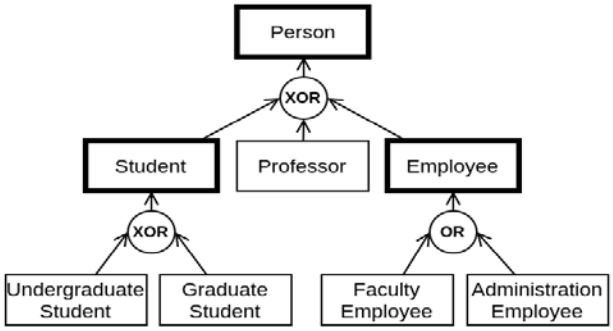
1. Наявність сильних зв’язків (див. рис. 5,а) між об’єктом та асоціацією, які запропоновані в AGILA MOD+, є показником віднесення цих елементів до документо-орієнтованих або об’єктно-реляційних структур персистентності

2. Наявність ієрархії типів об’єктів є показником віднесення цих елементів до документо-орієнтованих або об’єктно-реляційних структур персистентності. За допомого ієрархії типів об’єктів AGILA MOD підтримує успадкування об'єктів. На рисунку 7 a, b та c показані концептуальна модель і результати деривації успадкування.

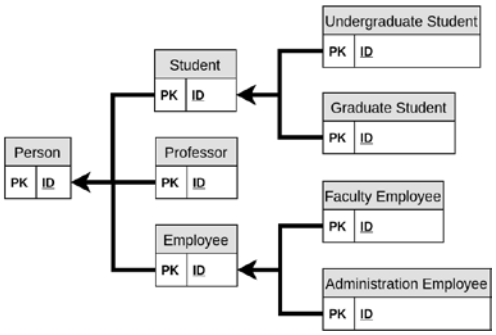
3. Для концептуальних моделей або їх частин, що містять значну кількість успадкувань, особливо коли успадкування є складним із використанням довільних комбінацій “OR”, “XOR” та “AND” доцільно використовувати документно-орієнтовані структури персистентності XML або JSON.

4. Наявність великої кількості асоціацій типу «Самоасоціація» (Self Association) коли межи зв’язків обьєктів-учасників дорівнюють "0: *" є

показником віднесення цих елементів до графо-орієнтованих структур персистентності




(a) концептуальна модель ієрархії типів об'єктів




(b) реляціна модель ієрахії типів об'єктів

```
<s:complexType name="studentType"
abstract="true">
   <s:complexContent>
      <s:extension          base="personType">
         <s:sequence>
            <s:element
   name="studyProgram" minOccurs="1"
                        type=" s:string"
            <s:element     name="semester"
                        type="s:positiveInteger"   maxOccurs="1" />
                  <s:element     name="enrolmentNo"

                        type="s:positiveInteger"
                     </s:sequence>
                  </ s:extension>
               </s:complexContent>
            </s:complexType>
```

(c) документо-орієнтована модель (XML)

Рисунок 7 – Модель та результати деривації успадкування

Апробація розроблених методів на прикладі проекту KampfrichterAnmelde-System (KRAS) – веб-додатку, який було створено для інформаційної підтримки Баварської Федерації дзюдо – дозволила розділити існуючу комплексну концептуальну модель ПрО на дві незалежні субмоделі для подальшої їх деривації в реляційну та LDAP моделі логічного рівня та створення відповідних БД. Дослідження показали, що було отримано скорочення в 2,3 рази часу на таке проектування двох БД в порівнянні з їх розробкою за класичною схемою.

Таким чином отримані другій та третій пункти наукової новизни а саме:

– вперше розроблено метод розділення комплексної концептуальної моделі даних на незалежні субмоделі що дозволило автоматизувати отримання відповідних логічних моделей даних включаючи реляційні та NoSQL структури та тим самим скоротити час проектування баз даних;

– вперше розроблено метод створення концептуальних субмоделей, які враховують особливості персистентності даних включаючи реляційні та NoSQL структури із комплексної концептуальної моделі даних, що дозволило обґрунтовано вибрати тип структур даних на логічному рівні і тим самим зменшити час розробки баз даних з урахуванням багатоваріантної персистентності.

Врахування удосконаленої структури концептуальної моделі даних та методів автоматизації розробки логічних моделей баз даних дозволило отримати четвертий пункт наукової новизни:

– удосконалено мову концептуального моделювання AGILA MOD+ за рахунок за рахунок удосконаленої структури концептуальної моделі даних та методів створення моделей даних концептуальному рівні, що дозволило враховувати багатоваріантну персистентність та скоротити час проектування баз даних інформаційних систем.

У *четвертому розділі* розроблено інтерфейс прикладного програмування (API) для забезпечення єдиного доступу до баз даних зі багатоваріантною персистентністю із бізнес-логіки

У *п'ятому розділі* проведено апробацію мови концептуального моделювання AGILA MOD+ та API при розробці БД в процесі виконання IT-проектів та показано суттєве зменьшення часу проектування, що підтверджено відповідними актами впровадження.

**Ключові слова:** Розробка баз даних, багатоваріантна персистентність, моделювання предметних областей (ПрО), мови концептуального моделювання, модель сутність-зв'язок, NoSQL бази даних, реляційні бази даних, система управління базами даних (СУБД)

## ABSTRACT

*Kolonko M.* Conceptual modeling language and methods for developing databases with polyglot persistence . - Qualifying scientific work on the manuscript.

Thesis for the PhD degree in specialty 122 «Computer science» (12 - Information Technology). - Odessa National Polytechnic University, Ukraine, Odessa, 2020.

The *introduction* verifies the topicality of solving the problem of developing an extension of the conceptual modelling language and methods for automating the development of databases with multivariate persistence in order to reduce the time of database design.

In the *first chapter* of the thesis paper, the existing languages of conceptual modelling, persistence technologies, methods for automating the process of developing databases with multivariate persistence are analyzed and the goal and objectives of the research are justified.

It is shown that today the process of designing a database according to the classic work of Ramez Elmasri and Shamkant B. Navathe requires IT-developer to solve a number of problems when carrying out a three-stage data modelling at the conceptual, logical and physical levels (Fig. 1).

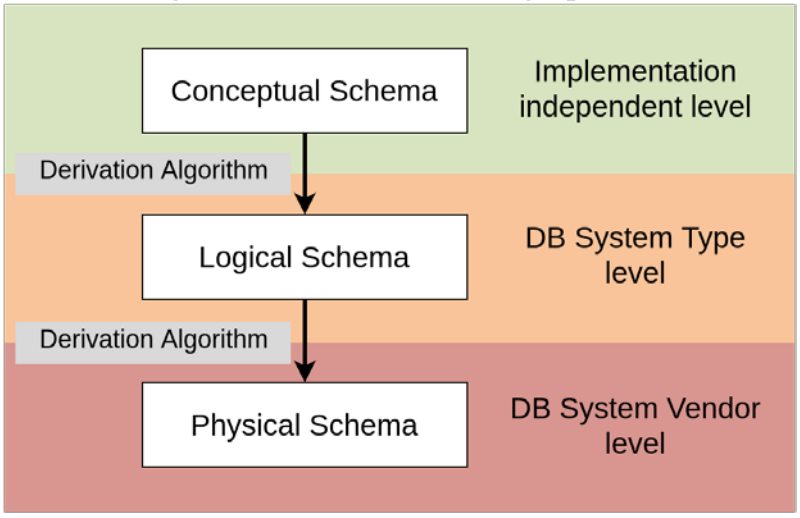



Figure 1 – Relationship between the stages of database design and their technological support

At the conceptual design stage, IT database developers, together with the Universe of Discourse (UoD) experts, create a conceptual model which is further converted into a logical one. The capabilities of the existing conceptual modelling languages have been analyzed, namely: ER, ORM / NIAM, IDEF1X, EXPRESS, EER, UML, Information Analysis and AGILA MOD, and it is shown that they all have the following disadvantages: they are partially inconsistent with the requirements of IT database developers; they are complex and incomprehensible to non-technical experts of the subject domain who are involved in reconciling the

functional requirements for the data; they do not have sufficient tools to automatically transform a conceptual schema into a data logic one with minimal manual intervention; they are focused on designing only relational databases and therefore are not used in the development of NoSQL or Polyglot Persistence databases. It is shown that it's the imperfection of the existing languages of conceptual modelling that leads to an increase in the time and manpower input (labour intensity) when designing the database.

From the point of view of the possibilities of converting a conceptual schema into a logical data schema, at the logical level the analysis of existing traditional and newest logical data models has been carried out. The continuity of traditional logical data models, such as hierarchical, network and relational ones, is shown. As well as within the study the continuity of traditional and new logical models has been also shown, for example, the relationship between relational and column-family models; between inter-hierarchical, network and graph-oriented; between object-oriented and document-oriented ones. The advantages of "key-value" logical data models for the creation of search databases have been substantiated.

It is shown that the use of different data storage technologies (Polyglot Persistence) within one application based on several databases with different logical models on the server-side has the following advantages: good scalability of the application, efficient management of heterogeneous data, faster response time and, as a result, higher performance.

But summing up what has been said about the process of designing a database at the conceptual and logical level, the author can conclude that the spread of NoSQL databases and the concept of multivariate persistence made the process of developing a database with their use even more time- and labour-consuming.

In connection with the above, the first section also formulates the purpose and objectives of the study.

The *second chapter* of the dissertation the language of conceptual modeling AGILA MOD + is improved for development of process automation of separate data conceptual models with multivariate persistence. AGILA MOD (Automatic

Generators for Information Representation, Logical DB structures and Applications - AGILA Modeling language - MOD) is an automatic generator for information representation, logical structures of databases. The general idea is to create a conceptual model by translating natural language as sentences that are not semantically simplified into graphic notation. The main structural elements are the objects O and the associations A between them are shown in Figure 1, and the general conceptual model is given as (1).


$$MOD =< O, A > \qquad (1)$$

Figure 2 – Objects-members of the association

The formal description is introduced: let there be a set O of objects of certain types, cardinality of set – $|O|$:

$$O = \{o(t)_1, o(t)_2, \dots o(t)_i, \dots o(t)_{|O|}\}, \qquad (2)$$

Type $t$ of object $o_i$ is defined as:

$$t = \begin{Bmatrix} \text{inheritance}\{o_i \cap o_j \vee o_i \cup o_j\}, abstract, \\ base\ \{\text{enumeration}, \text{set}\}, \\ system\ \{string, \text{integer}, \text{boolean}\} \end{Bmatrix} \qquad (3)$$

where $system$ – the usual data type, for example «String», «Integer» або «Boolean»; $base$ – basic type: enumeration is a fixed list of values and a set is an ordered collection without duplicates; $inheritance$ – specifies the inheritance between the supertype and its subtypes; $abstract$ – complex object type

A set $A$ of associations of certain types is given as:

$$A = \{a_1, a_1, \dots a_i, \dots a_{|A|}\} \qquad (4)$$

where $|A|$ – power set of associations

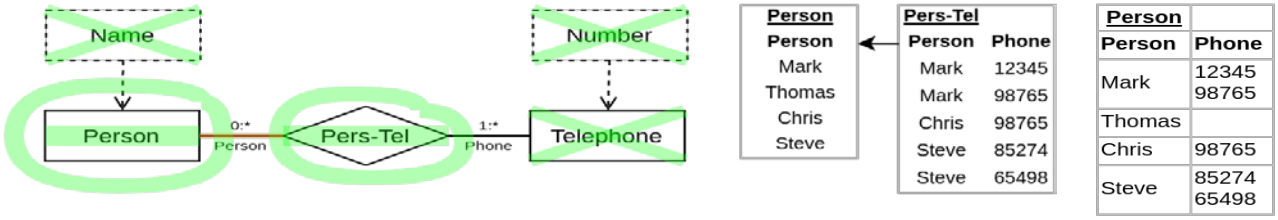
Then objects $o_i$ and $o_j$ are connected by association $a_i$, if:

$$\langle o_i, a_i, o_j \rangle = \{o_i \in O \wedge o_j \in O \wedge a_i \in A\} \qquad (4)$$

One of the most powerful but simple constructions of AGILA MOD is the concept of *objectification association*. Thus, the structured object is created. Objectification association allows the model developer to reuse the association as an object to model further sentences, which is not semantically simplified. The objectification of the association has the form:

$$\langle (o_i, a_i, o_j) \vee ((o_i, a_i, o_j), a_j, o_k) \rangle = \{o_i, o_j, o_k \in O \wedge (o_i, a_i, o_j) \in O \wedge a_i, a_j \in A\} (5)$$

Example of a conceptual model of a telephone book (Fig. 3, a) built with the help of existing structural elements of AGILA MOD shows the possibilities of AGILA framework for automated derivation of logical data models (Fig. 3, b, c).




(a) Conceptual model with derivation signs (b) Relational model (c) Object-relational model

Figure 3 – Examples of possible and impossible derivation of a conceptual model

It is shown that the existing syntax of AGILA MOD language has disadvantages in terms of defining and displaying Standalone objects, Cascading, and Nested Structures. Taking into account the disadvantages found, it is proposed to improve the syntax of AGILA MOD language (structure of the conceptual model) by introducing two structural elements: an *autonomous object* and *Strong Association Participants*.

An autonomous object expresses that the instances of the described element of UoD exist independently of its relations with other elements, is denoted by a rectangle with vertical double lines (Fig. 4, a). Its introduction allows to adjust the existing in AGILA framework about the need to cancel the participating objects of the type of association "with the lower limit "1"and is not part of the consistency component. The obtained relational model (Fig. 4, b) in contrast to the model (Fig. 3, b) shows that now An autonomous object "Telephone" receives its own relation, which is referred to by the relation "Pers-Tel".

Strong Association Participants means that the association is strongly associated with the connected object with «x: 1» than with the rest. That is, in the

association "Pers-Tel" "Person" owns "Telephone". In AGILA MOD +, such strong participation in association using a thick line (Figure 5, a).

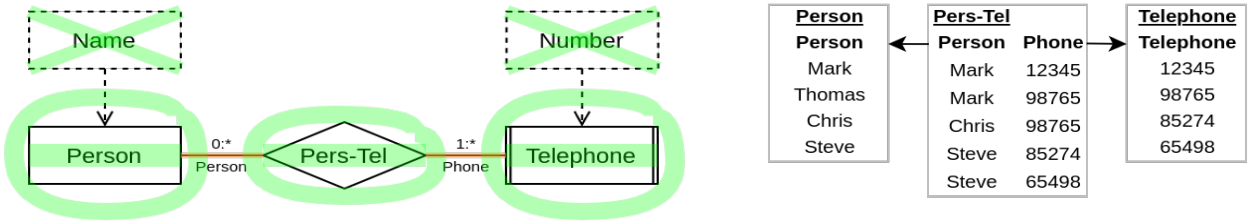
To derive from the conceptual model of the logical model into the AGILA framework, the following scenario is used:

1. one object type within a consistency component is referred to as a name;

2. all other object types within the relationship become identifying attributes;

3. all association type roles that lie inside the consistency component that point to an object type outside the consistency component become an attribute of the relationship;

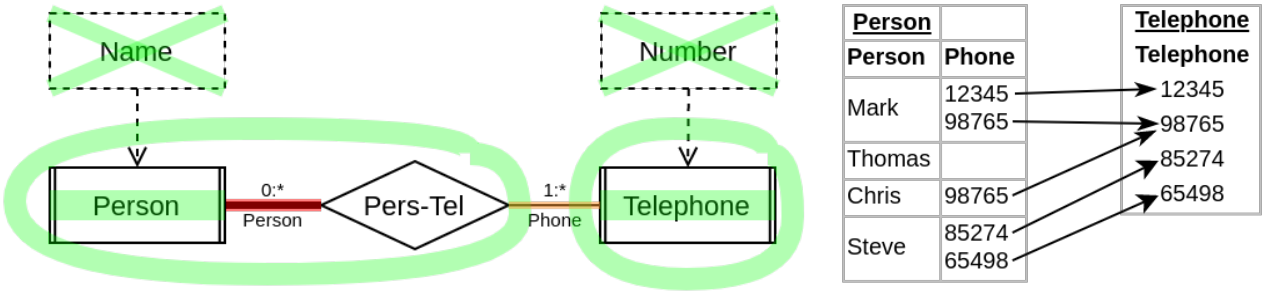
4. roles of this type, which are the result of marked strong participation with an upper limit bigger than 1, are encapsulated as nested attributes




(a) Conceptual model with derivation signs that take into account an in autonomous object

(b) Relational model

Figure 4 – The result of derivation using an autonomous object




(a) Conceptual model with derivation signs that take into account autonomous object and strong participation in association

(b) Object-relational model

Figure 5 – The result of derivation using an autonomous object and strong participation in association

In this case, paragraph 4 of this scenario is performed only in case of derivation of the conceptual model (Fig. 5, a) in the object-relational data model at the logical level (Fig. 5, b).

Thus, the second task of the dissertation research was completed and the first point of scientific novelty was obtained, namely, the structure of the conceptual data

model has been improved through the development of an independent type of objects and the introduction of a Strong Association Participants, which made it possible to improve logical model derivation rules taking into account the Polyglot persistence of data.

The *third chapter* of the dissertation represents methods of automation of database logical models development on the basis of conceptual submodels with multivariate persistence.

Adapted to the classical (see Fig. 1) database design process is represented as (Fig. 6):

1. Creation of a comprehensive conceptual model of UoD is with the help of extended language AGILA MOD +. Then an additional step is performed - division of a complex conceptual model into independent conceptual submodels, which take into account the features of data persistence at the logical level.

2. Sequential transformation using the algorithm of derivation of the obtained conceptual submodels into appropriate pathological models - "logical schemes", the appearance of which depends on the type of data persistence, which is supported by a specific database.

3. The corresponding data models are consistently transformed into the corresponding physical models (schemes) - internal structures of persistence of concrete DBMS.

To methodically support the actions of the first stage, a method of dividing the complex conceptual model of ABM into independent parts, the so-called "sections" of the conceptual model for their further derivation into appropriate logical models, was developed. The method includes the following stages:

1. Creating a copy of the existing complex conceptual model.

2. Surrounding with a closed, continuous translucent shell of elements of the complex conceptual model of objects, associations and lines of connections between them, from which the submodel is created.

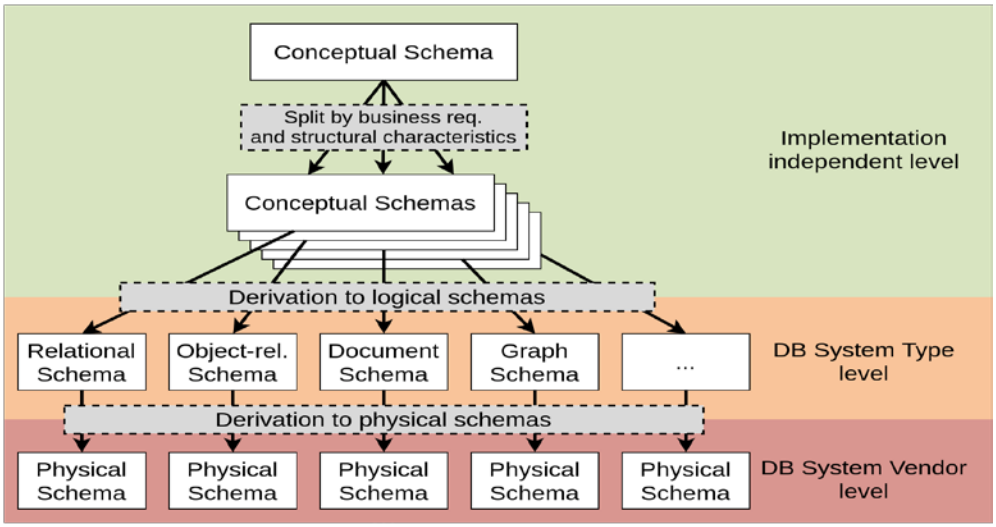
3. Delete all associations and connected lines that lie outside the shell.

4. Delete all system and base type objects that lie outside the environment shell and have no connection to objects that belong to the inner part of the environment.

5. Abandoning of all objects of the system and basic types that lie outside the shell of the environment and have connections with objects that belong to the inner part of the environment.

6. Replacement of the type of all abandoned objects that lie outside the shell of the environment and have a connection inside the type of "reference" (referencing) to the submodel to which the original object belongs.

Stages of the adapted database design process




Methodical support

**AGILA MOD +**

**Method of dividing** a complex conceptual model into submodels

**Method of creating** conceptual submodels

**API**

Figure 6 – The interconnection of the stages of the adapted process of database design, taking into account the Polyglot persistence and their developed methodological support

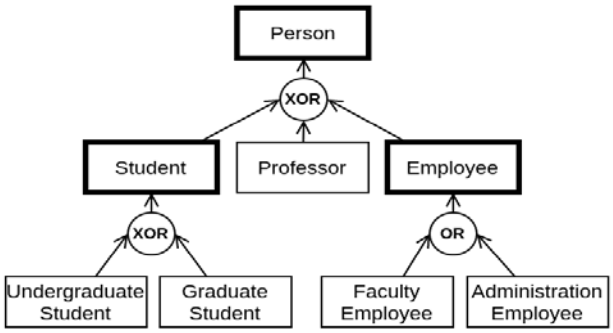
Thus, as a result of the division of a complex conceptual model, we have three sets of elements (objects, associations and lines of connection between them): «inside», «outside» and «remainder». The set «inside» includes elements surrounded by a shell in the second step, the «outside» - elements that were removed in the third and fourth steps. The set «remainder» consists of objects that have the type «link», lie outside the shell of the environment, and have connections with objects belonging to the set «inside».

To build shell – the environment of the complex conceptual model elements, a method of creating conceptual submodels that take into account the peculiarities of data persistence is proposed. The method involves the analysis of structural features

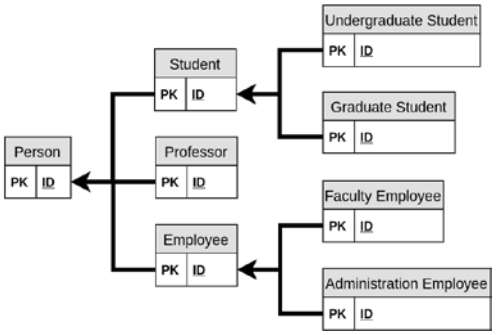
of many elements of a complex conceptual model, namely objects, associations and lines of connection between them:

1. The presence of strong links (see Fig. 5, a) between the object and the association, which are proposed in AGILA MOD +, is an indicator of the attribution of these elements to document-oriented or object-relational structures of persistence.

2. The presence of a hierarchy of object types is an indicator of the attribution of these elements to document-oriented or object-relational structures of persistence. AGILA MOD supports object inheritance through a hierarchy of object types. Figure 7 a, b and c show the conceptual model and results of inheritance derivation.




(a) conceptual model of object type hierarchy




(b) relational model of the hierarchy of object types

```
<s:complexType name="studentType" abstract="true">
  <s:complexContent>
    <s:extension base="personType">
      <s:sequence>
        <s:element name="studyProgram" minOccurs="1" type=" s:string"
        <s:element name="semester" type="s:positiveInteger" maxOccurs="1" />
        <s:element name="enrolmentNo" type="s:positiveInteger"
      </s:sequence>
    </ s:extension>
  </s:complexContent>
</s:complexType>
```

(c) document-oriented model (XML) inheritance output

Figure 7 – Model and results of inheritance derivation

For conceptual models or their parts that contain a significant number of inheritances, especially when inheritance is complex using arbitrary combinations of "OR", "XOR" and "AND", it is better to use document-oriented XML or JSON persistence structures.

3. The presence of a large number of associations such as "Self-Association" when the boundaries of the links of the objects-participants are "0: *" is an indicator of the attribution of these elements to graph-oriented structures of persistence.

Approbation of the developed methods on the example of the project KampfrichterAnmelde-System (KRAS) - a web application created for information support of the Bavarian Judo Federation - allowed to divide the existing complex conceptual model of UoD into two independent submodels for their further derivation into relational and LDAP models and creation of appropriate databases. Studies have shown that there was a reduction of 2.3 times the time for such design of two databases compared to their development according to the classical scheme.

In this way obtained the second and third points of scientific novelty, namely:

– for the first time a method of dividing a complex conceptual data model into independent submodels was developed, which allowed to automate the production of appropriate logical data models including relational and NoSQL structures and thus reduce database design time;

– for the first time a method of creating conceptual submodels that take into account the peculiarities of data persistence, including relational and NoSQL structures from a complex conceptual data model, which allowed to choose the type of data structures at the logical level and thus reduce database development time taking into account multivariate persistence.

Taking into account the improved structure of the conceptual data model and methods of automation of development of logical models of databases allowed to receive the fourth point of scientific novelty

– Improved conceptual modeling language AGILA MOD + due to the improved structure of the conceptual data model and methods of creating data models at the conceptual level, which allowed to take into account the multivariate persistence and reduce the design time of information system databases.

The *fourth chapter* develops an application programming interface (API) to provide unified access to databases with multivariate persistence in business logic.

*The fifth chapter* tests the conceptual modeling language AGILA MOD + and API in the development of databases in the process of IT projects and shows a significant reduction in design time, which is confirmed by the relevant implementing acts.

**Keywords:** Database development, Polyglot Persistence, Modeling of Universe of Discourse (UoD), Conceptual Modeling language, Entity-Relationship, NoSQL database, Relational databases, Database Management System (DBMS)

## СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

1. Елена Арсирий, Matthias Kolonko, Sabine Müllenbach, Борис Трофимов, Мария Глава Concept modeling for developing databases with polyglot persistence as a direction of business digitalization. *Устойчивое развитие: Международный журнал*. (Варна, Болгария) – 2020. - №2-2020. – С. 22-29. Періодиний журнал Технічного університету м. Варна, Болгарія. https://maurorg77.wixsite.com/maur-org/anotaciya

*2.* Müllenbach Sabine, Kern-Bausch Lore, Kolonko Matthias. Conceptual Modeling Language Agila Mod. *Herald of Advanced Information Technology, 2019* Vol. 2, No. 4, pp. 246-258. *Видання входить до міжнародних наукометричних баз Academia.edu, ROAD, Національна бібліотека України імені В. І. Вернадського, Djerelo, Україніка наукова, РИНЦ, Index Copernicus. https://hait.opu.ua/?fetch=articles&with=info&id=35*

3. Olena O. Arsirii., Maria G. Glava, Matthias Kolonko., Alina O. Glumenko. Information technology of supporting architectural solutions using polyglot persistence concept in learning management systems . *Applied Aspects of Information Technology.* 2020; Vol. 3 No. 2: – P 13-31 . *Видання входить до міжнародних наукометричних баз Academia.edu, ROAD, Национальная библиотека Украины имени В. И. Вернадского, Djerelo, Українікананаукова, РИНЦ, Index Copernicus* https://aait.opu.ua/?fetch=articles&with=info&id=42

4. M. Kolonko and S. Müllenbach, "Polyglot Persistence in Conceptual Modeling for Information Analysis," *2020 10th International Conference on*

*Advanced Computer Information Technologies (ACIT)*, Deggendorf, Germany, 2020, pp. 590-594, doi:10.1109/ACIT49673.2020.9208928.
https://ieeexplore.ieee.org/document/9208928 *Видання входить до міжнародних наукометричних баз* **Scopus, IEEE Xplore**.

5. Matthias Kolonko and Sabine Müllenbach. Modern Databases – What‘s really new about them?“ In: *Proceedings of the V. Ukrainian-German conference „Informatics. Culture. Technology.“ (IKT).* (Odessa, Ukraine). Odessa National Polytechnical University, May 2017, pp. 35–37. ISBN: 978-966-2666-18-2

6. Matthias Kolonko and Sabine Müllenbach. „From E³R to AGILA MOD – A Syntax Evaluation and Advancement towards a Comprehensive Conceptual Semantically Irreducible Modeling Language.“ In: *Proceedings of Modern Information Technology 2018.* (Odessa, Ukraine). Odessa National Polytechnical University, May 2018, pp. 172–173. UDC: 004.55. ISBN: 978-617-7046-50-8

7. Matthias Kolonko and Sabine Müllenbach and Elena Arsirii and Boris Trofymov. „Extensions to the Conceptual Modeling Language AGILA MOD“. In: *Proceedings of the VI. Ukrainian-German conference „Informatics. Culture. Technology.“ (IKT).* (Odessa, Ukraine). Odessa National Polytechnical University, September 2018, pp. 38–39. UDC: 004.652 ISSN: 2522-1523

8. Matthias Kolonko and Sabine Müllenbach. „An Analysis of Identification and Integrity Properties of Database Systems“ In: *Proceedings of the VII. Ukrainian-German conference „Informatics. Culture. Technology.“ (IKT).* (Odessa, Ukraine). Odessa National Polytechnical University, September 2019, pp. 53–57. UDC: 004.652. ISSN: 2522-1523

9. Глуменко А.О., Стельмах Д.Е., Маттиас Колонко, Арсирий Е.А. Использование многовариантного хранения и мультимодельных СУБД для обработки разнообразных данных. Сучасні інформаційні технології : мат. X Міжнар. наук. конф. студентів та молодих вчених, м. Одеса, 14 – 15 травня 2020 р. / МОН України; Одес. Нац. політех. ун-т; Ін-т комп’ют. систем. Одеса : Наука і техніка, 2020. С. 118–119.

ЗМІСТ

# ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ACID – Atomicity, Consistency, Isolation, Durability.

AGILA – Automatic Generators for Information representation, Logical DB structures and Applications.

ANSI – American National Standards Institute.

API – Application Programming Interface.

AUAS – Augsburg University of Applied Sciences.

BASE – Basic Availability, Soft State, Eventual Consistency.

CAP – Consistency, Availability, Partition tolerance.

CQL – Cassandra Query Language.

CRUD – create, read, update and delete.

DB – database.

DBMS – database management system.

DBS – database system.

DDL – Data Definition Language.

DN – Distinguished Name.

EER – Enhanced Entity-Relationship.

ER – Entity-Relationship.

JDBC – **J**ava **D**ata**b**ase **C**onnectivity.

JPA – Java Persistence API.

JPQL – Java Persistence Query Language.

JSON – JavaScript Object Notation.

KISS – Keep it simple, stupid.

KRAS – **K**ampf**r**ichter-**A**nmelde-**S**ystem.

LDAP – Lightweight Directory Access Protocol.

MVC – Model-View-Controller.

NFR – non-functional requirement.

NIAM – Natural language Information Analysis Methodology.

OID – object identifier.

OLAP – Online Analytical Processing.

OLTP – Online Transaction Processing.

OWL – Web Ontology Language.

PCR – Parent-child relationship.

PDO – PHP Data Objects.

PL/SQL – procedural language Structured Query Language (SQL).

POJO – Plain Old Java Object.

RDBMS – relational database management system.

RDF – Resource Description Framework.

RDN – Relative Distinguished Name.

SLA – Service Level Agreement.

SPARC – Standards Planning and Requirements Committee.

SQL – Structured Query Language.

UML – Unified Modeling Language.

UoD – universe of discourse.

URI – Unique Resource Identifier.

UUID – Universally Unique Identifier.

UWA – User Work Area.

W3C – World Wide Web Consortium.

XML – eXtensible Markup Language.

XSLT – XSL Transformation.

# ВСТУП

**Актуальність теми.** Процес розробки бази даних (БД), яка містить структуровану інформацію про конкретні об'єкти реального світу в будь-якій предметній області (ПрО), досі є довготривалим та трудомістким, незважаючи на сучасний рівень розвитку інформаційних технологій. Відомо що процес проектування БД є трьохетапним. На етапі концептуального проектування архітектор програмного забезпечення, враховуючи вимоги до даних та функціональні вимоги розробляє *концептуальну модель* ПрО – модель використання даних ПрО в конкретних цілях. На етапі логічного проектування із врахуванням існуючих концепцій персистентності (реляційної або NoSQL) концептуальна модель перетворюється в *логічну модель даних.* На етапі фізичного проектування в залежності від типу побудованої логічної моделі даних, обирається конкретна система управління базами даних (СУБД), засобами якої будується *фізична модель* зберігання даних.

Аналіз багаторічного досвіду розробки БД показує, що саме побудова концептуальної моделі ПрО визначає оперативність та якість процесу проектування БД в цілому та вимагає тісної співпраці між експертами з предметних питань та IT-спеціалістами. Ця взаємодія забезпечується за допомогою CASE-технологій моделювання – концептуальних мов моделювання, таких як Entity-Relationship (ER) у різних варіаціях та Unified Modeling Language (UML). Протягом останніх років Консорціумом всесвітньої павутини (World Wide Web Consortium – W3C) на основі Resource Description Framework (RDF) та Web Ontology Language (OWL) також був запроваджений семантичний підхід до розробки БД. Відома також мова концептуального моделювання AGILA MOD, яка займає проміжне місце між ER, UML та підходом W3C. Проведений аналіз існуючих CASE-технологій моделювання показав, що всі вони дозволяють скоротити час розробки БД за рахунок автоматизації побудови концептуальної моделі ПрО та перетворення її тільки в реляційну логічну модель даних з подальшим використанням однієї із

поширених реляційних СУБД (РСУБД), таких як Oracle Database, IBM DB2 и Microsoft SQL Server.

З іншого боку, із збільшенням обсягів даних що зберігаються, стали популярними нові технології персистентності – бази даних NoSQL, що мають такі переваги: швидкість обробки великих даних, масштабованість, розподіленість зберігання та обробки на стороні сервера. Сьогодні існує декілька типів NoSQL СУБД. Залежно від використовуваної ними логічної моделі даних відрізняють наступні типи: «ключ-значення», документно-орієнтовані, графові, стовбцеві (колоночні) сховища. Однак суттєвим недоліком NoSQL технологій персистентності є не універсальність використання, тобто прив'язка застосунків, які розробляються, не тільки до відповідної логічної моделі даних але і до конкретної NoSQL СУБД. Тому при необхідності зберігання гетерогенних даних ПрО використання будь якої однієї логічної моделі даних та конкретної NoSQL СУБД є неефективним. Крім того РСУБД все ще ефективно використовуються при обробці транзакцій. Тому на сучасному етапі проектування БД концепція *багатоваріантної персистентності* (БП), яка передбачає використання декількох СУБД з різними логічними моделями на стороні сервера, набуває особливої популярності. Але, як показав аналіз існуючих мов концептуального моделювання, поширення баз даних NoSQL та концепція багатоваріантної персистентності зробили процес розробки бази даних із їх використанням ще більш довготривалим та трудомістким.

Таким чином, актуальним завданням є розробка розширення мови концептуального моделювання та методів автоматизації розробки баз даних зі багатоваріантною персистентностю з метою зниження часу проектування БД

**Мета і задачі дослідження.** Метою даної роботи є скорочення часу на проектування баз даних зі багатоваріантною персистентністю шляхом удосконалення структури концептуальної моделі даних та методів створення моделей даних на логічному рівні

Для досягнення поставленої мети сформульовані та вирішені такі завдання:

– проаналізовано існуючі мови концептуального моделювання, технології персистентності, методи для автоматизації процесу розробки баз зі багатоваріантною персистентністю та обгрунтовано мету та задач дослідження;

– удосконалено мову концептуального моделювання AGILA MOD+ для автоматизації процесу розробки окремих концептуальних моделей даних зі багатоваріантною персистентністю;

– створено методи автоматизації розробки логічних моделей баз даних на основі концептуальних моделей зі багатоваріантною персистентністю;

– розроблено інтерфейс прикладного програмування (API) для забезпечення єдиного доступу до баз даних зі багатоваріантною персистентністю із бізнес-логіки;

– розроблені рішення (моделі та методи автоматизації та API) впроваджені для удосконалення мови концептуального моделювання AGILA MOD+, показано їх переваги при вирішенні практичних задач.

**Об’єкт дослідження** – процес розробки баз даних з урахуванням багатоваріантної персистентності даних.

**Предмет дослідження** – моделі та методів автоматизації проектування баз даних з урахуванням багатоваріантної персистентності даних.

**Методи дослідження.** При створенні наукових положень, висновків та рекомендацій автор застосовує дані сучасних наукових джерел, спирається на результати аналізу низки існуючих мов концептуального моделювання, власних досліджень базових та нових технологій персистентності, а також методів для автоматизації процесу розробки баз даних. Достовірність концептуального моделювання даних та методів розробки баз даних з урахуванням багатоваріантної персистентності підтверджується використанням існуючих засобів прямого та зворотного проектування при створенні відповідних логічних моделей та фізичних схем даних щодо обраних СУБД та практичним застосуванням розробленого за допомогою об’єктно-орієнтованого

програмування API для забезпечення доступу к персистентним структурам даних із бізнес логіки. Крім того, достовірність наукових положень, висновків та рекомендацій підтверджується даними комп'ютерних досліджень та практичними результатами, які підтверджуються наведеними актами організаційно-технічних випробувань, результатами практичного використання із позитивним ефектом.

**Наукова новизна одержаних результатів** полягає у вирішенні науково-практичної задачі удосконалення та розробки моделей та методів для автоматизації проектування баз даних зі багатоваріантною персистентністю.

1. Удосконалено структуру концептуальної моделі даних за рахунок розробки незалежного типу об'єктів та введення «сильного» типу звязку для об'єкта-учасника асоціації, що дало можливість вдосконалити правила деривації логічних моделей з урахуванням багатоваріантної персистентності даних.

2. Вперше розроблено метод розділення комплексної концептуальної моделі даних на незалежні субмоделі, що дозволило автоматизувати отримання відповідних логічних моделей даних включаючи реляційні та NoSQL структури та тим самим скоротити час проектування баз даних.

3. Вперше розроблено метод створення концептуальних субмоделей, які враховують особливості персистентності даних включаючи реляційні та NoSQL структури із комплексної концептуальної моделі даних, що дозволило обґрунтовано вибрати тип структур даних на логічному рівні і тим самим зменшити час розробки баз даних з урахуванням багатоваріантної персистентності

4. Удосконалено мову концептуального моделювання AGILA MOD+ за рахунок за рахунок удосконаленої структури концептуальної моделі даних та методів створення моделей даних концептуальному рівні, що дозволило враховувати багатоваріантну персистентность та скоротити час проектування баз даних інформаційних систем

**Особистий внесок здобувача.** Всі наукові положення, висновки та рекомендації, що містяться в дисертації, отримано здобувачем особисто. В роботах, які написано в співавторстві, автору дисертації належать: розробка елементів автономних типів об'єктів та введення «сильних» учасників асоціативних зв’язків між об'єктами в структурі концептуальної моделі даних [1,2,6,7]; метод розділення загальної концептуальної моделі даних на незалежні частини [1-3,8]; метод створення логічних моделей баз даних включаючи реляційні та NoSQL структури [1,2,4,9]; удосконалення мови концептуального моделювання AGILA MOD+ [1-9].

**Апробація результатів дисертації.** Основні положення дисертаційної роботи доповідалися та обговорювалися на наукових семінарах кафедри інформаційних систем ОНПУ та наукових конференціях: V Міжнародна науково-практична конференція магістрантів, аспірантів та науковців V, VI та VII Українсько-німецька конференція «Інформатика. Культура. Техніка» (м. Одеса, 2017, 2018 та 2019 рр.); Міжнародна науково-практична конференція «Сучасні інформаційні технології» (м. Одеса, 2020 р.) та на X Міжнародна конференція «Advanced Computer Information Technologies» (ACIT'2020, Scopus, IEEE Xplore) (м.Дюсельдорф, 2020 р.).

**Структура дисертаційної роботи.** Дисертація має загальний обсяг 208 сторінок і складається зі вступу, п’яти розділів, висновків, списку використаних джерел та трьох додатків. Основний текст дисертації викладено на 181 сторінках. Робота містить 80 рисунків, 3 таблиці та список використаних джерел з 76 найменувань.

**Практичне значення одержаних результатів** Практичне значення отриманих результатів полягає у скороченні часу на розробку баз даних зі багатоваріантною персистентністю завдяки використанню удосконаленої мови концептуального моделювання даних AGILA MOD+ для побудови незалежних моделей даних на концептуальному рівні, методів автоматизації розробки реляційних та NoSQL структур даних на логічному рівні та інтерфейсу прикладного програмування (API) для забезпечення єдиного доступу до баз

даних зі багатоваріантною персистентністю на фізичному рівні із бізнес-логіки.

Здійснені практичні випробування отриманих моделей даних, методів розробки баз даних та запропанованного API при створенні баз даних зі багатоваріантною персистентністю для веб-застосувань Федерації баварського дзюдо – Kampfrichter Anmelde-System (KRAS) та системи управління дистанційним навчанням «JustStar» показали скорочення часу на розробку баз даних в середньому на 50%. Практичне використання розроблених моделей, методів та API при розробці курсових проектів в рамках навчального процесу на кафедрі інформаційних систем ОНПУ та факультеті комп'ютерних наук Університету прикладних наук м. Аугсбург також показало скорочення часу на розробку баз даних на етапах концептуального, логічного та фізичного моделювання при значному зменшенню кількості помилкових рішень студентів.

Основні положення, висновки та рекомендації, що викладені в дисертаційній роботі використовуються в навчальному процесі Одеського національного політехнічного університету в дисциплінах «Організація баз та сховищ даних», «Сучасні технології баз даних» та «Імовірностні алгоритми та структури даних», які викладаються на кафедрі інформаційних систем для студентів спеціальності 122 – «Комп'ютерні науки».

# РОЗДІЛ 1
# АНАЛІЗ ІСНУЮЧИХ КОНЦЕПТУАЛЬНИХ МОДЕЛЕЙ ТА МЕТОДІВ АВТОМАТИЗАЦІЇ ПРОЦЕСУ РОЗРОБКИ БАЗ ДАНИХ

## 1.1. Аналіз існуючих підходів до проектування баз даних

У цьому пункті розглядається поточна ситуація щодо підходів до моделювання даних в процесі проектування БД. Розглянуто узагальнену концепцію процесу проектування баз даних, представлену в 1994 році в [10], яка є актуальною і сьогодні, аби бути представленою в якості базової системи для оцінки наявних на даний момент мов концептуального моделювання. Показані

Особлива увага приділяється існуючим мовам концептуального моделювання. Далі розглянуті структурні особливості існуючих традиційних та новітніх логічних моделей даних та засоби автоматизации процесу проектування БД

### 1.1.1. Процес проектування баз даних згідно Elmasri / Navathe

У класичній роботі [10] Ramez Elmasri and Shamkant B. Navathe (Рамес Ельмасрі та Шамкант Нават) представили у 1994 році узагальнену концепцію процесу проектування баз даних. Робота перевидавалася, останній раз у 2017 році [11]. Представлену концепцію можна знайти за різними посиланнями [12], [9]. У [7, гл. 3.1] автори описують процес проектування БД у більш широкому контексті програмної інженерії. Згідно узагальненої концепції процес проектування бази даних є взагалі складним та залежить від багатьох факторів, проте в ньому розглядають три основні етапи або так звані рівні розробки архітектури: концептуальний рівень, логічний рівень, фізичний рівень (Рис.1.1)

Учасники процесу розробки БД (IT-розробники та спеціалісти предметної області (ПрО)) спільно аналізують «вимоги» ПрО (universe of

discourse (UoD). А саме «вимоги до даних» та в деякій мірі «функціональні вимоги», хоча вони є основою для проектування програмного забезпечення. На основі аналізу вимог до даних будується «концептуальна схема». Вона є незалежною від специфіки будь-якої системи управління базами даних (СУБД). Після створення концептуальної моделі, наступним кроком є опис IT-розробники «фактичної реалізації бази даних із використанням комерційної СУБД». На другому етапі концептуальна модель перетворюється за допомогою алгоритму виведення у «логічну схему», на яку спирається зазначена СУБД. Вигляд логічної (даталогічної) моделі залежить від типу персистентності даних, який підтримується засобами конкретної СУБД.

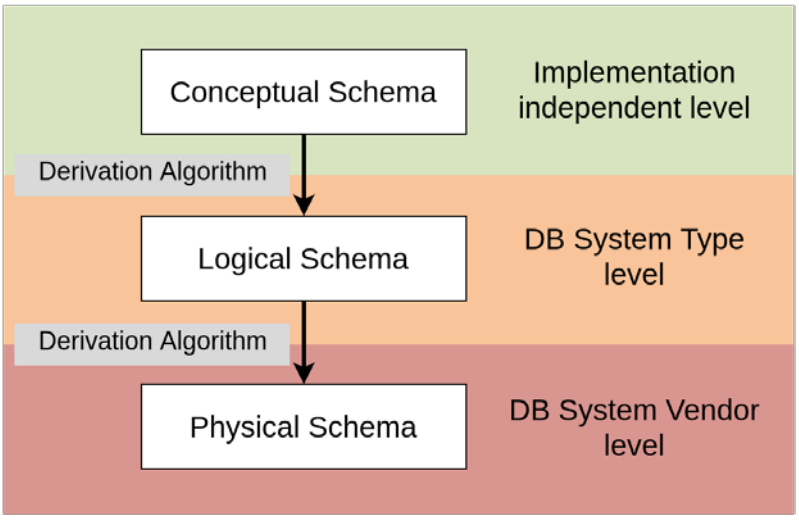



Рисунок 1.1 - Процес проектування бази даних
(адаптовано за [11, рис. 3.1, с. 91])

Таким чином, необхідно зазначити, що мова моделювання високого рівня, що використовується для побудови концептуальної моделі повинна задовольняти наступними вимогам:

– побудована концептуальна модель повинна виступати спільною платформою для технічних та нетехнічних користувачів, на якій вони узгоджують визначені вимоги та підтримувати ефективне спілкування в рамках проекту.

– побудована концептуальна модель, як письмовий документ може виступати в якості посилання в проектній документації.

– відображення концептуальної схеми в логічну схему даних повинно бути автоматизованим або частково автоматизованим, отже, ручне навантаження на даному етапі має бути мінімальним

Крім того необхідно зазначити, що Ельмасрі / Навате називають логічну схему як «модель реалізації даних» при цьому вказують, що «Більшість сучасних комерційних СУБД використовують модель реалізації даних, таку як реляційна (SQL) модель...» [11, гл. 3.1, с. 92] Отже, «модель реалізації даних», на яку посилаються Ельмасрі / Навате, не обов'язково є специфічною для СУБД, а описує певну концепцію *логічного представлення даних*.

На третьому етапі процесу проектування БД ІТ-розробники створюють «фізичну схему» БД, де визначаються фізичні аспекти БД, які включають «внутрішні структури зберігання, файлову організацію, індекси, шляхи доступу та параметри фізичного проектування файлів бази даних».

Таким чином, підсумком етапу концептуального проектування є незалежне від реалізації представлення структур даних, що визначено тільки вимогами до даних. Концептуальна модель даних не тільки незалежна СУБД, але і від будь-якої парадигми персистентності даних, визначеної для інформаційних технологій. Вона є відображенням структурних аспектів ПрО (UoD), які були проаналізовані та задокументовані як «вимоги до даних». Підсумком етапу логічного проектування – відображення концептуальної моделі з врахування певної парадигми персистентності даних в логічну модель. У цьому тлумаченні логічна модель представляє лише структури даних у відповідності до парадигми персистентності, наприклад, реляційну, ієрархічну, документо-орієнтовану, графову тощо. На цьому етапі не враховується жодної специфіки конкретної СУБД. На етапі фізичного проектування логічна модель відображається в конкретному вигляді СУБД, беручи до уваги всі згадані фізичні аспекти та особливості цієї СУБД.

У подальшому курсі [10] автори визначили алгоритми виведення різних логічних моделей із концептуальної моделі - зокрема для реляційної [10, гл. 6.8], ієрархічної [10, гл. 10.4], мережевої [10, гл. 11.5] та об’єктно-орієнтованої

[10, гл. 22.8] моделі. Отже, можна отримати крок від концептуальної до логічної моделі, широко автоматизованої, застосовуючи конкретний алгоритм. Це також було зазначено в [12]. Цей процес "виведення" також має бути можливим для відображення логічних структур у фізичну модель. На рисунку 1.1 показано адаптацію процесу проектування з урахуванням цієї вдосконаленої інтерпретації описаного процесу проектування бази даних, включаючи ідею застосування алгоритмів виведення до вищої моделі для отримання нижчої моделі.

Наступні міркування будуть приймати цю передову інтерпретацію процесу проектування бази даних в якості орієнтира для даного моделювання даних підходів анкетування та того, наскільки вони йому відповідають.

### 1.1.2. Класифікація та аналіз мов концептуального моделювання даних

З кінця 1960-х моделювання даних було предметом багатьох досліджень, і в наступні десятиліття з'явилося багато підходів. Розглядаючи сучасну практику, визначено два типи підходів до моделювання даних: сутність-зв’язок (Entity-Relationship ER ) у різних варіаціях та Unified Modeling Language (UML).

На рисунку 1.2 різні назви мов розташовані вертикально за роком їх розробки, починаючи із мови «Data Structure Diagrams» у 1969 році внизу. Стрілки вказують від попередників до наступників. Цілком природно, що конкретна мова може мати більше одного попередника Наприклад, підход Чена до створення мови ER базується на «Data Structure Diagrams», а також використовує поняття "BDM".

Крім розподілу на ER-орієнтовані та UML-орієнтовані засоби моделювання, на рисунку 2.1 показані межі приблизної класифікації мов концептуального моделювання за трьома різними типами вихідних моделей:

*Графічно орієнтовані моделі* зосереджені на графічному представленні узагальненого погляду на структуру конкретних даних.

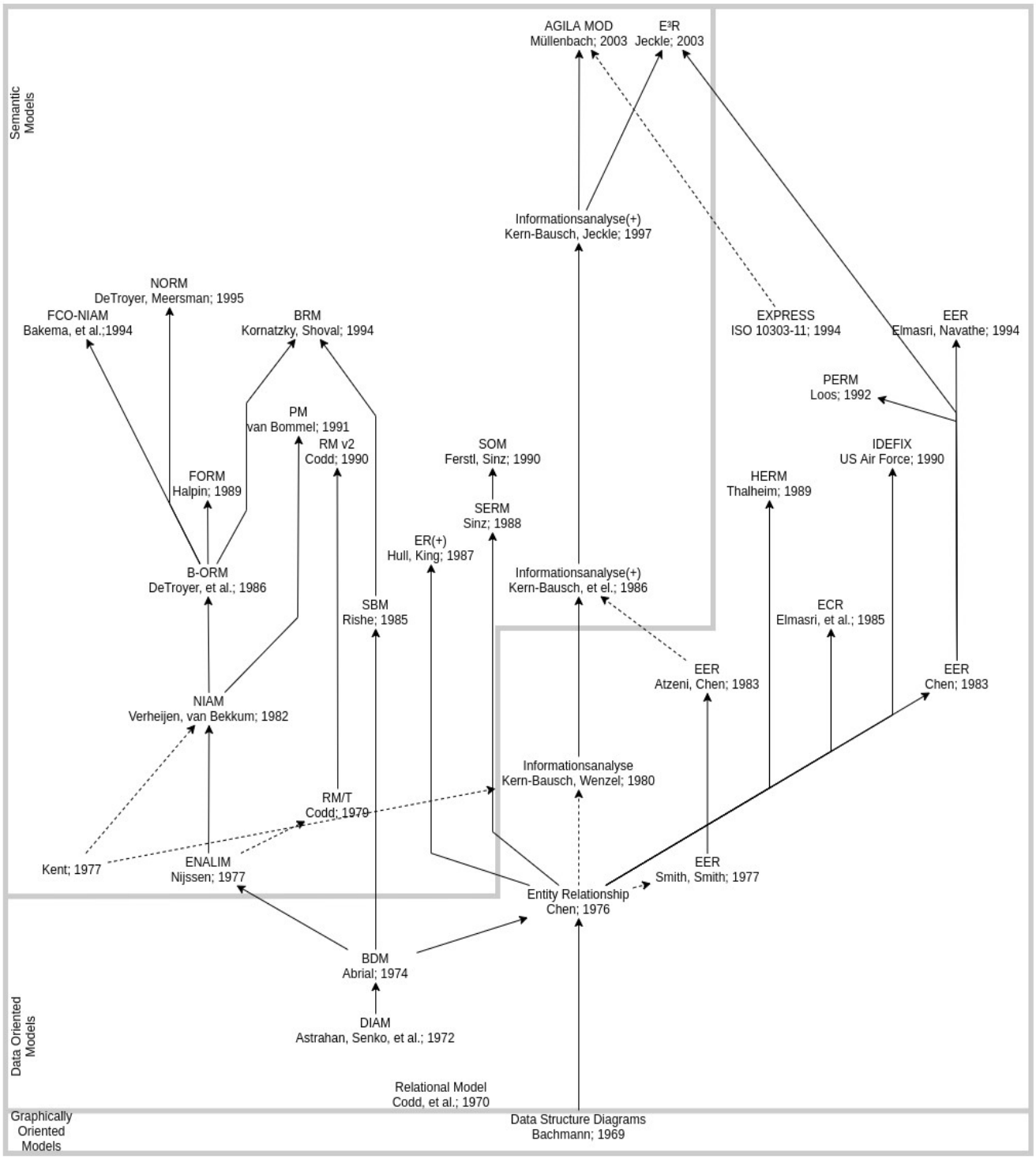



Рисунок 1.2 – Історичний взаємозв’язок підходів до моделювання даних [13]

*Моделі, орієнтовані на дані***,** зосереджені на описі структурних атрибутів даних для певного домену в структурованому вигляді. Цей опис не обов'язково повинен бути графічним.

*Семантичні моделі* більше зосереджуються на описі об’єктів та їх семантики, оскільки вони з’являються в реальному світі, а не просто на технічних структурах даних.

Необхідно також зазначити, що с часом мови концептуального моделювання відходять від суто технічної точки зору на структури цифрових даних і стають більш зрозумілими людині, яка не є IT-спеціалістом.

Ідея використання семантичних моделей при проектуванні БД в ІС набула великої популярності. Розвиток веб-технологій висунув кілька підходів, які також базуються на ідеї семантичного моделювання. Відомими прикладами тут є Resource Description Framework (RDF) та Web Ontology Language (OWL). Хоча ці засоби не є мовами концептуального в широкому снесу, так як використовують природну мову, передану технічним структурам, для зберігання «фактів» реального світу, як це роблять люди. Навіть більше, будуючи онтології за допомогою цієї техніки – що є декларованою метою OWL - цей підхід створює основу для штучного інтелекту та машинного навчання.

Розглянемо деякі засоби концептуального моделювання більш детально.

*Діаграми структури даних* від Бахманна (Data Structure Diagrams by Bachmann) [11] (Див. рис.1.2). В основі роботи Бахманна представлення «класів арифметичних тверджень» таких як *2 + 3 = 5, 17 + 4 = 21* або *5 + (- 3) = 2* за допомогою алгебраїчних тверджень типу $a + b = c$.

Бахманн представляє узагальненим графічним способом конкретні об'єкти – «сутності» у вигляді «класу сутності». Сутності, що належать до класу сутності, мають спільну основу, за якою їх можна об’єднати. Клас сутності представляється у вигляді простого прямокутника, що містить ім'я певного класу, який він представляє. Наприклад, «Міллер», «Фуллер» та «Сміт» можуть бути сутностями класу «Person», що включає ці сутності. Бахманн визначає термін «сукупності сутностей». Вони асоціюють групу сутностей одного класу з однією сутністю іншого класу. Таким же чином визначені «набори», які зв’язують сутності та «клас наборів». Ці класи наборів представлені як спрямовані кінці від одного класу сутності до іншого, де початкова точка

позначає клас сутності одного екземпляра, а кінцева точка класу сутності згрупованих екземплярів. На рисунку 1.3 показаний простий приклад діаграми структури даних, який взято з оригінальної статті.

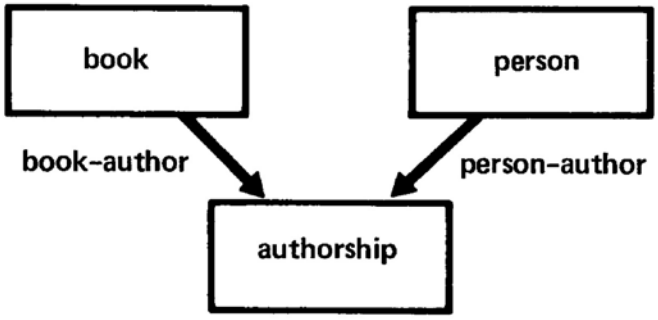



Рисунок 1.3 - Приклад діаграми структури даних. ([14, с. 8])

В той час Бахманн розроблял концептуальні моделі для мережевих баз даних, які на той час були вважалися сучасними. Засоби візуалізації структур даних від Бахмана є прокладом графічно орієнтованої моделі.

*Моделювання Чена «Сутність-зв'язок»* (Entity-Relationship by Chen) [15] (Див. рис.1.2). У своїй роботі Пітер Чен відштовхується від трьох логічних моделей «мережевої», «реляційної» та «сукупності сутностей» та пропонує «узагальнення (...) існуючих моделей» за рахунок «єдиного погляд на дані», що забезпечує «незалежність даних». Це є першою спробою використання семантичного аспекту моделювання даних в концептуальному моделюванні ПрО. На рисунку 1.4 наведено простий приклад оригінальної моделі Чена ER

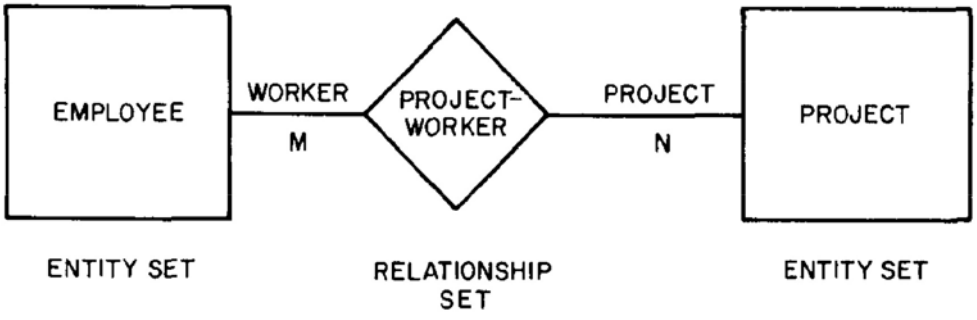



Рисунок 1.4 - Проста модель ER. ([15])

На погляд автора саме Чен заклав основу для процесу проектування бази даних від Ельмасрі / Навате, який був описаний раніше у [10] та таке інш.

Мова Чена «Сутність-зв'язок» має дві переваги над моделлю Бахманна:

– введення поняття «набор відносин», що зображено на рис.1.4 у визді ромбу замість спрямованих ліній (стрілок), що забезпечило більш природне та гнучке моделювання взаємозв’язків.

– відображення відносини між задіяними суб'єктами є більш широкім на відміну від моделі Бахманна, де можна визначити лише ролі..

– введення поняття залежних сутностей, які не можуть існувати без сутності, від якої вони залежать.

Однак в порівняно із загальноприйнятими сьогодні варіантами ER, цей оригінал не ввів жодних атрибутів до примітивів синтаксису.

Отже, модель «Сутність-зв'язок» - це перший підхід із використанням фактів реального світу в моделі, орієнтованої на дані. Отримані моделі сприймаються як узагальнення логічних моделей.

*Об’єктно-реляційне моделювання та NIAM* Object-Relational Mapping (ORM) and NIAM (Див. рис.1.2) . Часто акронім «ORM» використовується як для «об'єктно-рольового моделювання» (Object-Role Modeling) , так і для «об'єктно-реляційного відображення». ORM – група підходів до моделювання, які зосереджуються на ідеї семантичного моделювання. Методологія аналізу інформації про природну мову (NIAM) [16] була одним із найперших підходів, які запровадили дане  поняття в 1970-х роках. Даний підхід базується на ідеї використання природної мови для опису ПрО (UoD). Цей термін було введено для позначення домену. Експерти з питань домену на природній мові описують певні причинно-наслідкові зв'язки UoD – «факти» в ORM. Формалізовані речення переносяться на графічну модель, на яку можуть посилатися технічні спеціалісти IT-розробники. Опис доповнено відповідними прикладами даних та так званими  «контрприкладами», що створює чітку диференціацію щодо того, що вважатиметься правильним та неправильним. [17, гл. 1.2, с. 11] На рисунку показано приклад графічного подання моделі у поєднанні з прикладами (зеленим кольором) та контрприкладами (червоним кольором)

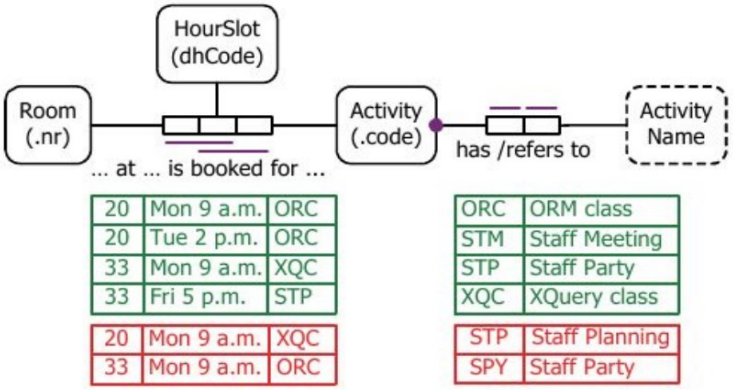



Рисунок 1.5 – Приклад ORM моделі ([17])

Основною вадою даного моделювання є той факт, що експерти з питань ПрО та розробники БД техніки посилаються на різне представництво з тієї ж справи.

*IDEF1X стандарт ISO* [18] (Див. рис.1.2). У стандарті вказано що це «мова, яка використовується для створення концептуальної схеми» Мова базується на роботі Чена і моделі [19-21]. Приклад моделі IDEF1X показаний на рисунку 1.6

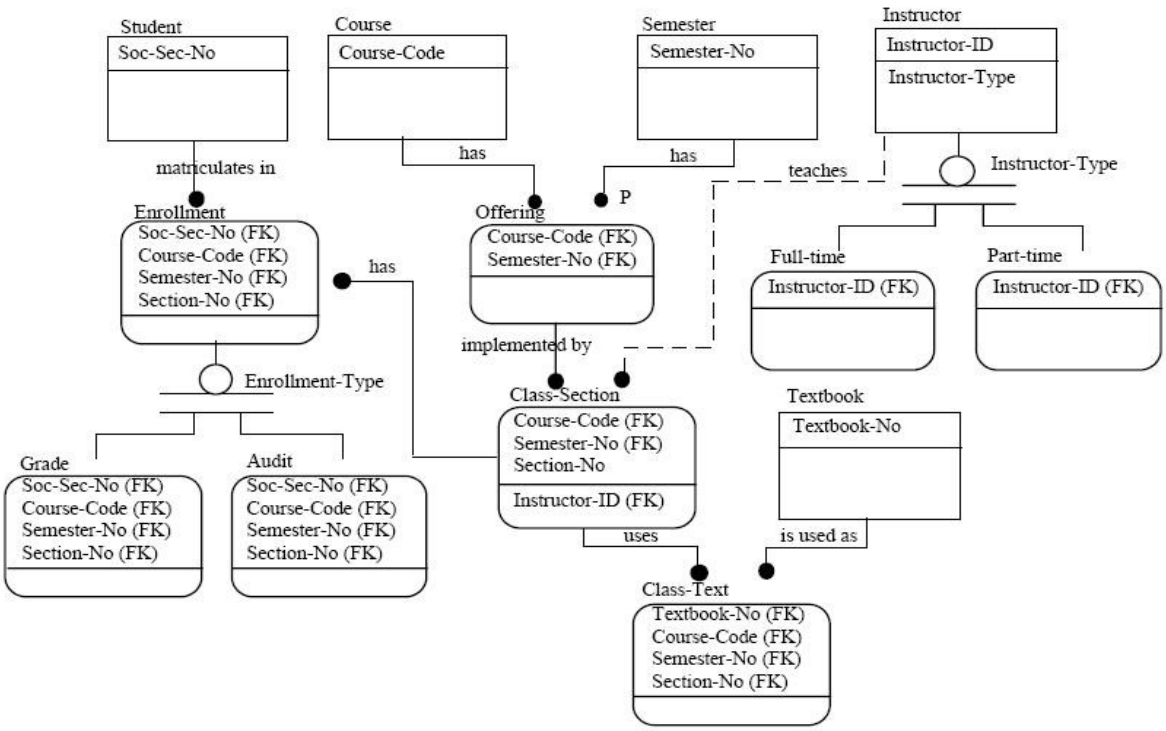



Рисунок 1.6 - Приклад моделі IDEF1X. (взято з [18])

IDEF1X в основному призначений для моделювання структур даних з метою введення їх у СУБД. [19]. IDEF1X використовує вже *атрибути* для

опису сутності більш детально. Основними недоліками IDEF1X є висока складність розробки, міцна прив'язка до реляційної моделі та всіх недоліків ER підходу

*EXPRESS є стандартом ISO* з 2004 р, розроблений як формат обміну даними, реалізує мову для створення концептуальних моделей, щодо проектування баз даних згідно Ельмасрі / Навате. Мова *EXPRESS* вважається незалежною від реалізації в рамках конкретної СУБД та призначається для опису структур даних для обміну між виробничими процесами. Проте EXPRESS можна використовувати для опису будь-якої структури даних на концептуальному рівні незалежно від подальшого практичного спрямування. Засоби мови EXPRESS спрямовані на опис обробки інформації, а не на опис структурної інформацію. EXPRESS була створена не як мова графічного моделювання, а як структуроване текстове представлення, яке одночасно є зручним для читання та комп'ютерної обробки. Але графічне представлення - EXPRESS-G (Рис 1.7), що було створено пізніше забезпечує візуалізацію моделі

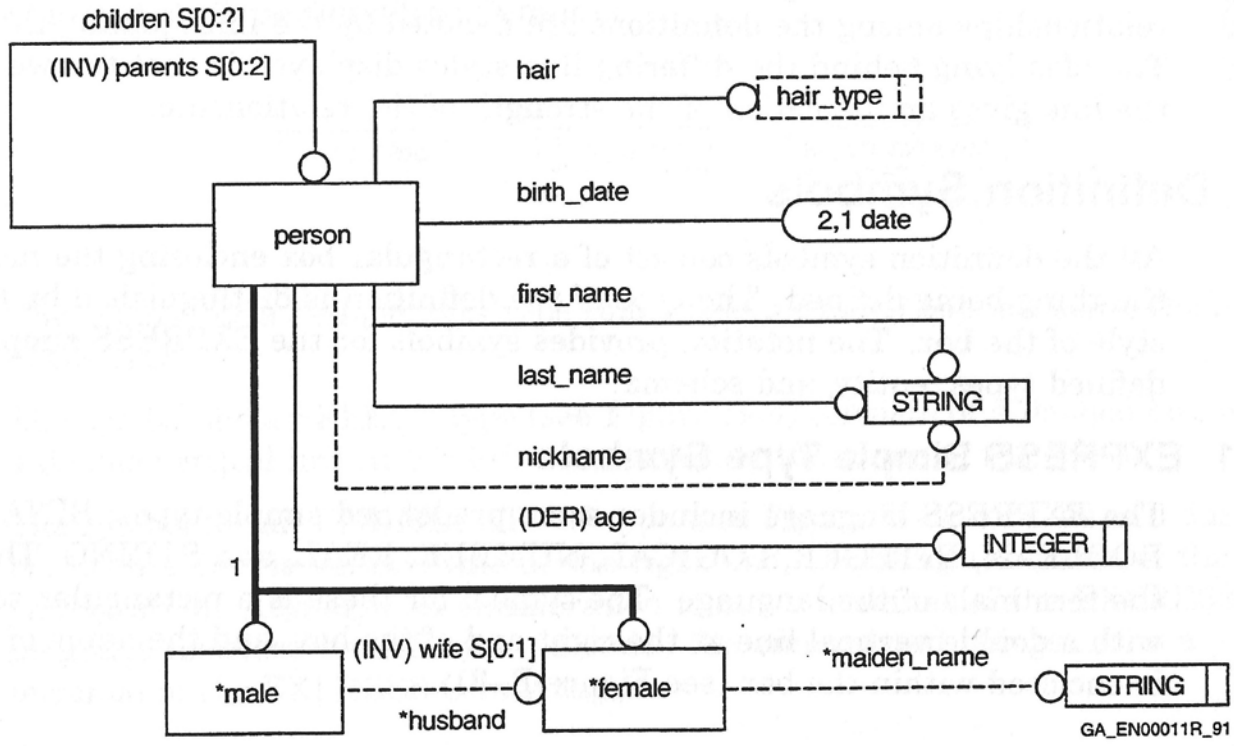



Рисунок 1.7 - Приклад моделі EXPRESS-G. ([19, рис. B-1])

До мови EXPRESS входять засоби опису сутностей з атрибутами та обмеженнями, яким ці сутності повинні слідувати, ці засоби відображають об'єктно-орієнтовані концепції узагальнення з «суб- та супертипами», а також абстрактними сутностями. До мови EXPRESS додано концепцію типів агрегування для атрибутів сутності, тобто списків значень та наборів.

Таблиця 1.1 Приклади лістингів визначення схеми та сутностей в EXPRESS [22]

| Визначення схеми БД | Визначення сутностей із схеми |
|---|---|
| SCHEMA example;<br>ENTITY entity1;<br>a: integer;<br>b: integer;<br>END ENTITY;<br><br>ENTITY entity2;<br>a: entity1;<br>b: integer;<br>END ENTITY;<br>END SCHEMA; | ENTITY unit_vector;<br>a, b: REAL;<br>c : OPTIONAL REAL;<br>WHERE<br>length_1 : a**2 + b**2<br>+ NVL(c, 0.0)**2 = 1.0;<br>END ENTITY; |

*(E)ER від Ельмасрі / Навате* (Elmasri / Navathe) Крім описаної вище трьохетапного процесу проектування БД у [10, гл. 3] Ельмасрі / Навате представили власну версію моделювання ER. В якості розширення (Enhanced Entity-Relationship) EER було введено додаткові синтаксичні примітиви (рис 1.8):

*Ідентифікуючі відносини* (*Identifying Relationship*) – спеціальний тип відносин, який сигналізує про те, що відносини ідентифікуються. Він позначається як ромб з подвійною суцільною лінією.

*Атрибути* (*Attributes*) були додані або до сутності, або до відносин, позначаються як еліпс. Атрибут може бути регулярним, ідентифікуючим, багатозначним, складеним – це атрибут, який створюється як комбінація інших атрибутів - або із інших атрібутів. Атрибути не можуть бути задяними у

відносинах. Вони пов'язані з певною сутністю або відношенням суцільною лінією.

*Загальні учасники* (*Total Participation*) Суб'єкт є пов'язаним із відносинами зазначеними в ромбі означає, що кожна сутність повинна брати участь у цих відносинах принаймні один раз. Не може бути екземпляра, який не був би задяний у цих відносинах. Цей випадок позначається подвійною лінією між зв'язком та сутністю.

*Мінімальна / Максимальна потужність* (*Min/Max Cardinalities*)**.** На додаток до потужності, яку Чен ввів для учасників відносин, Ельмасрі / Навате додали додаткові позначення, де можна визначити як мінімальну, так і максимальну межу участі.

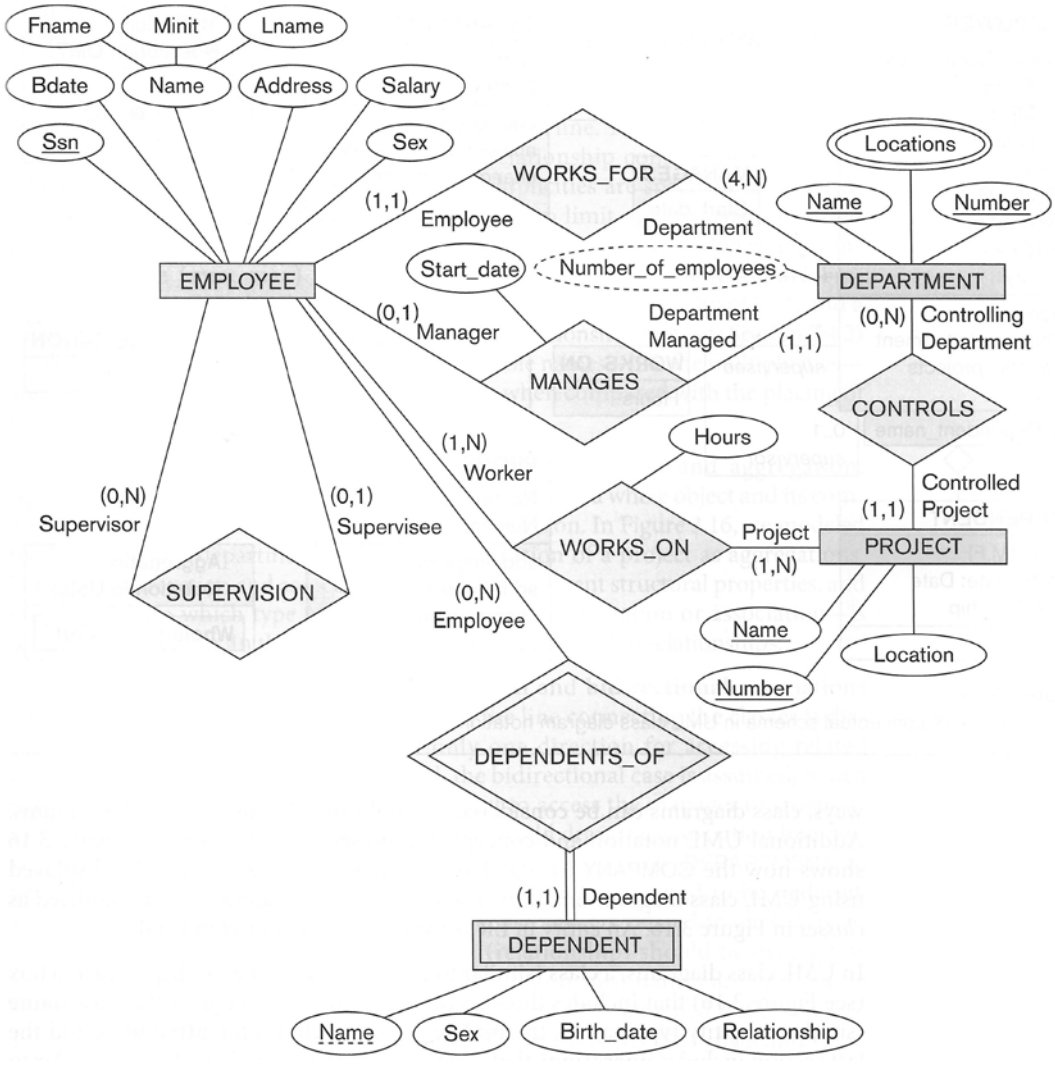



Рисунок 1.8 – (E)ER-модель Ельмасрі / Навате ([11, рис. 3.15, с. 115])

Крім того в EER було додано такі принципи об'єктно-орієнтованої концепції як узагальнення та успадкування (generalization and inheritance) (Рис.1.9).

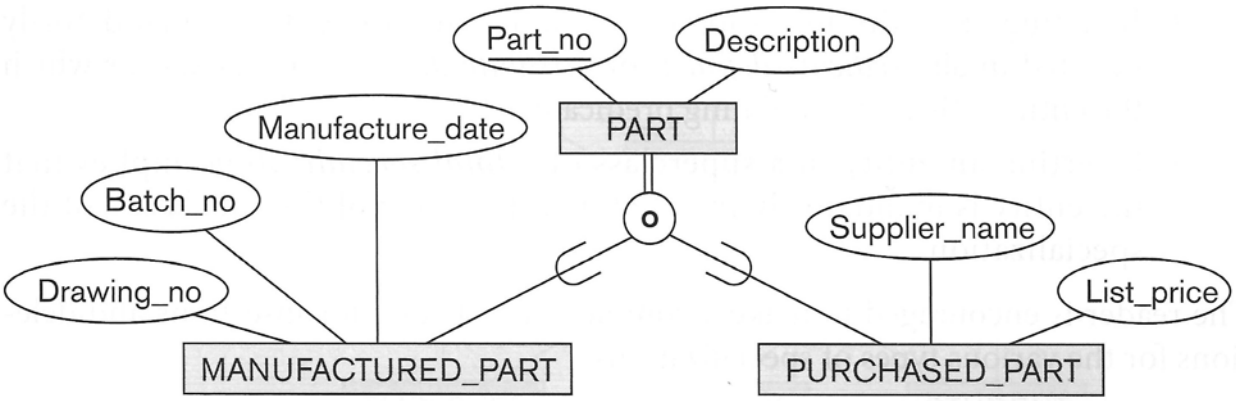



Рисунок 1.9 - Приклад узагальнення EER. [11, рис. 4.5, с. 145]

Але нові конструкції хоча і збільшують виразність EER, але одночасно й ускладнюють її. Оскільки зараз існує більше синтаксичних примітивів, які потрібно зрозуміти. Однак Elmasri / Navathe довели своєю роботою, що ER може перейти від простої логічної моделі даних до узагальненої моделі даних, яка може виступати в якості платформи зв'язку та документації, не вказуючи жодних деталей реалізації проекту.

*UML* (*Unified Modeling Language*) Розробка UML розпочалася в 1990-х роках і розвинута новими версіями й до сьогодні. [25, гл. 1.4] Ця мова є всеосяжною структурою для програмного моделювання, що містить елементи моделювання структур даних. UML є широко прийнятою серед IT-розробників. Спеціалістам із ПрО (UoD) ця мова не є зрозумілою. Крім того, UML містить абстрактні поняття об'єктно-орієнтованого програмування. Таким чином, моделі даних створені за допомогою UML є скоріше логічними, ніж концептуальними.

*Information Analysis ma* Information Analysis (+) – ця мова концептуального моделювання запропанована в 1980 та вдосконалена в 1986 році Lore Kern-Bausch and Bernd Wenzel.[26], базується на оригінальному підході ER Чена і залишається на рівні моделей, орієнтованих на дані. В якості вдосконалення було додано поняття семантичного моделювання із NIAM або

ORM відповідно, які згадувалися раніше. [27] Ще один огляд з використанням Information Analysis (+) був проведений в 1997 році [28].

На рисунку 1.10 наведено приклад моделі побудованої в Information Analysis (+). У цьому прикладі «Людина» створює фактор, що зв'язує, оскільки всі речення містять об'єкт такого типу.

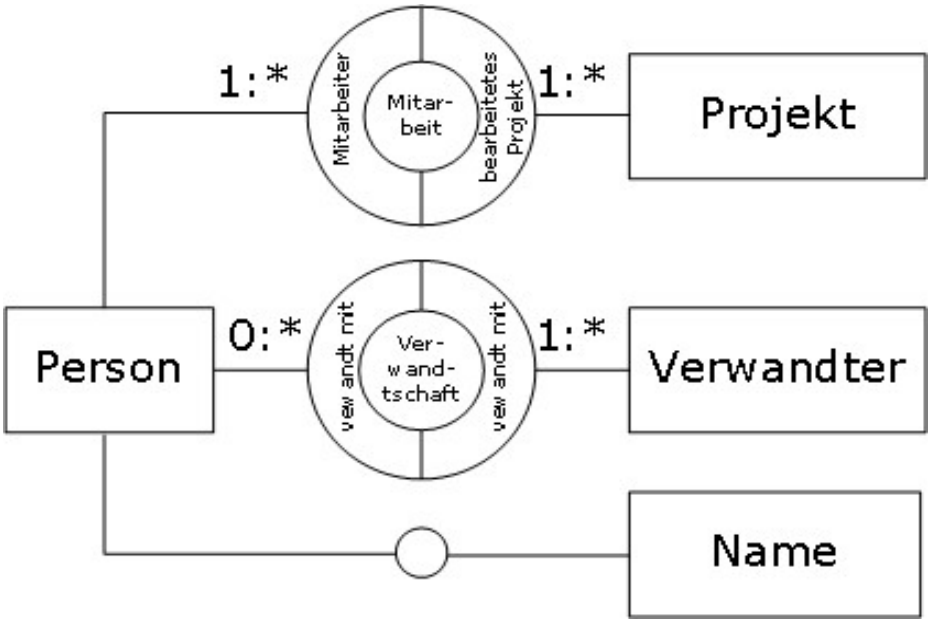



Рисунок 1.10 - Діаграма Information Analysis (+) із нотацією «коло». ([26])

Розбиття складних речень на прості не пов'язані речення полегшує завдання їх зображення в графічній моделі. Ця модель є зрозумілою для спеціалістів ПрО оскільки твердження, представлені графічних зображеннях, можна перекласти на природну мову однаково легко. Приклад на рисунку 1.10 представляє наступні семантично пов'язані прості речення:

- Людину *ідентифікують* за іменем.
- Людина *може* мати родичів.
- Родич належить *хоча б* одній людині.
- Людина працює *принаймні* над одним проектом.
- У проекті *є принаймні* один учасник.

Хоча Information Analysis (+) базується на оригінальному підході ER, рисунок 1.10 показує, що позначення відрізняються від звичних позначень Чена. Це свого роду недолік цього підходу, оскільки ще одне позначення моделювання повинно бути введене та вивчене практиками.

*AGILA MOD* (***A***utomatic ***G***enerators for ***I***nformation representation, ***L***ogical DB structures and ***A***pplications – AGILA ***Mod***eling language – MOD) – це автоматичний генератор для інформаційного представлення, логічних структур БД та застосувань. Можна вважати цю мову еволюцією мови Information Analysis (+). Вона була вдосконалена Сабіною Мюлленбах. Отже, AGILA MOD також базується на семантично пов'язаних простих речення, таких як Information Analysis (+), і цим можна визначити семантичний підхід до моделювання даних. AGILA MOD викладається в університеті прикладних наук м. Аугсбург з 2003 року. Основні положення опубліковано у [2].

В AGILA MOD основні синтаксичні примітиви були взяті з EER [10] замість того, щоб створювати новий синтаксис, як це має місце для Information Analysis (+). Водночас AGILA MOD також має всі переваги концепції семантично пов'язаних простих речень. Введений синтаксис щодо узагальнення був замінений на більш зручний запис, який походить від EXPRESS і може бути легко зрозумілий для людей, які знайомі з UML. Крім того, конструкції множин та перерахування, які існують у EXPRESS, були додані для посилення виразності - особливо коли мова йде про більш точне визначення особливостей певного виду об'єктів. Рисунок 1.11 показує приклад моделі, написаної в AGILA MOD, яка демонструє платформу для закупівель.

Перевага цього підходу полягає в тому, що він є ER підходом, але з філософією семантично пов'язаних простих речень із Information Analysis (+) (Див рис.1.10).

Таким чином, мова концептуального моделювання AGILA MOD об'єднує переваги семантичного та ER моделювання, може по-справжньому об'єднати IT-розробників БД та експертів ПрО, будучи загальною платформою комунікації та ідеальною базою документації для всіх учасників проекту.

*Висновок, щодо розглянутих мов концептуального моделювання*. Проведений аналіз існуючих основних мов концептуального моделювання дозволяє зробити основний висновок про те що методи та мови моделювання не зберегли темпу розвитку технологій баз даних. В даний час бракує декількох

елементів, щоб повністю скористатися перевагами концепцій автоматичних генераторів для інформаційного представлення, логічних структур БД та програм (AGILA) щодо сучасних розробок у технології проектування баз даних

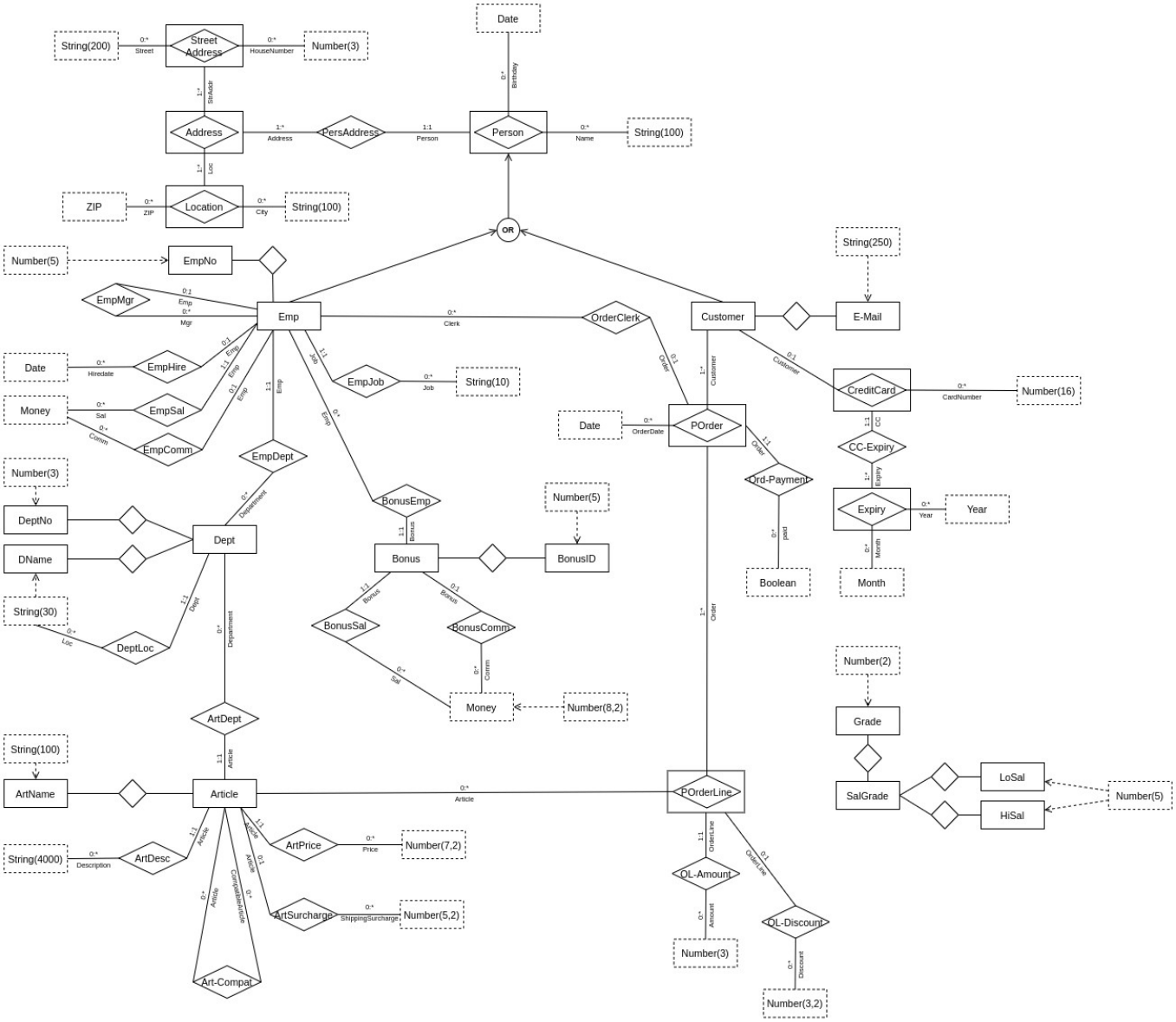



Рисунок 1.11 - Приклад моделі AGILA MOD

## 1.2. Аналіз типів логічних моделей даних в базах даних.

В цьому підрозділі міститься оцінка типів логічних моделей даних в БД, які на сьогоднішній день є актуальними для практичного використання. Розгянуті традіційні і новітні технології логічні моделі даних. Враховуючі щодо

концептуального моделювання та автоматизованої генерації логічної моделі даних із врахування типу персистентності фокус оцінки типів логічних моделей даних в БД розподілено:

*Ааспектами логічної структури,* а саме способу логічної організації даних. При цьому розглядають, як розділені блоки даних та якими є основні логічні елементи даних. Логічне представлення даних діє як фасад для розробників програмних застосувань і передбачає необхідне відображення між персистентністю та структурами в пам'яті. Тут у наш час відбувається так звана «об’єктно-реляційна неузгодженість імпедансу (object-relational impedance mismatch)» [30] із традиційними СУБД. Однак термін «проблема пам'яті / реляційного відображення (in-memory/relational mapping problem)», введений Мартіном Фаулером, може вважатися більш доречним. [31] Але сучасні In-Memory Data Grid (IMDG) здатні забезпечити надвисоку доступність даних за допомогою зберігання їх в оперативній пам'яті в розподіленому стані. Сучасні IMDG здатні задовольнити більшість вимог до обробки великих масивів даних..

*Технічними аспектами,* які не пов'язані з логічною структурою даних, а саме пояснюється, як зберігаються дані та як організовується їх зберігання для забезпечення довговічності та доступності. Дуже важливим моментом у цьому контексті є спосіб ведення операцій, що суперечать одна одній. [5].

### 1.2.1. Огляд та порівняння традиційних логічних моделей даних

З точки зору технічних вимог, всі традиційні типи логічних моделей даних пов'язані з ідеєю узгодженості даних. Це було оформлено згідно з принципами ACID. [11, гл. 20.3, с. 787] Це стосувалося внутрішньої узгодженості, тобто весь набір даних узгоджується згідно з правилами, виведеними з моделювання ПрО, як, наприклад, унікальність властивостей або посилальна цілісність. Це стосувалося зовнішньої узгодженості, що означає, що всі відтворені данні повинні мати однакові значення в кінці транзакції. Традиційні бази даних розроблялися в час мейнфреймів та централізованих

структур. Підвищення продуктивності обробки даних потребувало вдосконалення апаратного забезпечення центру обробки даних – так званого вертикального масштабування – тоді як реплікація на той момент не була основною метою. Але така обробка транзакцій є громіздкою та неефективною тому, що зміни реплікаційних даних змушують усі вузли бази даних взаємодіяти між собою для перевірки загальної узгодженості. Однак за цих обставин ACID дозволив полегшити погляд на збережені дані, і розробники могли зосередитись на бізнес-логіці без необхідності дбати про суперечливі дані. Розглянемо більш детально класичні типи логічних моделей даних в БД в порядку їх виникнення у часі та покажемо деяку їхню спадкоємність.

*Ієрархічна база даних* (Hierarchical Database) – найбільш оригінальний підхід до збереження даних у структурованому вигляді. Виділяють структурні елементи ієрархічних баз даних:

*Типи записів* визначають фіксовану структуру як групу полів або елементів даних, що представляють дані, що належать до загальної теми, наприклад «Персона».

*Типи відносин "Parent-child relationship"* (PCR) визначають взаємозв'язок 1: N між двома типами записів. Кожен тип запису може бути пов'язаний лише з одним типом запису "Parent" та довільною кількістю дочірніх типів записів – «*Child*».

Складні структури даних, де елементи мають більше батьківських елементів або посилаються на інші елементи десь у дереві, повинні бути вирішені шляхом копіювання їх в інші деревоподібні структури, що неминуче призводить до неконтрольованих дублікатів. Обмеженою альтернативою є “Віртуальні відносини батьків і дітей” (Virtual Parent-child relationships), які дозволяють підпорядкованому елементу бути віртуальним вказівником на елемент іншого дерева ієрархії, що порушує сувору ієрархічну структуру даних. Подальші обмеження цілісності - як унікальні поля - не надаються і не залишаються на боці бізнес- логики.

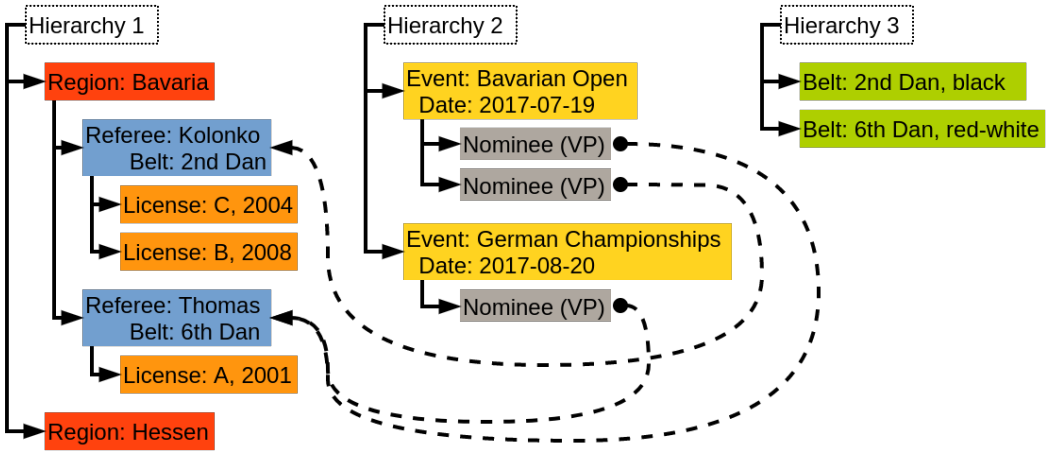



Рисунок 1.12 - Приклад, що представляє базу даних суддів дзюдо, які беруть участь у заходах (“VP” = “віртуальний покажчик”)

Доступ до цих баз даних забезпечує мова низького рівня, вбудована в хост-мову, де повинні бути створені так звані робочі області користувачів (User Work Areas, UWA), що відображають структуру бази даних. Дана концепція певною мірою відображена в сучасних структурах об'єктно-реляційного відображення (ORM), де класи сутності генеруються для відображення структури бази даних. Команди дозволяють отримувати доступ лише до одного запису одночасно. Складні маніпуляції повинні виконуватися хост-мовою, використовуючи ці основні команди. Отримати дані можна лише шляхом обходу ієрархії з кореневого запису. Отже, доступ до даних дуже складний і схильний до помилок.

*Мережеві бази даних* (Network Database). При організації даних в мережевих БД було усунуто багато обмежень, що містились в ієрархічних БД.. Структурними елементами мережевих баз даних є (рис. 1.13):

*Типи записів* мають таке ж саме значення, як і в ієрархічних базах даних, але в мережевих БД є додаткові функції для більш складного визначення даних: вектор може містити кілька значень одного типу всередині поля, група повторень - «вкладена таблиця», віртуальний або похідний елемент даних обчислюється за значеннями інших полів у записі.

*Типи наборів* визначають відносини між типом запису власника та типом запису члена. Кожен екземпляр типу набору містить один екземпляр типу

власника та довільну кількість екземплярів типу члена, тоді як кожен екземпляр типу елемента може мати місце лише один раз для типу набору. Це обмеження допускає взаємозв'язок лише 1: N. Типи наборів визначають упорядкування даних. Отже, дані не можуть бути впорядковані базою даних довільно, але тип набору для кожного бажаного впорядкування повинен бути доданим, значно збільшуючи складність.

Мережеві бази даних вже пропонують деякі можливості для визначення обмежень на дані. На рівні типу запису можливі унікальні комбінації атрибутів та перевірка значень. Для наборів типів можна визначити обмеження, вказуючи, чи є приналежність до цього типу набору обов’язковим для екземпляра запису члена, і чи це членство є незмінним.

Мова мережевих баз даних подібна до ієрархічних. Отримання даних є незручним. Мережа передачі даних повинна проходити з визначеної вихідної точки. Мережеві бази даних забезпечили багато вдосконалень та більшу гнучкість для ефективного зберігання даних. Але все-таки складне використання ускладнювало їх використання та обслуговування.

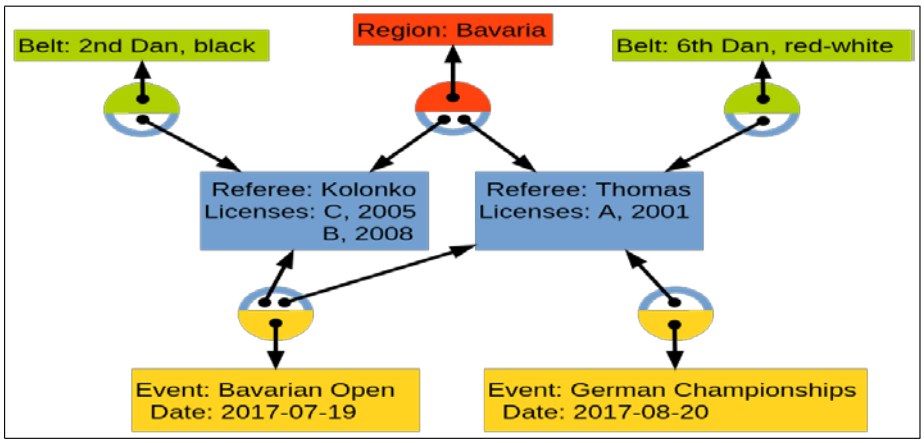



Рисунок 1.13 - Приклад арбітра дзюдо представлений у вигляді мережевої бази даних

*Реляційна база даних* (*Relational Database*) основана на теорії множин та реляційній алгебрі, що дозволило запропонувало набагато простіший спосіб зберігання даних. Структурними елементами реляційної баз даних є (рис. 1.14):

*Таблиці* визначаються як набір атрибутів, що описують пов'язані дані. Атрибути мають ім'я та тип даних. Збереження запису означає додавання рядка, який суворо відповідає визначеній структурі.

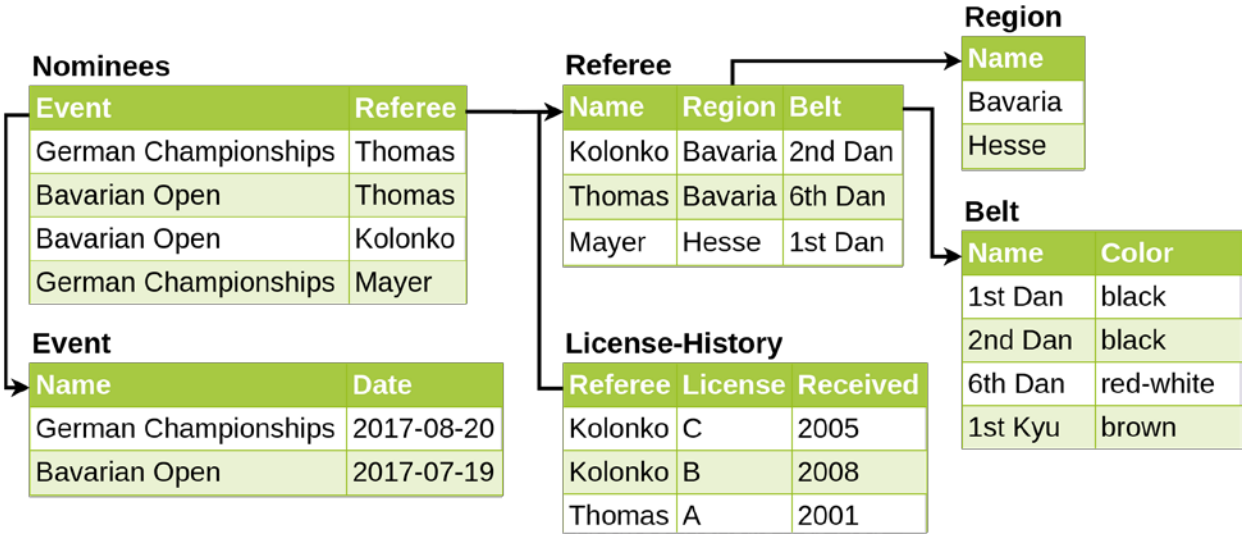



Рисунок 1.14 - Приклад арбітра дзюдо представлений у вигляді реляційної бази даних

*Зв'язки між таблицями* представлені обмеженнями зовнішнього ключа. Ці обмеження засновані на посиланні за значенням, тобто зовнішній ключ ділиться доменом із вказаним ключем. У цьому контексті досить часто ігнорується, що зовнішній ключ може посилатися на будь-який унікальний ключ - не тільки на первинний ключ. Покажчик, як у попередніх підходах, не використовується. Перевагою є те, що вміст атрибутів, які посилаються безпосередньо показує, на що саме посилається, без необхідності шукати вказаний запис.

Тим не менше, із зовнішніми ключами залишається обмеження, що представляє лише відносини 1: N. Для взаємозв’язків M: N необхідна таблиця перетину між таблицями, які пов’язані між собою. Реляційні бази даних забезпечують перевірку більш детальних та складних правил щодо узгодженості даних за допомогою обмежень перевірки та тригерів.

Мова реляційних баз даних набагато складніша, ніж мови попередніх підходів. Вона був визначена як стандарт SQL і може не має бути вбудованою в мову хосту. На відміну від попередників, SQL здатний обробляти кілька записів

одночасно за допомогою одного оператора. Для цього не потрібно залучати мову хосту. Вибір даних також набагато простіший. Застосовуються правила реляційної алгебри, і всі бажані дані повертаються відразу як блок даних.

Тим не менше, порівняно з мережевими базами даних, класичні реляційні бази даних також мають деякі обмеження щодо моделювання даних:

Вектори та групи, що повторюються, не існують завдяки реляційній моделі. Натомість потрібно створити окрему таблицю із зовнішнім ключем до вихідної таблиці.

Віртуальні похідні стовпці не передбачені в реляційній моделі. Однак постачальники баз даних почали додавати цю функцію до своїх продуктів реляційних баз даних. Наприклад, Oracle надає віртуальні стовпці.

*Об'єктно-орієнтована база даних* (Object-Oriented Database) основана принципах об’єктно-орієнтованого програмування до системи зберігання – як, наприклад, класи, екземпляри, успадкування, поліморфізм тощо. Вони є природним аналогом сховища для об’єктно-орієнтованих мов програмування, оскільки обидва дотримуються однієї парадигми. Ідея об’єктно-орієнтованих баз даних досить проста: замінити об'єкти із середовища виконання на систему баз даних, об'єкти не повинні завантажувати пам'ять більше, ніж це потрібно, зберігати їх протягом циклу, підтримувати загальний стан об’єктів для різних систем, що мають доступ до цієї системи баз даних.

В об'єктно-орієнтованих БД кожен об'єкт ідентифікується за допомогою незмінної системної ідентифікації об'єкта (object identifier – OID). Два об'єкти, що мають однаковий стан, не є однаковими. Щоб зменшити величезну кількість об’єктів, об’єктно-орієнтовані БД розрізняють мову програмування Java між реальними класами з екземплярами та власними простими значеннями. Об'єктно-орієнтовані БД вимагають як структури, так і поведінки збережених класів. Вони надають власну мову для визначення класу. Відповідно до принципів інкапсуляції та приховування інформації, визначеною структурою не можна маніпулювати безпосередньо ззовні. Тому мова обробки даних застаріла і є частиною поведінки класів, де розробник також повинен

впроваджувати структурні обмеження. Наперед визначені обмеження не існують. Унікальні обмеження стають досить важкими для реалізації, оскільки об’єкти не знають про інші об’єкти того ж типу, а лише про інші об’єкти, з якими вони пов’язані. Взаємодія для зміни об’єктів у базі даних реалізується у спілкуванні у стилі повідомлення. Повідомлення надсилаються, і відповідний об’єкт реагує на нього, використовуючи бажаний метод.

Відносини між об'єктами представлені як атрибути класу типу, на який посилається, що містить посилання на конкретний екземпляр. Отже, як в ієрархічних або мережевих базах даних, об’єкт, на який посилаються, завжди повинен бути завантажений для перевірки даних. Посилання саме по собі не містить жодної інформації. Відносини можуть бути визначені одно- або двонаправленими, щоб отримати доступ від одного примірника до іншого. А відносин 1: N, наприклад представлений простим атрибутом з одного боку та атрибутом списку з іншого боку. Відносини M: N складають списки з обох сторін.

До недоліків відноситься брак унікального визначення об’єктів та класів.

Не можна ігнорувати і таке запитання: Чи повинна бізнес-логіка знаходитись у базі даних, і якщо так, то в якій мірі? Коли поведінка визначається для класів у базі даних, межі між сховищем даних та логікою програми починають розмиватися. Досить сумнівно, чи слід дійсно націлювати на змішування аспектів персистентності та обробки. Це суперечило б загальновизнаному принципу програмної інженерії «розділення проблем».

*Об’єктно-реляційна база даних* (Object-Relational Database) При розробці об’єктно-орієнтованих БД, постачальники СУБД інтегрували об’єктно-орієнтовані принципи у свої продукти, створюючи гібридну систему БД, яка додатково до реляційних функцій дозволяла б, наприклад, додати сукупні поля, вкладені таблиці та систему типів, яка може визначати як тип, так і поведінку. Наприклад, Oracle включив увімкнені вкладені таблиці до XMLDB. Також надаються нові типи зображень, які можуть зберігати дані зображень. Однак об’єктні БД не є дуже популярними.

*Порівняння типів традиційних логічних моделей даних.*

У [10] показано, що основні структурні елементи всіх типів логічних моделей даних можуть бути покладені в матрицю відповідності із елементами (E)ER мови концептуального моделювання. В таблиці 1.2 автором викладене таке порівняння термінів щодо розглянутих типів традиційних логічних моделей із термінами концептуального моделювання, яке адаптоване до підходу запропонованого в [10, таблиця С.1, с. 811]

На завершенні підрозділу необхідно зробити висновок, що із розглянутих типів логічних моделей даних сьогодні на практиці використовуються тільки реляційні. РБД залишились єдиними, що змогли встояти. Всі інші БД не використовуються через їх недоліки. Ієрархічні бази даних не гнучко представляють інші структури, крім ієрархічних. Мережеві бази даних є більш гнучкими, але обмеження щодо подання та обробки даних ускладнили роботу зі складними структурами даних. Порівняно з вимогами колишніх часів, об'єктно-орієнтовані бази даних виявились занадто складними та не відповідають потребам системних архітекторів.

Реляційні бази даних пропонували гнучкий нестандартний спосіб представлення даних, які можна легко зрозуміти, що також полегшує адміністрування. З іншого боку, розширені можливості перевірки цілісності даних спростили завдання розробників додатків, оскільки вони могли покладатися на ці перевірки цілісності. Можливості реляційної алгебри рекомбінувати дані за допомогою операцій відбору, проекції та об'єднання, які покладено в SQL, при пошуку давали дуже потужну можливість показу збережених даних у будь-якій формі. Головною проблемою сьогодні є ефективне відображення між реляційними структурами та структурами в пам'яті. Це занадто часто призводить до посилань на підтаблиці для багатозначних або структурованих атрибутів або успадкованих сутностей, що створюють проблеми з продуктивністю через надмірне використання операцій об'єднання. Спрощенням проблеми було б використання об’єктно-реляційних

можливостей, таких як вкладені таблиці, що в основному сягає корінням концепцій мережевих баз даних, тобто векторів та повторюваних груп.

Таблиця 1.2 - Порівняння термінології традиційних баз даних з концептуальною ER-номенклатурою [10].

| Назва терміну на концептуальному рівні | Назва терміну на логічному рівні для типів традиційних моделей БД | | | |
|---|---|---|---|---|
| (E)ER *Conceptual* | Ієрархічні<br><br>Hierarchical | Мережеві<br><br>Network | Реляційні<br><br>Relational | Об’єктно-орієнтовані<br>Object-oriented |
| *Сутність* | Record type<br>Тип запису | Record type<br>Тип запису | Table<br>Таблиця | Class definition<br>Визначення класу |
| *Екземпляр сутності* | Record occurrence<br>Наявність запису | Record occurrence<br>Наявність запису | Row<br>Рядок | Object<br>Об'єкт |
| *Атрибут* | Поле або елемент даних | Поле або елемент даних | Стовпець | Атрибут |
| *Ідентифікатор* | Sequence Field<br><br>Послідовність<br>Поле | Ключ або унікальна комбінація полів | Первинний ключ | OID |
| *Унікальне обмеження* | (n/a)<br>(не застосовується) | унікальна комбінація полів | унікальний ключ | (за умови реалізації поведінки) |
| *Взаємозв`язки* | PCR | тип набору | зовнішній ключ | посилання на об’єкт |
| *Багатозначні Атрибути* | (n/a)<br>(не застосовується) | Вектор | (n/a)<br>(не застосовується) | List-/Set-Constructor<br>Список- / Набір-Конструктор |
| *Слабкі сутності* | (n/a)<br>(не застосовується) | повторювані групи | (n/a)<br>(не застосовується) | Tuple-Constructor<br>Корпусний конструктор |

### 1.2.2. Огляд та порівняння новітніх логічних моделей даних

З розвитком популярності веб-технологій, які вимагають необмеженої доступності до даних при розподіленої їх обробці на багатьох вузлах проблеми практичного використання реляційних БД ще зросли та загострились. Ці проблеми пов’язані із неможливість опрацювання великих даних та • паралельної обробки, з обмеження на застосування у проектах з великим навантаженням. Вони виникають із методологічного спрямування РБД на обробку транзакцій. Транзакція визначена як набір дій з даними, об'єднаний в логічну одиницю, яка виконується повністю або відкочується назад, для її реалізації використовується реляційна модель даних, яка задовольняє вимогам ACID [10, глава 17.3, с. 537]:

– нерозривність, атомарность (Atomicity) гарантує виконання всіх підопераций транзакції;

– узгодженість (correctness) стану системи забезпечується до початку транзакції і після її завершення;

– ізольованість (isolation) гарантує для безлічі транзакцій, що працюють паралельно, що результати будь-яких підопераций транзакції будуть приховані від всіх інших до її повного завершення;

– стійкість, довговічність (durability) гарантує фіксацію в базі даних на постійній основі результатів виконання транзакції навіть в разі аварійної зупинки.

В якості шляхів вирішення проблем РБД запропоновані нові технології персистентності – open source рішення NoSQL – нереляційні і розподілені бази даних з підтримкою гнучкої схеми бази даних і горизонтальної масштабованість, в яких в якості мови запитів не використовується SQL. Неможливість підтримки властивостей транзакцій ACID в якійсь мірі компенсована евристичним принципом, відомим як теорема CAP Згідно CAP, все розподілені СУБД забезпечують тільки два з трьох властивостей:

узгодженість (consistency) даних, доступність (availability) і стійкість до поділу (partition tolerance) (рис. 15) [3, 32, Глава 2.2.3, с. 33]:

– Consistency - наявності тільки однієї копії даних, яка відповідає останій за часом операції оновлення;

– Availability - доступність даних при наявності операцій оновлення;

– Partition tolerance - здатність вузлів, між якими немає зв'язку, продовжувати працювати незалежно один від одного.

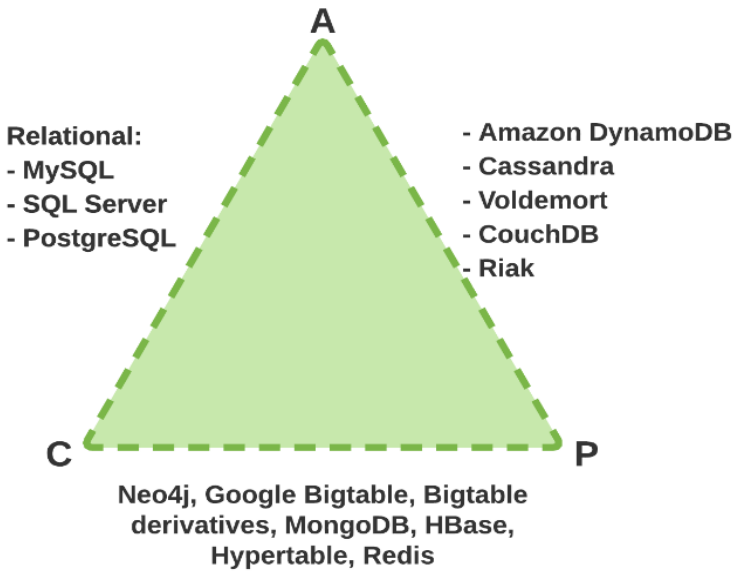



Рисунок 1.15 Ілюстрація взаємозв'язку CAP і популярних СУБД

Нестроге доведення теореми CAP засноване на простих міркуваннях [3,32]. Нехай розподілена система складається з N серверів, кожен з яких обробляє запити деякого числа клієнтських додатків. Під час опрацювання запиту сервер повинен гарантувати актуальність інформації, для чого потрібно попередньо синхронізувати вміст його власної бази з іншими серверами. Отже, серверу необхідно чекати повної синхронізації або генерувати відповідь з урахуванням несинхронізованих даних. Можливий і третій варіант, коли з яких-небудь причин синхронізація здійснюється тільки з частиною серверів системи. У першому випадку виявляється не виконаною вимога стосовно *доступності*, у другому – *узгодженості*, у третьому – *стійкості до поділу*.

В теперішній час виділено 4 типа новітніх NoSQL логічних моделей даних, які розглянемо більш детально.

*Колоночні бази даних* (Column-Family Databases), такі як Cassandra [33. 34], дуже схожі на РБД, складаються з рядків, які мають атрибути. Згідно з реляційною логічною моделлю рядки групуються в таблицях, а Column-family databases групують рядки у так звані Column-family databases. Column-family databases є менш обмежуючими і дозволяють рядкам мати підмножину можливих атрибутів, але це не така велика різниця, оскільки реляційні бази даних використовують значення NULL, щоб опустити певні атрибути. Отже, відмінності в логічних моделях низькі. Справжня відмінність полягає в додаткових обмеженнях для моделі, які передбачаються BASE, та способі реплікації даних Cassandra. Спосіб визначення первинного ключа в мові запитів Cassandra (CQL) є показником. Первинний ключ поділяється на ключ розділу та стовпці кластеризації, що визначають розподіл таблиці по вузлах. Отже, є відмінності, якщо первинний ключ визначено «(a, b, c)» або «((a, b), c)». У першому прикладі лише «a» є ключем розділу, тоді як у другому «a» та «b» будують разом ключем розділу. Подібно до мережевих баз даних, дані не можна отримати у довільному порядку. Від визначених структур даних та індексів залежить, чи може CQL приймати певні впорядкування чи ні.

Інші приклади цього типу, такі як Apache HBase або HyperTable (див. [35], тобто [36]), взагалі не мають визначення даних, що дозволяє мати будь-який атрибут у кожному рядку, який ідентифікується за допомогою певного ідентифікатора рядка. Це схоже на спосіб, яким документні БД ідентифікують свої документи. З зору логічного моделювання, це все ще дані, які знаходяться як атрибути в рядках і мають однакову базову структуру, як РБД.

*Документні (документо-орієнтовані) бази даних* (Document Databases), основною одиницею логічної моделі даних є документ тобто структурований текст з певним синтаксисом та технічний ідентифікатор, наприклад універсальний унікальний ідентифікатор (UUID). Прикладами Document Databases є CouchDB і MongoDB [37, 38], які зберігають документи в форматі JSON (Див. рис. 1.16). Іноді більш детальний XML є альтернативою. Оскільки не існує необхідного типу документа, мови визначення даних не існує. Будь-

який документ може мати будь-яку структуру, проте доцільно використовувати загальний тип документа для проекту, щоб мати можливість запитувати та обробляти дані у загальній формі. Зазвичай модель програмування MapReduce [32] використовується для вилучення даних з документів, який призначений для великих обсягів даних і дозволяє отримувати дані у різних формах та як один блок даних, аналогічно SQL.

З іншого боку логічна модель даних в Document Databases схожа на логічну модель даних в об’єктно-орієнтованих базах даних. Це підтверджується використанням JSON в Document Databases та для серіалізації об'єктів в об'єктно-орієнтованих мовах програмування, наприклад коли об’єкти (де-) серіалізовані через WebService для обміну даними. З об'єктно-орієнтованої точки зору Document database є сховищем серіалізованих об'єктів. Навіть посилання можливі із використанням OID об'єкта, відповідно до ID документа, на який він посилається. [38] Єдина реальна різниця між Document Databases і об'єктно-орієнтованої базою даних полягає в тому, що бази даних документів тільки зберігають стан, але не поведінку, забезпечуючи спільну мову для отримання даних. Бази даних документів в цьому сенсі є значним покращенням у порівнянні з класичними об'єктно-орієнтованими базами даних, оскільки вони усувають багато з проблем, згаданих раніше.

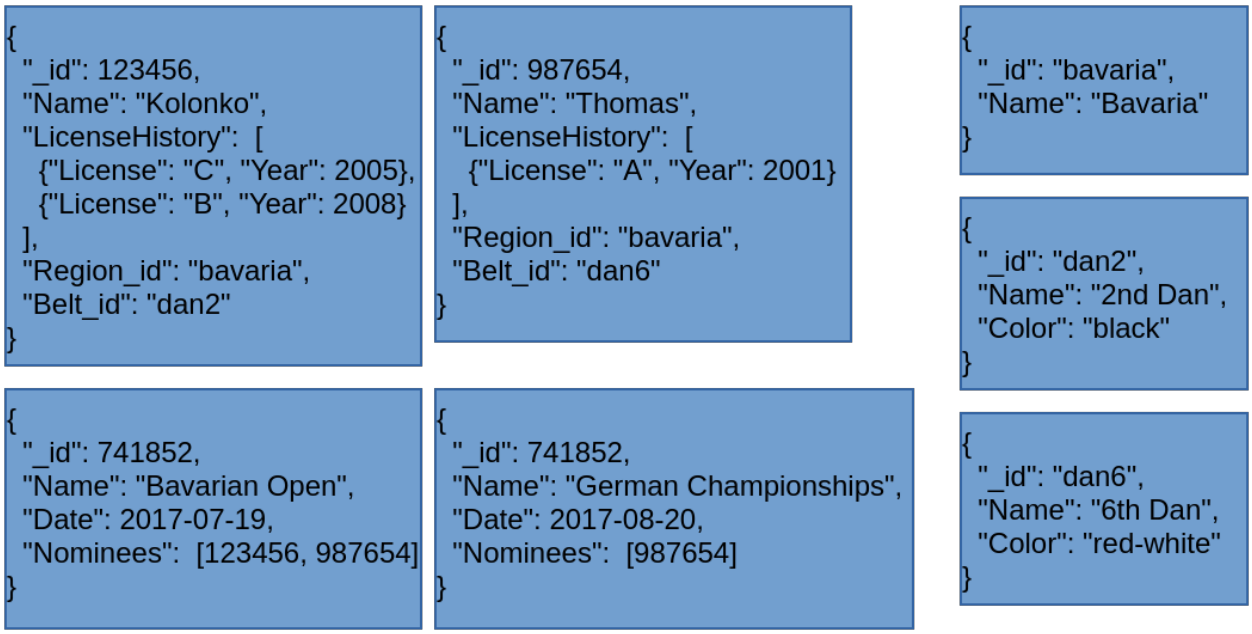



Рисунок 1.16 - Приклад судді дзюдо як об’єкта Document database

*Графові (графо-орієнтовані) бази даних* (Graph-Based Databases) логічні моделі даних графічних БД, таких як Neo4J [39] спрямовані на відображення відносин між об'єктами, а не на зберіганні великих обсягів структурованих даних. Структурними елементами графової моделі є (рис. 1.17):

*Вузли* – представляють об'єкти, які перебувають у взаємозв'язку з іншими об'єктами. Вони мають довільні атрибути. База даних не надає різних типів вузлів, залишаючи це бізнес-логіці.

*Стрілки* – представляють взаємозв'язки (посилання) між вузлами.

Вузли та стрілки можуть мати певну кількість атрибутів.

Таким чином, графові БД відображують мережу об’єктів, які довільно зв’язані. Таким чином вочевидь коріння графові БД сягає до традиційних мережевих БД. Обидві логічні моделі даних складаються із об'єкти і відносин. Графові БД повністю зосереджені на цих взаємозв’язках. Структури неможливо визначити для вузлів, як це можливо через типи записів для мережевих БД. Натомість самі ребра можуть містити атрибути. Це великий крок вперед у порівнянні з мережевими БД, де інформація, яка стосувалась саме цього відношення, повинна була зберігатися якимось чином в однієї із сутностей - учасників. Отже, графові БД можна як розвиток мережевих БД.

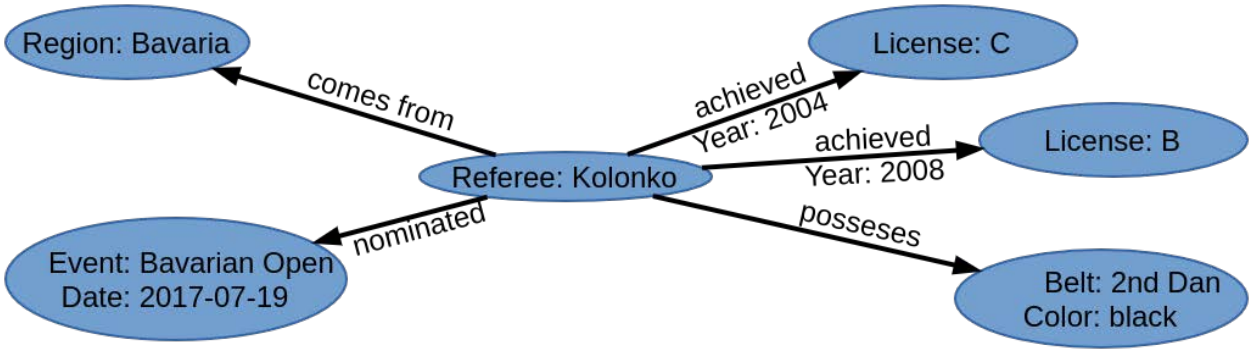



Рисунок 1.17 - Приклад судді представлений у вигляді графової БД

*«Ключ-Значення» бази даних* (Key-Value Stores) – найбільш неструктурований тип БД. Ці системи зберігання беруть лише пари унікального ключа та відповідного значення будь-якого типу, щоб зберігати їх на

постійному носії, наприклад, на жорсткому диску або в пам'яті. Саме сховище не володіє інформацією про характеристики даних. Основна увага приділяється ефективному збереженню та отриманню великої кількості неструктурованих даних, які є рівномірно доступними. Фактично, це дуже великі хеш-таблиці, де кожному ключу поставлене у відповідність значення.

Насправді, моделі «Ключ-Значення» не мають аналогів в типах традиційних моделей

БД «Ключ-Значення» можна сприймати як екстерналізацію засобів, які зазвичай є доступними за допомогою мови програмування. Наприклад, у Java вони відомі як карти, в PHP асоціативні масиви беруть на себе цю функцію, а Python має для цього словники. БД «Ключ-Значення» дозволяють розробникам ефективно зберігати велику кількість даних унікальним способом. Вони вирішують фізичні проблеми такі як кешування, зберігання та реплікація. Внутрішні засоби, що надаються мовами програмування, ніколи не могли ефективно справлятися з такими обсягами даних через обмежену пам'ять.

*Порівняння типів новітніх логічних моделей даних*

Як було показано раніше, нові логічні моделі даних мають досить багато аналогій традиційним моделями. Отже в таблиці 1.3 у відповідності до таблиці 1.2 автором також викладене порівняння термінів щодо розглянутих типів новітніх логічних моделей із термінами (E)ER концептуального моделювання

### 1.2.3. Модель багатоваріантної персистентності та мультімодельні СУБД

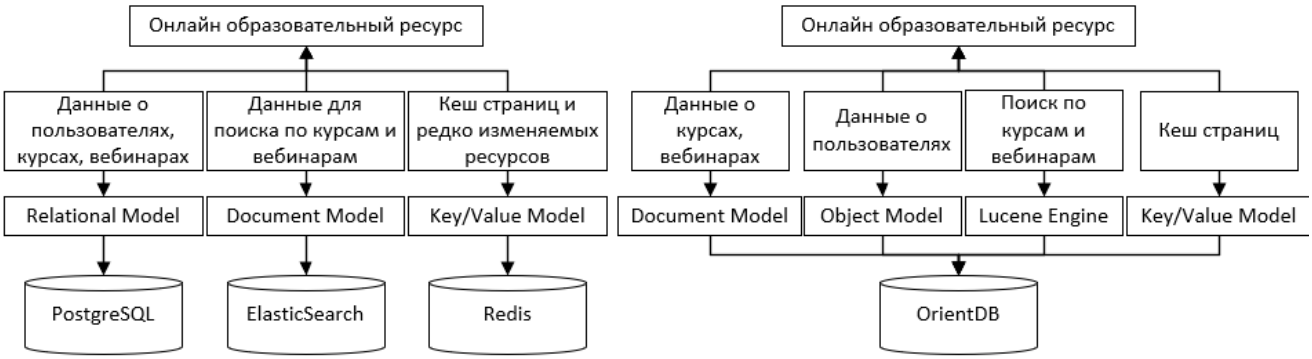
Проблема неефективного зберігання гетерогенних даних БД, побудованої на одній моделі залишалася невирішеною. Таким чином, виникла концепція *багатоваріантної пересистентності* (Polyglot Persistence), яка дозволяє використання декількох баз даних з різними логічними моделями на стороні сервера. Використання Polyglot Persistence вирішує проблему з обробкою гетерогенних даних, а застосування побудовані на його основі відрізняються високою продуктивністю та низькім часом реакції на запит (Див. рис. 1.18,а).

Але разом з цим модель Polyglot Persistence має і недоліки: немає гарантії, побудоване застосування буде відмово стійким, громіздкість коду, необхідність інтеграції даних, деяке дублювання даних, кожній СУБД вимагається різний час для синхронізації даних. Особливо значення ці недоліки мають при реалізації застосувань середніх і малих розмірів.

Таблиця 1.3 - Порівняння термінології новітніх підходів до баз даних із концептуальною ER-номенклатурою

| Назва терміну на концептуальному рівні | Назва терміну на логічному рівні для типів новітніх моделей БД | | |
|---|---|---|---|
| (E)ER<br>*Conceptual* | Column-Family<br>Колоночні | Document<br>Документні | Graph-based<br>Графові |
| *Entity set*<br>*Набор сутностей* | Column family | Залежно від типу документа | (n/a)<br>(не застосовується) |
| *Entity instance*<br>*Екземпляр сутності* | Row<br>Рядок | Document<br>Документ | Node<br>Вузол |
| *Attribute*<br>*Атрибут* | Attribute<br>Атрибут | Attribute<br>Атрибут | Attribute<br>Атрибут |
| *Identifier*<br>*Ідентифікатор* | Primary key<br>Первинний ключ | Unique identifier<br>Унікальний ідентифікатор | Unique identifier<br>Унікальний ідентифікатор |
| *Unique constraint*<br>*Унікальне обмеження* | (n/a)<br>(не застосовується) | (n/a)<br>(не застосовується) | (n/a)<br>(не застосовується) |
| *Relationships*<br>*Взаємозв`язки* | (n/a)<br>(не застосовується) | Attribute with ID reference | Edges<br>ребра |
| *Multivalued Attributes*<br>*Багатозначні Атрибути* | Lists / Collections<br>Списки / Колекції | List attributes<br>Атрибути списку | List attributes<br>Атрибути списку |
| *“Weak entities”*<br>*“Слабкі сутності”* | Super columns / Tuples<br>Супер колонки / Кортежі | Structured attributes<br>Структуровані атрибути | Structured attributes<br>Структуровані атрибути |

У той же  саме модель багатоваріантної персистентності покладено в створення готових рішень у вигляді мультімодельних СУБД, які спрямовані вирішити вищезазначені проблеми. Виходячі із назви мультімодельних СУБД вони спрямовані на об'єднання різних логічних моделей даних в єдиний інтегрований движок, який використовує уніфікований мову запитів і надає єдиний API, який використовується у всіх логічних моделях даних. Основна концепція полягає в тому, щоб на фізичному рівні зберігати всі дані в одній СУБД, а потім для обробки представляти інші моделі шляхом зіставлення моделей більш високого рівня з поданням нижчого рівня (Див. рис. 1.18,б) [3]. Таким чином, розробляючи системи малого та середнього масштабу доцільно використовувати саме цей підхід. Прикладами таких СУБД є Oracle, MarkLogic, OrientDB, DataStax. На рисунку 1.18 а,б показана схема реалізації системі дистанційної освіти (СДО) з використанням Polyglot Persistence та мультімодельной СУБД. У список вирішуваних завдань ресурсу входило: можливості проходження курсів та перегляду вебінарів, які складаються з відео, а також контроль знань після кожного уроку і модуля по завершенню курсу. Кожен користувач реєструється для забезпечення доступу до ресурсу і даних про власний прогрес. Крім того, на ресурсі реалізований пошук по основній текстової інформації курсів, а також забезпечено збереження кешованих сторінок для забезпечення їх швидкого завантаження.




а б

Рисунок 1.18 – Схеми реализації системи дистанційної освіти на боці сервера (а – Polyglot Persistence; б – мультімодельной СУБД

## 1.3. Аналіз CASE-засобів для проектування баз даних

Розглянемо існуючі інструменти та технології, що дозволяють автоматизувати трьохетапний процес проектування БД

Microsoft Visio до версії 2010 побудований з використанням векторної графіки надає візуальну підтримку для створення реляційної моделі даних. Графічні примітиви оснащені додатковими діалоговими вікнами властивостей та логікою для надання всієї необхідної інформації для створення необхідних операторів SQL, які автоматично формують фізичну модель у базі даних. Microsoft Visio також може підключитися до існуючої бази даних і порівняти її структури з моделлю файлу Visio. Зміни можуть бути застосовані безпосередньо до підключеної бази даних. Створення окремого сценарію для зміни структур здається недоступним [46]. Це досить незручно для процесу розробки, коли зміни системи дозволяється вносити лише в продуктивне середовище після того, як вона була протестована в безпечному середовищі тестування. Застосування змін автоматично з файлу Visio може спричинити більше проблем під час запуску. Отже, Microsoft Visio потрібен замість простого клієнта бази даних, де системний інтегратор може просто запустити згенерований сценарій знову у виробничому середовищі. Крім того, додаткова перевірка перед запуском згенерованого результату також неможлива. Іншим недоліком цього підходу є той факт, що в таблиці, що зазнали змін, необхідно внести додаткові атрибути для процесу порівняння. Вони не пов'язані з реальною моделлю даних і можуть викликати плутанину у інших учасників проекту. В підсумку в Microsoft Visio підтримується лише реляційну модель даних на логічному рівні, яка прив’язується до власних реалізацій СУБД Microsoft, таких як напр. MS SQL Server або MS Access. На рисунку 1.19 показано таке графічне зображення з панеллю властивостей внизу. Однак підтримка цих фігур у Microsoft Visio була скасована в останніх версіях Microsoft Visio.

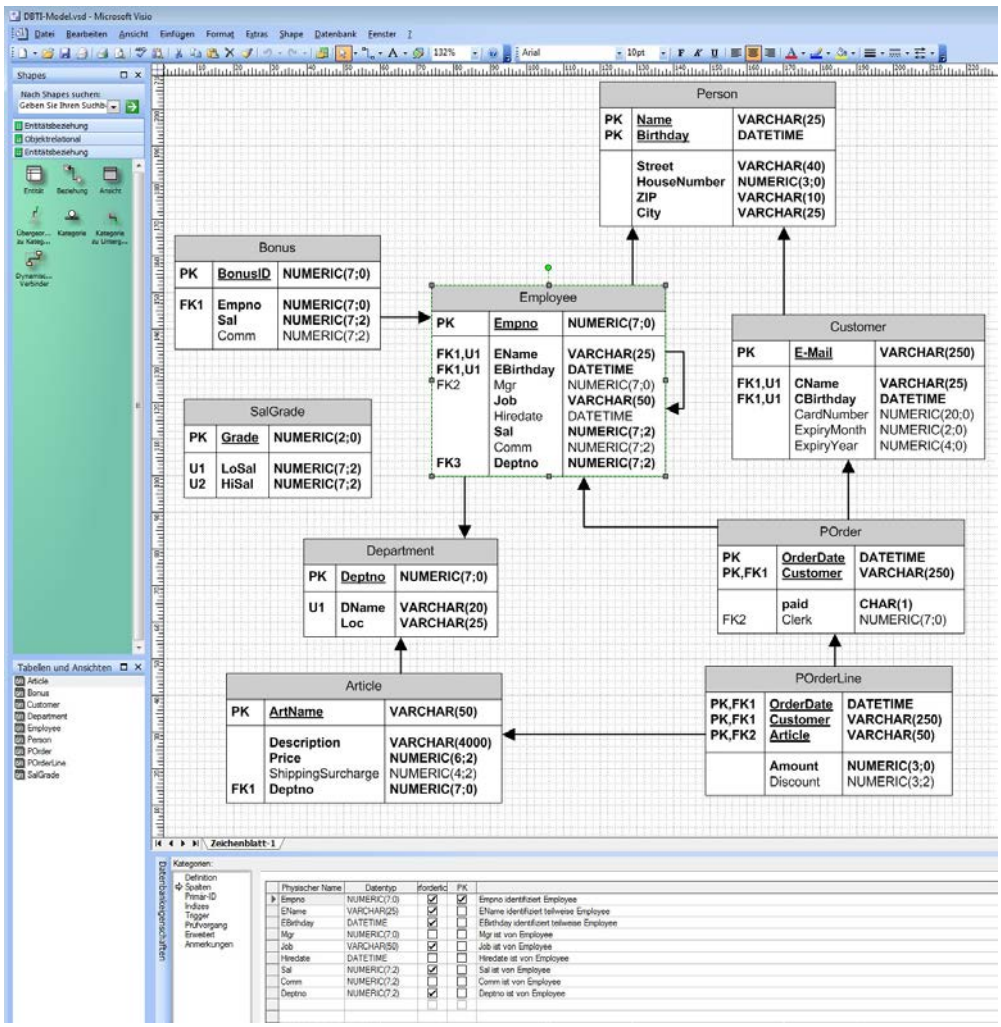
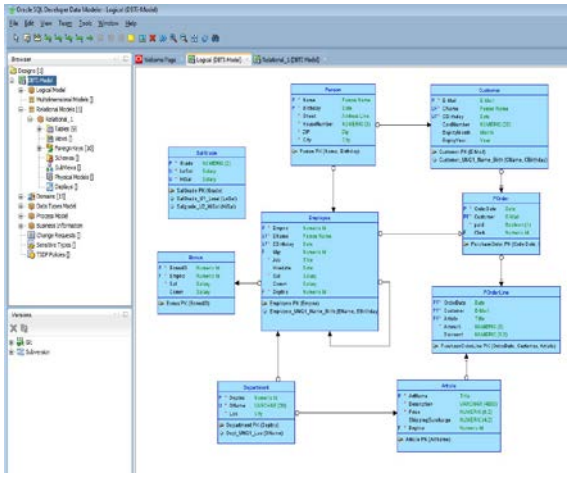



Рисунок 1.19 - Реляційна модель, створена за допомогою Microsoft Visio

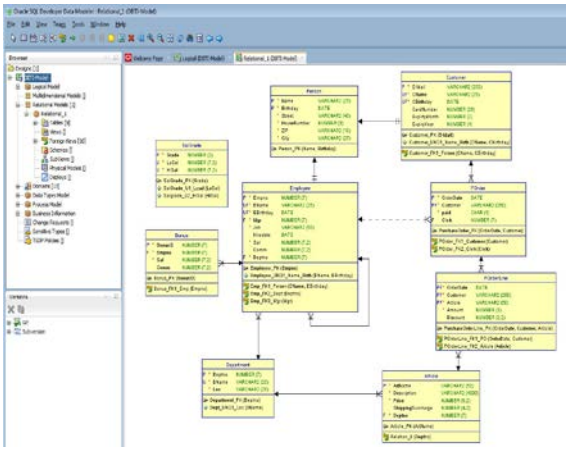
Сьогодні більш складною технологією для автоматизації проектування БД є Data Modeler від Oracle. Інструменти середовища Data Modeler Це дозволяють створювати графічні реляційні моделі даних, з усіма необхідними деталями, щоб сформувати сценарії створення мови визначення даних (DDL), який представляють фізичну модель даних.. Створення логічної моделі, виведення до реляційної моделі та генерація сценаріїв SQL для створення структур баз даних – це класичний підхід прямої інженерії, який Data Modeler використовує як термін у даному контексті. Він також підтримує зворотний інженерний процес шляхом підключення до існуючої бази даних та читання її структур. З прочитаної інформації створюється реляційна модель, і логічна модель може бути виведена користувачем.

Найголовніше, Data Modeler здатний створювати сценарії, що змінюють існуючу базу даних через змінену модель. Він пропонує можливість підключення до певної бази даних, виявляючи відмінності від даної моделі, а також порівнюючи дві версії моделі - надані через систему версій або вручну - в результаті чого скрипт змінює структури бази даних.

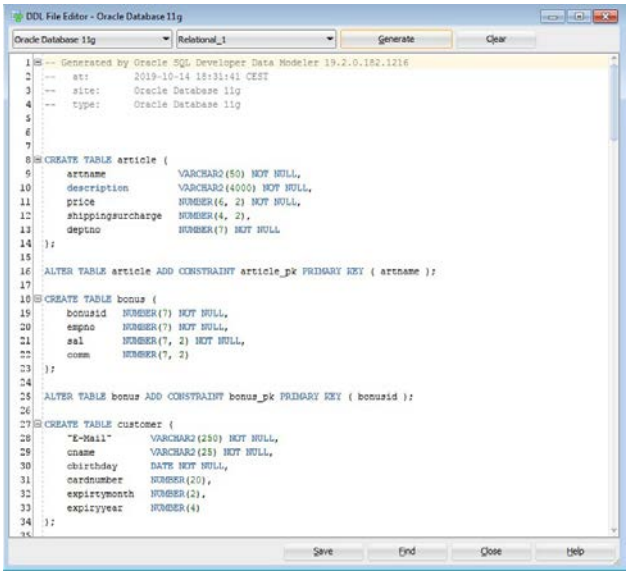
На рисунку 1.18 показано різні етапи прямого інженерного підходу від логічної моделі до сценарію DDL.



(а) Логічна модель



(б) Реляційна модель домени, замінені системними типами




(с) DDL-скрипт, сформований на основі реляційної моделі

Рисунок 1.20 - Різні етапи моделювання даних в Oracle Data Modeler

## 1.4. Висновки. Постановка задачі дослідження

У першому розділі дисертаційної роботи, проаналізовано існуючі мови концептуального моделювання, технології персистентності, методи для автоматизації процесу розробки баз зі багатоваріантною персистентністю та обгрунтовано мету та задачі дослідження

Показано, що і сьогодні процес проектування БД згідно класичної роботи Ramez Elmasri and Shamkant B. Navathe вимагає від IT-розробника вирішення низки задач при проведенні трьохетапного моделювання даних, а саме на концептуальному, логічному та фізичному рівні.

1. Проведено аналіз існуючих мов концептуального моделювання, а саме ER, ORM/NIAM, IDEF1X, EXPRESS, EER, UML, Information Analysis та AGILA MOD та показано що всі вони частково не відповідають вимогам, які висувають IT-розробники БД: є складними і не зрозумілими нетехнічним спеціалістам із ПрО для узгодження функціональних вимог, які висуваються до даних; не мають достатньої кількості засобів для автоматизованого відображення концептуальної схеми в логічну схему даних, щоб забезпечити мінімальне ручне втручання; орієнтовані на проектування тільки РБД і тому не використовуються при розробці NoSQL або Polyglot Persistence БД. Показано що саме недосконаліть існуючих мов концептуального моделювання призводить до підвищення витрат часу та трудомісткості проектування БД.

2. З точки зору можливостей відображення концептуальної схеми в логічну схему даних на логічному рівні проведено аналіз існуючих традиційних та новітніх логічних моделей даних. Показано спадкоємність структур з одного боку ієрархічних, мережевих та реляційних логічних моделей, а з іншого реляційних та колоночних; ієрархічних, мережевих та графо-орієнтованих; об’єктно-орієнтованих та документно-орієнтованих. Обгрунтовано переваги логічних моделей даних «ключ-значення» для створення пошукових БД.

3. , Показано, що використанням різних технологій зберігання даних – Polyglot Persistence в рамках одного застосування на основі декількох баз даних

з різними логічними моделями на стороні сервера має наступні переваги: хороша масштабованість застосування, ефективне управління різнорідними даними, більш швидкий час відгуку і як наслідок більш висока продуктивність. Але підводячи підсумки щодо процесу проектування БД на концептуальному та логічному рівні можна зробити висновок поширення баз даних NoSQL та концепція багатоваріантної персистентності зробили процес розробки бази даних із їх використанням ще більш довготривалим та трудомістким.

У зв'язку з перерахованим вище сформульована мета і завдання дослідження.

Метою даної роботи є скорочення часу на проектування баз даних зі багатоваріантною персистентністю шляхом удосконалення структури концептуальної моделі даних та методів створення моделей даних на логічному рівні. Для досягнення поставленої мети сформульовані та вирішені такі завдання:

– проаналізовано існуючі мови концептуального моделювання, технології персистентності, методи для автоматизації процесу розробки баз зі багатоваріантною персистентністю та обгрунтовано мету та задач дослідження;

– удосконалено мову концептуального моделювання AGILA MOD+ для автоматизації процесу розробки окремих концептуальних моделей даних зі багатоваріантною персистентністю;

– створено методи автоматизації розробки логічних моделей баз даних на основі концептуальних моделей зі багатоваріантною персистентністю;

– розроблено інтерфейс прикладного програмування (API) для забезпечення єдиного доступу до баз даних зі багатоваріантною персистентністю із бізнес-логіки;

– розроблені рішення (моделі та методи автоматизації та API) впроваджені для удосконалення мови концептуального моделювання AGILA MOD+, показано їх переваги при вирішенні практичних задач

# РОЗДІЛ 2
# СТВОРЕННЯ КОНЦЕПТУАЛЬНИХ МОДЕЛЕЙ ДЛЯ АВТОМАТИЗАЦІЇ РОЗРОБКИ БАЗ ДАНИХ

У відповідності зі трьохетапним предствленням згідно зі Elmasri та Navathe процесу проектування БД для автоматизації процесу розробки окремих концептуальних моделей даних БД зі багатоваріантною персистентністю удосконалено мову концептуального моделювання AGILA MOD+.

## 2.1. Детальна характеристика AGILA MOD

Наступний детальний опис AGILA MOD в основному взято з певними адаптаціями з [10]. Як було вказано в підрозділі 1.1.2 AGILA MOD (***A***utomatic ***G***enerators for ***I***nformation representation, ***L***ogical DB structures and ***A***pplications – AGILA ***Mod***eling language – MOD) – це автоматичний генератор для інформаційного представлення, логічних структур БД та застосувань. Загальна ідея полягає в тому, щоб створити концептуальну модель шляхом перекладу природної мови, як речень, які семантично не спрощуються, в графічну нотацію. AGILA MOD використовує здебільшого синтаксис ER або EER, введений Elmasri / Navathe в [10], і максимально зменшує примітиви синтаксису, щоб полегшити розуміння моделі

*Основні елементи синтаксису.*

Першим синтаксичним примітивом AGILA MOD є «Тип Об'єкта/ Object Type». Він представляє групу реальних об'єктів одного типу і представлений прямокутником із суцільною рамкою та назвою типу об'єкта всередині. Розрізняють лексичні та нелексичні типи об’єктів. Прикладами нелексичних типів об'єктів є «Person/Особа» , «Car/Автомобіль» або «Room/ Кімната», а лексичних типів об'єктів «Name/Ім'я», «Salary/Зарплата або «Birthday/День народження. Поняття лексичних та нелексичних типів об'єктів адаптоване з

мови моделювання NIAM, де детально описані "LOT" та "NOLOT".[56, глава 4.5]

Другим синтаксичним примітивом є «Тип асоціації/Association Type». Він включає всі асоціації (відношення) між двома або більше типами об'єктів за певної передумови та представлений ромбом із суцільною рамкою та назвою типу асоціації всередині, що позначає передумову асоціації. Примітив «Тип асоціації» з'єднує два примітиви «Тип Об'єкта», які беруть участь та пов'язані з «Типом Асоціації» ребрами.

Ребра відображаються як пряма суцільна лінія. Кожне з ребер повинно бути анотоване *роллю*, яку екземпляри підключеного типу об'єкта відіграють у кожній асоціації. Крім ролі для ребер задається потужнісь «Об'єкта» в «Асоціації»:

$\{min,max \mid min \in N0 \land max \in N \land 0 \leq min < +\infty \land 0 < max \leq +\infty \land min \leq max\}$.

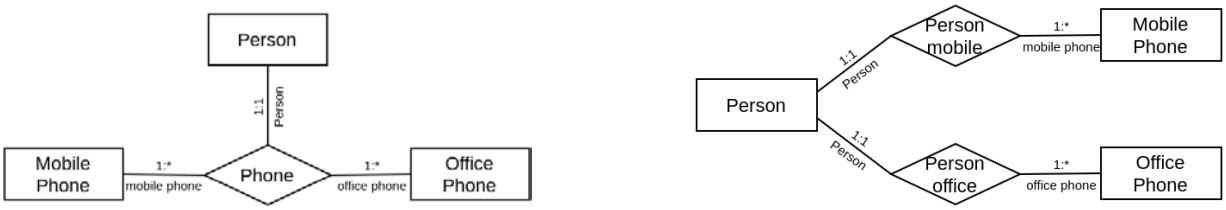
Межі потужності позначені "<min>: <max>". Максимальна межа + ∞ буде позначена зірочкою ("*"). Це дещо відрізняється від Elmasri / Navathe (пор. [11, с. 113]), де позначенням є «(<min>: <max>)», а + ∞ представлено як «N». Дужки опущені з міркувань простоти під час моделювання, а використання зірочки замість «N» запобігає плутанині. Згадування ролі може бути опущено, якщо роль ідентична імені типу об'єкта. Приклад «Об'єктів» та «Асоціації» зображений рисунку 2.1 та відповідає реченню, яке семантично не спрощується: «Особа може брати участь у довільних курсах»


Рисунок 2.1 – Тип асоціації – задає «учасників»

Ця модель може бути прочитана ще так – «Особа не повинна брати участь у довільних курсах» або «Курс повинна відвідувати принаймні одна людина». Отже, стає зрозумілим, що кожен тип асоціації містить стільки фактів, скільки «Об'єктів», оскільки його потрібно розглядати з точки зору кожного «Об'єкта»

Для «Типів асоціацій» існує лише *одне* обмеження: Якщо є «Об'єктів»з максимальною потужністю ребра "1", то «Тип асоціації» повинен з'єднувати

лише двох «Об’єктів». Якби було більше двох«Об’єктів», то така модель не представляла би речення, яке семантично не спрощується




(a) Помилковий тип асоціації (b) Виправлена модель

Рисунок 2.2 – Типові помилки при створенні речень, які семантично не спрощуються.

На рисунку 2.2a показано помилковий тип асоціації, тому що у зворотному напрямку прочитана як речення «У людини є мобільний телефон і службовий телефон», яке можна розділити на два речення, не втрачаючи при цьому жодної інформації: «Людина має мобільний телефон. У людини є службовий телефон ». Коректну модель показано на рисунку 2.2,b.

Крім цього обмеження подальших обмежень для «Тип асоціації» немає.

*Спеціалізовані типи примітиву ««Тип асоціації»*

Наприклад, дозволяється одному і тому ж «типу об'єкта» брати участь більше одного разу в «типі асоціації». На рисунку 2.3 наведено приклад коли: «Начальник працівника - це сам працівник» Цей приклад також пояснює, чому ролі учасників на ребрах є важливими. «Типу об'єкт» не повинен брати участь більше одного разу з однаковою роллю в «типі асоціації».

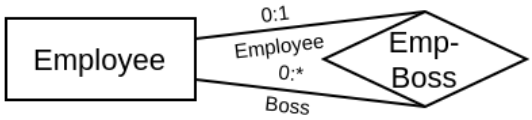



Рисунок 2.3 – Тип асоціації – Самоасоціація

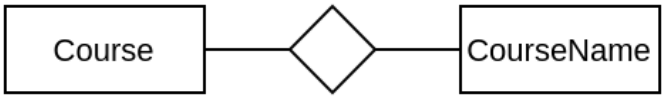



Рисунок 2.4 – Тип асоціації – Конвенція про іменування

Крім того, був доданий спрощений «Тип асоціації» для ідентифікації відносин іменування, який називається «конвенція іменування». У ньому лише два «Об’єкта», із max межею “1”. Ця скорочена форма «Типу Асоціації»

полегшує розпізнавання ідентифікаційних зв’язків типа «Курс» та «Назва курса» – семантичне речення «Курс називається»

*Формальний опис*

Введемо формальний опис викладеного: Нехай є множина $O$ об’єктів певних типів, потужність множини – $|O|$:

$$O = \{o_1, o_2, \dots o_i, \dots o_{|O|}\} \quad (2.1)$$

та множина $A$ асоціацій певних типів, потужність множини – $|A|$:

$$A = \{a_1, a_1, \dots a_i, \dots a_{|A|}\} \quad (2.2)$$

Тоді об’єкти $o_i$ та $o_j$ зв’язані за допомогою асоціації $a_i$,якщо:

$$\langle o_i, a_i, o_j \rangle = \{o_i \in O \wedge o_j \in O \wedge a_i \in A\} \quad (2.3)$$

А загальна концептуальна модель задається як:

$$MOD =< O, A > \quad (2.4)$$

*Об'єктивізація*

Однією з найпотужніших, але простих конструкцій AGILA MOD є концепція об'єктивізації «Типу асоціації». Таким чином створюється так званий «Тип структурованого об'єкта» «structured object type».

На рисунку 2.5 показаний приклад рисунка 2.1, вдосконалений датою подання заявки з використанням об’єктивації.

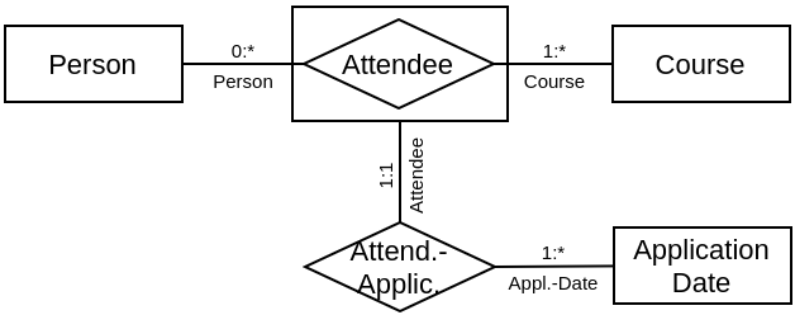



Рисунок 2.5 – Об'єктивізація «Типу асоціації»

Як і в оригінальному прикладі, який вже було запропоновано, поєднання «особи» та «курсу» можна вважати «учасником». З семантичної точки зору

цілком очевидно, що «учасник» може сприйматись також як тип об'єкта. Отже, об'єктивізація цього типу асоціації дозволяє розробнику моделі повторно використовувати цей тип асоціації як тип об’єкта для моделювання подальших речення, яке семантично не спрощується, коли поєднання виступає як «тип об’єкта». У цьому прикладі новим реченням є: “Учасник має дату подання заяви”. Тип об'єднання слід об'єктивувати лише у разі потреби як тип об'єкта. В іншому випадку об’єктивація є зайвою. При графічному зображенні лінія внутрішнього типу асоціації не повинна торкатися або перетинати лінію об’єктивізації асоціації за допомогою прямокутника. Витоки об'єктивізації можна знайти в мові IDEF1X [18], а також у існуючих діалектах ER. [56, гл. 4.2]

З врахуванням об'єктивізації «Типу асоціації» (2.3) має вигляд:

$$\langle(o_i, a_i, o_j) \vee \big((o_i, a_i, o_j), a_j, o_k\big)\rangle = \{o_i, o_j, o_k \in O \wedge (o_i, a_i, o_j) \in O \wedge a_i, a_j \in A\} \quad (2.5)$$

*Базові та системні типи*

AGILA MOD підтримує моделювання «Типів Об'єктів» базових типів і системних типів. Обидва ці типи можна сприймати як особливу форму «Типів Об'єктів». Системний тип представляє собою звичайний тип даних, наприклад «String», «Integer» або «Boolean». Аналоги таких типів є в EXPRESS [24] В AGILA MOD це додано, для забезпечення технічного представлення (Рис.2.6)

Базовий тип узагальнює поняття системного типу на будь-який тип даних, який сам по собі не має безпосереднього посилання на реальний світ. Імовірним прикладом є базовий тип money «гроші» (Рис. 2.7). Наприклад, зарплата виплачується в грошах, але самі по собі «гроші» не є «Тип Об'єкта»


Рисунок 2.6 – Системний тип

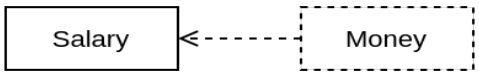



Рисунок 2.7 – Базовий тип

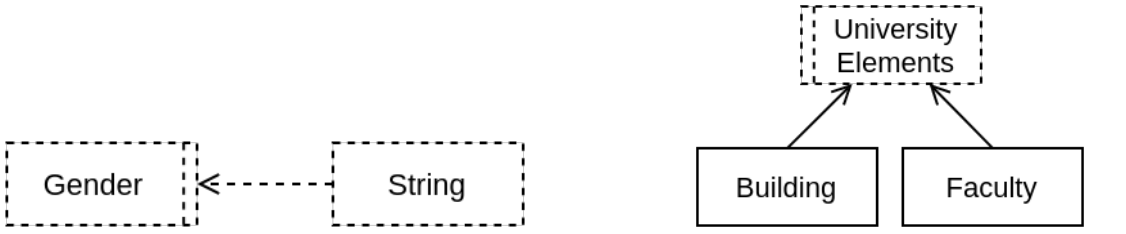
Базові типи можна сприймати як проміжний елемент синтаксису між «Тип Об'єкта» та системним типом. Це найкраще описано як повністю

формалізований тип лексичного об'єкта, як це можна знайти в синтаксисі NIAM. [56, гл. 4.5].

*Допоміжні елементи для представлення базового типу*

В синтаксисі AGILA MOD передбачено можливість представити два допоміжні значення даних базового типу для «Типу об’єктів»: перелік та набір.

*Перелік* (enumeration) – це допоміжна конструкція діє як базовий тип, але його значення обмежені фіксованим списком значень. З графічної точки зору перелік визначається як прямокутник з подвійною рамкою з правого боку. Лінії кордону пунктирні, оскільки це особлива форма базового типу. Назва переліку зазначається всередині прямокутника. Можливі значення переліку можна записати як текстову метаінформацію на краю моделі. Перелік слід використовувати у випадках, коли дозволені значення для цього типу постійно визначаються з самого початку як фіксований список унікальних невпорядкованих значень (Рис. 2.8,а)




(a) Тип переліку (b) Тип набору

Рисунок 2.8 - Допоміжні елементи для представлення базового типу

Поняття типу даних перелік також не є новим, але запозичене в синтаксисі та значенні з мови EXPRESS. [23, с. 240]

Подібним чином запозиченим з EXPRESS для AGILA MOD є тип набір для «Тип Об'єкта» .

*Набір* (set) - це впорядкована колекція без дублікатів, що включає екземпляри об’єктів довільних типів об’єктів. Це дещо відрізняється від визначення набору в EXPRESS, де набір – це невпорядкована колекція однорідних об'єктів.

Графічно набір представлений у вигляді прямокутника з подвійною рамкою з лівого боку. Через те, що набір не представляє певний «Тип об'єкта», а є лише контейнером із довільними об'єктами, межові лінії виділяються пунктиром. Типи об'єктів, які повинні внести свої об'єкти до набору, мають стрілку від типу об'єкта до набору (2.8,б).

Використання наборів у «Типі асоціації» підлягає певним обмеженням. Набір може бути лише частиною типу асоціації з величиною “0:*”. У той же час аналог у цьому типі асоціації може брати участь лише з верхньою межею “1”, тобто “$x : 1$” з $x \in \{0,1\}$. З огляду на вищезазначені правила щодо учасників типу асоціації, це обмеження передбачає, що набір може бути лише частиною «Типу асоціації», що містить двох учасників. Рисунок 2.9 демонструє наочний приклад використання набору типів для типу асоціації.

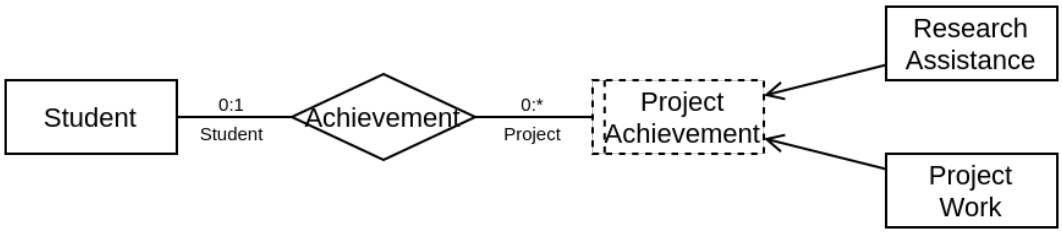



Рисунок 2.9 – Приклад використання типу «набір»

Таким чином, з урахуванням введених базового та системного типів даних формулу (2.1) можна представити як:

$$O = \{o(t)_1, o(t)_2, \dots o(t)_i, \dots o(t)_{|O|}\},$$
$$t = \begin{cases} base\ \{\text{enumeration, set}\} \\ system\ \{\text{string, integer, boolean}\} \end{cases} \quad (2.6)$$

*Успадкування* (Inheritance)

Як зазначалось при обговоренні наборів для «Об'єкт» , AGILA MOD включає концепцію узагальнення (generalization) з суб- та супертипами, а також абстрактними типами, що бере свій початок від об'єктно-орієнтованих розробок, наприклад EER Elmasri/Navathe. [11, гл. 4] (Рис. 2.10).

Синтаксично, AGILA MOD дотримується цієї концепції EER, даючи можливість виразити, що підтипи супертипу є неперервними наборами, які не

перекриваються або перекриваються [11, pp. 144]. Тут використовує синтаксис, UML, де спрямоване суцільне ребро веде від підтипу до супертипу. Дійсне також множинне успадкування з одним підтипом, що успадковує більше ніж один супертип.

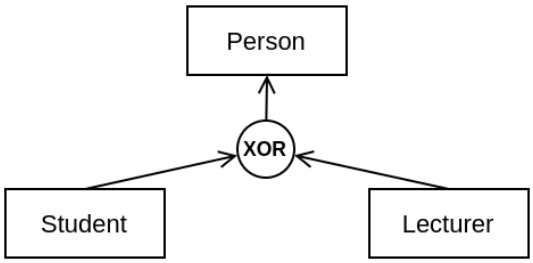



Рисунок 2.10 – Успадкування

Щоб додати концепцію EER не перекриваються «XOR» або перекриваються «OR» підтипів в AGILA MOD було додано позначення кола для логічного оператора встановленого між супертипом та його підтипами. Це позначення також дотримується ідеї речень, які семантично не спрощуються: «Особа – студент, або викладач» – успадкування «XOR».

AGILA MOD також дозволяє використовувати оператор «AND» у колі, що може бути корисним для більш складних ієрархій, як на рисунку 2.11. Можна опустити коло та підключити підтипи безпосередньо до супертипу. Це позначення еквівалентно позначенню кола з використанням «OR».

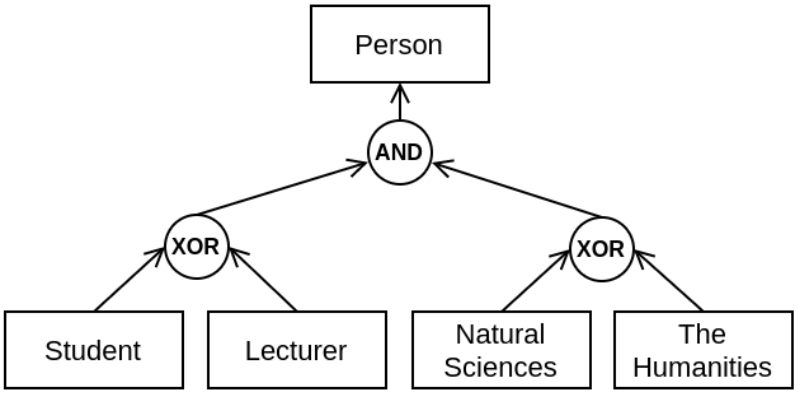



Рисунок 2.11 – Використання AND в ієрархії типів

Крім того враховуючі позначення типів для «Об'єкт» в EER [11, ch. 4.3, p. 145] будь який тип об’єкта можна оголосити *абстрактним*. Це також стосується структурованих типів об'єктів. Абстрактний тип об’єкта позначається зображенням прямокутника товстою лінією (Рис. 2.12)

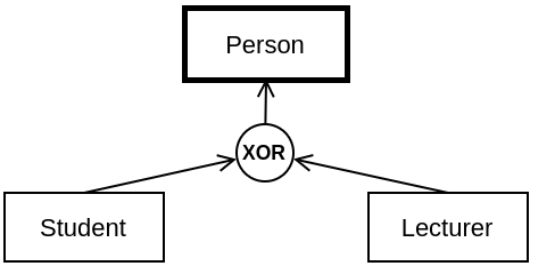



Рисунок 2.12 - Використання абстрактного типу для«Об'єкт»

З урахуванням введених позначень успадкування та абстрактного типу формальне представлення (2.6) можна представити як:

$$O = \{o(t)_1, o(t)_2, \dots o(t)_i, \dots o(t)_{|O|}\},$$

$$t = \begin{cases} \text{inheritance}\{o_i \cap o_j \vee o_i \cup o_j\}, \\ \text{abstract}, \\ base\ \{\text{enumeration}, \text{set}\}, \\ system\ \{\text{string}, \text{integer}, \text{boolean}\}. \end{cases} \quad (2.6)$$

*Запобігання використання повторень найменувань*

Усі найменування «Тип Об’єкта» та «Тип Асоціації» повинні бути унікальними, але елементи однієї моделі можуть бути повторно використані в інших моделях. Щоб мати можливість однозначно посилатися на цей елемент, сама модель повинна отримати унікальний ідентифікатор для посилань (Рис. 2.13)

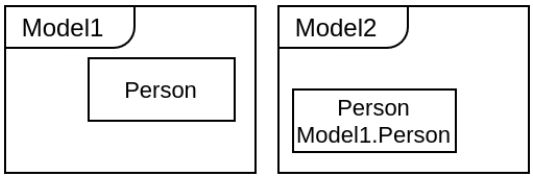



Рисунок 2.13 - Використання ««Тип Об’єкта»» в іншої моделі

*Платформа AGILA*

Мова концептуального моделювання синтаксис якої був описаний AGILA MOD є центральним елементом фреймворка AGILA. Незалежно від мови моделювання фреймворк AGILA надає можливості для подальшого створення логічних моделей та фізичних моделей даних із готової концептуальної моделі. Для створених моделей передбачені структури даних, які однозначно зберігають та обробляють ці моделі. Обробка включає автоматизовану систему

деривації (автоматизованого виведення), яка може генерувати фізичні структури даних та API для унікального доступу до даних з бізнес-логіки, яка повинна отримувати доступ до даних.

Важлива частина системи деривації пов’язана з визначенням застосовних правил виведення для отримання логічної моделі із концептуальної моделі. На даний час існують правила виведення реляційної логічної моделі, які практично використовуються для навчання студентів у рамках курсу «Бази даних» в університеті Прикладних наук м.Аугсбург. [58]

Технології, які були використані для реалізації фрейморку AGILA, включають процедурну мову SQL (PL / SQL) для серверної частини та фреймворку  eclipse відповідно до веб-технологій для інтерфейсу. Більше того, в PL / SQL було створено унікальний API для доступу до серверної серверної.

Ці розробки застосовуються і використовуються в якості основних в подальшому для розробки удосконаленого API четвертому розділі даної роботи.

## 2.2. Розробка розширення мови AGILA MOD +

AGILA MOD – це концептуальна мова, яка не залежить від деталей реалізації та відображає семантику ПрО, і з одного боку це потужний, а с другого простий підхід для створення реляційної логічної моделі. По суті, однією з сильних сторін AGILA MOD є простота синтаксису. Але ця простота супроводжується також спрощенням мови та ймовірною втратою цінних деталей семантики ПрО. До цього часу ефекти цього спрощення на логічну модель оцінювались лише на основі реляційної парадигми. Однак із зростанням баз даних, які підпадали під термін NoSQL протягом останнього десятиліття, ситуація змінилася. Тому необхідно проаналізувати синтаксис AGILA MOD щодо логічних моделей NoSQL баз даних для визначення слабких сторін AGILA MOD щодо його виразності. При цьому необхідно зосередитись на аспектах, які справді важливі в контексті проектування NoSQL баз даних. Ця

стратегія дозволяє збалансувати між простотою синтаксису та виразністю, яка є важливою для отримання всебічної картини ПрО, яка може бути ефективно перетворена на довільні логічні моделі. Звичайно, недоліком цього дедуктивного аналізу є спрямованість на NoSQL логічні моделі

Для демонстрації результатів аналізу для всіх подальших рішень використано простий приклад: Звичайна телефонна книга це «Особи» та їхні «Телефонні номери» Кожна особа має довільні телефони(будь-яку кількість телефонних номерів), а телефон присвоюється принаймні одній особі. Особа представлена своїм іменем, а телефон - своїм номером телефону. Модель AGILA MOD у цього прикладу показана на рисунку 2.14.

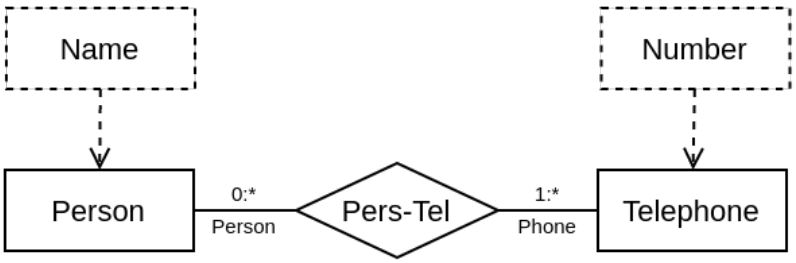


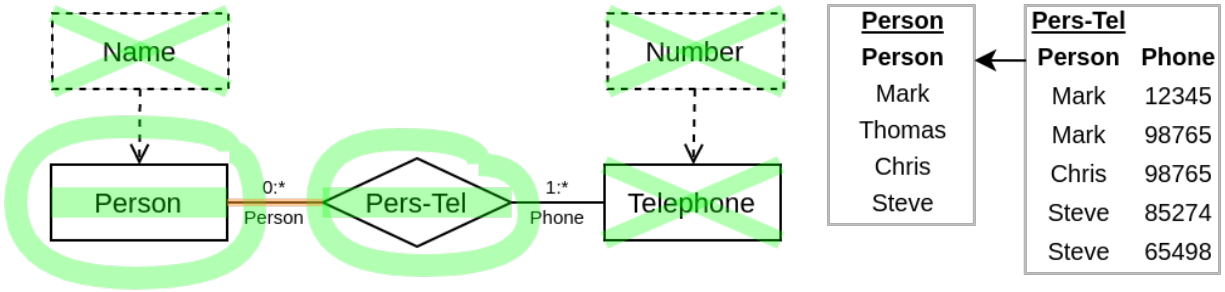

Рисунок 2.14 – Проста телефонна книга як концептуальна модель.

### 2.2.1 Аналіз недоліків існуючого синтаксису AGILA MOD щодо правил дерівації

Застосовуючи правила фрейморку AGILA для дерівації реляційної моделі із концептуальної моделі приклада (Рис. 2.14) отримаємо маркування, як показано на рисунку 2.15,a. Побудовано два компоненти узгодженості, тобто «Person» та «PersTel». Вони підключаються за допомогою посилання зовнішнього ключа, де «Pers-Tel» вказує на «Person». Всі інші елементи не відображатимуться як окрема структура в реляційній моделі даних. Однак це не означає, що вони зовсім не мають значення. Вони відіграють лише певну роль усередині створених структур.

Фінальні реляційні структури даних, які є результатом застосування вищезазначених правил виведення у [58], показані з прикладними даними на рисунку 2.15b. Відношення «Person» має лише один атрибут «Person» оскільки

відповідний компонент узгодженості не містить жодних інших елементів, крім самого типу об'єкта «Person».




(a) Концептуальна модель із знаками деривації (b) Реляційна модель зі екземплярами даними

Рисунок 2.15 – Приклад деривації концептуальної моделі.

Зв'язок «Pers-Tel» має два атрибути «Person» та «Phone». Такий автономний «тип асоціації» отримує стільки атрибутів, скільки в ньому є учасників. Імена атрибутів стосуються ролі, а не «типу об'єкта», до якого веде з'єднання. Тому атрибут називається «Phone», а не «Telephone». У той час як елементи атрибута «Person» підтримуються шляхом зовнішнього ключа до відношення «Person», значення атрибута «Phone» записуються негайно у відношення без посилання, оскільки «тип об'єкта» «Telephone» скасовано за правилами фреймворку AGILA .

Причина неоднакового ставлення до «типів об'єктів» «Person» та «Telephone» полягає в тому, як правила деривації трактують різні значення потужності зв’язку участників в «типу асоціації» «Pers-Tel» – «Особа *не повинна* мати номер телефону», тобто можуть бути Особи, які не відображаються в «Pers-Tel». Отже, для зберігання всіх бажаних осіб необхідне окреме відношення. З іншого боку, кожен екземпляр Телефону з’являється принаймні раз у цьому типі асоціації. Для цього немає необхідності зберігати телефонні номери окремо, оскільки недоцільно зберігати телефонні номери, не присвоєні конкретній особі.

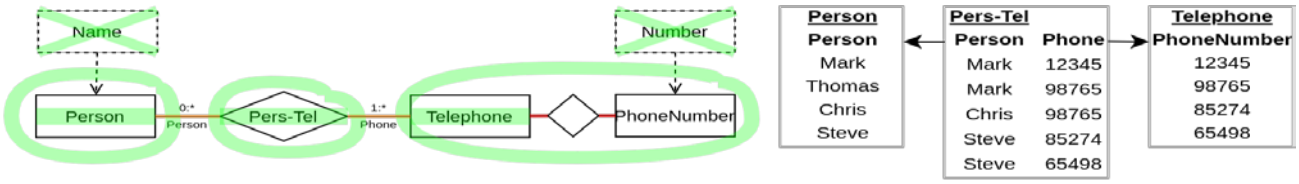
Базові типи «Name» та «Number» також скасовуються за правилами фреймворку AGILA. Оскільки базові типи є узагальнених формою системних

типів, вони визначають лише область різних атрибутів. Це буде трансльовано у фізичні типи під час деріваці у фізичні структури даних, наприклад:
CREATE TABLE→ «Name» → VARCHAR (50), «Number» →NUMBER (20).

Відображення результату деріваці (рис. 2.15, б) викликає різні запитання, які будуть пояснені далі.

*Автономные «Типи Об'єктів»* (Standalone Object Types)

Пояснення щодо неоднакового поводження з «типами об’єктів», які беруть участь у «типах асоціацій», зрозумілі з урахуванням наявних на даний момент синтаксичних примітивів. Для цих автономних (standalone) «типів об’єктів» повинно бути зважування між наданням їм окремої структури в логічній моделі та уникненням надмірної кількості відношень окремих атрибутів та обмежень зовнішнього ключа в базі даних. Доречним виглядає рішення скасувати ті «типи об’єктів», які беруть участь у будь-якому «типі асоціації» з нижньою межею потужності зв’язку, а саме кожен екземпляр повинен з'являтися принаймні один раз. Застосування є значнішим тому, що цю умову можна виправити, додавши до «типу об'єкта» «тип асоціації» – конвенція про іменування (див. рис 2.4). На рисунку 2.16,а оригінальний приклад був змінений з використанням конвенції про іменування «типу об'єкта» «Telephone»: «Телефон ідентифікується за номером телефону».




а – концептуальна модель телефонної книги із конвенцією про іменування

(b) Реляційна модель зі екземплярами даними

Рисунок 2.16 – Приклад деріваці концептуальної моделі із дотриманням конвенції про іменування

Це, звичайно, правильно, і результат є визначним: Оскільки конвенції про іменування є короткою формою «типу асоціації» з відношенням «1:1», ці зв’язки позначені як сильні, що автоматично ведуть до окремого компонента узгодженості. В результаті встановлюється додатковий зовнішній ключ між

«Pers-Tel» та «Telephone», оскільки зараз у реляційній моделі існує додаткова структура, на яку можна посилатися. Реляційна модель даних, яку зображено на рисунку 2.15b, показано з поправками на рисунку 2.16,б. Важливо зазначити, що у відповідності до концептуальної моделі (рис. 2.16,б) в «типу об'єкта» «Telephone» не повинно бути жодного телефонного номера, на який немає посилань. Це має бути забезпечене таким впровадженням – як, наприклад, через налаштування тригерів.

Описане використання автономних «типів об’єктів» призводить до наступних наслідків:

1. провокує непотрібне розростання концептуальної моделі, додаючи конвенцію про іменування кожного разу, коли необхідно відобразити «тип об’єкта» у побудовані далі логічній моделі даних.

2. з семантичної точки зору не повинно бути різниці, чи має «тип об’єкта» конвенцію про іменування чи просто ідентифікується сам по собі. Отже, чому існує різниця в логічній моделі?

3. виникає ризик того, що розробник моделі виконує свою роботу з логічною моделлю як результат подальшого виведення в його свідомості, що залежить тільки від його кваліфікації, але це явне протиріччя визначеним цілям концептуального моделювання, тобто зобразити ПрО без врахування аспекту реалізації.

4. недоліком синтаксису AGILA MOD є те, що не може бути різниці у зображенні між «типами об’єктів», які можна скасувати, та тими, які не можна скасувати.

Телефонна книга є добрим прикладом: використання телефонного номера є не такий простим, як здається на перший погляд. Придивившись уважніше, можна виявити різницю у двох випадках використання: коли особа отримує новий номер (в випадку переїзда) та телефонне з'єднання отримує новий номер (від нового провайдера).

В першому випадку точки зору концептуальної моделі, спочатку потрібно змінити «тип асоціацію» «Pers-Tel», де особі відповідає її номер

телефону, а при необхідності заздалегідь слід додати новий екземпляр в «тип об’єкта» «Telephone» якщо такого номера немає в базі даних. У другому випадку потрібно змінити «Telephone» додавши новий номер в «PhoneNumber». У деріваціі на реляційну модель (див. 2.15,b) другий варіант використання спричиняє проблему, оскільки тут немає відношення «Telephone», яке можна оновити. Натомість усі записи стосовно «Pers-Tel» потрідно оновлювати в ручному режімі. Хоча це не є великим викликом у SQL, відсутність обмеження зовнішнього ключа дозволяє змінювати телефонні номери окремо без перевірки. Це несе ризик класичної аномалії модифікації, як це описано в [11, гл. 14.1.2].

*Каскадування (Cascading) Каскадне виділення та оновлення*

Ще один недолік пов’язаний з каскадуванням було виявлено під час аналізу синтаксису концептуального моделювання щодо деріваціїї реляційної логічної моделі. Посилання між компонентами узгодженості (coherence) які створюються в процесі деривації, призводять до відношенням зовнішнього ключа в РБД. Фактично ці обмеження зовнішнього ключа можуть бути визначені як каскадні або ні, тобто, повинен чи ні дочірній запис автоматично видалено при видаленні вказаного батьківського запису.

Різницю можна спостерігати на прикладі адаптованому телефонної книги на рисунках 2.16 а,б. Тут під час деривації з’являються два посилання; «Pers-Tel» на «Person» та «Pers-Tel» на «Telephone». У реляційній моделі це призводить до відповідних обмежень зовнішнього ключа, які показані стрілками на рисунку 2.16,б. Однак із відображення неможливо зрозуміти із строк участників «Pers-Tel», має бути одне із результуючих посилань каскадним або ні.

Єдина синтаксична різниця, яку можна використовувати для отримання алгоритмічного рішення – це визначений тип відношення, де найнижча межа дорівнює «0» для «Person» та «1» для «Telephone». Однак це не є виправданим, тому що найнижча межа виражає лише те, як часто екземпляр «типу об’єкта» -

учасника повинен з’являтися у зв’язку визначеного «типу асоціації». Отже, вказані межі виражають обмеження лише для екземплярів «типів об’єктів» - учасників. Але вказана різниця не вказує жодних умов щодо екземплярів «типу асоціації». Єдине, що стосується «типу асоціацій», це те, що вони мають стільки атрибутів, скільки учасників є у відповідного «типу асоціації», і комбінація атрибутів повинна бути унікальною для всіх екземплярів цього типу.

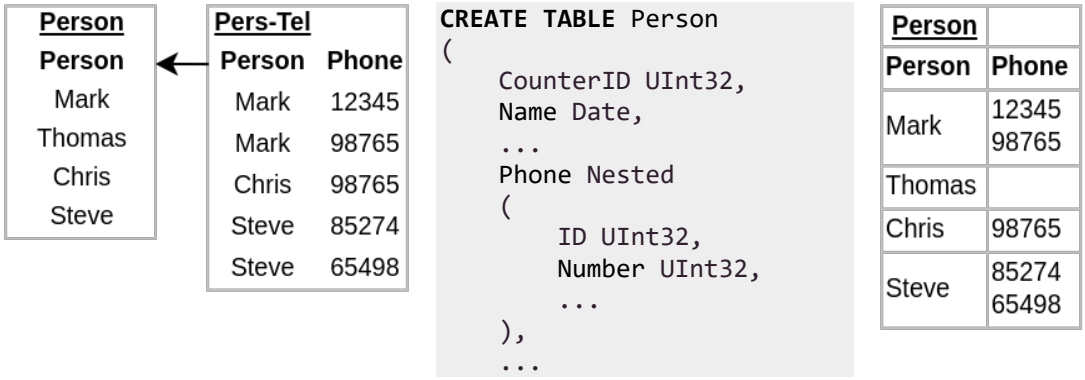
Але якщо розглядати можливе каскадування із семантичної точки зору використовуючи приклад телефонної книги то вочевидь при видаленні екземпляра «Person» з телефонної книги необхідно також видалити всі повьязані з ним телефонні номери, що зберігаються, відповідно до «типу асоціації» «Pers-Tel» цього екземпляру «Person» з телефонними номерами. Але ніхто не погодиться видалити всі телефонні номери, перш ніж можна буде видалити саму особу. Очікується, що це станеться автоматично. З іншого боку, телефонний номер слід видаляти, лише якщо більше немає жодної особи, яка посилається на цей номер телефону. В іншому випадку запис в телефонній книзі людини може самовільно втратити номер телефону.

Отже, «типи об’єктів»-учасників «типів асоціацій» можна семантично розділити на тих, що *мають зворотний вплив* на існуючі асоціації, та тих, що не мають такого впливу Однак AGILA MOD не має можливості представити цю відмінність для учасників типу асоціації.

*Вкладені структури даних*

Недолік синтаксису AGILA MOD щодо зображення вкладених структур даних стає більш помітним при порівнянні реляційних структур даних на рисунку 2.15b, яка була повторена для порівняння на рисунку 2.17,а подібними можливими структурами даних NoSQL. Наприклад документної базі даних телефонні номери особи, швидше за все, зберігатимуться як список значень всередині документа, що представляє особу. Зберігання телефонних номерів як списку значень всередині елемента «Person» як єдиної структури даних є більш природним способом представлення цих фактів, а потім їх детальної заміни на

окрему структуру даних. Об'єктно-реляційний приклад показаний на рисунку 2.17, де стовпець «Phone» містить довільні значення для одного запису.



| Person |
|---|
| Person |
| Mark |
| Thomas |
| Chris |
| Steve |

| Pers-Tel | |
|---|---|
| Person | Phone |
| Mark | 12345 |
| Mark | 98765 |
| Chris | 98765 |
| Steve | 85274 |
| Steve | 65498 |

```
CREATE TABLE Person
(
    CounterID UInt32,
    Name Date,
    ...
    Phone Nested
    (
        ID UInt32,
        Number UInt32,
        ...
    ),
    ...
```

| Person | |
|---|---|
| Person | Phone |
| Mark | 12345<br>98765 |
| Thomas | |
| Chris | 98765 |
| Steve | 85274<br>65498 |

(а)Реляційна модель (б)Приклад вкладеної таблиці (в)Об'єктно-реляційна модель

Рисунок 2.17 – Варіанти реалізації телефонної книги

Але мова SQL підтримує вкладені структури даних (вкладені таблиці). Параметри вкладеної структури даних - імена і типи стовпців, вказуються так само, як у запиту CREATE. Кожному рядку таблиці може відповідати будь-яку кількість рядків вкладеної структури даних (рис. 2.17,б)

В AGILA MOD не має можливостей для відображення семантичних відмінності вбудованих структур даних всередині основних переважаючих.

Розглядаючи приклад концептуальної моделі AGILA MOD на рисунку 2.14, таку відмінність необхідно було б відобразити в контексті типу асоціації «Pers-Tel». Але лінії, що з’єднують учасників цього «типу асоціації» не сигналізують про переважання одного з «типів об’єктів»-учасників, що може призвести до бажаного алгоритмічного рішення.

В даний час це може бути виявлено лише через вказані межі потужності участі «x:1» (з $x \in \{0,1\}$), оскільки вони описують сильну прив’язку між «типом асоціації» та «типом об’єкта». Однак в такому випадку немає можливості змінити межу потужності участі «Person», оскільки це повністю змінило б семантику цієї моделі. Зміна межі участі з «0: *» на «0: 1» означала б зміну речення, яке семантично не спрощується «Особа може мати довільні телефонні номери» на «Особа може мати номер телефону». Зменшити кількість записів

телефонної книги для людини точно не є правильним. Отже, має бути поправка, яка зможе виразити описане поширення без зміни будь-якого інтервалу участі.

### 2.2.2 Розширення синтаксису мови AGILA MOD

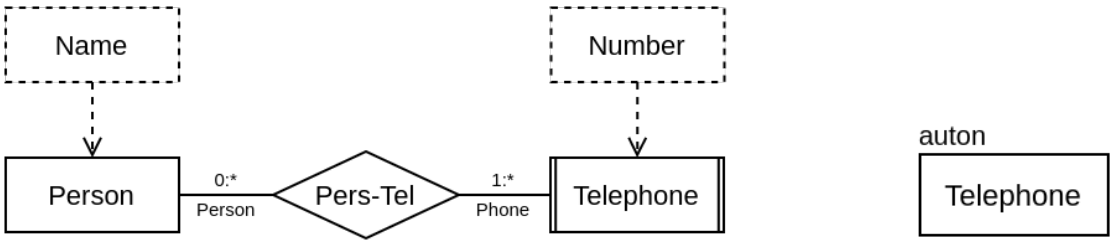
Знайдені проблеми у підрозділі 2.2.1 показують, що існуючому набору синтаксичних примітивів AGILA MOD бракує виразності щоб детально представити всі необхідні семантичні кореляції концептуальної моделі, а алгоритму деривації в середовище AGILA бракує правил, щоб коректно її перетворити в логічні структури даних незалежно від обраної парадигми персистентності (реляційної або NoSQL). Для їх вирішення запропано розширення мови AGILA MOD у вигляді додавання двох структурних елементів, які легко адаптуються до загальної концепції мови, яка стосується представлення речень, які семантично не спрощуються за допомогою «типів об’єктів» та «типів асоціацій». З врахування запропанованого концептуальна мова моделювання AGILA MOD підвищується до AGILA MOD +

Незалежний «Тип об’єкта» (Autonomous Object Types)

Що стосується першого недоліка про брак коректного представлення автономних «типів об’єктів» (standalone object types), то розробнику потрібно дати можливість відрізняти на концептуальному рівні, «типи об’єктів», які є незалежними. Для цього запропановано ввести додатковий тип «незалежний» для «типа об’єкт». Незалежний тип елемента «тип об’єкт» виражає, що екземпляри описаного елемента ПрО існують незалежно від співвідношень з іншими елементами описаної області. Тобто цей тип об'єктів абсолютно незалежний і має власне існування. Незалежний тип об’єкта семантично сигналізується певними прикметниками автономії для конкретного типу, наприклад, «Незалежний» (independent, self-contained, standalone, sovereign, self-supporting, distinct or by itself) Таким чином, речення, яке семантично не спрощуються в моделі телефонної книги щодо «типу об’єкта» «Telephone» на рисунку 2.14 може бути доповнене додатковим речення, яке семантично не

спрощуються, наприклад: «Телефон є незалежним» (A telephone is self-contained)

Позначення незалежного типу об'єкта в моделі AGILA MOD + є однаковим із звичайним типом об'єкта, тобто прямокутником із суцільною лінією. Для представлення незалежного характеру вертикальні лінії прямокутників мають подвійну суцільну лінію. Крім того було запропоновано альтернативне представлення незалежного типу з додаванням позначки «auton» у верхньому правому куті прямокутника. На рисунку 2.18,а показаний приклад телефонної книги з незалежними типом об’єкта «Telephone», а на рисунку2.18,б – альтернативне представлення незалежного типу




(a) Тип об'єкта "Телефон" є незалежним. (b) Альтернативні позначення.

Рисунок 2.18 - Застосування автономного типу об’єкта в прикладі телефонної книги.

Тоді в перелік формалізованих типів елементів «тип обьєкту» в (2.6) додається тип *autonomous*

$$t = \left\{ \begin{array}{c} \text{inheritance}\{o_i \cap o_j \vee o_i \cup o_j\}, abstract, \\ base\ \{\text{enumeration, set}\}, \\ system\ \{\text{string, integer, boolean}\}, autonomous \end{array} \right\} \qquad (2.7)$$

Подібно до інших елементів синтаксису AGILA MOD, це позначення запозичене з UML, де воно представляє так званий «активний об'єкт» (active object), як це описано в [59, гл. 13.3.8, с. 438]. Більшість розробників знайомі з UML і цей доданий структурний елемент є досить легким для розуміння, тому існує певна схожість між цими елементами синтаксису.

Розглядаючи існуючі правила деривації для створення реляційних логічних моделей в фрейморку AGILA, як вони були визначені в [56], для прикладу телефонної книги на рисунку 2.15 можно записати:

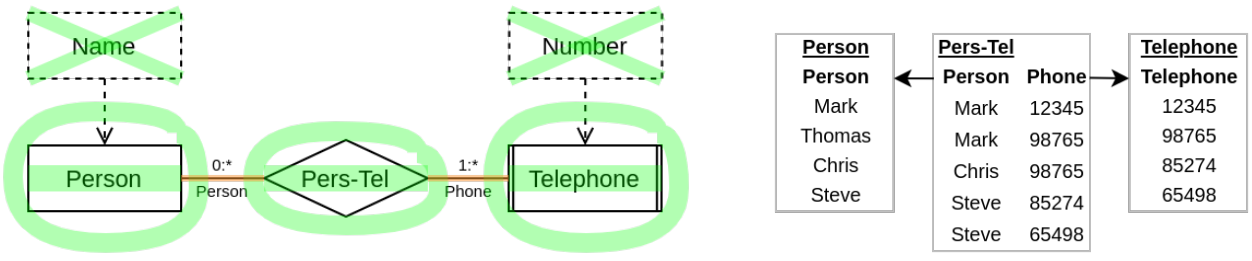
*Правило деривації 1: 1. скасуйте елементи «тип об'єкта», які не частиною компонента узгодженості та є учасником у будь-якому «типі асоціації» з нижньою межею «1».*

*2. залиште елементи «тип об'єкта», які є учасником у будь-якому «типі асоціації» з нижньою межею "0", саме вони стають їх власним компонентом узгодженості*

Додавання незалежного типу для елемента «тип об'єкта» практично еквівалентно доданню до «типу об'єкта» «тип асоціації» – конвенція про іменування (див. рис 2.4) та прикладу деривації концептуальної моделі із дотриманням конвенції про іменування.

З додаванням незалежного об'єкта маємо *Правило деривації 2: Незалежні «типи об'єктів», які ще не є частиною компонента узгодженості , стають власним компонентом узгодженості.*

У порівнянні з вхідною моделлю (рис. 2.4), деривація реляційної структури для телефонної книги, як показано на рисунках 2.19a,b, демонструє різницю у типі об'єкта концептуальної моделі «Telephone», який зараз визначений як окремий компонент узгодженості, що дозволяє встановити додаткове посилання, вказавши «тип асоціації» "Pers-Tel" до типу об'єкта «Telephone». Отримані реляційні структури (2.19,b) показують, що тепер телефонні номери отримують своє власне відношення, на яке посилається відношення «Pers-Tel». Крім того на фізичному рівні БД використання, наприклад, тригерів гарантує, що на кожен номер телефону в «Telephone» справді посилається хоча б одна особа в «PersTel».




(а)Концептуальна модель зі знаками деривації (б) Реляційна структура.

Рисунок 2.19 - Результат деривації з використанням автономних типів об'єктів.

Сильні учасники асоціації (Strong Association Participants)

Для подолання недоліків щодо каскадування (cascading) та вкладених структур (nested structures) у синтаксису AGILA MOD необхідно передбачити можливість відображення структурних відмінностей для «типів асоціацій», які дозволяють своєчасно вирішувати, чи слід обробляти певні елементи окремо із взаємними посиланнями між ними та чи так що один з них є частиною іншого. У прикладі телефонної книги на рисунку 2.14 особа може швидко вирішити, який елемент є переважним у типі асоціації «Pers-Tel». Вочевидь , що особа володіє своїми телефонними номерами, що змушує асоціації між особами та телефонними номерами схилятися в бік особ. Отже, цю структурну відмінність треба передбачити у концептуальній моделі.

При аналізі недоліків в пункті 2.2.1 вже зазначалося, що межі участі в асоціації «x: 1» (з $x \in \{0,1\}$) вже сигналізують про сильну прив'язку. Проте це constellation не можна використовувати загалом, оскільки семантичні наслідки в цілому сягають набагато далі, ніж передбачається у цьому випадку. Отже, концепція «сильного зв’язку» повинна виходити за рамки цього дуже вузького контексту.

Тому синтаксис типів асоціацій доповнюється можливістю оголосити одного учасника сильним. Подібно ситуації з межами участі в асоціації «x: 1», сильний учасник наголошує, що «тип асоціації» сильніше пов’язаний із типом підключеного об’єкта, ніж з рештою. У моделі AGILA MOD + сильна участь в асоціації, або сильний об’єкт-учасник підключається до типу асоціації за допомогою товстої лінії. На рисунку 2.20, «Person» є сильний учасник «типу асоціації» «Pers-Tel» у прикладі телефонної книги.

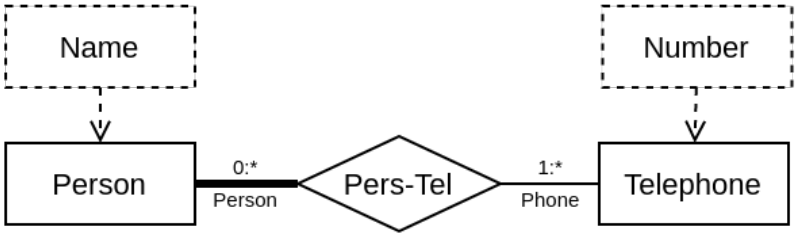



Рисунок 2.20 - Приклад телефонної книги, де «Person» є сильним учасником асоціації .

Для рукописних концептуальних моделей таке позначення можна виконати «strong» між лінією та роллю участі (рис. 2.21).

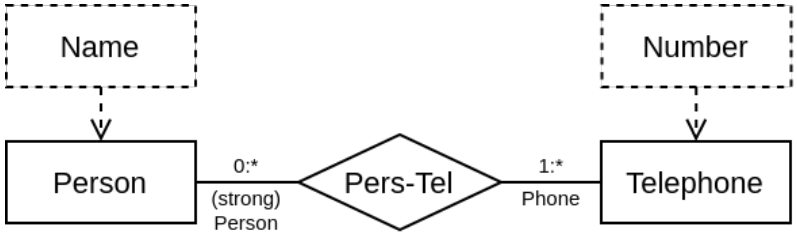



Рисунок 2.21 - Альтернативні позначення сильної участі у типах асоціацій

При деривації з концептуальної моделі AGILA MOD + із сильним учасником «Типу Асоціації» для логічної моделі персистентності, яка допускає вкладені структури вкладені структури (nested structures), тепер може обробляти сильних учасників так само як учасників з межами участі в асоціації «x: 1». Отже, тип асоціації буде включено до компонента узгодженості типу об'єкту, який є задіяним в асоціації. Це в свою чергу призведе до атрибута списку для конкретної структури даних, що містить решту учасників цього «типу асоціації». З іншого боку, алгоритм деривації логічної моделі певного типу персистентності даних, яка не підтримує вкладені структури - як, наприклад, реляційна модель - може просто ігнорувати введених сильних учасників та застосовувати попередні звичайні правила із [58].

Однак отримання логічних моделей об'єктно-реляційних баз даних, які мають вкладені структури, такі як список значень та вкладені таблиці до стандартного набору правил деривації, потрібно додати ще декілька додаткових позначень. Правило:1 «*Позначте всіх учасників з інтервалом «1: 1» і «0: 1»*» розширюється до «*Позначте всіх учасників з інтервалом "1: 1" і "0: 1", а також сильних учасників*».

Крім того, правило, яке визначає, як створити відношення з компонента узгодженості, має бути змінено. Для реляційних баз даних Правило 2:

*«Один тип об’єкта всередині компонента узгодженості позначається як ім’я відношення. Усі інші типи об’єктів усередині відношення стають ідентифікуючими атрибутами. Усі ролі типу асоціації, що лежать усередині*

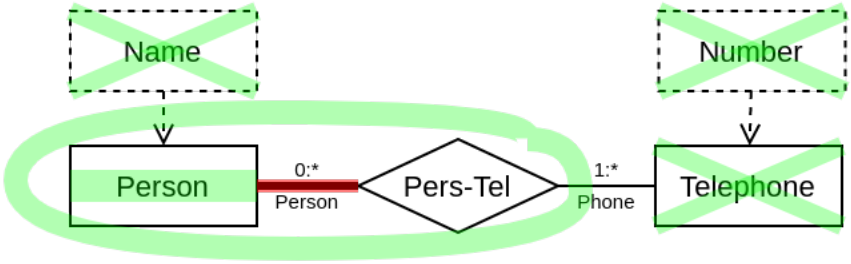
*компонента узгодженості, які вказують на тип об'єкта поза компонентом узгодженості, стають атрибутом відношення»*

Для об'єктно-реляційної логічної моделі Правило2 має бути адаптоване таким чином:

*«Один тип об'єкта всередині компонента узгодженості позначається як ім'я відношення. Усі інші типи об'єктів усередині відношення стають ідентифікуючими атрибутами. Усі ролі типу асоціації, що лежать усередині компонента узгодженості, які вказують на тип об'єкта поза компонентом узгодженості, стають атрибутом відношення. Ролі цього типу, які є результатом позначеної сильної участі з верхньою межею, більшою за 1, інкапсулюються як вкладені атрибути»*

Це гарантує, що всі елементи, які були додані до відношення завдяки сильній участі, дадуть список відповідних кортежів.

На рисунку 2.22,а показані знаки деривації, які можуть бути застосовані до об'єктно-реляційної моделі із згаданим розширенням правил. Крім раніше, зв'язок участі між "Person" та "Pers-Tel" був позначений, і тип асоціації "Pers-Tel" тепер об'єднаний з типом об'єкта "Person" в один компонент узгодженості.




| **Person** | |
|---|---|
| **Person** | **Phone** |
| Mark | 12345<br>98765 |
| Thomas | |
| Chris | 98765 |
| Steve | 85274<br>65498 |

(а) Концептуальна модель із знаками деривації, які враховують «сильних» учасників «типів асоціації»

(б) Об'єктно-реляційна модель

Рисунок 2.22 - Результат деривації з використанням «сильних» учасників «типів асоціації»

Результат деривації, націленої на об'єктно-реляційні структури, зображений на рисунку 2.22,b. Залишився єдиний компонент узгодженості «Person» на рисунку 2.22,а переводиться у відношення «Person», що містить

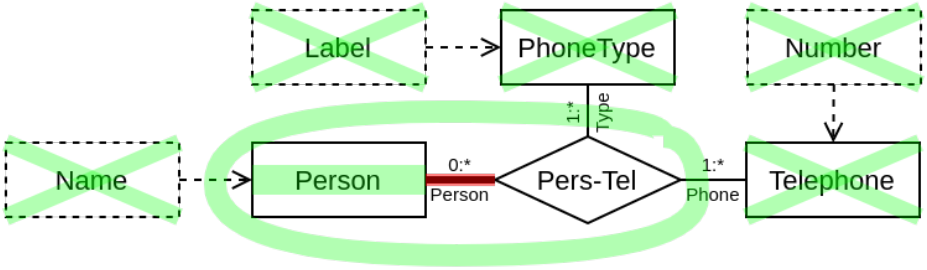
два стовпці, а другий стовпець «Phone» є атрибутом списку, який може містити довільні значення для одного запису.

Наведемо ще два приклади «сильного» типу зв’язку для об’єкта-учасника асоціації

Перший приклад на рисунку 2.23a показує телефонну книгу з типом асоціації “Pers-Tel”, яка отримала третього учасника «PhoneType». Тип телефону може бути “mobile”, “private”, “Fax” тощо. Речення, які семантично не спрощуються в цьому випадку будуть такими: *Кожна особа може володіти будь-якими комбінаціями телефону та типу телефону. Кожен телефон належить принаймні одній особі з типом телефону. Кожен тип телефону належить принаймні одному телефону особи.*

На рисунку 2.23a також показано знаки деривації. Було відзначено сильний зв’язок з «Person» та побудовано компонент узгодженості з назвою «Person». Решта елементів було скасовано або через те, що вони мають базовий тип, або дотримуються правила, що «типи об’єктів» вже присутні у «типі асоціації» через нижню межу, яка дорівнює “1”, як це описано для автономних типів об’єктів. На рисунку 2.23b показано об’єктно-реляційну реалізацію цієї концептуальної моделі. Компонент узгодженості «Person» трансформується в таблицю «Person». Він отримує атрибут «Person» завдяки тому, що сама особа представлена іменем базового типу. Крім того, він отримує атрибут вкладеної таблиці, що містить атрибути «Type» та «Phone» відповідно до решти ролей учасників «Pers-Tel», які не є сильними. Відповідно до значеннями меж між «Person» та «Pers-Tel» ця структура відповідає концептуальній моделі. Наведені приклади даних також детальніше демонструють змодельовані факти. З трьома учасниками «Pers-Tel» допускається будь-яка комбінація, наприклад особа «Steve» має два номери типу «Work», тоді як особа «Ben» має один номер телефону типу«Work» та «mobile». Через нижню межу «0» між типом об’єкта «Особа» та типом асоціації «Pers-Tel» також є дійсним, що телефонних номерів взагалі немає, як у особи «Thomas» (див. рис 2.23b). Відповідно до концептуальної моделі, кожен запис у вкладеній таблиці повинен містити

обидва атрибути. Це пов'язано з характером типу асоціації, де кожному екземпляру потрібен повний набір усіх учасників. Інтервал "0: *", наприклад для типу об’єкта «PhoneType» - це означало, що не кожен екземпляр «PhoneType» повинен бути частиною асоціації «Pers-Tel».




(а) Концептуальна модель із знаками деривації, які враховують «сильний» тип зв’язку для об’єкта-учасника асоціації

| Person | |
|---|---|
| Person | |
| Mark | Type Phone<br>Work 12345<br>Home 98765 |
| Thomas | |
| Chris | Type Phone<br>mobile 98765 |
| Steve | Type Phone<br>Work 85274<br>Work 65498 |
| Ben | Type Phone<br>Work 85274<br>mobile 85274 |

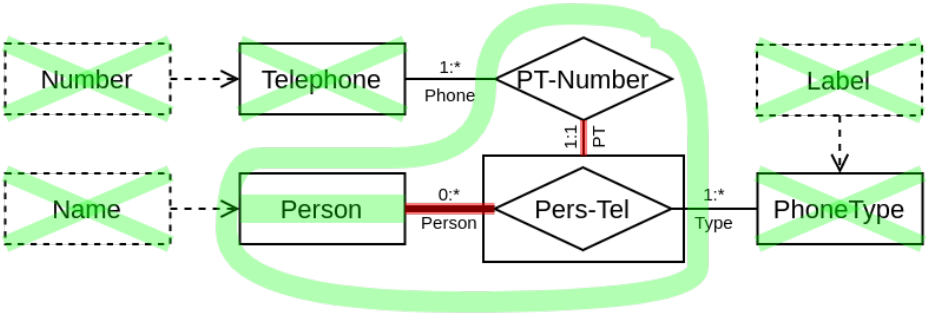
(б) Об'єктно-реляційна модель

Рисунок 2.23 – Складний приклад для активних учасників. Особа може володіти довільними комбінаціями типу телефона та номеру телефону

Модифікуємо розглянутий приклад, застосовуючи об’єктивацію для типу асоціації «Pers-Tel» (див. рис. 2.24a). Речення, які семантично не спрощуються в цьому випадку змінюються таким чином: *Особа може мати типи телефонів. Тип телефону належить принаймні одній особі. Кожен телефонний тип особи отримує рівно один телефон. Кожен телефон належить принаймні одному типу телефону особи.*

За допомогою цієї модифікації додається додатковий тип асоціації, який також стає частиною компонента узгодженності завдяки інтервалу "1: 1" – це «PT-Number». Найголовніше, що кількість атрибутів для логічної моделі не змінюється із цією зміненою моделлю. Але відносини між ними змінюються. Тепер кожна особа може мати кожен тип телефону лише один раз, і цій комбінації присвоюється рівно один номер телефону. Рисунок 2.24b демонструє, що об’єктно-реляційна реалізація все ще має однакові атрибути з атрибутом вкладеної таблиці для типів телефонів та номерів. Однак у цій

модифікованій моделі атрибут «Type» тепер унікальний для кожної вкладеної таблиці. Отже, тепер не можливо, щоб особа «Steve» мала два номери телефонів типу «Work». Усі інші твердження залишаються як і раніше. Цікаво в цьому контексті, що атрибут вкладеної таблиці «Phone» став би необов'язковим, якщо б значення межі участі для ролі «PT» змінився з "1: 1" на "0: 1". Отже, навіть можливо мати вкладені таблиці з необов'язковими атрибутами.




(а) Концептуальна модель із знаками деривації, які враховують «сильний» тип зв'язку для об'єкта-учасника асоціації

| Person | |
|---|---|
| **Person** | |
| Mark | **Type** **Phone**<br>Work 12345<br>Home 98765 |
| Thomas | |
| Chris | **Type** **Phone**<br>mobile 98765 |
| Steve | **Type** **Phone**<br>Work 85274<br>~~Work~~ ~~65498~~ |
| Ben | **Type** **Phone**<br>Work 85274<br>mobile 85274 |

(б) Об'єктно-реляційна модель

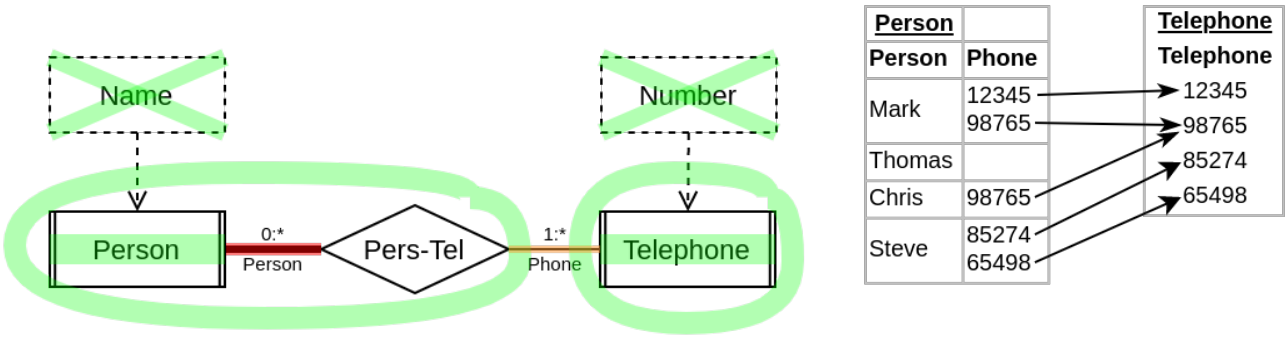
Рисунок 2.24 – Складний приклад для активних учасників.

В цілому, можливі навіть складні ситуації з кількома учасниками типу асоціації або об'єктивованих типів асоціацій, які мають активного учасника, і не втручаються синтаксично або під час деривації.

«Сильний» тип зв'язку для об'єкта-учасника асоціації також не заважають використанню автономних «типів об'єктів». Ці два доданих структурних елемента можна комбінувати. На рисунку 2.25, а показано приклад телефонної книги з використанням автономних типів об'єктів для «Person» та «Telephone». Результат для автономних типів об'єктів залишається таким же, як і раніше. Ті автономні типи об'єктів, які вже є частиною компонента узгодженості, залишаються такими, як вони є, як у цьому прикладі типу об'єкта «Person». Ті автономні типи об'єктів, які були скасовані раніше, тепер стають

власним компонентом узгодженості - як у цьому прикладі «типу об’єкта» «Telephone».

Об’єктно-реляційний результат цього остаточного прикладу зображено на рисунку 2.2 5b. Модель надає посилання всередині вкладеної таблиці. Об'єктно-реляційна реалізація Oracle не дозволяє створювати таку конструкцію. Однак це є допустимим в інших парадигмах персимтентності, таких як документо-орієнтовані БД, де може бути посилання на інший документ у списку значень. Отже, алгоритми деривації повинні подбати про такі випадки. Це не повинно впливати на концептуальну модель, оскільки семантика є достовірною та чіткою.




(а) Концептуальна модель із знаками деривації, які враховують «сильний» тип зв’язку для об’єкта-учасника асоціації та автономний «тип об’єкта»

(б) Об'єктно-реляційна модель

Рисунок 2.25 - Комбіноване використання автономного типу об’єкта та активна участь.

Таким чином, введення «сильного» типу зв’язку для об’єкта-учасника асоціації надає можливість виразити семантичну близькість типу асоціації до типу об’єкта та враховують особливості рішень щодо логічних структур будь-якої парадигми персистентності.

## 2.3. Висновки до другого розділу

У другому розділі отримано наступні результати:

1. Показано що загальна ідея AGILA MOD, як автоматичного генератора для інформаційного представлення, логічних структур БД, полягає в тому, щоб

створити концептуальну модель шляхом перекладу природної мови, як речень, які семантично не спрощуються, в графічну нотацію. Основні структурні елементи є об’єкти та асоціації між ними , запропановано їх формальне представлення.

2. На прикладі концептуальної моделі телефонної книги (рис. 2,а) побудованої за допомогою існуючих структурних елементів AGILA MOD розглянуті можливості фреймворку AGILA щодо автоматизованої деривації логічних моделей даних. Показано, що існуючий синтаксис мови AGILA MOD має недоліки щодо визначення та відображення автономних (Standalone) об'єктів, каскадного видалення та оновлення (Cascading), а також вкладених структури (Nested Structures)

3. З врахуванням знайдених недоліків запропоновано удосконалити синтаксис мови AGILA MOD (структуру концептуальної моделі) за рахунок введення двох структурних елементів: незалежний об’єкт (*autonomous*) та «сильної» участі об’єкта в асоціації. Незалежний об’єкт виражає, що екземпляри описаного елемента ПрО існують незалежно від його відношень іншими елементами. Сильна участь об’єкта в асоціації означає що асоціація сильніше пов’язана із підключеним об’єктом з межами участі в асоціації «x: 1» ніж з рештою. Для деривації із концептуальної моделі логічної моделі в фрейморку AGILA використовують наступний сценарій:

– один тип об’єкта всередині компонента узгодженості позначається як ім’я відношення;

– усі інші типи об’єктів усередині відношення стають ідентифікуючими атрибутами;

– усі ролі типу асоціації, що лежать усередині компонента узгодженості, які вказують на тип об’єкта поза компонентом узгодженості, стають атрибутом відношення;

– ролі цього типу, які є результатом позначеної сильної участі з верхньою межею, більшою за 1, інкапсулюються як вкладені атрибути».

При цьому останній пункт зазначеного сценарію виконується тільки в разі деривації концептуальної моделі в об’єктно-реляційну модель даних на логічному рівні.

Таким чином виконане друге завдання дисертаційного дослідження та отримано перший пункт наукової новізни, а саме удосконалено структуру концептуальної моделі даних за рахунок розробки незалежного типу об'єктів та введення для об'єктів «сильної» участі в асоціації, що дало можливість вдосконалити правила деривації логічних моделей з урахуванням багатоваріантної персистентності даних.

.

# РОЗДІЛ 3
# МЕТОДИ АВТОМАТИЗАЦІЇ РОЗРОБКИ ЛОГІЧНИХ МОДЕЛЕЙ БАЗ ДАНИХ ЗІ БАГАТОВАРІАНТНОЮ ПЕРСИСТЕНТНІСТЮ

Раніш в роботі було показано що поширення баз даних NoSQL та концепція багатоваріантної персистентності зробили процес проектування бази даних із їх використанням ще більш довготривалим та трудомістким особливо на концептуальному рівні. Для скорочення часу розробки було вдосконалено структуру концептуальної моделі за рахунок додавання спеціалізованих елементів. Це дозволило запропонувати сценарії деривації концептуальної моделі у відповідну реляційну або нереляційну логічні моделі  В цьому розділі процес проектування бази даних за Ельмасрі / Навате розглядається з урахуванням багатоваріантної персистентності, створено методи автоматизації розробки логічних моделей баз даних на основі концептуальних моделей зі багатоваріантною персистентністю.

## 3.1 Процес проектування бази даних з врахуванням багатоваріантної персистентності

Як зазначалося раніше, концептуальна модель при відображені об’єктів ПрО розглядає вимоги до даних якими займається проект її структура не залежить від особливостей персистентності даних на логічному та фізичному рівнях. Крім того представлення моделі має бути якомога простішим, щоб усі зацікавлені сторони проекту могли легко зрозуміти зміст, використовуючи модель як загальну основу  для обговорення особливостей проекту. Перераховані особливості концептуального моделювання не повинні змінюватись при додавання умов багатоваріантної пресистентності (polyglot persistence) до процесу проектування бази даних. Все це дозволить автоматизувати процес проектування баз даних за Ельмасрі / Навате та врахувати концепцію багатоваріантної пресистентності.

Адаптований до класичного (див. рис. 1.1) процес проектування БД представлено наступним чином (3.1):

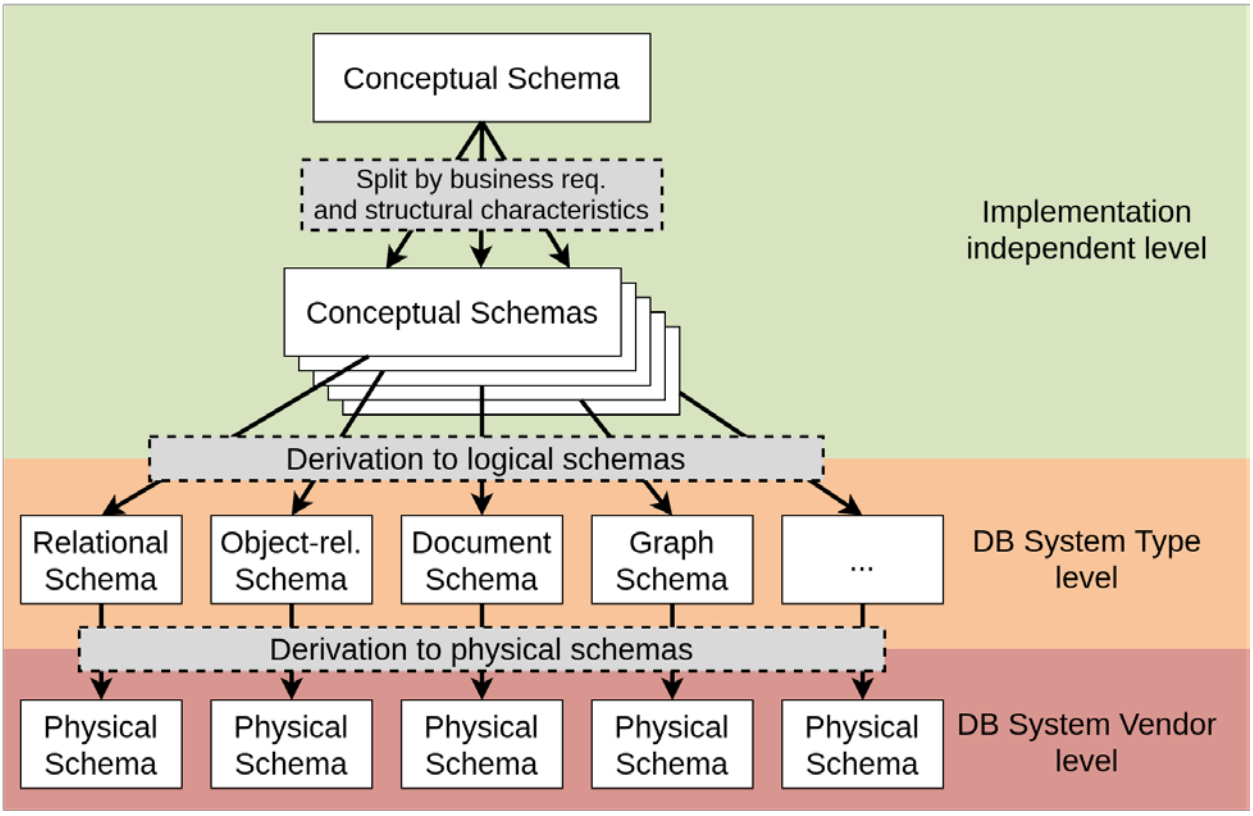



Рисунок 3.1 - Процес проектування бази даних з врахуванням polyglot persistence

1. На першому етапі створення комплексної концептуальної моделі ПрО відбувається за допомого засобів розширеної мови AGILA MOD+, як описано в розділі 2.

2. На першому етапі виконується додатковий крок – розділення комплексної концептуальної моделі на незалежні концептуальні субмоделі, так звані «розділи» концептуальної моделі для подальшої їх деривації у відповідні логічні моделі, які враховують особливості персистентності даних на логічному рівні.

3. На другому етапі отримані концептуальні субмоделі послідовно перетворюється за допомогою алгоритму виведення у відповідні датологічні моделі – «логічні схеми», вигляд яких залежить від типу персистентності даних, який підтримується засобами конкретної СУБД.

4. На третьому етапі процесу проектування БД відповідні датологічні моделі послідовно перетворюються у відповідні фізичні моделі (схеми) – внутрішні структури персистентності конкретної СУБД.

Для методичної підтримки процесу проектування БД на першому етапі концептуального моделювання розроблено метод розділення комплексної концептуальної моделі ПрО на незалежні частини – субмоделі

## 3.2 Метод розділення комплексної концептуальної моделі даних на субмоделі

З врахуванням розширеної мови концептуального моделювання AGILA MOD+ та правил деривації логічних моделей даних запропановано метод розділення комплексної концептуальної моделі ПрО на незалежні субмоделі – так звані «розділи» концептуальної моделі для подальшої їх деривації у відповідні логічні моделі. Метод включає такі етапи:

1. Створення копії існуючої комплексної концептуальної моделі
2. Оточення елементів концептуальної моделі («типів об’єктів», «типів асоціацій» та ліній зв’язків між ними) із яких створюється субмодель замкненою, суцільною напівпрозорою оболонкою
3. Видалення усіх «типі асоціацій» та підключених до них ліній зв’язків, що лежать зовні оболонки оточення.
4. Видалення усіх «типів об'єктів» системних та базових типів, що лежать зовні оболонки оточення, і не мають з'єднання «типами об'єктів» що належать до внутрішньою частини оточення.
5. Залишення усіх «типів об'єктів» системних та базових типів що лежать зовні оболонки оточення, і мають з'єднання «типами об'єктів» що належать до внутрішньої частини оточення.
6. Заміна усіх залишених типи об'єктів, що лежать зовні оболонки оточення і мають з'єднання всередині на тип «посилання» (referencing) на розділ моделі, до якого належить оригінальний «тип об'єкта».

Таким чином, в результаті розділення комплексної концептуальної моделі маємо три множині елементів (об’єктів, асоціацій та ліній зв’язків між ними): «inside», «outside» та «remainder».

До множини «inside» належать елементи оточені оболонкою на другому кроці, до «outside» – елементи які було видалено на третьому та четвертому кроці.

Множина «remainder» складається із об'єктів що мають тип «посилання», лежать зовні оболонки оточення, і мають з'єднання з об'єктами, що належать множини «inside».

Формально відношення зазначених підрозділів комплексної моделі виглядає наступними чином:

$$MOD = \left(\{o_i, a_i, o_j\}_{in} \cup \{o_i, a_i, o_j\}_{out}\right) - \left(\{o_i, a_i, o_j\}_{in} \cap \{o_i, a_i, o_j\}_{out}\right)_{rem} \quad (3.1)$$

Процес розділення комплексної концептуальної моделі згідно запропонованого методу запускається стільки раз скільки незалежних концептуальних субмоделей необхідно отримати.

Також допускається, що оточення перекриваються, що призводить до множення неодноразово оточених елементів, оскільки вони з’являються у кожній субмоделі, оболонкой якою вони були оточені.

Для елементів синтаксису, не згаданих в описі, в контексті процесу розділення застосовується наступне:

– Базові типи об'єктів та типи переліку (див. рис.2.8,а)діють як системні типи.

– Об'єктивізовані «типи асоціацій» (див. рис. 2.5) повинні бути віднесені до конкретного кроку за роллю, яку вони відіграють у відповідній ситуації, тобто якщо об'єктивізований тип асоціації, яка належить до множині «outside», підключений як тип об'єкта що належить до множини «inside» то до прямокутника об’єктивізації, необхідно використовувати правило кроку 6. У всіх інших випадках об'єктивізований тип асоціації можна видалити.

– Для типів об'єктів «успадкування» (див. рис.2.10) та (2.6), тип об’єкта посилання завжди повинен бути створений по відношенню логічного оператора, який належить до «inside». Всі «типи об'єктів», підключених до логічного оператора, що належать «outside», потрібно замінити типом об'єкта, посилання згідно правилу кроку 6. Якщо немає логічного оператора тобто є один підтип, тоді в субмоделі, що містить підтип, потрібно створити тип об'єкта посилання, як у кроці 6.

– Тип набору об'єктів – усі типи об'єктів, підключені до типу набору, що лежить в іншої субмоделі ніж набір, потрібно обробляти, як у кроці 6.

Апробацію запропанованого методу розділення комплексної концептуальної моделі даних на незалежні субмоделі показано на прикладі проекту KampfrichterAnmelde-System (KRAS) – веб-додатку, який було створено для інформаційної підтримки Баварської Федерації дзюдо показано в п’ятому розділі даної дисертаційної роботи (див. рис. 5.2-5.5). Подальше проведення досліджень показало, що було отримано скорочення в 2,3 рази часу на таке проектування двох БД в порівнянні з їх розробкою за класичною схемою.

Таким чином, вперше розроблено метод розділення комплексної концептуальної моделі даних на незалежні субмоделі, що дозволило автоматизувати отримання відповідних логічних моделей даних включаючи реляційні та NoSQL структури та тим самим скоротити час проектування баз даних.

Далі розглянуті питання про специфічні причини розділення концептуальної моделі та про побудову так званої оболонки – оточення елементів концептуальної моделі.

## 3.3 Взаємоьязок функціоналіних вимог та вимог до даних

Як відомо аналіз вимог (Requirements Engineering) [60,61] – частина процесу розробки програмного забезпечення (ПЗ), що включає в себе збір

вимог до ПЗ, їх систематизацію, виявлення взаємозв'язків, а також документування). Цей процес регламентується стандартом ISO [62]. А рішення щодо налаштування структур даних базуються на визначених вимогах до ПЗ. Таким чином, метод прийняття рішення про розподіл комплексної концептуальної моделі даних повинен розглядатися як невід'ємна частина аналізу вимог.

В теорії вимоги розподіляють на функціональні та нефункціональні [60, 61]. Але саме коректне визначення функціональних вимог має вплив на структуру даних – особливо можливе розділення моделі даних. З урахуванням багатоваріантної персистентності, набір функціональних вимог проекту тепер може призвести не тільки до однієї конкретної технології персистентності, але і до кількох. Коли деякі системні частини мають суперечливі вимоги, системний інженер тепер має можливість розділити модель даних, як описано, замість того, щоб змушувати знаходити компроміси щодо використання структури даних. Отже, ретельний аналіз вимог проекту з точки зору багатоваріантної персистентності включає методологію вибору чітко структурованого та прозорого способу розподілу концептуальної моделі.

Функціональні вимоги визначають, що повинна робити система [63], згідно ISO / IEC / IEEE 24765: 2017: Функціональні вимоги – твердження, яке визначає, які результати повинен дати продукт або процес; вимога, яка визначає функцію, яку повинна виконувати система або компонент системи. [62]

З іншого боку, згідно з [11, гл. 3.1] на рисунку 3.2 видно, що створення концептуальної моделі в якості специфікації «у максимально детальній та повній формі» так званих «вимог до даних», які були вилучені із збору та аналізу вимог раніше. Тут Elmasri / Navathe створюють зв'язок між функціональними вимогами та вимогами до даних, що є результатом згаданого збору та аналізу вимог. Цей зв'язок дуже чітко видно на рисунку 3.2.

При об'єднанні цих двох точок зору, порівняно з наведеними вище визначеннями щодо функціональних вимог, вимоги до даних, як вони

згадуються в [11], визначають ні то, що «система повинна робити», а то, що «система повинна зберігати».

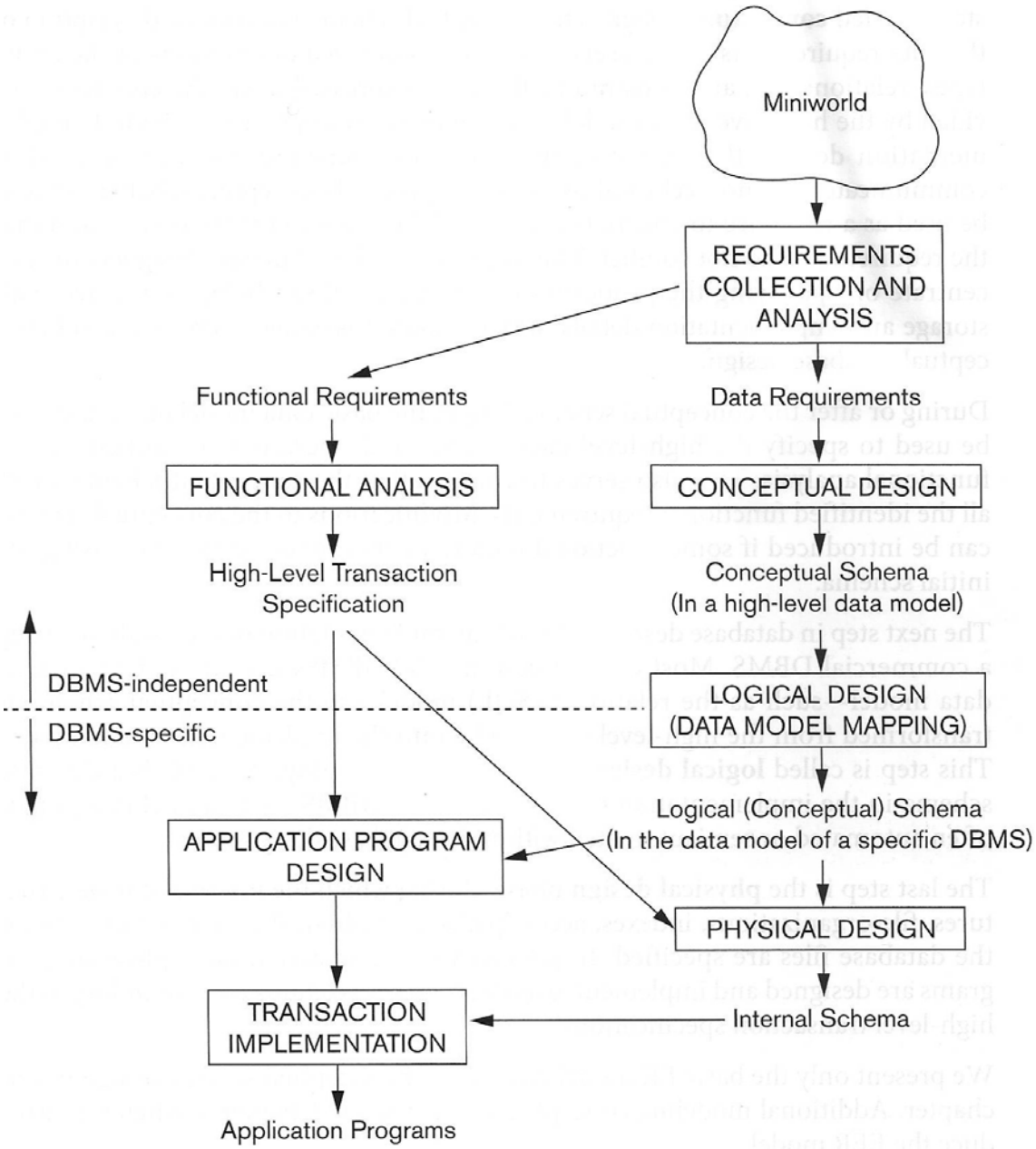



Рисунок 3.2 – Основні етапи проектування бази згідно [11, рис. 3.1, с. 91]

З цієї точки зору, вимоги до даних можна сприймати як зворотню сторону медалі функціональних вимог, що концентруються на структурних аспектах, які повинні бути на місці, щоб система могла забезпечити необхідні функції. Рисунок 3.6 чітко підтримує цю точку зору. Отже, розглядаючи фактори впливу функціональних вимог на розділення концептуальної моделі даних, робиться висновок, що *сама концептуальна модель є результатом аналізу функціональних вимог*.

Підсумовуючи, збір та аналіз функціональних вимог призводить дозволяє сформулювати вимогі до даних, які, в свою чергу, знайшли своє відображення у концептуальній моделі даних. Це не означає, що функціональні вимоги або вимоги до даних не впливають на розподіл концептуальної моделі даних. Але сама концептуальна модель даних повинна містити всю інформацію щодо структурних характеристик проекту. Це призводить до висновку, що концептуальна модель даних, яка була створена на основі самого аналізу вимог, повинна бути досліджена на структурну значимість, яка вказує на використання певної структури даних для реалізації. Ці структурні особливості будуть розглянуті далі.

## 3.4 Метод створення незалежних концептуальних субмоделей

Як вже зазначалося раніше, концептуальна модель даних є результатом аналізу вимог і описує структурні потреби в проектних даних шляхом трансляції речень, які семантично не спрощуються, в графічну нотацію. Сама концептуальна модель не вказує, яку структуру даних або базу даних слід використовувати. Метою концептуального моделювання є повне та детальне відображення структурних аспектів ПрО.

Однак іманентність зображених структур може включати певні ознаки, що вказують на використання певної структурної реалізації. У будь-якому випадку, ці вказівки матимуть лише структурний характер через те, що концептуальна модель також відображає лише структури даних проекту. У свою чергу, структурні аспекти дадуть показники логічної структури даних, яку рекомендується використовувати при реалізації.

Таки висновки не є новими. Само поняття багатоваріантної персистентності включає припущення, що IT-розробники користуючись власним досвідом вирішують, яку логічну структуру даних вони воліють використовувати для деривації логічної моделі від попудованиї

концептуальної. Необхідно зазначити, що досвід базується на аналізу дяких структурних показників.

Таким чином для побудови оболонки (див. Рис 5.3) – оточення елементів комплексної концептуальної моделі запропановано метод створення концептуальних субмоделей, які враховують особливості персистентності даних. Метод предбачає проведення аналізу структурних обобливостей множині елементів комплексної концептуальної моделі, а саме об’єктів, асоціацій та ліній зв’язків між ними:

1. Наявність сильних зв’язків між обьєктом та асоціацією, які запропановані в AGILA MOD+, є показником віднесення цих елементів до документо-орієнтованих або об’єктно-реляційних структур персистентності

2. Наявність ієрархії типів об’єктів є показником віднесення цих елементів до документо-орієнтованих або об’єктно-реляційних структур персистентності. За допомого ієрархії типів об’єктів AGILA MOD підтримує успадкування об'єктів..

3. Для концептуальних моделей або їх частин, що містять значну кількість успадкувань, особливо коли успадкування є складним із використанням довільних комбінацій “OR”, “XOR” та “AND” доцільно використовувати документно-оріентовані структури персистентності XML або JSON.

4. Наявність великої кількості асоціацій типу «Самоасоціація» (Self Association) коли межи зв’язків обьєктів-учаників дорівнюють "0: *" є показником віднесення цих елементів до графо-орієнтованих структур персистентності

Розглянемо кроки методу на конкретних прикладах деривації в логічні структури даних

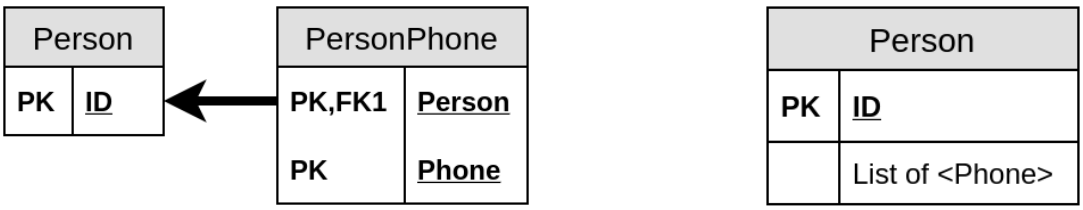
*Сильні зв'язки*

В AGILA MOD +, поняття сильних зв’язків було додано до примітивів синтаксису. Сильний тип звязку об’єкта-учасника асоціації означає що асоціація сильніше пов’язана із підключеним об’єктом з межами участі в

асоціації «x: 1» ніж з рештою В результаті тип асоціації стане частиною компонента узгодженості, до якого належить сильно пов'язаний тип об'єкта. Рисунок 3.3 є прикладом такого зв’язку. Там зазначено, що телефонні номери, пов’язані з особою, сильно пов’язані з цією особою.


Рисунок 3.3 – AGILA MOD + концептуальна субмодель з відображенням сильного типу зв'язку

При деривації такої моделі до простих реляційних структур цей сильний зв’язок не може бути врахований. Під час виведення до реляційних структур компонент узгодженності стає відношенням. Дотримуючись вищезазначеного принципу сильних зв’язків, тип асоціації «PersonPhone» повинен стати частиною компонента узгодженості «Person», що, веде до необхідності формування багатозначного атрибуту. Оскільки реляційна структура підтримує лише скалярні атрибути, сильний зв’язок тут повинен розглядатися як звичайний зв’язок. Результат можна побачити на рисунку 3.4а.




(а) Реляційна модель (б) Об'єктно-реляційна модель

Рисунок 3.4 - Порівняння деривації в реляційні та об'єктно-реляційні структури.

Той факт що використання реляційної структури не дозволяє коректно відобразити сильні зв'язки, слід розглянути деривацію такої концептуальної субмоделі в документно-орієнтовані або об'єктно-реляційні структур даних, оскільки вони можуть мати багатозначні атрибути. Деривація в об'єктно-реляційну модель призведе до простого відношення, яке містить ідентифікатор особи «ID» та атрибут «List of Phone», який містить усі телефонні номери

людини (рис. 3.4,b). Цей результат деривації є набагато простішим, т.к не вимагає доступу до іншої таблиці, як у випадку деривації в реляційну модель.

В випадку деривації в документо-орієнтовану структуру, основану, наприклад, на використанні, розширюваної мови розмітки (eXtensible Markup Language - XML) отримаємо наступну схему XML (рис. 3.5).

```
<element name="PersonList">
  <complexType>
    <sequence>
      <element name="Person" maxOccurs="unbounded">
        <complexType>
          <sequence>
            <element name="ID" type=" string " />
            <element name="PhoneList">
              <complexType>
                <sequence>
                  <element    name="Phone" type="string"
                          maxOccurs="unbounded" />
                </sequence>
              </complexType>
            </element>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
  <key name="personId">
    <selector xpath="ID" />
    <field    xpath="Person" />
  </key>
</element>
```

Рисунок 3.5 – Приклад дерівації сильних зв'язків в документо-орієнтовану структуру (схему XML)

У цьому прикладі елемент «Person» отримує елемент «PhoneList», що містить незв'язане входження елементів «Phone». За замовчуванням мінімальний показник - "1". Отже, інтервал "1: *" на рисунку 3.7 є вже включеним.

Таким чином, існування сильних зв'язків у концептуальній моделі є чітким показником необхідності використовувати для деривації документо-орієнтовані або об'єктно-реляційні структури в перевагу над чистими реляційними структурами.

*Ієрархії типів об'єктів*

Іншим показником, що вказує на певну логічну структуру даних, є ієрархії типів об’єктів. AGILA MOD підтримує успадкування типів об'єктів класичним об'єктно-орієнтованим способом. Такого роду структури даних є серйозною проблемою для реляційних баз даних. Ця тема широко обговорюється в роботах [29] та [30]. Як там вже згадується, насправді існують різні підходи до відображення успадкованих структур до реляційних структур даних. Але всі вони мають певні недоліки.

На рисунку 3.6 показаний приклад, який використовує успадковані структури для моделювання різних типів об’єкта «Person», який може бути студентом, професором або працівником. Кожен студент повинен бути або студентом, або аспірантом, тоді як кожен працівник повинен бути викладачем, працівником адміністрації або тим і іншим.

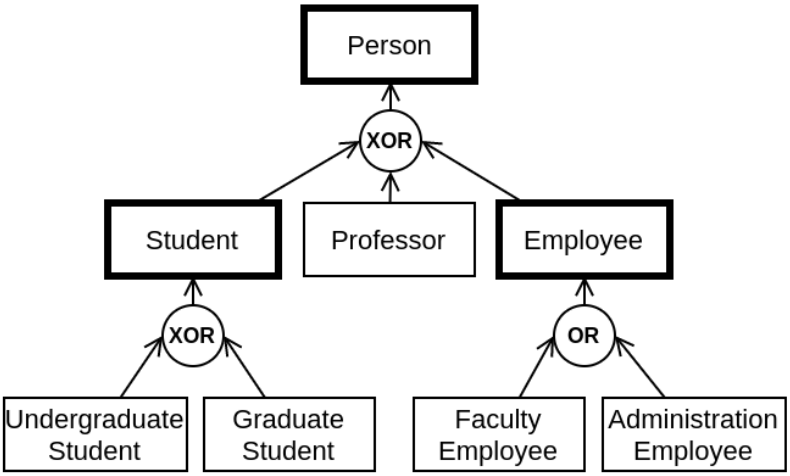



Рисунок 3.6 – Приклад концептуальної субмоделі «успадкування»

Одна з можливостей деріваці цієї концептуальної субмоделі в реляційну структуру зображена на рисунку 3.7. Отримана реляційна модель гарантує, що кожен запис підпорядкованого відношення правильно посилається на батьківський тип. Наприклад, аспірант посилається на «студента» на «особу». Але реляційна модель не може гарантувати, що студент не буде одночасно доданий як працівник або професор. Тобто реляційна структура на рис. 3.7 не можуть гарантувати, що кожна особа матиме посилання на запис лише в одного студента, професора чи працівника. Щоб забезпечити це, у базу даних потрібно

впровадити тригер, який ускладнює впровадження та, можливо, перешкоджає технічному обслуговуванню. Більше того, досить великий обсяг взаємозв`язків, який породить ця деривація, також погіршить доступ до даних, створюючи додаткові звернення до декількох баз даних лише для того, щоб отримати всю інформацію однієї особи.

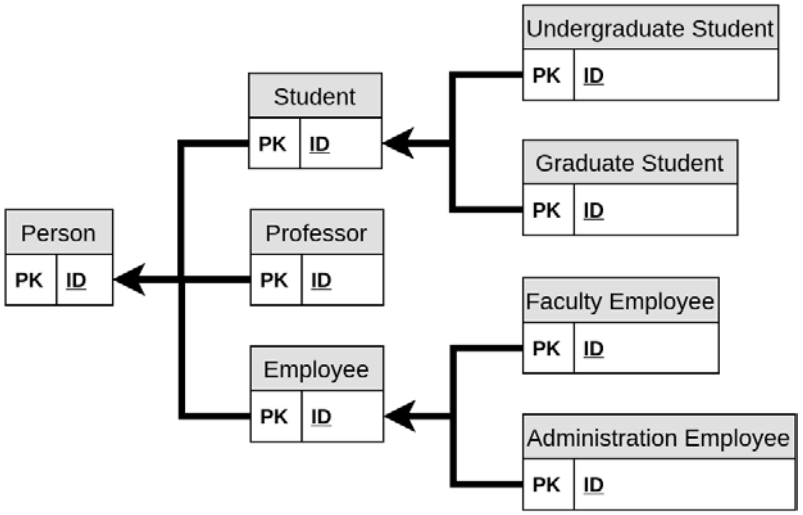



Рисунок 3.7 – Деривація успадкування в реляційну модель

Звичайно, є інші можливості для виведення реляційних структур, як наприклад звести все до одного співвідношення. Однак це створює інші проблеми: під час додавання запису потрібно бути впевненим, що вводяться лише зручні атрибути для конкретного типу особи.

Це може стати досить складним завданням із збільшенням кількості типів в ієрархії. Більше того, посилання на певний тип, наприклад, студент також стане більш складним завданням, коли все буде в одному відношенні.

На відміну від деривації концептуальної субмоделі в реляційну структуру вибір документо-орієнтованої структури (XML) є набагато простішим. У XML усі необхідні обмеження можна занестиу файл, який можна використовувати для перевірки записів. На рисунку 3.8 показано визначення типу студента. Тут тип студента відразу визначається як абстрактний. Успадкування виражається як "розширення" до "personType". Типом студента також може бути лише аспірант або студент.

На рисунку 3.9 показано випадок успадкування «OR» щодо співробітників.

```
<s:complexType name=”studentType” abstract=”true”>
  <s:complexContent>
    <s:extension  base=”personType”>
      <s:sequence>
        <s:element                        name=”studyProgram”
  minOccurs=”1”
                  type=” s:string”      maxOccurs=”1” />
        <s:element                        name=”semester”
  minOccurs=”1”
                  type=”s:positiveInteger”   maxOccurs=”1” />
        <s:element                        name=”enrolmentNo”
  minOccurs=”1”
                  type=”s:positiveInteger”   maxOccurs=”1” />
      </s:sequence>
    </ s:extension>
  </s:complexContent>
</s:complexType>
```

Рисунок 3.8 – Приклад деріваціі успадкування  в схему XML для визначення типу студента

```
<s:complexType name=”employeeType”>
  <s:complexContent>
    <s:extension base=”personType”>
      <s:sequence>
        <s:element   name=”entry”      minOccurs=”1”
                     type=” s:date ”   maxOccurs=”1” />
        <s:element   name=”wage”       minOccurs=”1”
                     type=”moneyType”  maxOccurs=”1” />
        <s:choice>
          <s:sequence>
            <s:group ref=”administrationEmployee”
                     minOccurs=”1”     maxOccurs=”1” />
            <s:group ref=”facultyEmployee”
                     minOccurs=”0”     maxOccurs=”1” />
          </s:sequence>
          <s:group   ref=”facultyEmployee”
                     minOccurs=”1”     maxOccurs=”1” />
        </ s:choice>
      </s:sequence>
    </ s:extension>
  </s:complexContent>
</s:complexType>
```

Рисунок 3.9 – Приклад деріваціі успадкування «OR» для співробітників в схему XML

Таким чином, доцільно використовувати документо-орієнтовані структури такі як XML або JavaScript Object Notation (JSON) – для деріваціі концептуальних субмоделей, які містять значну кількість успадкувань,

особливо коли успадкування є складним із використанням довільних комбінацій “OR”, “XOR” та “AND”.

*Тип асоціації Самоасоціація* (Salient Association Type Structure)

Приклад коли тий самий «тип об'єкта» приймає участь більше одного разу в «типі асоціації» (Див. рис. 2.3) теж є показником для надання переваги певним логічним структурам. На рисунку 3.10 наведено приклад концептуальної субмоделі адміністрування веб-сторінок, де статті можна розміщувати на сторінці, де люди зможуть «лайкнути» сторінку, виконувати певні дії за допомогою певних інструментів та бути частиною груп користувачів. У цьому прикладі помітно, що значення меж участі в асоціації дорівнюють "0: *". Таким чином, кожен «тип об’єкта» бере участь в «Самоасоціації». Кожен екземпляр може брати участь у існуючих типах асоціацій, довільно, як зображено.

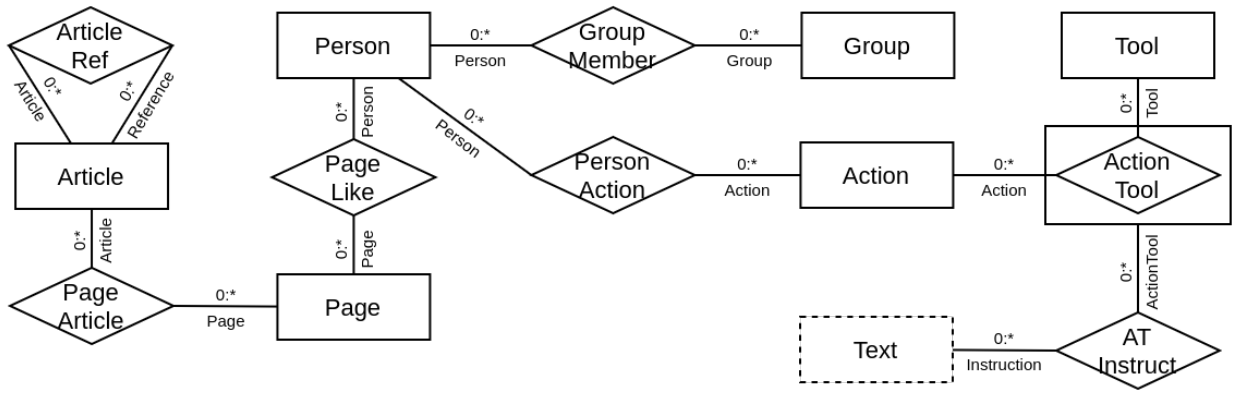



Рисунок 3.10 - Приклад концептуальної субмоделі, щодо адміністрування веб-сторінок

При дерівації субмоделі (рис. 3.10) в реляційну логічну модель, результатом буде наявність взаємозв`язків із великою кількістю обмежень зовнішнього ключа (рис. 3.11). Показано тринадцять взаємозв`язків і стільки ж зовнішніх ключів. Тобто отримано складну логічну структуру для простої концептуальної субмоделі. Її використання в бізнес-логіці, не є ефективним через безліч дрібних елементів та посилань, які потрібно враховувати. Тобто наявність того факту, *що багато невеликих об’єктів потрібно пов’язати між собою,* вказує на можливість використання графо-орієнтованих логічних структуру для дерівації субмоделі на рисунку 3.10.

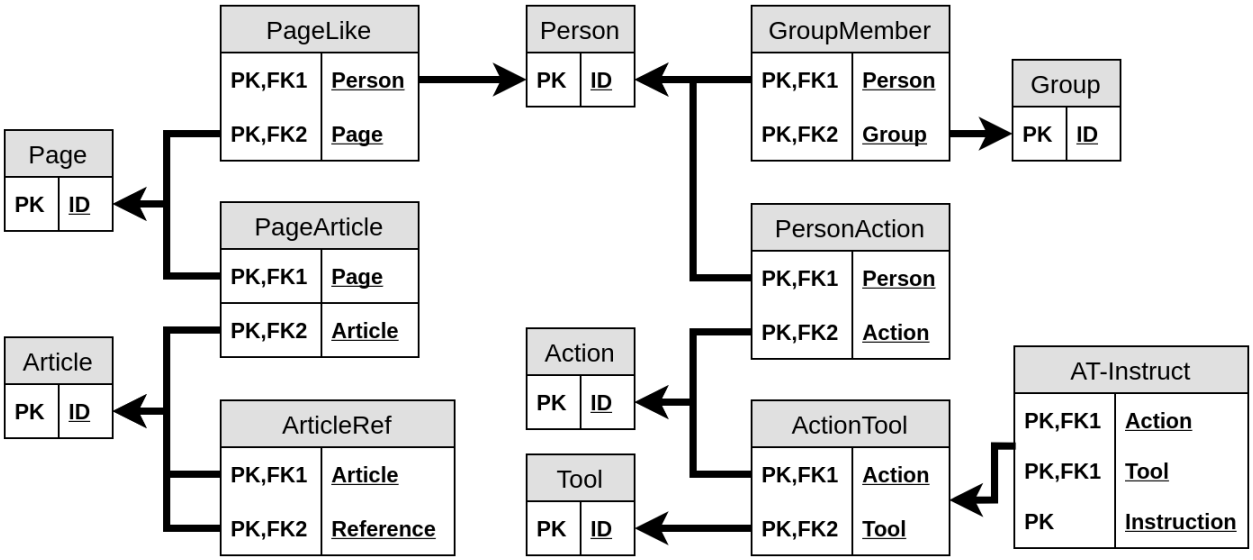



Рисунок 3.11 – Результат деріваційї концептуальної субмоделі, щодо адміністрування веб-сторінок в реляційну логічну модель

В графо-орієнтованих логічних моделях кожен «тип об’єкта» стає вузлом певного типу – тобто. «Person», «Article», «Page» та інш. Кожна асоціація може стати зв`язком (ребром) певного типу. Для об'єктивізованого «типу асоціації» (див. рис 2.5) «Action Tool» результат деріваційї залежить від можливостей графо-орієнтованої логічної моделі. Наприклад ArangoDB (пор. [62]) ребро може бути початком або ціллю іншого ребра, тоді об'єктивізована асоціація «Action Tool» стає ребром, яке вказує на довільну кількість текстових елементів через ребро «Instruction». Інакше ці асоціації повинні стати вузлом, який вказує рівно на один вузол «Tool» і рівно на один вузол «Action», а через ребро «Instruction» на довільні текстові елементи.

Апробацію запропанованого методу створення незалежних концептуальних субмоделей показано далі як і для поперднього метода на прикладі проекту KRAS (див. підрозділ 5.2). Дослідження показали, що було отримано скорочення в 2,3 рази часу на запропановане проектування двох БД в порівнянні з їх розробкою за класичною схемою.

Таким чином отримано третій пункти накової новизни а саме:

– вперше розроблено метод створення концептуальних субмоделей, які враховують особливості персистентності даних включаючи реляційні та

NoSQL структури із комплексної концептуальної моделі даних, що дозволило обґрунтовано вибрати тип структур даних на логічному рівні і тим самим зменшити час розробки баз даних з урахуванням багатоваріантної персистентності.

Врахування удосконаленої в другому розділі структури концептуальної моделі даних та розроблених в третьому розділі методів автоматизації розробки логічних моделей баз даних дозволило отримати четвертий пункт наукової новізни:

– удосконалено мову концептуального моделювання AGILA MOD+ за рахунок за рахунок удосконаленої структури концептуальної моделі даних та методів створення моделей даних концептуальному рівні, що дозволило враховувати багатоваріантну персистентность та скоротити час проектування баз даних інформаційних систем.

## 3.3 Висновки до третього розділу

У третьому розділі отримано наступні результати:

1. Запропановано в процесі проектування БД з урахуванням polyglot persistence на першому етапі розробити комплексну концептуальної моделі ПрО за допомого засобів розширеної мови AGILA MOD+, а потім розділити її на незалежні концептуальні субмоделі для подальшої їх деривації у відповідні логічні моделі, які враховують особливості персистентності даних на логічному рівні.

2. Показано, аналіз вимог (Requirements Engineering) з урахуванням polyglot persistence включає в себе збір вимог до программного забезпечення, їх систематизацію та прийняття рішеннь щодо налаштування структур даних базуються на визначених вимогах до ПЗ. Таким чином, метод прийняття рішення про розподіл комплексної концептуальної моделі даних на незалежні субмоделі повинен розглядатися як невід’ємна частина аналізу вимог.

3. Для методичної підтримки процесу проектування БД на першому етапі концептуального моделювання розроблено метод розділення комплексної концептуальної моделі даних на незалежні субмоделі, що дозволило автоматизувати отримання відповідних логічних моделей даних включаючи реляційні та NoSQL структури. На прикладі IT-проекту KRAS показані можливості розробленого методу, що до сторочення часу проектування БД.

4. Для побудови оболонки – оточення елементів комплексної концептуальної моделі для її розділення запропановано метод створення концептуальних субмоделей, які враховують особливості персистентності даних. Метод предбачає проведення аналізу структурних об’єктів комплексної моделі на наявність сильних зв’язків між обьєктом та асоціацією, ієрархії типів об’єктів, значної кількісті успадкувань, особливо із використанням довільних комбінацій “OR”, “XOR” та “AND”, а також великої кількості асоціацій Self Association.

5. Врахування удосконаленої в другому розділі структури концептуальної моделі даних та розроблених методів автоматизації розробки логічних моделей баз даних дозволило удосконалити мову концептуального моделювання AGILA MOD+ для враховування багатоваріантної персистентності та скоротити час проектування баз даних інформаційних систем.

# РОЗДІЛ 4
# РОЗРОБКА ІНТЕРФЕЙСУ ПРИКЛАДНОГО ПРОГРАУМАННЯ ДЛЯ ДОСТУПУ ДО БАЗ ДАНИХ З БАГАТОВАРІАНТНОЮ ПЕРСИСТЕНТНІСТЮ

В четвертому розділі розроблено інтерфейс прикладного програмування (***a****pplication* ***p****rogramming* ***i****nterface API*) у вигляді опису способів (набір класів, процедур, функцій, структур або констант) для забезпечення доступу до більш ніж одного джерела із бізнес логіки, що в подальшому буде використоне програмістами при написанні додатків.

## 4.1 Аналіз вимог до API

При аналізі вимог необхідно зазначити, що розробка API спрямована на забезпечення уніфікованого та прямого доступу бізнес-логіки до використовуваних сховищ даних для застосувань спрямованих на обробку онлайн-транзакцій (OLTP). Має бути продемонстровано можливість та показано можливий спосіб однозначного доступу до даних для створення (create), читання (read), оновлення (update) та видалення (delete) тобто CRUD операцій з абстрагуванням від деталей реалізації структури даних. Високопродуктивні застосування та застосування з OLAP не є предметом розгляду, оскільки отримані додаткові вимоги та складність, які необхідно враховувати для бажаного рішення, виходять за рамки цієї роботи.

Огляд літературних та інтернет-джерел щодо аналізу можливостей звичайних фреймворків ORM [30, 31, 67,68] по організації обробки структур даних з багатоваріантною персистентністю дозволив сформулювати наступні вимоги до розроблювального API.

API повинен надавати можливість отримати данні у формі, яка підходить для конкретного варіанту використання та бажаного використання даних. У будь-якому випадку, структури із даними не повинні впроваджуватись та використовуватись як своєчасно відображені та підтримані джерелом даних – у

таких структурах ORM, як JPA, це називається з’єднаними сутностями (attached Entity). Натомість вони повинні бути простими об'єктами-носіями даних, що забезпечують унікальний, безпечний за типом та простий доступ до даних.

API повинен запобігати «мовним» розривам у бізнес-логіці, як це є в ORM коли мова, подібна до SQL, вбудована як рядок у код. Звичайно, такого розриву взагалі не можна уникнути. У певний момент запит на дані повинен бути перекладений мовою певного сховища даних, наприклад, SQL. Але такий розрив повинен бути інкапсульований всередині API і виконаний унікальним та безпечним способом. Створення операторів SQL, наприклад, просте об’єднання рядків та даних, які введені користувачем не можна сприймати як безпечне, оскільки це відкриває можливість вбудовування SQL. Замість цього необхідно примусово використовувати підготовлені оператори. Окрім цього, також слід уникати буквального використання імен елементів даних. Це може спричинити суттєві проблеми, коли відбувається рефакторинг структур даних. Лістинг, який показано на рисунку 4.1 демонструє перераховані проблеми

```
String badStmt = ″ SELECT name, birthday FROM person ″;
StringbetterStmt =  ″ SELECT ″ + Person.NAME.attr +
                  ″       , ″ + Person.BIRTHDAY.attr +
          ″ FROM    ″ + Person.TABLENAME;
```

Рисунок 4.1 – Способи створення оператора SQL на Java

В першому рядку присвоєння змінній “badStmt”, є простим оператором SELECT, як це може записано на будь-якій консолі SQL. Оскільки він не містить жодного динамічного вводу користувача, по сам оператор захищений від атак, таких як SQL ін’єкція. Однак при рефакторингу це може стати проблемою. Наприклад, перейменування атрибуту “birthday" до “birth date” потребує детального пошука у коді для його використання в будь-яких рядках, де слово “birthday” також може використовуватися в інших різних випадках. У гіршому випадку це помічається лише під час виконання, коли виникають повідомлення про помилки.

В другому рядку, присвоєння змінній “betterStmt” дозволяє уникнути цих проблем. Він використовує клас визначення (у цьому прикладі “Person”), який, можливо, був створений раніше. Він несе необхідну інформацію визначення. Перейменування атрибута в найкращому випадку може означати лише регенерування цього класу визначення, де поле “attr” потім несе нове ім’я поля. У гіршому випадку регенерування може призвести до зміни самого визначення класу, що призведе до помилок під час компіляції. Але виправлення цих помилок відбувається швидше, надійніше і менш стомлює користувачів, оскільки під час роботи помилок не виникає. Крім того, перший підхід є теж не дуже загальним.

Попит на інкапсуляцію розриву мови йде паралельно із вимогами безпечних за типом операцій на основі бізнес логіки. Системи типів мов програмування та систем баз даних досить різні і відрізняються своїм дизайном. Отже, це є важливою особливістю API доступу до даних, яка дозволяє компенсувати ці відмінності дозволяє бізнес-логіки безпечно впливати на дані в системі типів мови програмування, на якій вона написана.

Принцип інкапсуляції не повинен обмежуватися лише тими частинами доступу до даних, де трапляється розрив мови. Тобто усі операції доступу до даних повинні бути інкапсульовані всередині рівня доступу до даних. Цей рівень повинен контролюватися тими членами команди проекту, які спеціалізуються на даній темі. Отже, всі інші програмісти зобов'язані здійснювати пошук даних і маніпулювати ними тільки за допомогою виклику функцій цього рівня. Таким чином, рівень доступу до даних виконує роль фасаду в класичному розумінні відповідного паттерна проектування. [69, с. 185] Така організація API та результуючих шарів даних конкретного для проекту також підтримує ще одну важлива ціль в контексті polyglot persistence: в структурах рівня доступу до даних, побудованих на API, різниця між окремими системами баз даних, які використовуються стосовно обробки транзакцій може бути зменшено та приховано від реалізації бізнес логіки. Традиційний ACID-підхід до реляційних баз даних та BASE-підхід, якого

дотримується більшість представників NoSQL, сильно відрізняються. Розроблений API повинен обробляти це внутрішньо і повинен надавати відповідні можливості, особливо щодо питань посилань, які були згадані раніше.

Для забезпечення цього достатньо представити API як базовий фрейм, який можна розширити відповідно до потреб проекту. Основною задачею API є приховування складності налаштування polyglot та забезпечити необхідні операції, а також функціональні можливості для спеціалістів по даним, щоб ефективно реалізовувати операції з даними конкретного проекту, створюючи цілістний рівень доступу до даних для конкретного проекту. Таким API в першу чергу повинен обмежуватися інкапсуляцією доступу до даних та вбудовуванням даних в об’єктно-орієнтовані структури, без спроби примусово перетворити логічні структури даних в об'єктно-орієнтовану форму (Рис. 4.2)

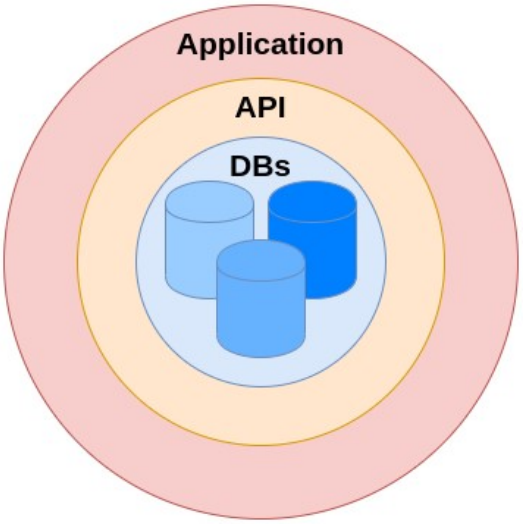



Рисунок 4.2 – Шар API, який призначений для інкапсуляції та ізоляції доступ до даних.

Отже, бажаний API повинен бути спроектований таким чином, щоб забезпечував можливості для реалізації рівня доступу до даних для конкретного проекту, створюючи модель, подібную до тої яка шаблоном проектування Model-View-Controller (MVC). [70, гл. 5.6] Крім того, API повинен бути спроектований таким чином, щоб змушувати проекти створювати вказаний рівень доступу до даних, де вся обробка даних є централізованою. Це

забезпечує ефективну та унікальну обробку даних, тоді як сама бізнес-логіка не містить цих проблем, а розробники можуть зосередитися на бізнес-кейсах.

Таким чином, дизайн API повинен враховувати наступні аспекти:

– API повинен бути розширюваним, щоб він міг стати основою рівня доступу до даних програмного проекту.

– Мовні розриви обробляються тільки в API, відповідно на рівні доступу до даних, побудованому на ньому.

– Відмінності між використовуваними системами баз даних повинні бути пом'якшені і приховані від бізнес-логіки.

– API створює фасад бізнес-логіки, інкапсулює всю обробку доступу до даних.

– API не діє як ORM. Об'єкти даних не підключені і не відслідковуються. Вони служать виключно для структурованої і типобезпечної обробки даних.

– Загалом, API повинен виступати в якості основи для проектів для побудови моделей у сенсі шаблона проектування MVC.

## 4.2 Характеристика структури розроблювального API

Концепція API основана API PHP Data Objects (PDO), який забезпечує простий, але загальний доступ до даних з реляційних баз даних без накладних витрат на звичайні ORM-фреймворки (пор. [71], [72, с. 286] і [73]) PDO описується як рівень абстракції, що забезпечує унікальний доступ до реляційних баз даних, подібних до Java Database Connectivity (JDBC). Однак він виходить за межі функціональних можливостей JDBC, надаючи можливість негайно упакувати результати вибору в об'єкти певного класу, який надається спільно з викликом. Дані заповнюються в об’єкти з використанням омонімії атрибутів об’єктів та повертаємих стовпців. Ці об'єкти є простими носіями даних і не мають зв'язку з джерелом даних під шаром даних, як це часто зустрічається в структурах ORM. Ідеї PDO були прийняти та отримали

подальший розвиток для застосування у випадку використання polyglot persistent.

Частини API, що стосуються підключення та обробки конкретних сховищ даних, були реалізовані для реляційної бази даних та полегшеного протоколу доступу до директорій/каталогів (Lightweight Directory Access Protocol LDAP), які є двома зразками представників сховищ даних з різною логічною структурою та технічною підготовкою. Вони були обрані відповідно до вимог практичного прикладу, який буде детально описано у п’ятому розділі. Звичайно, інші сховища даних також можна підключити, реалізувавши для них певні інтерфейси так само, як це було зроблено для згаданих двох типів..

Далі конкретні елементи API вказуються за допомогою діаграм класів.

### 4.2.1 Сутності (Entities)

Термін «Entities» використовується в більшості інших типів API. Отже, його слід використовувати також і для цього API, а також для надання певної орієнтації розробникам, які намагаються використати його у своєму проекті.

Сутність API повинна представляти структуру певної одиниці даних, яка зберігається в певному сховищі даних. Для реляційних баз даних це дорівнює таблиці, для документо-орієнтованої бази даних – документу, а для каталога LDAP – запису певного класу.

У будь-якому випадку «Сутність» у сенсі API транслює лише структуру конкретного блоку даних, щоб забезпечити безпечну обробку цих даних у бізнес логіці, а також загальну обробку всередині самого API. Отже, «Сутність» не тільки містить структуру блока даних, вона також несе інформацію, що стосується обробки даних, під час обміну даними із сховищем даних. Ця інформація повинна включати відображення атрибутів між API-сутностями та блоком сховища даних щодо назви атрибута, типу та конкретних обмежень (наприклад, є цей атрибут обов'язковим чи ні), а також більш загальну

інформацію про блок даних у цілому (наприклад, що ідентифікує екземпляр певного блоку даних і, що відрізняє його від інших екземплярів).

Щоб створити «Сутності» унікальним чином та обробляти їх в API було створено абстрактний клас Entity, а також абстрактні підкласи для конкретних сховищ даних. Крім того, створено абстрактний підклас PolyglotEntity, який може представляти частину даних, які концептуально належать друг другу (є цілостними), але розподілені між кількома сховищами даних через розділення концептуальної моделі, як було описано в третьому розділі.

Клас Entity містить загальні функції, які повинні забезпечувати всі сутності, незалежно від місця походження даних. Підкласи забезпечують подальшу функціональність, що залежить від конкретного сховища даних. Самі сутності повинні бути позбавлені будь-якої функціональності. Екземпляри сутностей повинні містити лише атрибути. Класи сутностей повинні містити визначення, як описано вище, яке є незмінним під час виконання.

Діаграма класів зображена на рисунку 4.3, а згадані функціональні можливості описані за допомогою зображених елементів.

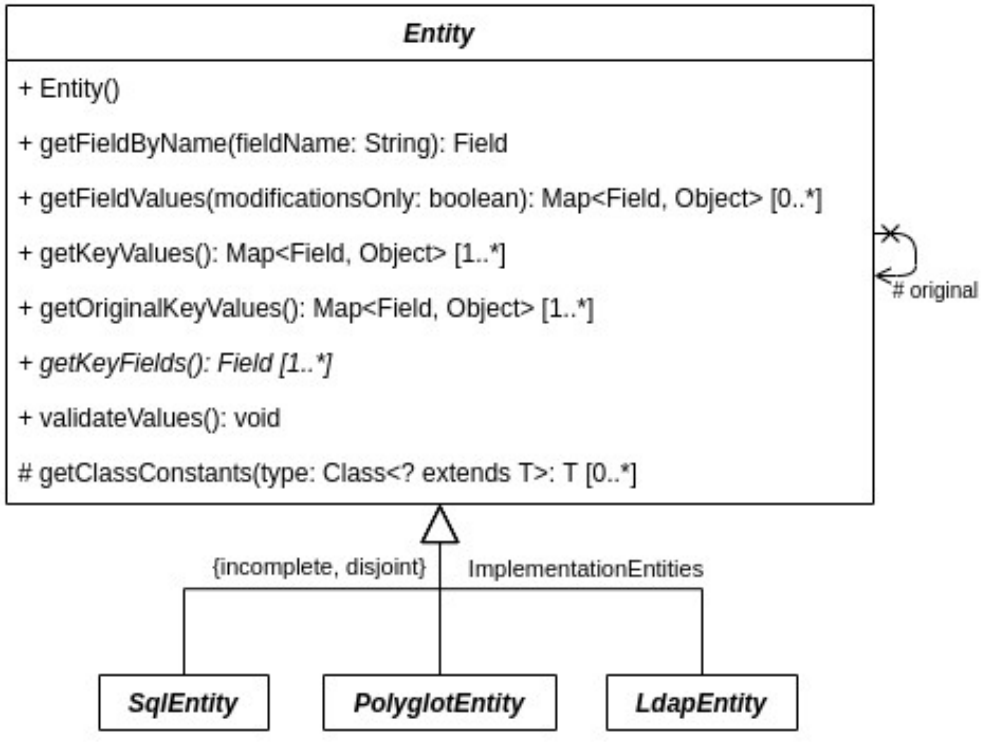



Рисунок 4.3 – Діаграма класів класу Entity та підкласів SqlEntity, PolyglotEntity, LdapEntity

**Original** Кожен екземпляр Entity має власний клон\копію, який представляє стан екземпляра таким, яким він був, коли він був завантажений із сховища даних. Оригінал атрибута захищений і тому доступ до нього можна отримати лише з entity instance.

**Entity** Кожен клас, який є підкласом Entity, повинен мати пустий конструктор. Це важливо для підходу всередині API, який обробляє різні сутності в загальному вигляді. Тому йому потрібно створювати нові екземпляри класів сутностей та заповнити їх за допомогою відображення. Без пустого конструктора створення екземпляра за допомогою відображення є дуже складним. Як правило, не бажано мати такий конструктор, оскільки передбачається що екземпляр Entity є лише носієм даних. Однак можна додати додаткові конструктори, які можуть заповнювати атрибути при створенні екземпляра

Детальній опис інших функціональних можливостей класу Entity наведено у Додатку В

Конкретний клас сутності не може безпосередньо розширити клас Entity. Він повинен успадковуватись від одній з реалізаційних сутностей (Implementation Entities), які детальніше вказує походження даних цієї сутності. Вже існуючі реалізаційні сутності будуть описані нижче. Однак кількість цих реалізаційних сутностей може збільшитися, коли до API будуть включені додаткові сховища даних. Отже, ця ієрархія класів позначається "неповною" (incomplete) на діаграмі класів. Оскільки кожна реалізаційна сутність представляє певний тип персистентності, ця ієрархія класів також позначається як «непересічна» «disjoint».

*SqlEntity*

Абстрактний клас SqlEntity – це клас сутностей реалізації, які успадковуються, якщо одиниця даних, яку вони представляють, походить виключно з реляційної бази даних. Діаграма класів зображена на рисунку 4.4, а методи, додаткові до успадкованих з класу Entity, детально визначено у додатку В

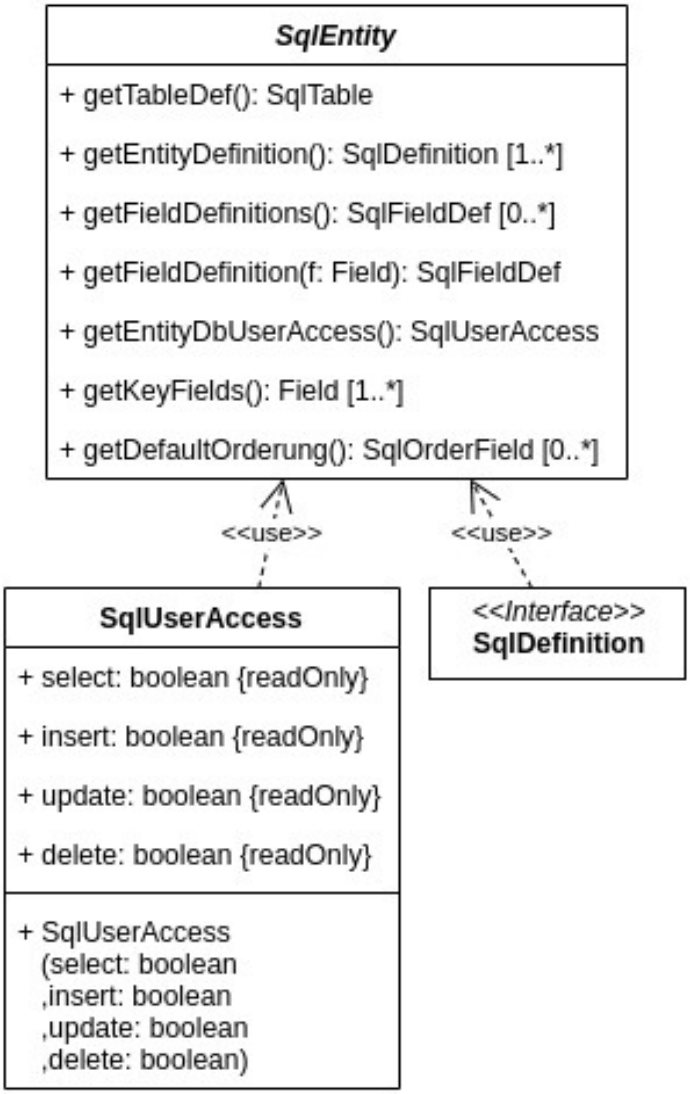



Рисунок 4.4 – Діаграма класу класу SqlEntity та залежностей

Як згадувалося в описах методів (Додаток В) для класу SqlEntity, було додано кілька допоміжних класів, які повинні бути використані для визначення зв'язків між сутністю та реляційною таблицею, до якої вона належить. Усі ці класи реалізують інтерфейс маркеру SqlDefinition для позначення своєї належності та спрощення доступу, наприклад, для виклику методу getEntityDefinition, де всі елементи можна отримати одночасно.

Елементи визначення - Plain Old Java Objects (POJO), що містять конкретну структуровану інформацію та є константами екземпляра, які можна встановити лише при створені екземпляра. Різні елементи цього фреймворку

визначень також використовують інші елементи для чіткого та структурованого представлення.

Різні елементи зображені як діаграма класів на рисунку 4.5 і будуть описані у Додатку В.

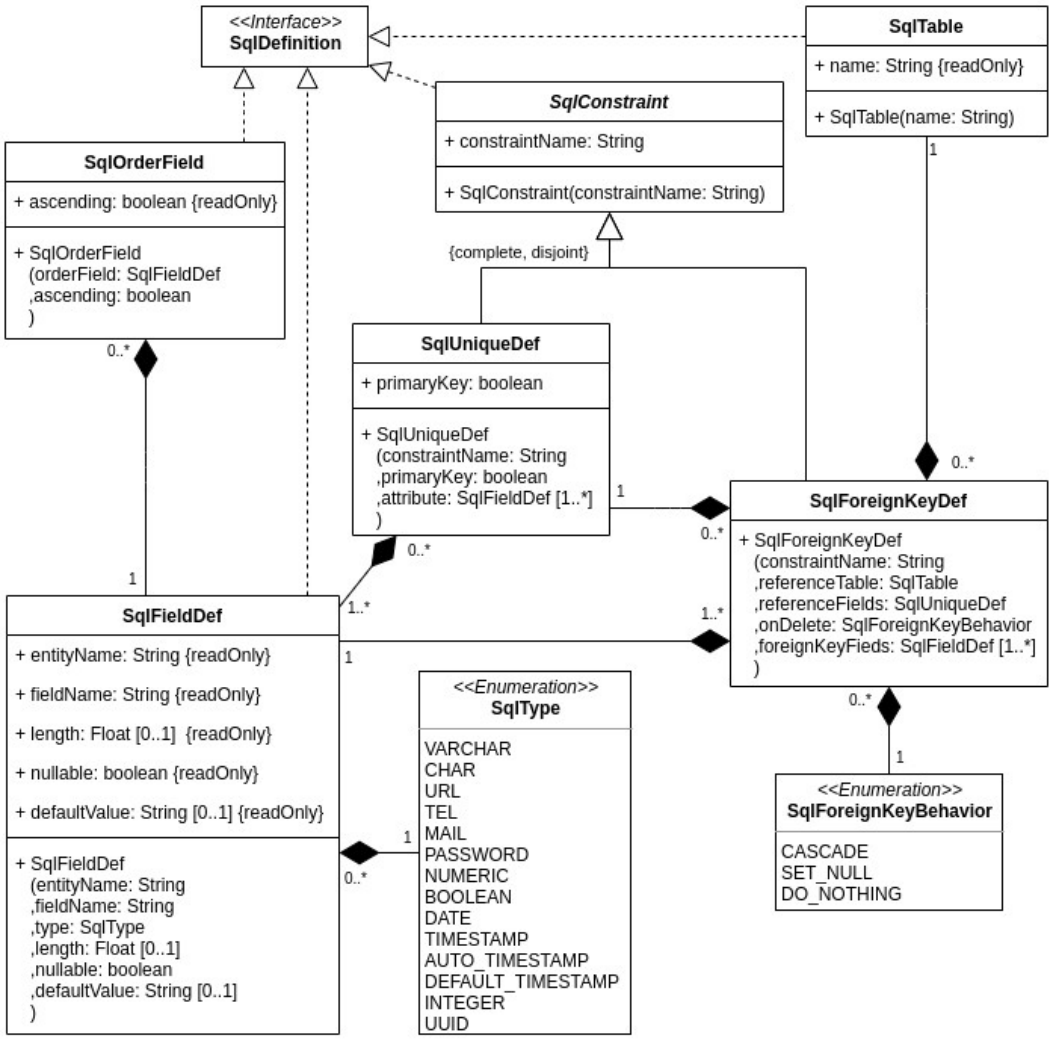



Рисунок 4.5 – Діаграма класів схеми SqlDefinition для SqlEntity

Представлення про реалізацію SqlEntity можна отримати із наступного коду:

```
public class Student extends SqlEntity {

    public static final SqlTable TABLE =
        new SqlTable(Student.class.getSimpleName());

    public static final SqlFieldDef FD_ENR_NO =
        new SqlFieldDef(”enrolmentNo”, ”enrolment_no”,
```

```
                    SqlType.VARCHAR, 10f, false, null);

    public static final SqlFieldDef FD_FIRST_NAME =
        new SqlFieldDef(”firstName”, ”first_name”,
                    SqlType.VARCHAR, 50f, false, null);

    public static final SqlFieldDef FD_LAST_NAME =
        new SqlFieldDef(”lastName”, ”last_name”,
                    SqlType.VARCHAR, 50f, false, null);

    public static final SqlFieldDef FD_FACULTY =
        new SqlFieldDef(”faculty”, ”faculty”,
                    Faculty.FD_NAME.type,
                    Faculty.FD_NAME.length,
                    true, null);
    public static final SqlUniqueDef CO_PK_NO =
        new SqlUniqueDef(”STUDENT_PK_NO”, true, FD_ENR_NO);

    public static final SqlForeignKeyDef CO_FK_FACULTY =
        new SqlForeignKeyDef(”STUDENT_FK_FACULTY”,
                        Faculty.TABLE,
                        Faculty.CO_PK_NAME,
                        SqlForeignKeyBehavior.SET_NULL,
                        FD FACULTY);

    public String enrolmentNo;
    public String firstName;
    public String lastName;
    public String faculty;
}
```

За допомогою цього коду продемонстровано SqlEntity «Student» з чотирма атрибутами, включаючи посилання на «Faculty». Атрибути – це прості загальнодоступні рядки, які можна довільно змінювати. Визначення містяться статично як константи, включаючи визначення таблиці, чотири визначення полів, одне унікальне визначення, що представляє первинний ключ, та визначення зовнішнього ключа, що посилається на «Faculty».

*LdapEntity*

Абстрактний клас LdapEntity – це клас сутностей реалізації, які успадковуються, якщо одиниця даних, яку вони представляють, походить

виключно з виключно з каталогу LDAP. Діаграма класів зображена на рисунку 4.6, а методи, додаткові до успадкованих з класу Entity, детально визначено у додатку В

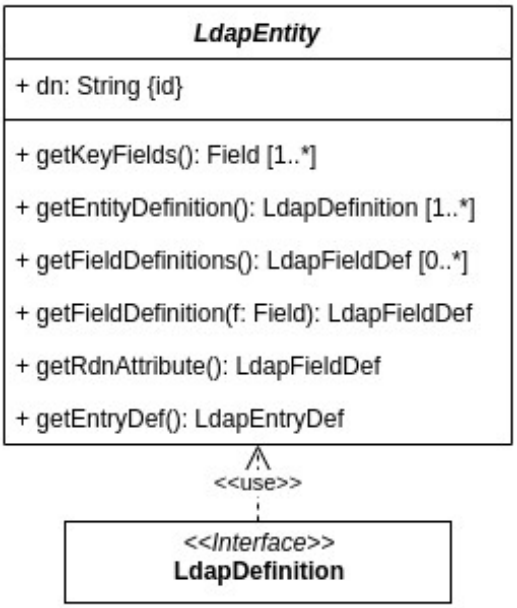



Рисунок 4.6 – Діаграма класу класу LdapEntity та залежностей

**Dn** У каталозі LDAP кожен запис отримує Distinguished Name (DN), яке однозначно ідентифікує запис для цілого каталогу. Він складається з Relative Distinguished Name (RDN), яке ідентифікує запис каталогу всередині вузла, в якому він знаходиться, і шлях до цього вузла від кореневого вузла. Оскільки кожен запис в обов'язковому порядку отримує цей DN, відповідне поле безпосередньо реалізовано в класі LdapEntity, тому його мають усі класи, що успадковують LdapEntity. Як і всі інші атрибути сутностей у цьому API, цей також є загальнодоступним.

Подібно до визначення SqlEntity, LdapEntitiy також має структуру визначення, яка використовується для забезпечення безпечного визначення структури LDAP для цієї конкретної сутності. Усі елементи визначення реалізують інтерфейс маркера LdapDefinition так само, як це було визначено для фреймворка SqlDefinition. Схема класів зображена на рисунку 4.7. Детальніший опис показано в Додатку В:

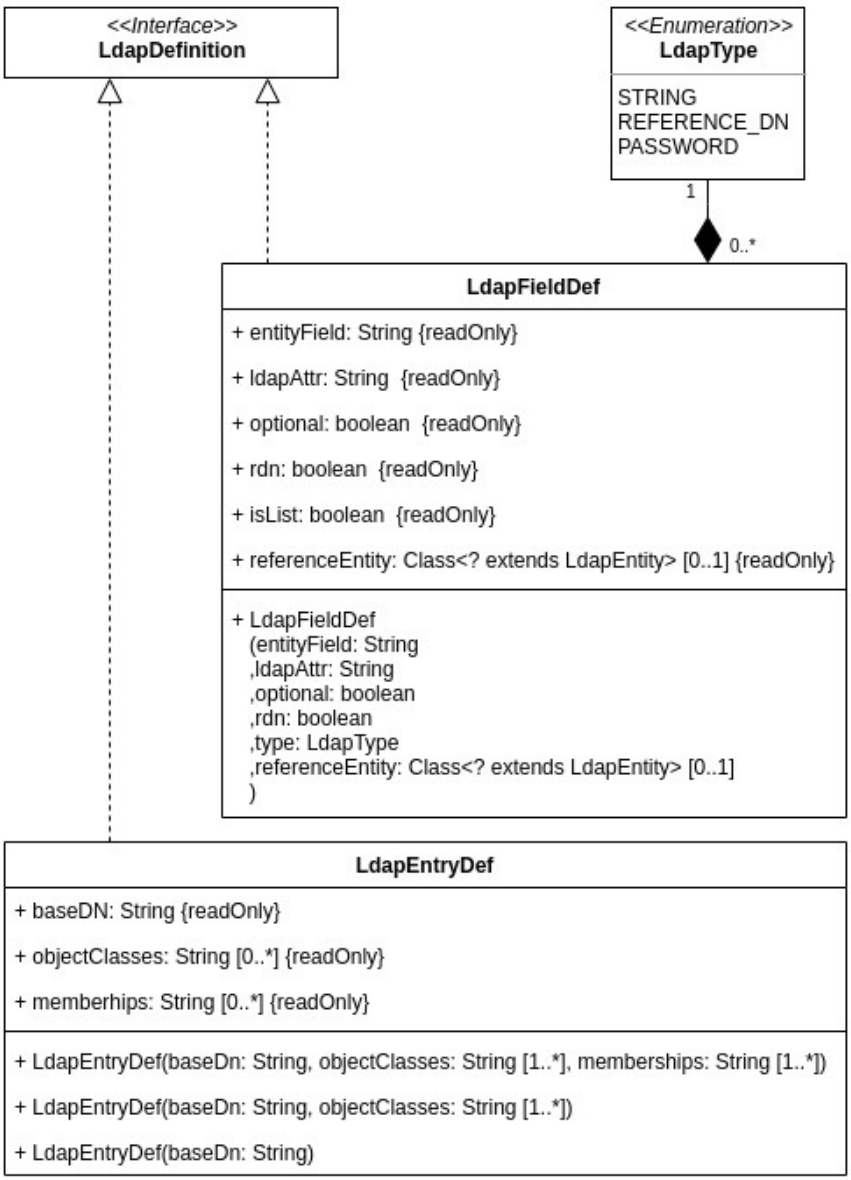



Рисунок 4.6 - Діаграма класів схеми LdapDefinition для LdapEntity

Представлення про реалізацію LdapEntity можна отримати із наступного коду:

```
public class Person extends LdapEntity {

    public static final LdapEntryDef ENTRY =
        new LdapEntryDef(”ou=users”,
                         new String[]{”mailUser”, ”inetOrgPerson”},
                         new
String[]{”cn=member,ou=groups,dc=acme”});

    public static final LdapFieldDef FD_USERNAME =
        new LdapFieldDef(”username”, ”uid”,
                         false, true,
```

```
                        LdapType.STRING, false);

    public static final LdapFieldDef FD_PASSWORD =
        new LdapFieldDef(”password”, ”Password”,
                        false, false,
                        LdapType.PASSWORD, false);

    public static final LdapFieldDef FD_SURNAME =
        new LdapFieldDef(”surname”, ”sn”,
                        false, false,
                        LdapType.STRING, false);

    public static final LdapFieldDef FD_FIRSTNAME =
        new LdapFieldDef(”firstname”, ”givenName”,
                        false, false,
                        LdapType.STRING, false);

    public String username;
    public String password;
    public String surname;
    public String firstname;
}
```

*PolyglotEntity*

У попередніх обговореннях про розділення моделі, виявилося, що елементи, які стали б одним компонентом узгодженості та одним блоком даних можуть бути розділеними з певних причин. Тоді результатом буде блок даних у кожній з розділених моделей, який має спільний ідентифікатор, який дозволяє посилатися один на одного. Однак, з точки зору бізнес-логіки, це розділення може бути нерелевантним, і блок даних повинен розглядатися як одна сутність. Тому було визначено окрему сутність реалізації, який називається PolyglotEntity. Цей клас сутностей представляє цілісну одиницю даних у бізнес-логіці так, ніби між сховищами даних не було розподілу. Внутрішньо PolyglotEntity зберігає сутності розділів, які належать до будь-якого іншого типу - так звані «постійні сутності» (persistent entities) – разом із інформацією про те, як атрибути з PolyglotEntity розподіляються між цими постійними сутностями і яка комбінація атрибутів є елементом, який зв’язує «постійні сутності».

Зрештою, клас, що успадковується від PolyglotEntity, повинен відображатися в бізнес-логіці, як і будь-який інший клас сутності, який є простим носієм даних із загальнодоступними атрибутами. Коли цей тип сутності обробляється для маніпулювання даними за допомогою API, дані «постійних сутністей» які поміщаються в PolyglotEntity, синхронізуються з даними PolyglotEntity, а «постійні сутності» витягуються і пересилаються в певну частину API, що взаємодіє з відповідним сховищем даних. У разі отримання даних цей процес виконується в зворотному порядку, тобто витягуються елементи з різних сховищ даних, поміщуються у PolyglotEntity, а потім синхронізуються атрибути у визначеному порядку.

Як показано для інших підкласів Entity, підклас PolyglotEntity також потребує певного виду визначення, тобто з яких «постійних сутностей» він складається, які поля при необхідності відображаються між persistent та polyglot екземплярами, та що є комбінацією звьязуючих атрибутів. Що стосується розділення концептуальної моделі, реалізація здійснюється від провідної сутності, ідентифікація якої використовується рештою сутностями як посилання, тобто атрибута підключення. Це визначення було поміщено до окремого класу PolyglotDefinition. Подібно до інших підкласів сутностей, які вимагають певних констант для власних конкретних сутностей, а кожен підклас PolyglotEntity повинен надавати рівно одну константу цього типу. Крім того, PolyglotReference можна налаштувати у випадках, коли немає розділення сутності як такої, але вона посилається на іншу сутність, яка знаходиться в іншому сховищі даних. Діаграму класів представлено на рисунку 4.7.

Визначення PolyglotDefinition детально представлено у Додатку В

Елементи самого PolyglotEntity, показані на рисунку 4.8 та визначені в Додатку В.

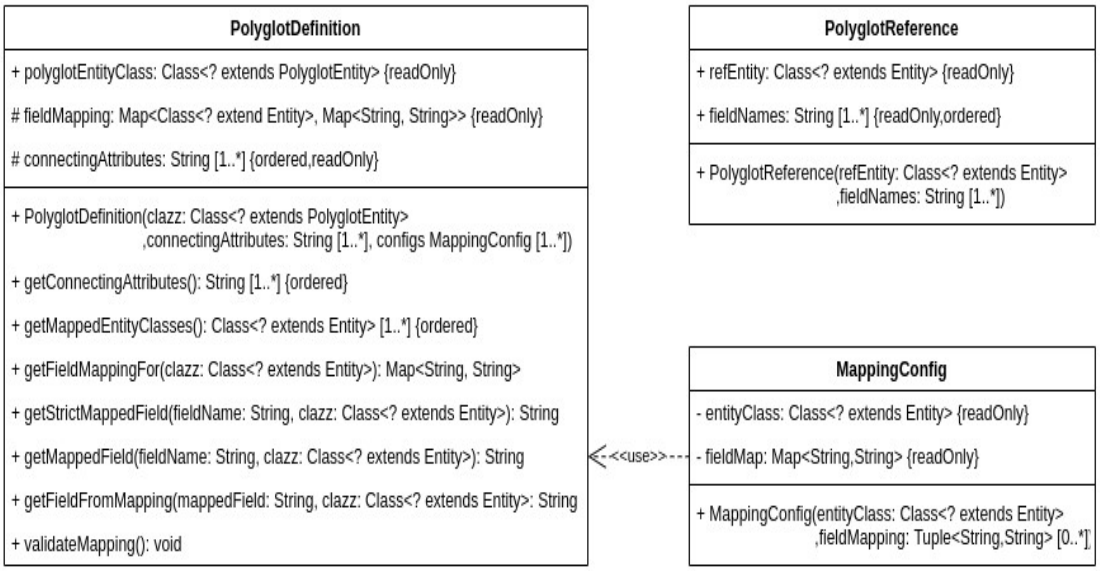



Рисунок 4.7 - Діаграми класів PolyglotDefinition та PolyglotReference

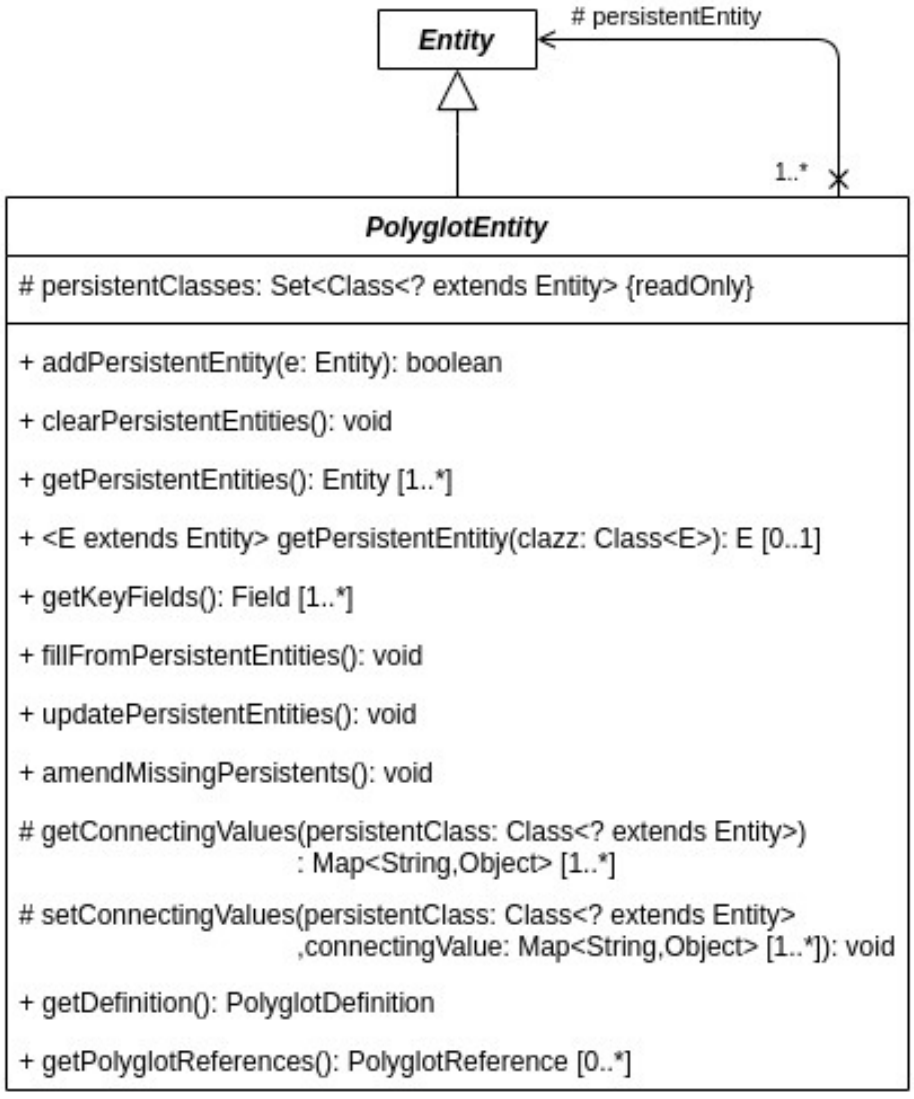



Рисунок 4.8 - Діаграма класу PolyglotEntity

### 4.2.2 Загальний інтерфейс даних як фасад

Загальний інтерфейс даних як фасад (Generic Data Interface as Facade – GenericDataIF) є центральним елементом, що забезпечує простий фасад бізнес-логіки. Шаблон фасад (Facade) – структурний шаблон проектування, що дозволяє приховати складність системи шляхом зведення всіх можливих зовнішніх викликів до одного об'єкту, який делегує їх відповідним об'єктам системи. Як уже згадувалося в вимогах до API, GenericDataIF містить базові функції для унікального доступу до сутностей будь-якого типу. Інтерфейс використовує загальний підхід, тобто його можна реалізувати для будь-якого класу в ієрархії сутностей. Це дозволяє створювати окремі реалізації для будь-якої сутності реалізації, які потім можуть бути зібрані разом в міру необхідності в залежності від потреб проекту.

Для концепції цього API важливо розуміти, що проекти, ймовірно, будуть розширювати цей інтерфейс додатковими методами відповідно до конкретних вимог проекту, наприклад, вилучення зі спеціалізованими вимогами до фільтрації та форматуванню або модифікації, які вимагають одночасної обробки декількох сутностей. Це дозволяє проекту створити дуже індивідуальний фасад для бізнес-логіки для доступу до даних так, як це необхідно.

Діаграма класів інтерфейсу GenericDataIF зображена на рисунку 4.9, а необхідні для нього методи детально визначені у Додатку В

На рис. 4.9 також показано, що для кожної реалізації класу Entity, описаного вище, існує відповідна реалізація GenericDataIF, а також для самого класу Entity. Ця настройка дозволяє API обробляти різні типи класів сутності за допомогою спеціальної реалізації та тим самим впроваджувати концепцію «розділення завдань».

Різні реалізації більш детально представлені нижче. Методи, які вже були описані вище для інтерфейсу, знову не використовуються для окремої

реалізації, тому що вони просто реалізують описане введення для конкретного класу Entity, до якого вони були прив'язані

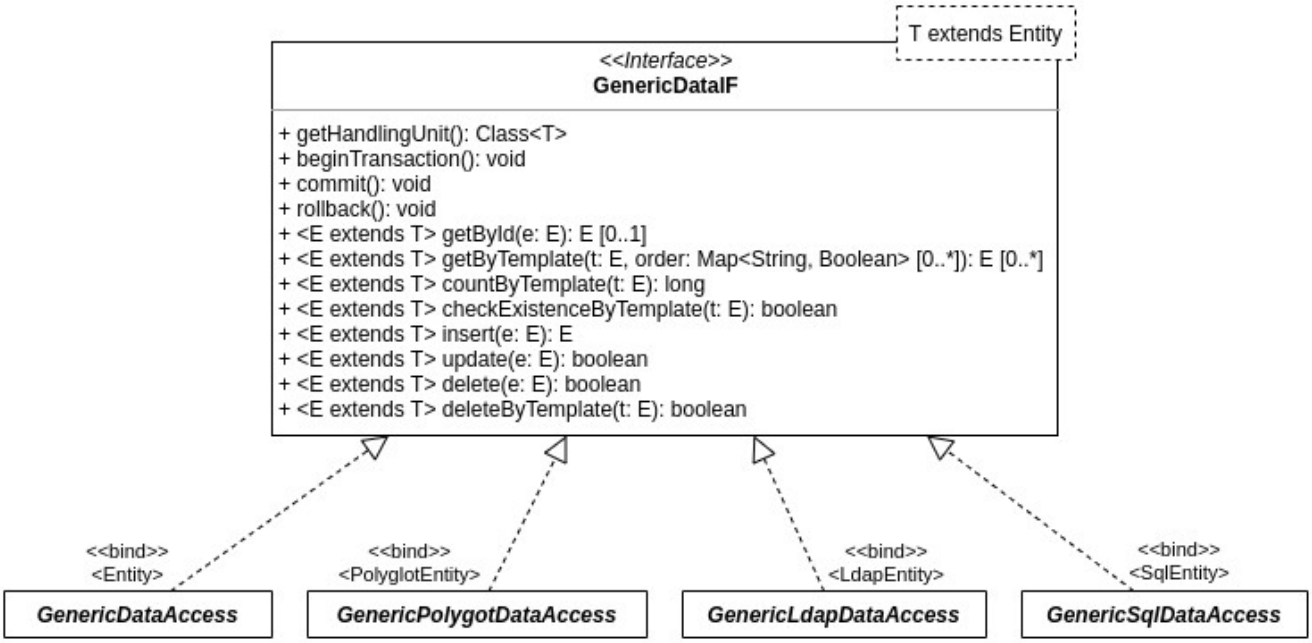



Рисунок 4.9 – Діаграма класів загального інтерфейсу даних як основного фасаду бізнес-логіки

*GenericDataAccess*

Абстрактний клас GenericDataAccess є центральною точкою входу бізнес-логіки на рівень доступу до даних. Він реалізує GenericDataIF, що зв'язує загальний «E» з самим класом Entity, тобто з кореневим класом в ієрархії класів Entity. При цьому будь-який клас, що успадковує від Entity, може бути переданий для обробки або вилучення даних. Отже, він завершує бажану реалізацію фасаду, оскільки бізнес-логіці потрібен тільки екземпляр цього класу, відповідно, розширення для конкретного проекту і класи Entity самого проекту, в той час як обробка за фасадом залишається чорним ящиком, і немає різниці, який вид of Entity різні класи (SqlEntity, LdapEntity, PolyglotEntity, ...).

Для досить слабо типизованної мови, такої як PHP, використання цього класу може бути призупинено, якщо використовується тільки одне сховище даних. У цьому випадку було б достатньо надати реалізацію для цього конкретного сховища даних. Якщо всі дані зберігаються, наприклад, в

реляційній базі даних, тоді бізнес-логіка може бути надана екземпляром конкретного розширення GenericSqlDataAccess для проекту. Якщо це зміниться в ході подальшого виконання проекту, і будуть додані інші сховища даних, наданий екземпляр все одно можна буде замінити специфічним для проекту розширенням GenericDataAccess – код існуючої бізнес-логіки взагалі не потрібно буде змінювати, оскільки змінні не мають типу, а параметри методу оцінюються тільки під час виконання.

З іншого боку, для строго типізованої мови, такої як Java, використання цього класу не має альтернативи для унікального надання цього фасаду і без будь-яких змін в бізнес-логіці, навіть якщо реалізація сховища даних зміниться в довгостроковій перспективі. Причина досить проста: існує чітке визначення, якого типу має бути наданий фасад. Наступна зміна цього типу – навіть якщо змінюється тільки загальна частина – призводить до змін в коді самої бізнес-логіки. В якості ілюстрації: якщо бізнес-логіка використовує змінну *GenericDataIF <SqlEntity> dataLayer*, яка відповідає реалізації *GenericSqlDataAccess* і, отже, робить екземпляр цього призначаємого, ці змінні повинні бути змінені на *GenericDataIF <Entity> dataLayer*, якщо *GenericDataAccess* повинен використовуватися через зміни в реалізації сховища даних, що використовує не тільки реляційну базу даних. Це небажано, так як це підірве концепцію фасаду, а код бізнес-логіки буде менш стабільним.

Як видно на рисунку 4.10, цей клас в основному реалізує методи, визначені GenericDataIF. Фактично, цей клас діє як проксі, тобто реалізації GenericDataIF або його розширення зареєстровані для різних типів. Коли викликається метод, реалізація аналізує тип переданої сутності, шукає підходящу реалізацію і перенаправляє виклик, залишаючи обробку спеціалізованої реалізації. Клас визначено за допомогою універсального «I», яке має розширювати GenericDataIF. Це було зроблено для карти реалізацій. Таким чином, спрощується розширення цього класу для конкретних цілей проекту, коли GenericDataIF було розширено.

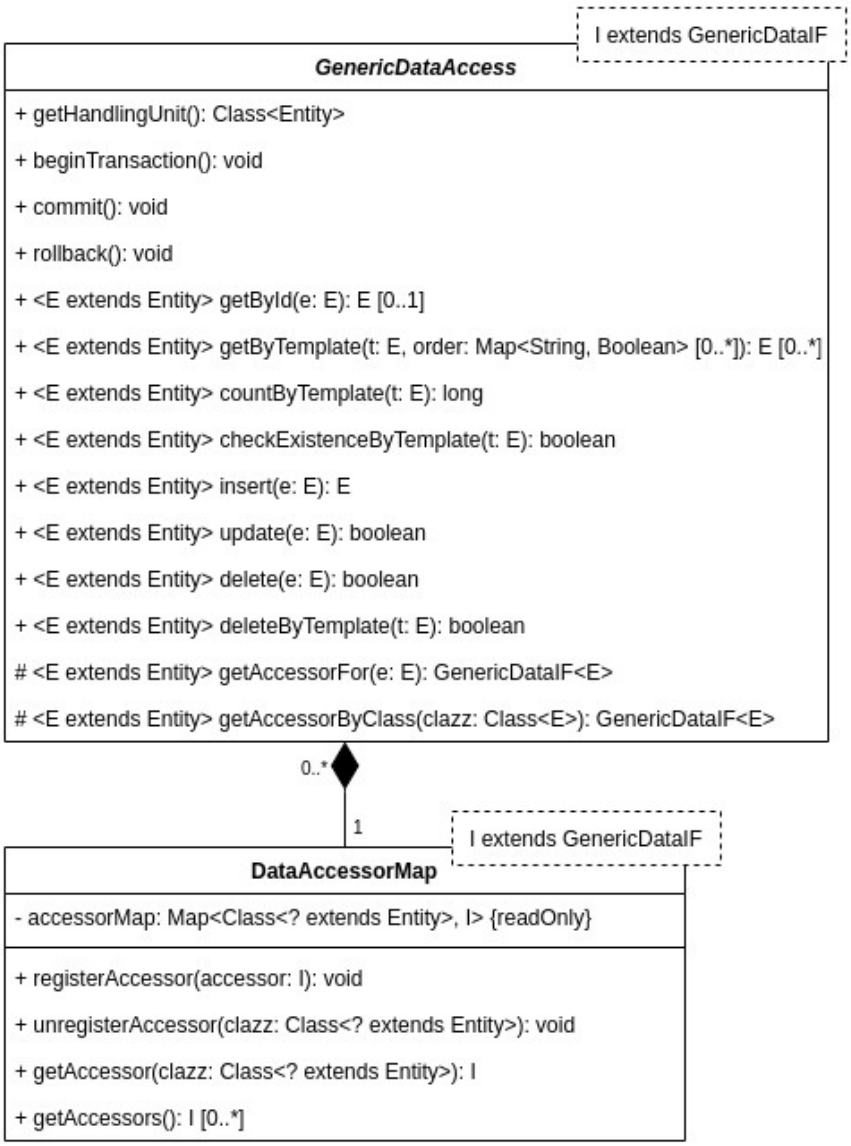



Рисунок 4.10 – Діаграма класів загального доступу до даних, що є проксі-сервером, який розподіляє виклики

Для обробки відображень різних спеціалізованих реалізацій, які повинні передаватися викликами, був визначений клас *DataAccessorMap*. Він інкапсулює карту розподілу реалізацій визначеного інтерфейсу, який повинен бути визначений у загальному вигляді, а також повинен розширити GenericDataIF до конкретного класу Entity, для реалізації якого забезпечується реалізація. Цей клас забезпечує обробку та перевірку безпеки типів, так що код виклику може абстрагуватися від коду, що має чітку поведінку та просте кодування. Клас був створений з універсальним «I», а також для забезпечення можливості повторного використання з типовою безпекою навіть для

розширення GenericDataIF, які потрібні для реалізації конкретних проектів. Детальний опис елементів, визначено у Додатку В

*GenericSqlDataAccess*

Важливо, що реалізація SQL є найскладнішою з усіх реалізацій. Відповідна схема класу зображена на рисунку 4.11.

Основною причиною цього є той факт, що SQL є стандартом і не є одночасно. В даний час існує, звичайно, стандарт SQL, який був узгоджений. Тим не менше, більшість постачальників реляційної бази даних додали свої власні поправки до SQL, що призвело до різних діалектів мови, яка насправді стандартизована. Простим прикладом тут є функції для управління даних під час виконання оператора. Ці функції відрізняються для кожного постачальника в конкретної назві і навіть у сигнатурі параметра. Наприклад, кожна реляційна база даних має функцію заміни NULL-значення іншим значенням. Це часто використовується для запобігання неправильному результату через особливу природу NULL в SQL. Тим не менше, функції відрізняються за своєю назвою: в Oracle функцією є NVL, в MySQL вона називається IFNULL, а в PostgreSQL – COALESCE. І таких прикладів є багато.

Наявність таких умов знаходить своє відображення у реалізації GenericDataIF для реляційної бази даних. Тому був створений внутрішній інтерфейс, який інкапсулює безпосередню взаємодію з базою даних під назвою SqlAccessIF. Цей інтерфейс повинен бути реалізований для кожного постачальника реляційних баз даних окремо, інкапсулюючи особливості та забезпечуючи унікальний інтерфейс для класу GenericSqlDataAccess.

Детальніший опис різних методів наведено у Додатку В.

*GenericLdapDataAccess*

Реалізація GenericDataIF для каталогів LDAP не настільки складна, оскільки немає діалектів, які слід враховувати. Отже, все це може бути реалізовано в одному класі. Загалом, макет схожий на GenericSqlDataAccess через той самий інтерфейс, який реалізований (рис. 4.12).

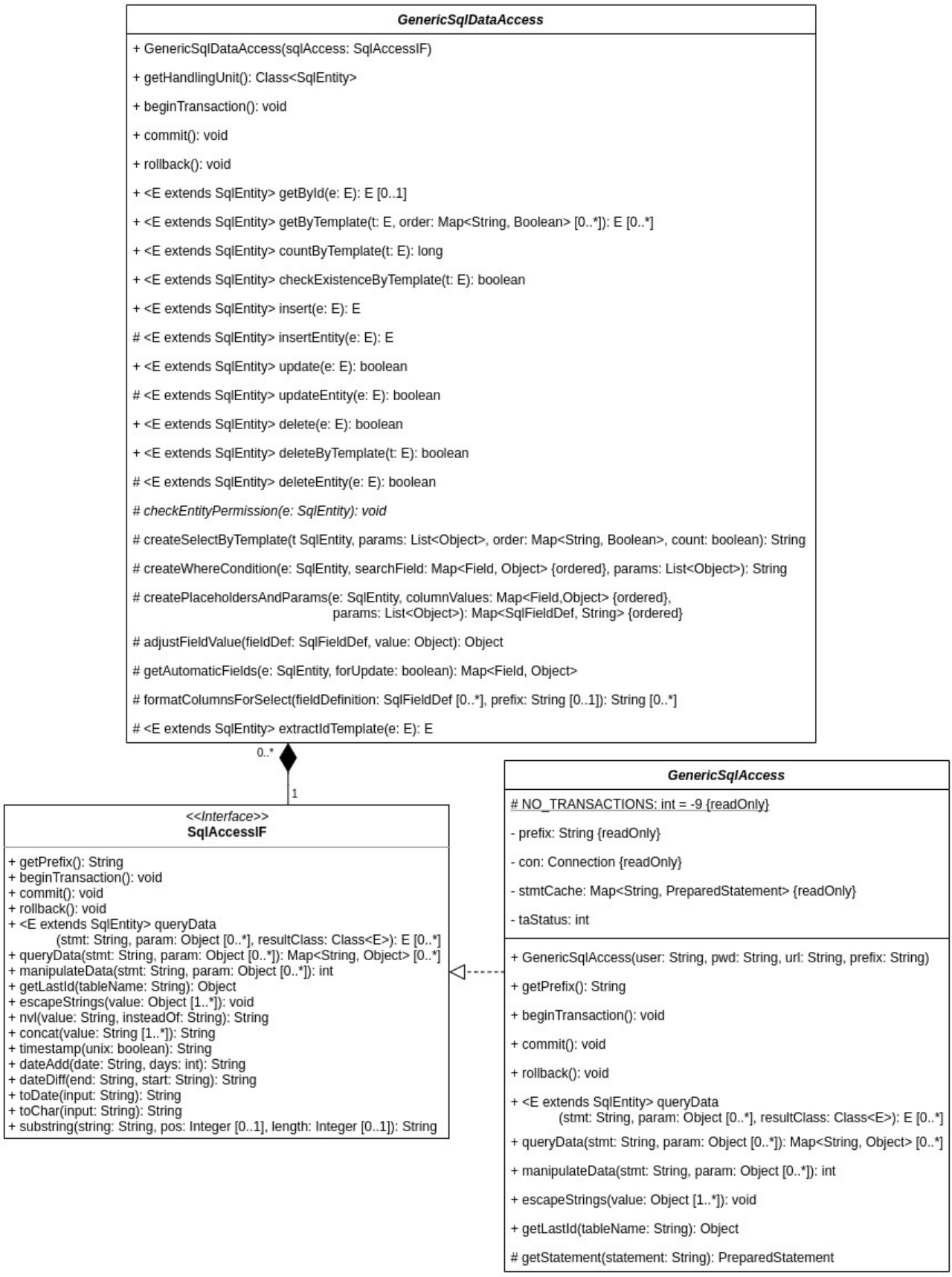



Рисунок 4.11 - Діаграма класів загального доступу до даних SQL, що реалізує фасадний інтерфейс для реляційних баз даних

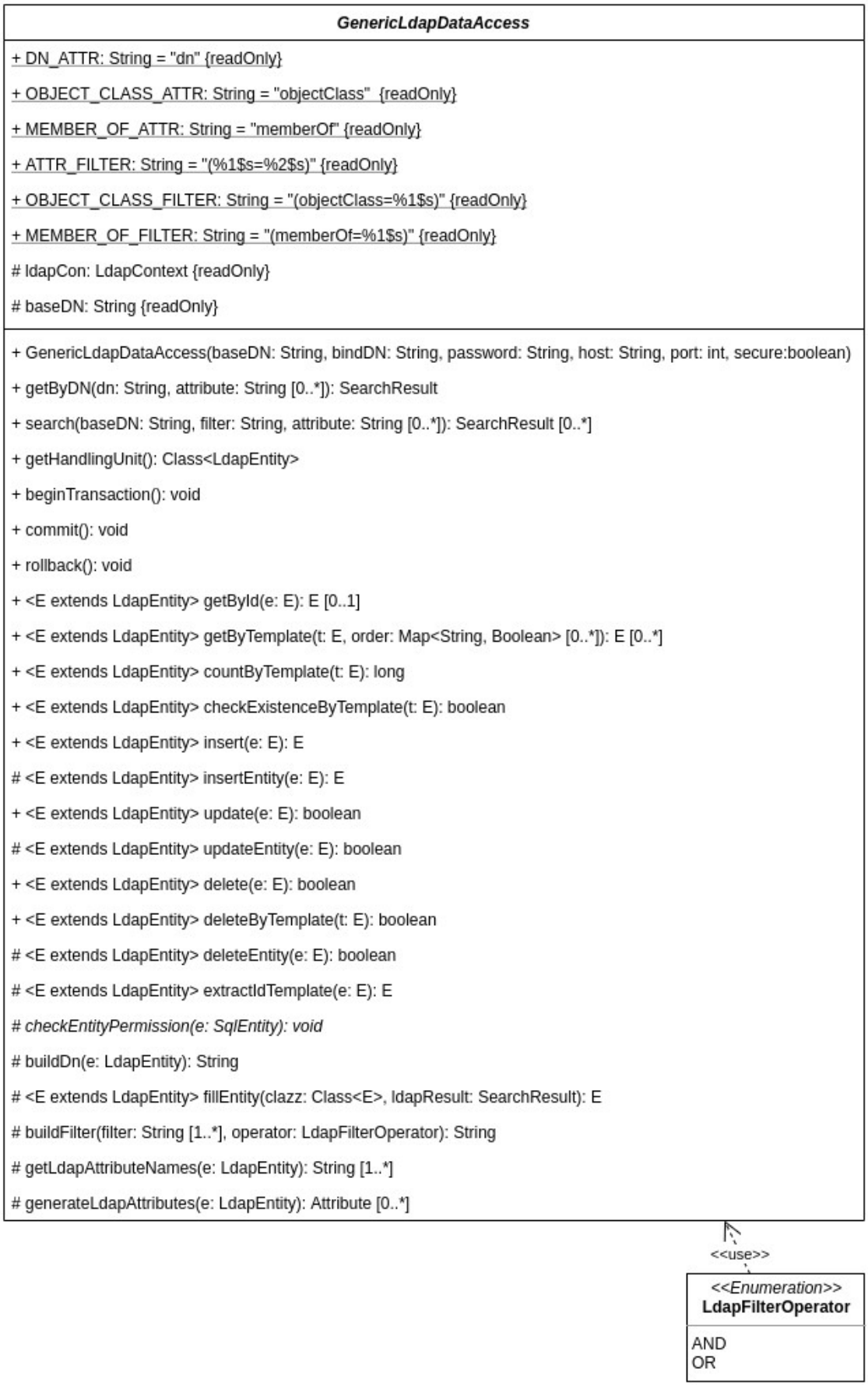



Рисунок 4.12 - Діаграма класів загального доступу до даних LDAP, що реалізує фасадний інтерфейс для каталогів LDAP

Аналогічно, реалізація LDAP також має захищені дзеркальні методи для вставки, оновлення та видалення, а також метод checkEntityPermission. Вони мають те саме призначення, що і ті, які присутні в реалізації SQL, і не будуть описані далі.

Елементи, які є специфічними для цієї реалізації, детально описано у Додатку В

*GenericPolyglotDataAccess*

GenericPolyglotDataAccess виконує всю обробку для бізнес-логіки, де беруть участь підкласи PolyglotEntity. Аналогічно класу PolyglotEntity, який безпосередньо не відображає конкретний блок даних у сховищі даних, він не взаємодіє з будь-яким сховищем даних безпосередньо, але обробляє збірку та розбирання екземпляра PolyglotEntity на екземпляри Entity, що представляють частини з різних сховищ даних, де містяться дані, і передає ці екземпляри Entity цих розділів до відповідної реалізації GenericDataIF (див рис 4.13). Ця остання частина стосується того самого механізму диспетчеризації, що вже реалізований для GenericDataAccess, який використовує DataAccessorMap для цього завдання. Отже, цей клас може бути легко використаний тут для цієї мети.

Реалізація цього класу однозначно позначає екземпляри сутності розділу як «Persistents», оскільки вони будуть ефективно зберігатися на відміну від PolyglotEntity, який є лише віртуальною одиницею даних, що конденсує дані різних частин і представляє їх як єдину одиницю для бізнес-логіки. Це полегшує розробникам реалізацію обробки даних, оскільки їм не потрібно дбати про те, куди їх розміщувати.

Як і інші класи доступу, GenericPolyglotDataAccess використовує інформацію, яку вже містять класи Entity. Однак, оскільки основним завданням є розділення та розподіл викликів до відповідних джерел даних, додаткову функціональність можна звести до мінімуму, оскільки всю інформацію можна отримати за допомогою конфігурації, розміщеної в конкретному підкласі PolyglotEntity. Єдиними методами, специфічними для класу в

GenericPolyglotDataAccess, які не реалізують необхідні методи інтерфейсу, є конструктор, зручні методи getAccessorFor та getAccessorByClass та метод checkPolyglotReference що є додано до GenericDataAccess з тим самим іменем. Метод checkPolyglotReferences перевіряє, чи модифіковані поля належать до одного з визначених посилань. Якщо це так, посилання перевіряється, шляхом запиту екземпляру Entity із його відповідними значеннями в якості ключових значення для вилучення екземпляра Entity. Якщо нічого не знайдено, створюється виняток. Він використовується, коли дані додаються або оновлюються, і запобігає посиланню на елементи, які не існують. Як уже зазначалося раніше, елементи посилання в цьому сузір'ї завжди повинні бути незмінними. Однак це може лише запобігти лише активним посилання на помилки під час обробки даних. Пасивні посилання на помилки, які трапляються, наприклад видаливши блоки даних, на які посилаються, тут виявити їх не можна, оскільки, звичайно, немає посилальної цілісності.

Отже, слід згадати ще одну цікаву деталь: як це було вказано в загальних міркуваннях щодо цього API, проблема посилань, що виходять за межі єдиного сховища даних, повинна бути зменшена лише посиланням на незмінні ідентифікатори, які, як правило, передбачає штучні ідентифікатори, які в основному не містять ніякої інформації самі по собі. Це запобігає появі каскадних оновлень зі посиланнями на блоки даних, які важко підтримувати в декількох сховищах даних, особливо коли, принаймні, одне із цих сховищ дотримується BASE замість принципів ACID. Однак все одно може знадобитися наявність надійного механізму посилань, який контролює цілісність посилань, навіть якщо задіяна база даних BASE.

Відповідь на це завдання можна також знайти за допомогою можливостей цього API, як описано раніше: додавши “координаційну базу даних” до налаштувань Polyglot, які мають єдине завдання відстежувати посилання, що виходять за межі однієї база даних. Цей координатор повинен підтримувати цілісність посилань і, отже, дотримуватися принципів ACID. Класи PolyglotEntity потім отримають персистентний клас Entity, що представляє

конкретну частину координатора. Потім він буде автоматично врахований цією реалізацією під час вставки та видалення та тим самим виконає бажане завдання, використовуючи вже наявні функціональні можливості.

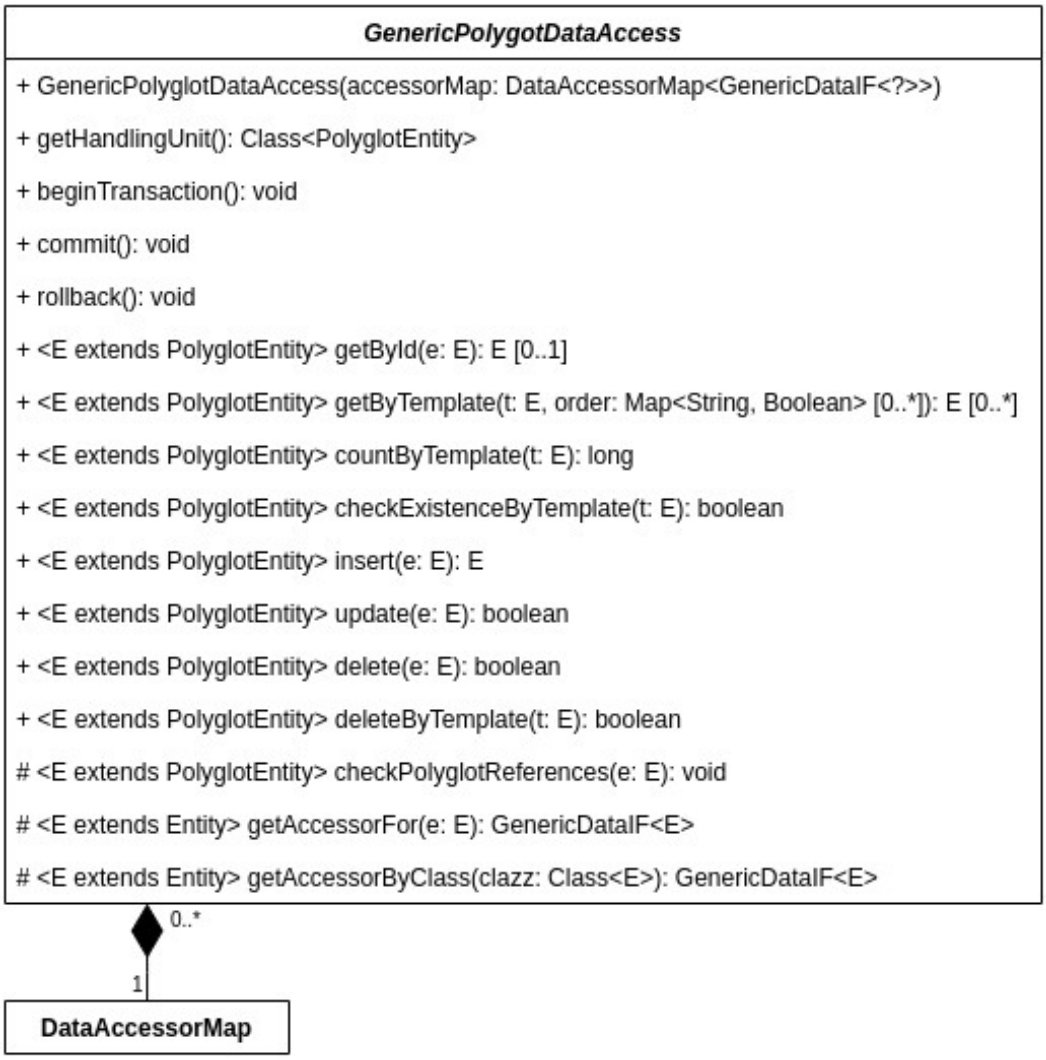



Рисунок 4.13 - Діаграма класів загального доступу до даних із багатоваріантною персистентністю, що реалізує фасадний інтерфейс для екземплярів PolyglotEntity

## 4.3 Деталі реалізації

Як вже показано на діаграмах класів, мову програмування Java було обрано, щоб довести, що концепція API є дійсною. Причина вибору Java досить зрозуміла: це одна з найбільш типізованих об'єктно-орієнтованих мов програмування, яка дозволяє писати дуже точні інтерфейси та створювати

добре структурований код. Система суворих типів також дозволяє виявляти проблеми вже під час компіляції, що корисно як для реалізації API, так і для його використання для створення рівня даних під час проекту.

З іншого боку, вибір мови програмування залежить від багатьох факторів, для яких рівень доступу до даних, як правило, відіграє не головну роль. Просте веб-застосування навряд чи буде написаний на Java, оскільки передумови щодо інфраструктури занадто високі. Тут інші мови, такі як PHP або Python, є кращими. Отже, важливо, щоб цей API працював для різних об'єктно-орієнтованих мов. Тому було прийнято рішення перевірити спектр об'єктно-орієнтованих мов не лише одним строго набраним представником - тобто Java - але також досить слабко типізованим представником. У цьому випадку це PHP. Слід визнати, що ця мова має досить багато питань, коли мова йде про проблеми безпеки та безпечне впровадження. Багато проблем часто можна знайти лише під час виконання, хоча їх можна було виявити під час компіляції з Java. Це піднімає зусилля для тестування. З іншого боку, є певні переваги, які виникають як у системі слабшого типу, так і внаслідок певних елементів синтаксису. В якості одного з прикладів слід зазначити можливості для визначення функцій. PHP пропонує тут можливість точно визначити для параметрів, а також для типу повернення, чи може бути передано "NULL" чи ні. Це дуже корисна здатність, яка зводить необхідність перевірки Null та ризик винятків нульових покажчиків до мінімуму.

Отже, два прототипи API були впроваджені для затвердження його конструкції. Наступні аспекти реалізації ілюструють певні цікаві моменти того, як вирішувались певні завдання та де можуть бути знайдені особливі відмінності в програмуванні тієї чи іншої мови програмування.

Більш детальніше запропоновану концепцію API описано в п'ятому розділі на прикладі розробки веб-застосування для IT-проекту KRAS

## 4.4. Висновки до четвертого розділу

В четвертому розділі отримано наступні результати.

1. На основі аналізу вимог до розроблювального API показано, що дизайн API повинен враховувати наступні аспекти:

— API повинен бути розширюваним, щоб він міг стати основою рівня доступу до даних програмного проекту.

— мовні розриви обробляються тільки в API, відповідно на рівні доступу до даних, побудованому на ньому.

— відмінності між використовуваними системами баз даних повинні бути пом'якшені і приховані від бізнес-логіки.

— API створює фасад бізнес-логіки, інкапсулює всю обробку доступу до даних.

— API не діє як ORM. Об'єкти даних не підключені і не відслідковуються. Вони служать виключно для структурованої і типобезпечної обробки даних.

— загалом, API повинен виступати в якості основи для проектів для побудови моделей у сенсі шаблона проектування MVC.

2. Введено поняття «Сутність API» яка транслює лише структуру певної одиниці (блока) даних, яка зберігається в певному сховищі даних, щоб забезпечити безпечну обробку даних у бізнес-логіці, а також загальну обробку всередині самого API. Для реляційних баз даних це дорівнює таблиці, для документо-орієнтованої бази даних – документу, а для каталога LDAP – запису певного класу. Отже, «Сутність API» крім структурної інформації зберігає відображення атрибутів між API-сутностями та блоком сховища даних щодо назви атрибута, типу та конкретних обмежень, а також більш загальну інформацію про блок даних у цілому.

3. Для створення «Сутності API» та їх обробки розроблено ієрархію класів з кореневим абстрактним класом Entity, а також абстрактними підкласи для конкретних сховищ даних SqlEntity та LDAPEntity та також PolyglotEntity,

який може представляти частину даних, які концептуально належать друг другу (є цілостними), але розподілені між кількома сховищами даних через розділення концептуальної моделі.

4. Створено загальний інтерфейс даних як фасад бізнес-логіки (Generic Data Interface as Facade – GenericDataIF) показано, що для кожної реалізації класу Entity (SqlEntity, LDAPEntity та PolyglotEntity) існує відповідна реалізація GenericDataIF, а також для самого класу Entity. Ця настройка дозволяє API обробляти різні типи класів сутності впроваджувати концепцію «розділення завдань».

5. Абстрактний клас загального інтерфейсу GenericDataAccess є центральною точкою входу бізнес-логіки на рівень доступу до даних. Він реалізує GenericDataIF, що зв'язує загальний «E» з кореневим класом в ієрархії класів Entity. Для підкласів для конкретних сховищ даних SqlEntity та LDAPEntity та також PolyglotEntity розроблено класи загального інтерфейсу відповідно GenericSqlDataAccess, GenericLDAPDataAccess та GenericPolyglotEntityDataAccess

# РОЗДІЛ 5
# ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ЗАПРОПАНОВАНИХ РІШЕНЬ

## 5.1 Приклад проектування БД для веб-застосування KRAS

Результати роботи, а саме удосконалена мова концептуального моделювання та розроблений API повинні бути продемонстровані та безперешкодно перевірені на практичному прикладі, який вже набув практичної актуальності: проект KampfrichterAnmelde-System (KRAS) – це веб-застосування, першу версію якого було розроблено в університеті Прикладних наук м. Аугсбург ще у 2005 році за підтримки Баварської Федерації дзюдо. Мета проекту полягає у створенні інформаційної підтримки виконання професійних обов'язків суддів та провідної суддівської комісії. Судді можуть подавати заявки на участь у заходах, а член комісії може проводити розпорядження про події після цього. Проект був реалізований з використанням PHP як мови програмування та бази даних MySQL на рівні персистентності. В подальших версіях, було додано рівень абстракції даних для підтримки інших реляційних баз даних, і додано реалізацію цього рівня для PostgreSQL. Останні вдосконалення розроблені в теперішний час. Популярність вдосконалених версій застосування зростає та поширюється на решту Німеччини та інші країни світу, наприклад Швейцарія, Чехія та США.

### 5.1.1 Розробка комплексної концептуальної моделі при проектуванні БД для KRAS

Поточна концептуальна модель даних, написана в AGILA MOD, показана на рисунку 2.31.

Ця модель була оцінена щодо можливостей використання нових елементів AGILA MOD + і було знайдено кілька місць, де було б доцільним з

семантичної точки зору використання автономних об’єктів або сильних зав’язків учасників асоціації (див рис. 2.18-2.25 другого розділу) . На відміну від цього, було також знайдено кілька місць, де використання цих елементів синтаксису не вказано.

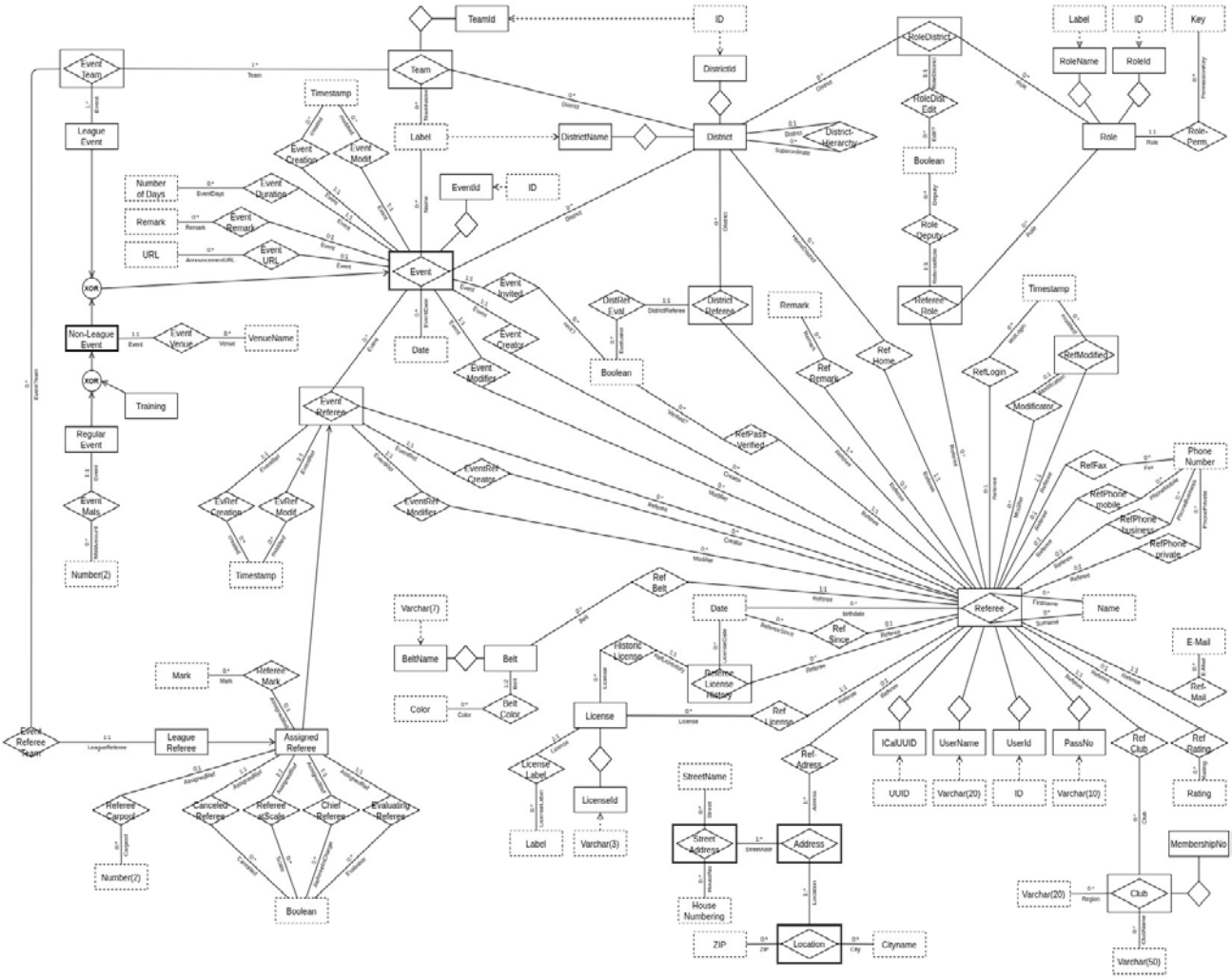



Рисунок 5.1 – Оригінальна модель AGILA MOD проекту KRAS.

Приклади подання будуть вибрані з цієї оцінки, щоб проілюструвати зміни та причини прийняття конкретного рішення:

– Об’єкти-учасники асоціації з інтервалом "x: 1" (з $x \in \{0,1\}$) попередньо не позначені, оскільки вони вже мають сильні зв’язки через вказаний інтервал, і вказані інтервали не будуть змінюватися. І все-таки їх можна було б позначити, як описано раніше. Подібне можна побачити в асоціації типу «RolePerm» у верхньому правому куті. Тут учасник «Role» був позначений сильним як виняток з інших учасників “x: 1”. Причина тут полягає

в тому, що інтервал може змінитися в майбутньому, коли дозволи більше не будуть представлені як ключ, а як окремі екземпляри дозволів.

– Для типу асоціації «RoleDistrict» учасник «Role» був позначений як сильний. Причина тут полягає в тому, що ролі, які можуть бути призначені суддям, також повинні мати певний дозвіл щодо округів. Оскільки це призначення важливе для ролей, а не для округів, розумно відзначити цю участь сильною.

– Об’єктивізована асоціація типу «EventReferee» не отримала жодного сильного учасника. У цьому випадку обидва учасники «Event» та «Referee» мають однакову вагу в цьому типі асоціації. Інформація, що належить цьому елементу, є релевантною для обох цих типів об’єктів. Для «Referee» важливо знати, на які «Event» та як він призначається. Для «Event» важливо знати, які «Referee» подали заявки, а кого призначили.

– Об’єктивована асоціація типу «Team» була оголошена автономною, оскільки спортивна команда завжди має своє власне існування, і її слід враховувати незалежно від оточуючих елементів. Автономна декларація не має негайного ефекту, оскільки вже існує конвенція про іменування. Однак це все ще робиться, оскільки важливо сформулювати із семантичної точки зору, як це вже обговорювалося раніше.

– Об’єктивізовану асоціацію типу «Club» не оголошено автономним. Це пов’язано з обставинами ПрО цього проекту. Клуб не має абсолютного значення для відділу суддівства, і тому йому не потрібно існувати окремо.

Результат вдосконалення концептуальної моделі проекту KRAS від AGILA MOD до AGILA MOD + показаний на рисунку 5.2. Вдосконалення мають скоріше візуальний характер, але вони підтримують зрозумілість моделі з точки зору семантичних умов ПрО. Зокрема, додані сильні зв’язки для учасників асоціації є надзвичайно корисними для прийняття рішення на основі цієї моделі для вкладених структур.

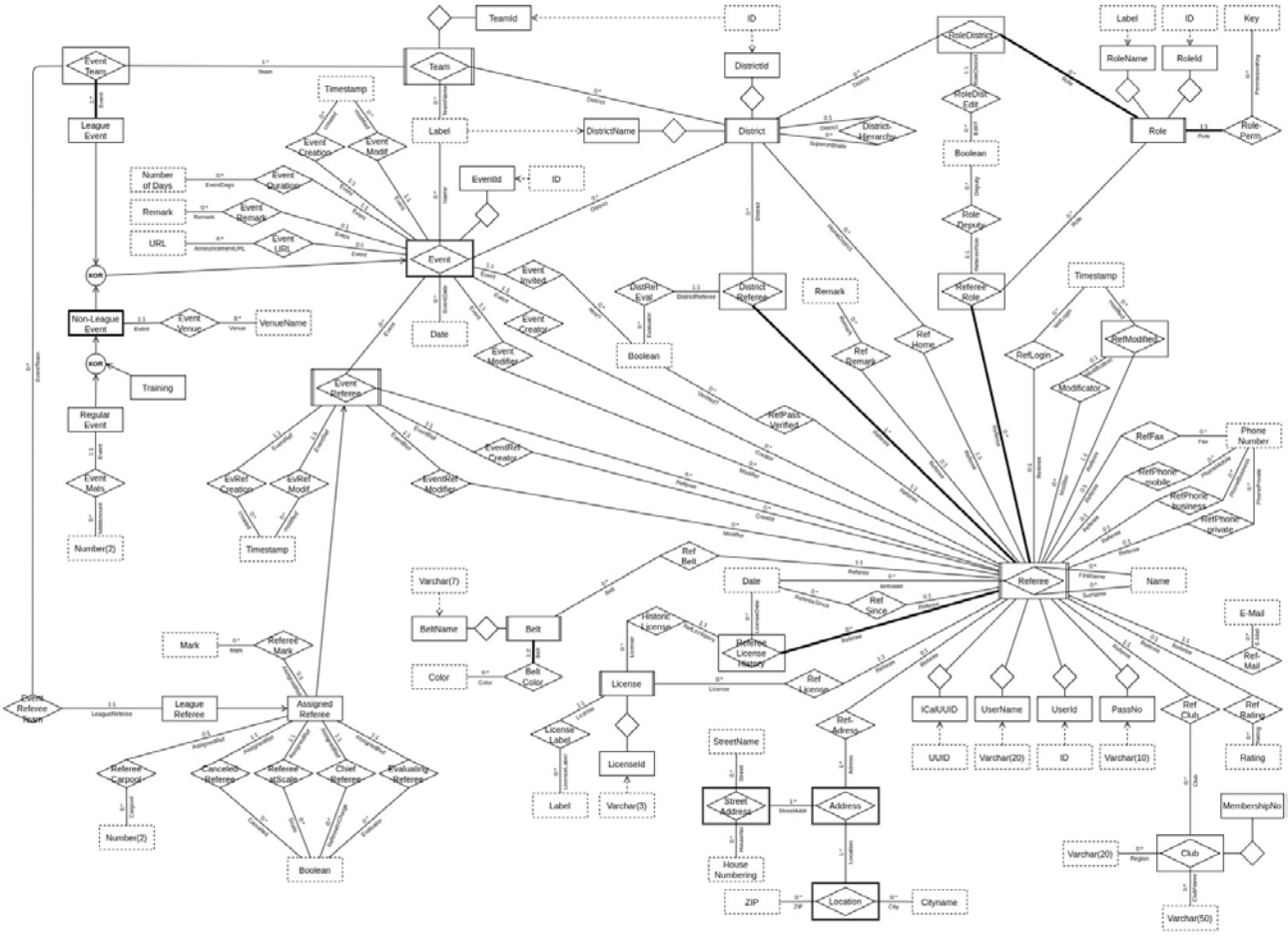



Рисунок 5.2 – AGILA MOD + комплексна концептуальна модель проекту KRAS

### 5.1.2 Розділення комплексної концептуальної моделі проекту KRAS на субмоделі

З врахуванням розширеної мови концептуального моделювання AGILA MOD+ та правил деривації логічних моделей у фреймворку AGILA побудовано комплексну концептуальну модель для проекту KRAS. В підрозділі 3.2 розроблено метод розділення комплексної концептуальної моделі ПрО на незалежні субмоделі – так звані «розділи» концептуальної моделі для подальшої їх деривації у відповідні логічні моделі.

Апробацію методу розділення комплексної концептуальної моделі даних на незалежні субмоделі показано на прикладі проекту KRAS.

В результаті виконання шості кроків запропонованого методу комплексну концептуальну модель розділено на дві субмоделі в результаті побудови оболонки оточення (рис. 5.3)

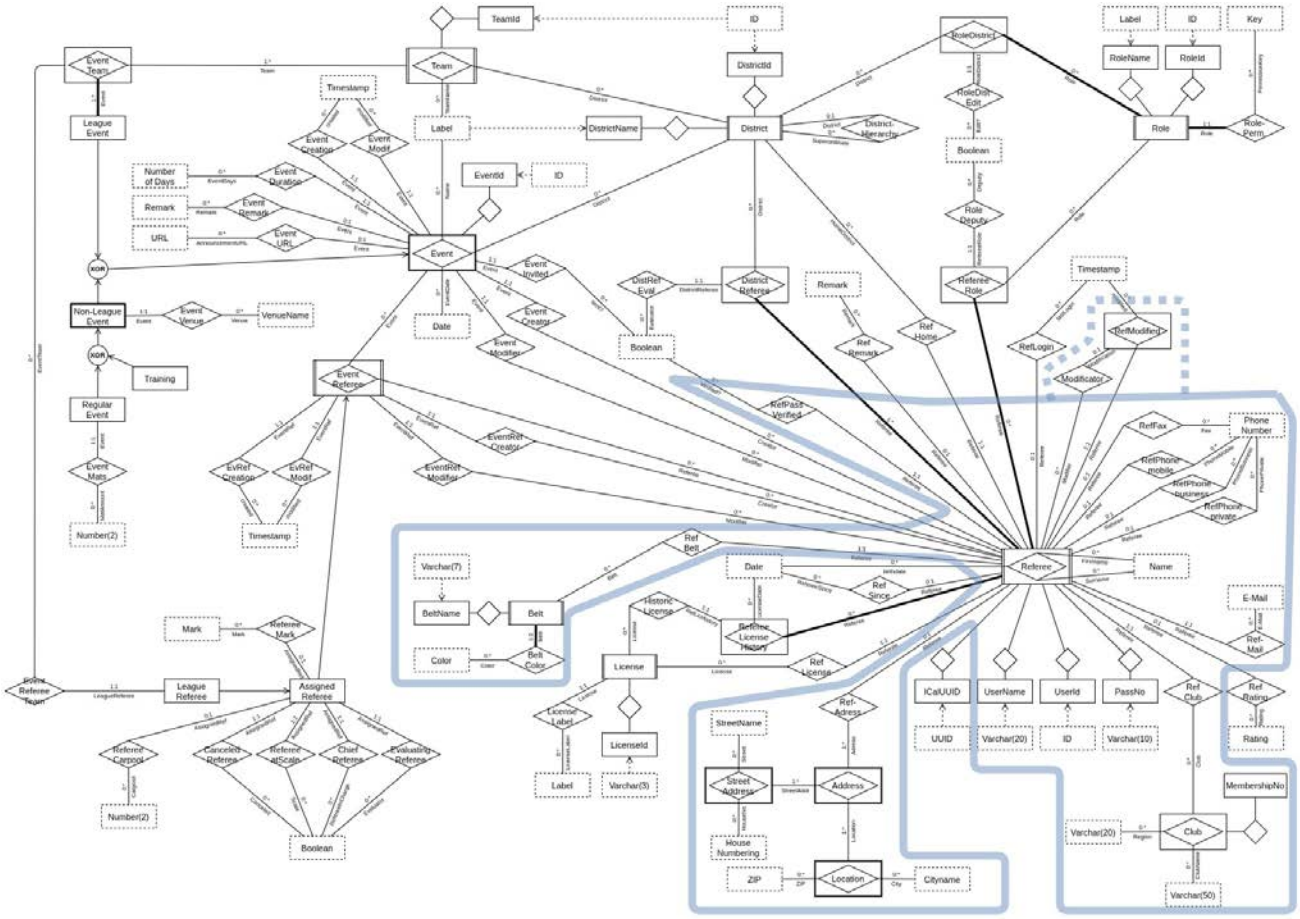



Рисунок 5.3 – Концептуальна модель проекту KRAS з побудованою оболонкою оточення (partitioning marks)

В даний час у Федерації дзюдо триває ініціатива щодо уніфікації даних та їх стисненню у найбільш підходящих місцях. Оскільки KRAS містить дані своїх користувачів, наприклад, телефонні номери, адресу та облікові дані також в інших інформаційних системах організації, виникла ідея об'єднати елементи, що перекриваються, разом у каталог LDAP, де ними можна управляти централізовано. Це хороший варіант прикладу використання для застосування багатоваріантної персистентності до програмного застосування з можливостю для розділення концептуальної моделі ПрО, як описано вище.

Таким чином, із розділення показаного на рисунку 5.3 не всі деталі концептуальної моделі представляють інтерес. Найважливішими є елементи в правому нижньому куті. У цій частині зображений суддя, який є користувачем системи з його залежностями, що представляють субмодель для деривації в каталог LDAP.

Згідно методу (див. підрозділі 3.2 та рис.5.3) розділ розпочато з кроку 1. Субмодель, яка має бути реалізованою в парадигмі персистентності LDAP, позначено оболонкою- оточенням товстішою прозорою лінією. Як бачимо деякі елементи комплексної концептуальної моделі ПрО, які є зв'язаними з виключно типом об'єкта «Referee», не включені до субмоделі. Це пов'язано з рішенням проекту, згідно з яким інформація, яка представляє інтерес лише для суддівської комісії - наприклад, ліцензія судді - не повинна додаватися до каталогу LDAP. Це рішення також враховує німецьке та європейське законодавство про захист даних. Це показує, що на розділення можуть впливати безпосередньо вимоги проекту, які не є технічно зумовленими, а є результатом простого управлінського рішення або правових норм.

Більше того, приклад демонструє випадок перекриття субмоделей, як описано раніше. Це видно посередині праворуч, де є два елементи, оточені штриховою лінією, що означає, що ці елементи належать як до першої субмоделі, оточеною оболонкою, так і до другої субмоделі, який можна сприймати як оточений білою лінією. Це перекриття субмоледей відбувається елементів, що відображають інформацію про зміну даних користувача. Інформація про те, коли і хто змінив певні дані, представляє інтерес для обох моделей даних. Включаючи ці елементи в обидві субмоделі, адміністратори можуть розрізняти зміни даних у будь-якому з розділів даних. Детально це означає, що якщо дані користувача були змінені в першої субмоделі, модифікація буде записана там, інакше вона буде записана в другої моделі даних. Отже, наявність накладених фрагментів розділених моделей, як описано, є бажаною і її не можна проігнорувати.

Отже після розділення отримані концептуальні субмоделі показані на рисунках 5.4 та 5.5  На рисунку 5.4 показаний результат розробленого методу розділення комплексної моделі для субмоделі, націленої на реалізацію LDAP.

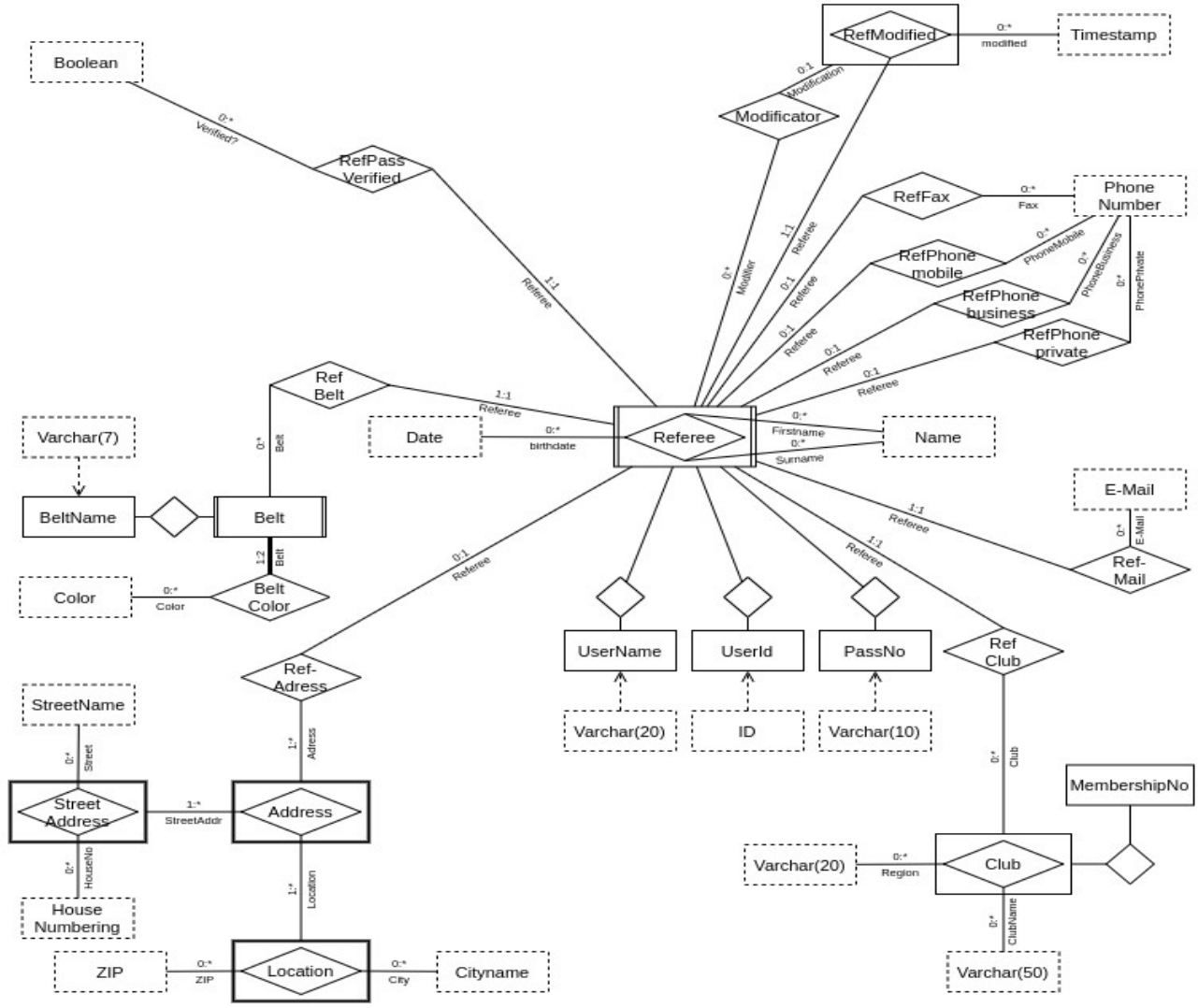



Рисунок 5.4 – Концептуальна субмодель (Conceptual partition models) для деривації в логічну модель типу LDAP

Друга субмодель створена для деривації в реляційну  модель на логічному рівні (рис. 5.5). Зображена реляційна субмодель демонструє лише відповідний фрагмент моделі, щоб отримати більш детальне уявлення про неї. Елементи з лівого боку в реляційно-спрямованій субмоделі на рисунку 5.5 були переміщені ближче до решти елементів з міркувань простору space reasons.

Схожість між двома відображеннями цієї субмоделі на рисунку 5.3 – ліва частина та на рисунку 5.5 можно помітити.

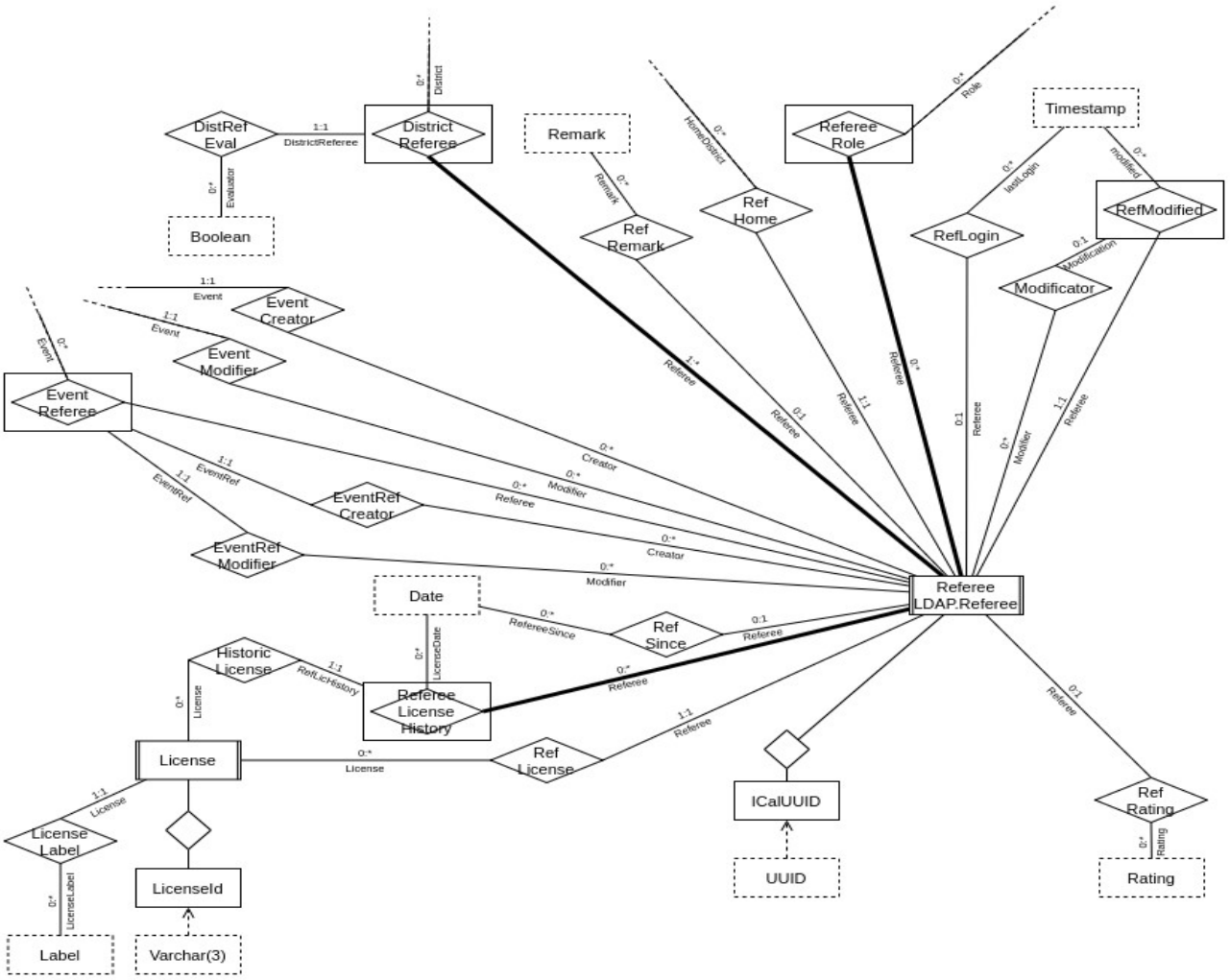



Рисунок 5.5 – Концептуальна субмодель (Conceptual partition models) для деривації в логічну реляційну модель

Важливо також зазначити, що реляційно-спрямована субмодель на рисунку 5.5 тепер має тип об’єкта, який позначається „LDAP.Referee”, що означає, що цей тип об’єкта є посиланням з концептуальної субмоделі спрямованої на LDAP. Більше того, є множина елементів (обьектів системних типів), що знаходяться в обох субмоделях, хоча в комплексну моделі ці елементи  входили лише один раз. Це подвоєння не є проблемою, оскільки об’єкти системних типів представляють лише вже існуючі елементи і можуть повторно використовуватися так часто, як це потрібно.

У прикладі розділення, який розглядається складнішим є випадок перетину успадкованих взаємозв`язків. Цей випадок повинен бути проілюстрований на прикладі подій у моделі даних KRAS.

На рисунку 5.6 представлено спрощене подання ієрархії типів об’єктів та успадкувань, які детально описані в третьому розділі (див. рис. 3.6). Як зазначено в методі якщо лінія розділення проходить через спрямовану лінію зв’язку успадкування між “XOR” і типом об’єкта «Event», тоді «Event» буде частиною субмоделі, яка відповідає множині елементів «inside» - Partition1, тоді як спрямована лінія зв’язку, яку була вирізано, тепер повинен вказувати на тип об’єкта, на який посилаються. «Partition1.Event», який належить множині «outside».

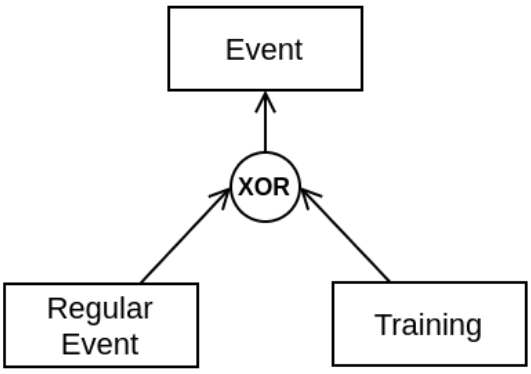
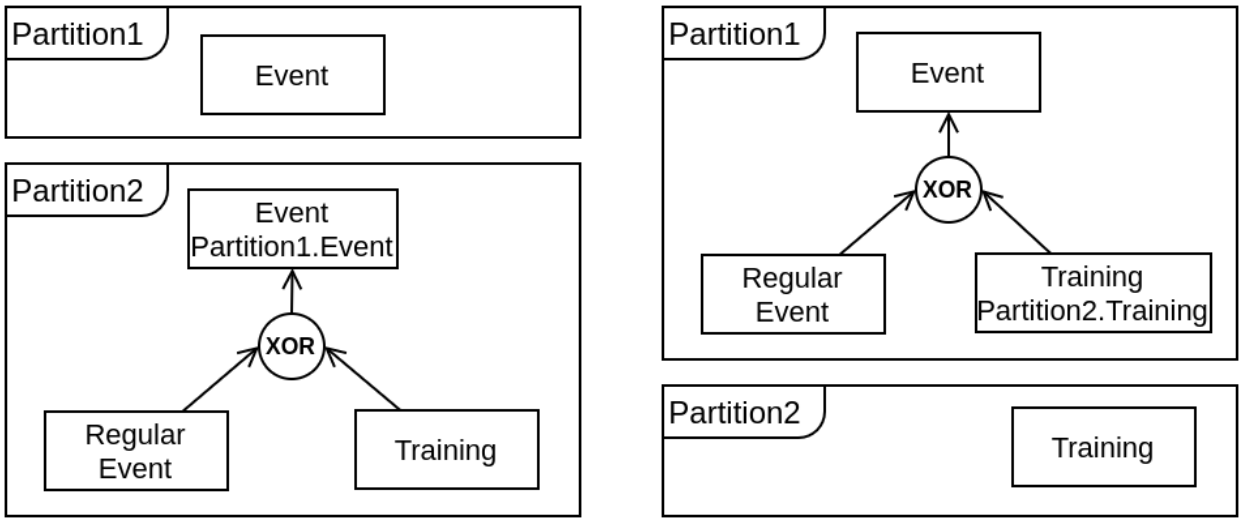



Рисунок 5.6 – Розділення елементів успадкування (Події – спрощено)

Результат можна побачити на рисунку 5.7а. З іншого боку, якщо лінія розподілу проходить через спрямовану лінію зв’язку між типом об’єкта «Training» та «XOR», «Training» буде частиною субмоделі, яка відповідає множині елементів «inside» - Partition2, тоді як тоді як спрямована лінія зв’язку, яку була вирізано, тепер повинен вказувати на тип об’єкта, на який посилаються. «Partition2. Training», який належить множині «outside» цей результат показано на рисунку 5.7б.

Отже, можна дійти висновку, що вихідна мета економії часу підтверджується цим підходом. Вже існуюче виведення на основі правил є більш швидким і менш схильним до помилок, ніж ручний підхід. Він

спирається на чіткі правила і навряд чи вимагає будь-яких інтелектуальних зусиль. За допомогою показаного процесу розділення, це виведення на основі правил тепер можна застосовувати до кожної часткової моделі окремо, що спричиняє ефект масштабування на перевагу економії часу.




(a) Розділення між Event та XOR (b) Розділення між XOR та Training

Рисунок 5.7 – Результати розділення успадкованих елементів.

Кількість часу необхідно для виведення логічної моделі множиться з кількістю часткових моделей, як і перевага в економії часу. Дослідження при апробації запрованованих рішень показали що було отримано скорочення в 2,3 рази часу на таке проектування двох БД в порівнянні з їх розробкою за класичною схемою, що підтверджено думкою експертів, власним досвідом розробника та автора цієї роботи.

## 5.2 Розробка API для веб-застосування KRAS

На прикладі розробки проекту KRAS буде показані можливості для розробників ПЗ щодо доступу до даних через розроблений API як доказ концепції.

Як було проілюстровано раніше, певні частини даних, які обробляються в системі KRAS, повинні бути передані з реляційної бази даних у каталог LDAP. Природно, тут основна увага приділяється особистим даним користувачів, які

регулярно зберігаються в каталозі LDAP для різних випадків використання, наприклад функції аутентифікації та авторизації користувачів або адресної книги, що забезпечують центральне сховище даних навіть для таких загальних програм, як клієнти електронної пошти. Оскільки LDAP забезпечує відкритий і дуже гнучкий інтерфейс для визначення структур, дані, що зберігаються, не обмежувалися стандартними атрибутами класів об’єктів за замовчуванням, які вже включені. Натомість рішення, яку інформацію додати до каталогу LDAP, зорієнтоване на питання: «Яка інформація є спільним інтересом для зацікавлених сторін Баварської федерації дзюдо та може використовуватися також в інших IT-системах?» Сюди входить - крім звичайних особистих даних, таких як адреси та номери телефонів, - кваліфікація особи, тобто її пояса, та клуб, в якому ця особа зареєстрована. Тому будуть визначені додаткові класи об’єктів, які визначають структуру цих порцій даних.

Перша версія веб-застосування KRAS написана на PHP і вже багато років працює з попередником API, представленим у цій роботі, який концентрується виключно на SQL із використанням PDO та абстрагуванням від можливих діалектів мови SQL від різних постачальників. Проте концепція та базовий інтерфейс майже однакові. API, представлений у цій роботі, додав поняття багатоваріантної пресистентності, зберігаючи філософію попередника, як описано. Реалізація всієї програми була виконана таким чином, щоб створити структуру найкращого можливого захисту, тобто надати певній частині системи якомога більше доступу та якнайменше (much access as needed and as less as possible). Це підтримує запобігання помилковому доступу, коли це можливо, як з боку операторів, так і розробників. Це допомагає розробникам підтримувати код чистим та запобігати помилкам при повторній зміні коду після тривалого періоду непрацювання над ним. Отже, така структура навіть корисна, якщо розробка являє собою класичну " one-man-show", оскільки він або вона, швидше за все, не пам'ятає деталей системи і буде правильно керуватися захищеними структурами.

Шляхом розширення *GenericDataIF*, яке вже показано на прикладі попередньої версії, був створений рівень даних для конкретного проекту. Рівень містить основний *DataIF*, який повинен бути використаний для основних інтерфейсних дій, і кілька розширень до нього, що використовуються для спеціалізованих вимог: *BackendDataIF*, який використовується для обслуговування розширених вимог до адміністративного серверного сервера, ReportDataIF, що забезпечує спеціальні оцінки даних та InterchangeDataIF, який обробляє весь доступ до даних, необхідний для обміну даними між різними екземплярами системи KRAS. Ця установка була обрана, щоб звести до мінімуму доступ до даних для різних частин програми про те, для чого вони дійсно уповноважені. Налаштування зображено на рисунку 5.8.

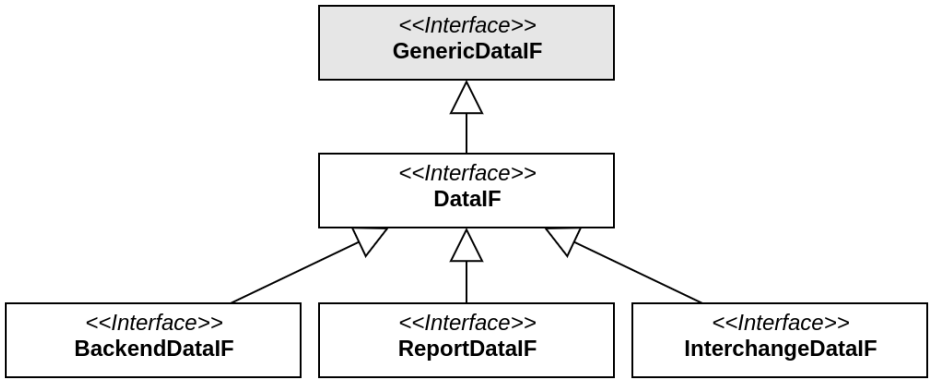



Рисунок 5.8 – Інтерфейс рівня даних KRAS. (Елементи API сірим кольором)

Для реалізації цих інтерфейсів класи реалізації були створені симетрично інтерфейсам, тобто для кожного інтерфейсу існує одна реалізація. Реалізація верхнього рівня *GeneralSqlDataAccess* успадковується від *GenericSqlDataAccess* API, тоді як інші успадковують від реалізації верхнього рівня. Крім того, існують реалізації для *DataIF* та аналогів бекенда та обміну, кожна з яких охоплює особливості *MySQL* при використанні з механізмом *InnoDB*, який не підтримує посилальну цілісність. Ці реалізації агрегують реалізації оригінального інтерфейсу та виконують необхідні завдання до або після фактичної обробки вихідної реалізації. Отже, ця частина реалізації відповідає шаблону декоратора (Decorator implementation) [69, с. 175] Приклад коду

наведено далі, де певні перевірки виконуються до того, як відбувається фактична маніпуляція даними.

```
public function assignEventTeam(Event_Team $et) {
     //* check MySQL
     $t = new Team();
     $t->team_id = $et->team;
     if (!$this->checkExistenceByTemplate($t))
          throw new Exception(“The team doesn’t exist!”);

     $t = new Event();
     $t->event_id = $et->event;
     if (!$this->checkExistenceByTemplate($t))
          throw new Exception(“The event doesn’t exist!”);
     //* end

     $this->genDataAccess->assignEventTeam($et);
}
```

Результат справді дещо складний, що також можна побачити на діаграмі класів на рисунку 5.9. Однак складність ховається за інтерфейсом і, її, не видно для бізнес-логіки. З іншого боку, це гарантує розмежування проблем, не поєднуючи регулярну обробку SQL зі спеціалізованою обробкою, та захищений доступ до даних на всіх рівнях.

Рівень даних KRAS також містить приклади, які демонструють згадану концепцію реалізації, що має внутрішній метод, додатковий до методу інтерфейсу. У наведеному далі коді зображено реалізацію методу вставки, яка передбачена інтерфейсом.

```
public function     insert(SqlEntity $e):    SqlEntity {
     $class=get_class($e);
     switch ($class) {
          case Referee::class:
               return $this->insertUser($e);
          case Event::class:
               return $this->insertEvent($e);
          case Referee_Role::class:
          case District_Referee::class:
          case Event_Team::class:
          case Event_Referee::class:
               throw new Exception(
                    ”$class cannot be inserted directly.”.
                    ”Use appropriate method instead!”
               );
          default: return $this->insertEntity($e);
     }
}
```

Як бачимо, метод за замовчуванням безпосередньо передає виклик внутрішньому методу insertEntity. Однак для певних класів, таких як наприклад Referee або виклик події перенаправляється на спеціалізовані методи, які застосовують спеціальну обробку до даного екземпляра і викликають метод insertEntity тоді. Більше того, деякі класи навіть відхиляються, оскільки відповідний випадок використання вимагає особливого ставлення, яке реалізується окремими методами.

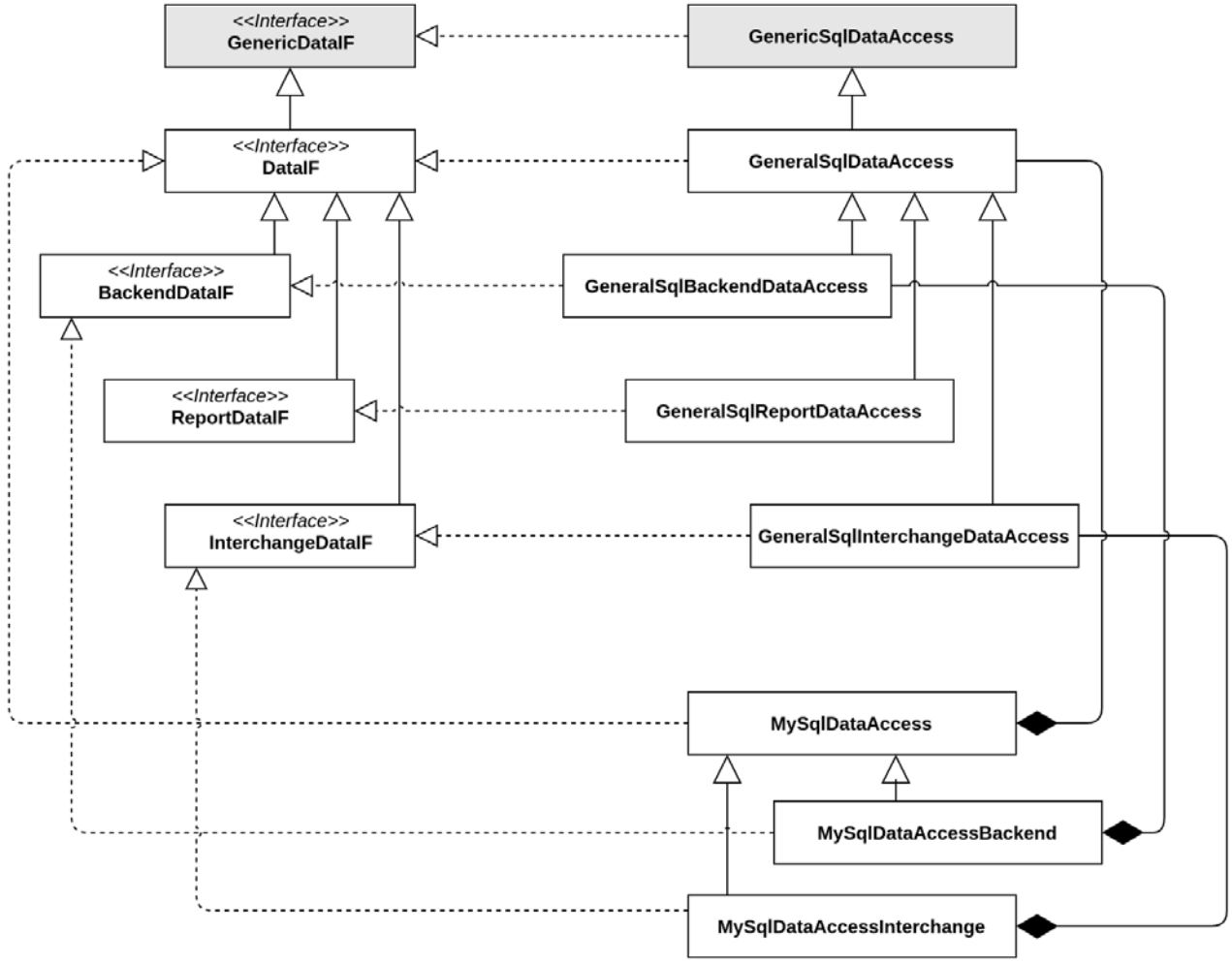



Рисунок 5.9 - Огляд класів доступу до даних KRAS для SQL.
(Елементи API сірим кольором)

Завдання подальшого розвитку, необхідного для інтеграції LDAP в застосування, полягає в тому, щоб, з одного боку, виконати цю інтеграцію без зміни інтерфейсу рівня даних для бізнес-логіки, щоб не потрібно було

змінювати бізнес-логіку. З іншого боку, код повинен бути організованими таким чином, щоб як і раніше були можливі обидві настройки, тобто SQL і Polyglot Persistence. Зрештою, застосування має бути доступно і для інших організацій і тому має сприйматися як продукт з усіма наслідками для розробки застосувань. Тому що вибір повинен бути залишений за тими, хто управляє конкретним екземпляром, чи використовують вони тільки реляційну базу даних або інтегруються з каталогом LDAP.

Перш за все, необхідно змінити порядок класів *Entity*, щоб їх можна було використовувати і для налаштування *Polyglot*. У цьому випадку для спрощення було використано функції PHP. Усі класи сутностей, які відображають реляційну таблицю, що використовуються в будь-якому налаштуванні, були представлені в одному загальному файлі. Класи *Entity*, які відрізняються між налаштуваннями *Polyglot* та *SQL*, були розміщені у двох окремих файлах. Ці два файли посилаються на файл із загальними класами *Entity* через визів *require_once* на початку, як це відбувається в PHP. При цьому можна посилатися на будь-який з двох файлів за допомогою вибору *require_once* під час налаштування доступу до даних для отримання правильного набору класів *Entity*

Поділ моделі даних призвів до чіткої прив’язки або до реляційної базі даних, або каталогу LDAP для більшості класів *Entity*, за виключенням сутності Referee. Інформація цього класу розбита відповідно до питанням про початок того, що актуально для всіх частин федерації дзюдо. Отже, особиста інформація, випуск та призначення клубу зберігаються в *LdapEntity*, а вся інформація, яка стосується виключно відділу суддівства, наприклад, Суддівська ліцензія або спеціальні інструкції, що стосуються роботи судді, залишаються в вирізаному *SqlEntity*. Це призвело до цікавої ситуації, так як урізане представлення *Referee* для налаштування *Polyglot* і аналог зі всіма атрибутами для настройки SQL мають багато спільних атрибутів. Тому файл з загальними класами *Entity* отримав абстрактний клас, в якому знаходиться визначення загальних частин, щоб не мати надлишкового коду. Останні класи *Entity*, що

представляють *Referee*, успадковуються від цього класу і додають необхідні атрибути. Для виділеного класу *Entity*, використовуваного для налаштування *Polyglot*, залишилося додати тільки *DN* в якості атрибута для зв'язку . Нарешті, для налаштування *Polyglot* створюється клас *Referee PolyglotEntity*, який містить всі атрибути, такі як повноцінний *SqlEntity*, так що не буде відмінностей для бізнес-логіки при використанні будь-якого з цих класів. Цей *PolyglotEntity* налаштований для включення очищеного *SqlEntity* під назвою «*SqlRefereePart*» і *LdapEntity* «*LdapRefereePart*», що містить дані з каталогу *LDAP*. З класами *Entity*, розташованими таким чином, бізнес-логіка може продовжувати працювати без будь-яких змін, оскільки класи як і раніше мають ті ж імена і ті ж атрибути. Для бізнес-логіки не має значення, від чого успадковується конкретний клас *Entity*, оскільки він просто отримує і повертає екземпляри *Entity* на рівень даних, який обробляє їх відповідно. Класи Entity додані в Додатку В для довідки і перегляду.

Звичайно, класи *Entity* – це лише половина завдання. Реалізація рівня даних також повинна бути розроблена для налаштування *Polyglot*, щоб екземпляри *Entity* могли бути належним чином оброблені. У цьому випадку також чітко зазначається: бізнес-логіка не повинна торкатися. Отже, визначені інтерфейси повинні обслуговуватися так само, як і раніше. Для цього підкласи класу *GenericDataAccess* були створені так само, як і клас *GeneralSqlDataAccess*, і його підкласи зображені на рисунку 5.9. Ці нові класи діють як класи маршрутизації так само, як і ті, що походять з API: вони приймають виклик і передають їх певному користувачеві. Для маршрутизації, що стосується конкретних методів проекту, вихідні класи реалізації SQL можна використовувати повторно для тих викликів методів, які є загальними для обох налаштувань. Решта методів також не обробляються цим класом маршрутизації – він залишається лише для цього завдання – маршрутизувати виклики за необхідності. Натомість буде визначено клас *PolyglotDataAccess*, який успадковується від *GenericPolyglotDataAccess*, який отримує всі залишки викликів методів і обробляє їх. Для підтримки чіткості коду методи в цій

реалізації *PolyglotDataAccess* отримують те саме ім'я, що і вихідні методи інтерфейсу. Звичайно, цей клас не реалізує жодного інтерфейсу рівня даних, оскільки він охоплює лише невелику підмножину функціональних можливостей.

Як було описано для API раніш, будь-який підклас *GenericPolyglotDataAccess* вимагає реалізації *GenericDataIF*, яка відповідає постійним об'єктам створених класів *PolyglotEntity*. Тому визначений підклас *GenericLdapDataAccess,* який може обробляти звичайні визові класів *LdapEntity,* а також отримуватиме спеціалізовані методи, які потребують нового класу *PolyglotDataAccess* для сбору та обробки необхідної інформації, що стосується каталогу *LDAP*. Теж саме необхідне зробити для частини *SQL*, оскільки вже реалізовані класи не будуть використовуватися в цих випадках і потребує додаткової обробки *SQL*. Огляд цієї реалізації показаний на рисунку 5.10. Тут відображаються лише елементи верхнього рівня. Спеціалізовані рівні, які стосуються серверної частини, звіту та обміну, було пропущене, щоб не створювати повністю незрозумілого образу.

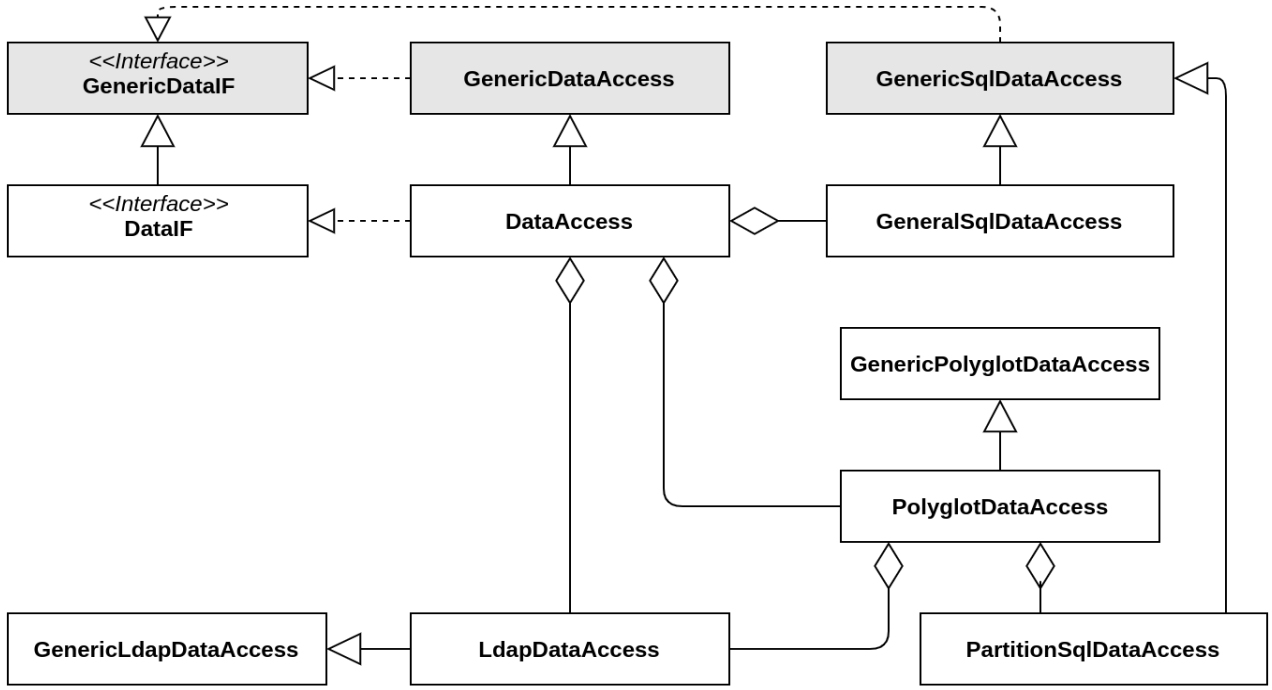



Рисунок 5.9 – Діаграма класів для для налаштування *Polyglot* проекту KRAS на верхньому рівні. (елементи API сірим кольором)

Визначення інтерфейсів для класів доступу *Polyglot*, *LDAP* і *SQL*, які були тільки що описані, не було зроблено, оскільки накладні витрати переважують переваги побудови внутрішніх фасадів всередині рівня даних. Взаємозалежності на цьому етапі досить сильні, що не заважає, оскільки це стосується внутрішніх компонентів цього шару

Підводячи підсумок, можна сказати, що додавання поняття *Polyglot Persistence* до KRAS, було необхідним здолати деякі завади, особливо з-за вимог мати вибір щодо персистентності даних. Однак зміни стосувалися лише рівня даних та були приховані за певними інтерфейсами. Сама бізнес-логіка не змінилась. Таким чином, фасадний підхід виявився успішним, і тепер застосування може працювати як з *Polyglot Persistence*, так з чисто *SQL*. Під час реалізації зрозуміло, що класи *Entity* можуть бути предметом генерації через структуру, що використовує концептуальну модель у якості джерела, тому що розділення може бути виконане за допомогою чітких правил на основі інформації з моделей даних. Стало також ясно, що реалізація рівня даних не може бути виконана автоматично, оскільки логіка даних вимагає занадто багато різних факторів та варіантів використання. Однак цього і не передбачається. Рівень даних повинен бути створений групою доступу до даних, яка, як було сказано раніше, є частиною групи розробників. Але час на розробку прикладного програмного (ПЗ) забезпечення скоротився за думкою експертів приблизно в три рази в порівнянні з ручною розробкою такого ПЗ «з нуля».

Таким чином, дизайн API підтверджено.

## 5.3 Інші практичні приклади проектування БД

Практична цінність удосконаленої AGILA MOD+, розроблених методів та API для автоматизації створення моделей даних на логічному та фізичному рівнях полягає в скороченні часу на проектування БД при розробці різних IT-проектів. Запропоновано провести оцінку загальної кількості необхідного часу в порівнянні між ручним підходом та використанням засобів AGILA MOD+, як

описано. Часові затрати було розраховано на основі емпіричних значень, отриманих за час практичної роботи студентів AUAS в рамках лекцій, практичних занять, проектів та дисертацій.

Затрати часу розраховувались на наступних кроках розробки (Development Step):

1. Створення концептуальних моделей субмоделей (partitions), тобто

– для AGILA MOD +: Аналіз ПрО, створення комплексної концептуальної моделі, аналіз критеріїв розділення та розділення на основі правил, як описано

– для ручного створення: Аналіз ПрО, розділення на основі критеріїв та створення субмоделей (partitions) для обмеженого контексту ПрО

2. Деривація логічних та фізичних моделей даних, включаючи сценарії налаштування, з концептуальних моделей даних

3. Створення API для кожної необхідної системи баз даних та фасадного API для бізнес-логіки

Крім того були визначені певні додаткові параметри:

– Рівень кваліфікації розробника

– Рівень складності ITпроекту

– Кількість субмоделей за встановленими критеріями

– Кількість мов програмування, для яких вимагається API

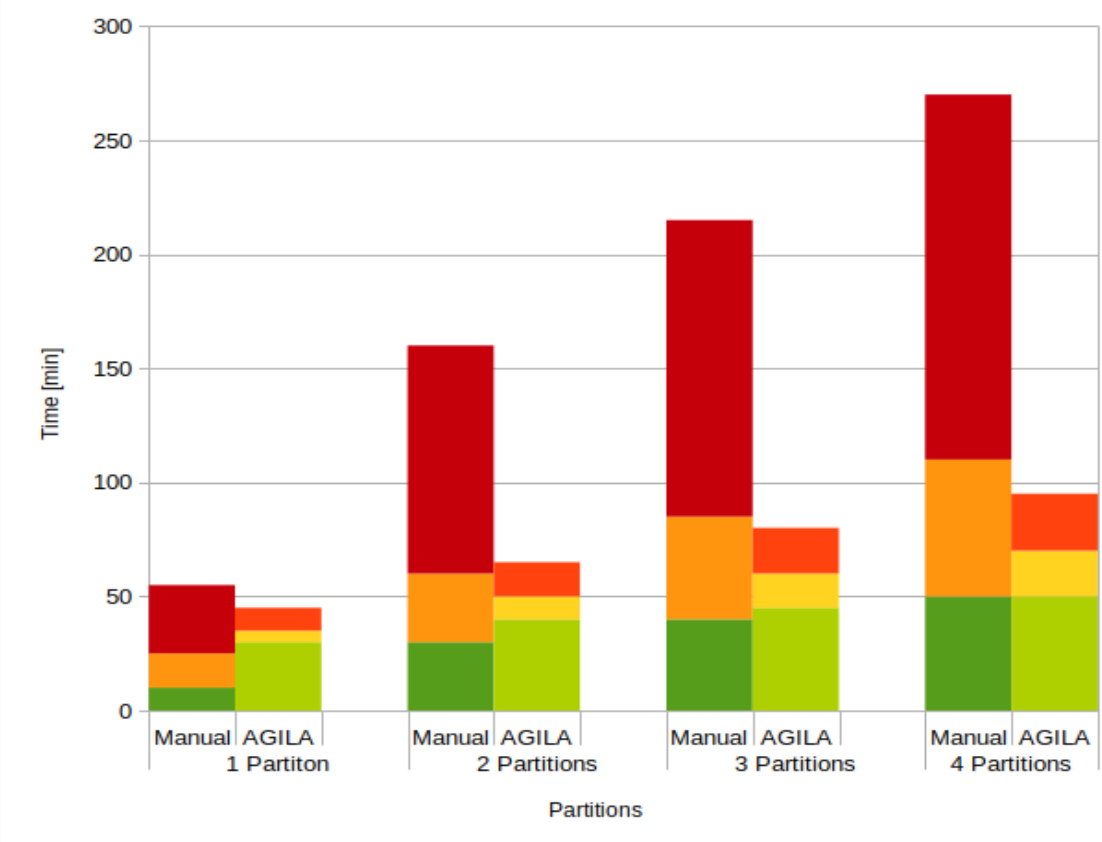
Для отримання результатів, які можна дійсно порівняти, кількість додаткових параметрів звужено до кількісті необхідних субмоделей. Для інших були зроблені наступні допущення: передбачається, що досвідченому розробнику потрібен мінімальний час для етапів розробки вручну, IT-проект вважається таким же складним, як семестрове завдання по курсу баз даних в AUAS, а API-інтерфейси потрібні лише для однієї мови програмування. Крім того, затрати часу на процес прийняття рішень про розділення не було враховано. З врахуванням зазначеного вище на основі оцінок експертів для ручного (Manual) та автоматизованого режимів (AGILA) розраховано затрати часу в хвилинах на виконання кожного кроку проектування (5.10, а). На основі

отриманих оцінок було створено діаграми необхідних часових затрат для різної кількості субмоделей (рис. 5.10,б).

| | Manual | | AGILA MOD+ | |
|---|---|---|---|---|
| Development Step | general | per DBS | general | per DBS |
| 1 | 10 | 10 | 30 | 5 |
| 2 | 0 | 15 | 0 | 5 |
| 3 | 40 | 30 | 5 | 5 |

**Remark:** For projects with only one partition,
- „general“ is omitted for steps 1 and 3 for „Manual“.
- „general“ of step 1 is reduced to 20 for „AGILA MOD+“
as no partitioning is needed.

а




б

Рисунок 5.10 – Порівняльні характеристики, щодо витрат часу на проектування БД (а – оцінка часу в хвилинах для кожного кроку проектування БД; б – діаграми необхідних часових затрат для різної кількості субмоделей з урахуванням кроків проектування)

Крім того необхідно зазначити, що було зафіксовано так званий «загальний» час розробки який залежить від кількості системних баз даних, які збираються використовувати. Він додається лише один раз за крок для будь-якої кількості субмоделей. З іншого боку, час розробки «на DBS» множиться на кількість субмоделей.

Таким чином, результати цієї роботи вже були і будуть основою для бакалаврських та магістерських дисертацій AUAS та ОНПУ, наприклад. для розробки подальших алгоритмів виводу (як це вже робиться в [75] для баз даних документів) або комплексного середовища моделювання, яке може супроводжувати і спростити певний процес проектування бази даних, щоб звільнити розробників від повторюваних завдань.

Переваги розроблених моделей та методів також продемонстровано на прикладі реінжинірингу існуючої системи дистанційного навчання (СДН) JustStart.

У список вирішуваних завдань СДН JustStart входили: можливості проходження курсів і перегляду вебінарів, які складаються з відео, а також контроль знань після кожного уроку і модульний контроль по завершенню курсу. Кожен користувач реєструється для забезпечення доступу до ресурсу і до даних про власний прогрес. Крім того, на ресурсі було реалізовано пошук за основною текстовою інформацією курсів і вебінарів, а також забезпечено збереження кешованих сторінок для забезпечення їх швидкого завантаження

Існуюча система була реалізована на основі монолітного архітектурного підходу, що уповільнювало роботу системи. Після її переведення на мікросервісне архітектурне рішення з урахуванням багатоваріантної персистентності  час доступу користувачів до СДН значно знизився при збільшені їхньої кількості.

JustStart реалізовано з використанням можливостей трьох різних СУБД: MariaDB ColumnStore, Redis, ElasticSearch. Взаємодія клієнта з ресурсом здійснюється за допомогою розробленого API. Дані користувача зберігаються в MariaDB ColumnStore. СУБД Redis використовується для кешованих сторінок і

часто оновлюваних або витягуваних даних. Зберігання інформації курсів та вебінарів, а також пошук серед освітнього контенту здійснюється за допомогою ElasticSearch.

Для оцінки швидкодії розробленої СДН JustStart було проведено її навантажувальне тестування, результати якого показані на рисунку 5.11.

Аналіз отриманих результатів показує, що завдяки поділу навантаження між трьома базами даних, середній час відгуку системи при одночасній роботі 150 користувачів склав 1,2 с. У той час як, при моделюванні роботи цієї ж кількості користувачів тільки з однієї СУБД час відгуку зріс і склав в середньому близько 2,6 с.

Таким чином, використання удосконаленої структури концептуальної моделі даних та методів створення моделей даних на логічному рівні зі багатоваріантною персистентністю дозволило зменшити трудовитрати розробника, а також спроектувати і реалізувати СДН, швидкодія якої у випадку її одночасного використання численною аудиторією в середньому в два рази більше, ніж у середньостатистичного освітнього ресурсу, представленого на ринку.


Рисунок 5.11 – Результати навантажувального тестування СДН

## 5.4 Висновки до четвертого розділу

В п’ятому розділі проведено апробацію мови концептуального моделювання AGILA MOD+ та API при розробці БД в процесі виконання IT-проектів та показано суттєве зменшення часу проектування, що підтверджено відповідним актом впровадження в Додатку Б.

З врахуванням розширеної мови концептуального моделювання AGILA MOD+ та правил деривації логічних моделей у фреймворку AGILA побудовано комплексну концептуальну модель для веб-застосування в рамках проекту KampfrichterAnmelde-System (KRAS). За допомогою розроблених методів розділення та створення отримано дві SQL- та LDAP-орієнтовані субмоделі, таку деривація було підтверджено висновками експертів Федерації дзюдо щодо уніфікації даних. Показано що данні про користувачів, наприклад, телефонні номери, адреси та інші облікові дані, які зберігаються SQL БД перекриваються, тому їх об'єднання у каталозі LDAP, де ними можна управляти централізовано, дає переваги щодо часу доступу до них із бізнес-логіки..

В результаті використання розробленого API показано, що додавання *Polyglot Persistence* до KRAS спрямовано на забезпечення вибіру щодо персистентності даних. При цьому зміни в програмному забезпеченні стосувалися лише рівня даних та були приховані за певними інтерфейсами. Сама бізнес-логіка не змінилась. Таким чином, фасадний підхід виявився успішним, і тепер застосування може працювати як з *Polyglot Persistence*, так з чисто *SQL*. Час на розробку прикладного програмного (ПЗ) забезпечення скоровився за думкою експертів приблизно в три рази в порівнянні з ручною розробкою такого ПЗ «з нуля»

Запропоновано провести оцінку загальної кількості необхідного часу в порівнянні між ручним підходом та використанням засобів AGILA MOD+. Часові затрати було розраховано на основі емпіричних значень, отриманих за час практичної роботи студентів AUAS в рамках лекцій, практичних занять, проектів та дисертацій. В порівнянні зі ручною розробкою БД показано значне

скорочення часу проектування з використанням фреймворку AGILA MOD+ , а саме в залежності від кількості субмоделей: на етапі концептуального моделювання час розробки скоротився в середньому на 20%; на етапі автоматизованої деривації в логічні та фізичні моделі даних, включно зі сценаріями налаштувань час обробки збільшився на в середньому 15%; на етапі створення API для кожної СУБД та фасадного API для бізнес-логіки рас розробки відповідного ПЗ зменшився в середньому в чотири рази.

Переваги розроблених моделей та методів також продемонстровано на прикладі реінжинірингу існуючої системи дистанційного навчання (СДН) JustStart. СНД реалізовано з використанням можливостей трьох різних СУБД: MariaDB ColumnStore, Redis, ElasticSearch. Взаємодія клієнта з ресурсом здійснюється за допомогою розробленого API. Дані користувача зберігаються в MariaDB ColumnStore. СУБД Redis використовується для кешованих сторінок і часто оновлюваних даних. Зберігання інформації курсів та вебінарів, а також пошук серед освітнього контенту здійснюється за допомогою ElasticSearch. Аналіз отриманих результатів тестування СНД показує, що завдяки поділу навантаження між трьома базами даних, середній час відклику системи при одночасній роботі 150 користувачів склав 1,2 с. У той час як, при моделюванні роботи цієї ж кількості користувачів тільки з однієї СУБД час відклику складає в середньому 2,6 с.

# ВИСНОВКИ

Дисертаційна робота містить раніше не захищені наукові положення та одержані автором нові науково-обґрунтовані результати, які полягають в розробці моделей та методів для автоматизації проектування баз даних зі багатоваріантною персистентністю.

1. Удосконалено структуру концептуальної моделі даних за рахунок розробки незалежного (*autonomous*) типу об'єктів та введення для об'єктів «сильної» участі в асоціації (*Strong Association Participants*). Введення цих елементів що дало можливість вдосконалити правила деривації із концептуальної моделі в логічної моделі даних зі багатоваріантною персистентністю в фрейморку AGILA.

2. Запропоновано в процесі проектування БД з урахуванням polyglot persistence на першому етапі розробити комплексну концептуальної моделі ПрО за допомого засобів розширеної мови AGILA MOD+, а потім розділити її на незалежні концептуальні субмоделі для подальшої їх деривації у відповідні логічні моделі, які враховують особливості персистентності даних на логічному рівні.

3. Для методичної підтримки процесу проектування БД на першому етапі концептуального моделювання розроблено метод розділення комплексної концептуальної моделі даних на незалежні субмоделі, що дозволило автоматизувати отримання відповідних логічних моделей даних включаючи реляційні та NoSQL структури. На прикладі IT-проекту KRAS показані можливості розробленого методу, що до скорочення часу проектування БД.

4. Для побудови оболонки – оточення елементів комплексної концептуальної моделі для її розділення запропоновано метод створення концептуальних субмоделей, які враховують особливості персистентності даних. Метод передбачає проведення аналізу структурних об’єктів комплексної моделі на наявність сильних зв’язків між об’єктом та асоціацією, ієрархії типів

об’єктів, значної кількісті успадкувань, а також великої кількості асоціацій Self Association.

5. Врахування удосконаленої структури концептуальної моделі даних та розроблених методів автоматизації розробки логічних моделей баз даних дозволило удосконалити мову концептуального моделювання AGILA MOD+ для врахування багатоваріантної персистентності та скоротити час проектування баз даних інформаційних систем.

6. Розроблено інтерфейс прикладного програмування (API) для забезпечення єдиного доступу до баз даних зі багатоваріантною персистентністю із бізнес-логіки. Проведено аналіз вимог до розроблювального API. Для створення «Сутностей API» та їх обробки розроблено ієрархію класів з кореневим  абстрактним класом Entity та підкласами для конкретних сховищ даних SqlEntity та LDAPEntity та також PolyglotEntity. Створено загальний інтерфейс даних як фасад бізнес-логіки (Generic Data Interface as Facade – GenericDataIF).

7. Здійснені практичні випробування отриманих моделей даних, методів розробки баз даних та запропонованого API при створенні баз даних зі багатоваріантною персистентністю для веб-застосувань Федерації баварського дзюдо - KampfrichterAnmelde-System (KRAS) та системи управління дистанційним навчанням «JustStar» показали скорочення часу на розробку баз даних в середньому на 50%. Практичне використання розроблених моделей, методів та API при розробці курсових проектів в рамках навчального процесу на кафедрі інформаційних систем ОНПУ та факультету комп’ютерних наук AUAS також показало скорочення часу на розробку баз даних на етапах концептуального, логічного та фізичного моделювання при значному зменшенню кількості помилкових рішень студентів..

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Елена Арсирий, Matthias Kolonko, Sabine Müllenbach, Борис Трофимов, Мария Глава Concept modeling for developing databases with polyglot persistence as a direction of business digitalization. *Устойчивое развитие: Международный журнал*. (Варна, Болгария) – 2020. - №2-2020. – С. 22-29. Періодиний журнал Технічного університету м. Варна, Болгарія. https://maurorg77.wixsite.com/maur-org/anotaciya

*2.* Müllenbach Sabine, Kern-Bausch Lore, Kolonko Matthias. Conceptual Modeling Language Agila Mod. *Herald of Advanced Information Technology, 2019* Vol. 2, No. 4, pp. 246-258.. *https://hait.opu.ua/?fetch=articles&with=info&id=35*

3. Olena O. Arsirii., Maria G. Glava, Matthias Kolonko., Alina O. Glumenko. Information technology of supporting architectural solutions using polyglot persistence concept in learning management systems . *Applied Aspects of Information Technology*. 2020; Vol. 3 No. 2: – P 13-31 . https://aait.opu.ua/?fetch=articles&with=info&id=42

4. M. Kolonko and S. Müllenbach, "Polyglot Persistence in Conceptual Modeling for Information Analysis," *2020 10th International Conference on Advanced Computer Information Technologies (ACIT)*, Deggendorf, Germany, 2020, pp. 590-594, doi:10.1109/ACIT49673.2020.9208928.

https://ieeexplore.ieee.org/document/9208928 *Видання входить до міжнародних наукометричних баз* ***Scopus, IEEE Xplore***.

5. Matthias Kolonko and Sabine Müllenbach. Modern Databases – What‘s really new about them?“ In: *Proceedings of the V. Ukrainian-German conference „Informatics. Culture. Technology.“ (IKT).* (Odessa, Ukraine). Odessa National Polytechnical University, May 2017, pp. 35–37. ISBN: 978-966-2666-18-2

6. Matthias Kolonko and Sabine Müllenbach. „From E³R to AGILA MOD – A Syntax Evaluation and Advancement towards a Comprehensive Conceptual Semantically Irreducible Modeling Language.“ In: *Proceedings of Modern Information Technology 2018.* (Odessa, Ukraine). Odessa National Polytechnical University, May 2018, pp. 172–173. UDC: 004.55. ISBN: 978-617-7046-50-8

7. Matthias Kolonko and Sabine Müllenbach and Elena Arsirii and Boris Trofymov. „Extensions to the Conceptual Modeling Language AGILA MOD“. In: *Proceedings of the VI. Ukrainian-German conference „Informatics. Culture. Technology.“ (IKT).* (Odessa, Ukraine). Odessa National Polytechnical University, September 2018, pp. 38–39. UDC: 004.652 ISSN: 2522-1523

8. Matthias Kolonko and Sabine Müllenbach. „An Analysis of Identification and Integrity Properties of Database Systems“ In: *Proceedings of the VII. Ukrainian-German conference „Informatics. Culture. Technology.“ (IKT).* (Odessa, Ukraine). Odessa National Polytechnical University, September 2019, pp. 53–57. UDC: 004.652. ISSN: 2522-1523

9. Глуменко А.О., Стельмах Д.Е., Маттиас Колонко, Арсирий Е.А. Использование многовариантного хранения и мультимодельных СУБД для обработки разнообразных данных. Сучасні інформаційні технології : мат. X Міжнар. наук. конф. студентів та молодих вчених, м. Одеса, 14 – 15 травня 2020 р. / МОН України; Одес. Нац. політех. ун-т; Ін-т комп’ют. систем. Одеса : Наука і техніка, 2020. С. 118–119.

10. R. Elmasri and S. B. Navathe, Fundamentals of Database Systems. The Benjamin/Cummings Publishing Company, Inc., 1994, isbn: 0-8053-1748-1.

11. , Fundamentals of Database Systems, 7th ed. Pearson Education Limited, 2017, isbn: 978-1-292-09761-9.

12. P. Sauer, “Informationsmodellierung,” in Taschenbuch Datenbanken, T. Kudraß, Ed., 2nd ed. Fachbuchverlag Leipzig im Carl Hanser Verlag, 2015, ch. 2, isbn: 978-3-446-43508-7.

13. L. Kern-Bausch and M. Jeckle, “Informationsmodellierung und logischer Datenbankentwurf,” in Taschenbuch der Informatik, U. Schneider and D. Werner, Eds., 4th ed. Fachbuchverlag Leipzig im Carl Hanser Verlag, 2001, ch. 14.2, isbn: 3-446-21753-3.

14. C. W. Bachmann, “Data Structure Diagrams,” Data Base, vol. 1, no. 2, pp. 4–10, 1969.

15. P. P.-S. Chen, “The Entity-Relationship Model - Toward a Unified View of Data.,” ACM Transactions on Database Systems, vol. 1, no. 1, pp. 9–36, Mar. 1976. [Online]. Available: https://dl.acm.org/ft_gateway.cfm?id=320440&ftid=6223&dwn=1&CFID=24967371&CFTOKEN=f4004ca2054b3bfc7E54B1B0-F4CE-8A6A-AB8144D3E71B65CA.

16. J. Wintraecken, The NIAM Information Analysis Method: Theory and Practice. Springer Science & Business Media, Dec. 2012, isbn: 9789400904514.

17. T. Halpin and T. Morgan, Information Modeling and Relational Databases 2nd ed. Morgan Kaufmann, Jul. 2010, isbn: 978-0-12-373568-3.

18. ISO Central Secretary, “Information technology — Modeling Languages — Part 2: Syntax and Semantics for IDEF1X97 (IDEFobject),” International Organization for Standardization, Geneva, Switzerland, Standard ISO/IEC/IEEE 31320-2:2012, Sep. 2012. [Online]. Available: https://www.iso.org/standard/60614.html.

19. Knowledge Based Systems, Inc. (Nov. 28, 2019). IDEF1X - Data Modeling Method, [Online]. Available: http://www.idef.com/idef1x-data-modeli ng-method/.

20. Wikipedia contributors. (Nov. 18, 2019). IDEF1X. accessed 2019-11-28, [Online]. Available: https://en.wikipedia.org/wiki/IDEF1X.

21. itl.nist.gov. (Dec. 3, 2019). Figure A3 21 Example of Phase Three Function View Diagram, [Online]. Available: https://commons.wikimedia.org/wiki/File:A3_21_Example_of_Phase_Three_Function_View_Diagram. jpg.

22. Digital Equipment Corporation, EXPRESS Language Reference Manual. Maynard, Massachusetts: Digital Equipment Corporation, Feb. 1992.

23. D. Schenck and P. Wilson, Information Modeling the EXPRESS Way. Oxford: Oxford University Press, 1994, isbn: 0-19-508714-3.

24. ISO Central Secretary, “Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual,” International Organization for Standardization, Geneva, Switzerland, Standard ISO/IEC 10303-11:2004, Nov. 2004. [Online]. Available: https://www.iso.org/standard/38047. html.

25. C. Kecher and A. Salvanos, UML 2.5 - das umfassende Handbuch. Bonn, Germany: Rheinwerk-Verlag, 2015, isbn: 978-3-8362-2977-7.

26. L. Kern-Bausch and B. G. Wenzel, “Informationsanalyse,” Leibnitz Rechenzentrum, Munich, Tech. Rep., 1980.

27. ——, “Database Design for Relational System: Why, Who, How?” European ORACLE Users Group Newsletter, no. 10, 1986.

28. L. Kern-Bausch and M. Jeckle, “From a Semantically Irreducible Formulated Conceptual Schema to an UML Model,” in The Unified Modeling Language: Technical Aspects and Applications. Physica-Verlag, 1998, pp. 32–44, isbn: 3-7908-1105-X.

29. M. Jeckle, “Ableitung objektorientierter Strukturen aus konzeptuellen Schemata,” Unpublished PhD thesis (Defense did not take place due to premature death of author.) Ulm University, May 2004.

30. T. Neward. (Jun. 26, 2006). The Vietnam of Computer Science, [Online]. Available: http://blogs.tedneward.com/post/the-vietnam-of-computer-science.

31. M. Fowler. (May 8, 2012). OrmHate, [Online]. Available: https://martinfowler.com/bliki/OrmHate.html.

32. S. Edlich, A. Friedland, J. Hampe, B. Brauer, and M. Bru¨ckner, NoSQL - Einstieg in die Welt nichtrelationaler Datenbanken. Carl Hanser Verlag München, 2011, isbn: 978-3-446-42753-2.

33. E. Hewitt, Cassandra - the definitive guide. North Sebastopol, California, USA: O’Reilly Media, Inc., Nov. 2010, isbn: 978-1-4493-9041-9.

34. A. Singh, Instant Cassandra Query Language. Packt Publishing, Sep. 2013, isbn: 978-1-78328-271-5.

35. D. Vohra, Apache deepak2016 Primer. Apress, Nov. 2016, isbn: 978-1-48422423-6.

36. C. Miceli, M. Miceli, S. Jha, H. Kaiser, and A. Merzky, “Programming Abstractions for Data Intensive Computing on Clouds and Grids,” in 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, 2009, p. 478, isbn: 978-1-4244-3935-5.

37. J. C. Anderson, J. Lehnhardt, and N. Slater, CouchDB: The Definitive Guide. O'Reilly Media Inc., Jan. 2010, isbn: 978-0-596-15589-6.

38. D. Garrett, P. Bakkum, K. Banker, S. Verch, and T. Hawkins, MongoDB in Action, second. Manning Publications, Apr. 2016, isbn: 978-1-61729-160-9.

39. S. Gupta, Neo4J Essentials. Packt Publishing, Feb. 2015, isbn: 978-1-78355517-8.

40. M. Fowler. (May 25, 2004). IntegrationDatabase, [Online]. Available: https: //martinfowler.com/bliki/IntegrationDatabase.html.

41. ——, (May 25, 2004). ApplicationDatabase, [Online]. Available: https: //martinfowler.com/bliki/ApplicationDatabase.html.

42. M. Aslett. (Apr. 6, 2011). What we talk about when we talk about NewSQL, [Online]. Available: https://blogs.the451group.com/information_ management/2011/04/06/what-we-talk-about-when-we-talk-aboutnewsql/.

43. ——, (Apr. 15, 2011). NoSQL, NewSQL and Beyond: The answer to SPRAINed relational databases, [Online]. Available: https://blogs.the451group.com/information_management/2011/04/15/nosql-newsql-and-beyond/.

44. M. Fowler and P. Sadalage, NoSQL Distilled. Addison-Wesley Professional, 2012, isbn: 978-0-321-82662-6. [Online]. Available: https://martinfowler .com/books/nosql.html.

45. M. Fowler. (Nov. 16, 2011). PolyglotPersistence, [Online]. Available: https: //martinfowler.com/bliki/PolyglotPersistence.html.

46. C. Buckett. (Jan. 9, 2007). Using Visio 2003 for Enterprise Architects to Design SQL Server Databases, [Online]. Available: https://www.codeproject.com/Articles/16992/Using-Visio-2003-for-EnterpriseArchitects-to-Desi.

47. P. Johannesson, M. L. Lee, S. W. Liddle, A. L. Opdahl, and Ó. P. López, Eds., Proceedings of the 34th International Conference on Conceptual Modeling (ER), (Stockholm, Sweden), Springer, Cham, Oct. 2015, isbn: 978-3319-25263-6.

48. I. Comyn-Wattiau, K. Tanaka, I.-Y. Song, S. Yamamoto, and M. Saeki, Eds., Proceedings of the 35th International Conference on Conceptual Modeling (ER), (Gifu, Japan), Springer, Cham, Nov. 2016, isbn: 978-3-319-46396-4.

49. H. C. Mayr, G. Guizzardi, H. Ma, and O. Pastor, Eds., Proceedings of the 36th International Conference on Conceptual Modeling (ER), (Valencia, Spain), Springer, Cham, Nov. 2017, isbn: 978-3-319-69903-5.

50. J. C. Trujillo, K. C. Davis, X. Du, Z. Li, T. W. Ling, G. Li, and M. L. Lee, Eds., Proceedings of the 37th International Conference on Conceptual Modeling (ER), (Xi'an, China), Springer, Cham, Oct. 2018, isbn: 978-3030-00846-8.

51. A. H. F. Laender, B. Pernici, E.-P. Lim, and J. P. M. de Oliveira, Eds., Proceedings of the 38th International Conference on Conceptual Modeling (ER), (Salvador, Brazil), Springer, Cham, Nov. 2019, isbn: 978-3-030-33222-8.

52. I. Holubová, M. Svoboda, and J. Lu, "Unified Management of Multi-model Data (Vision Paper)," in Proceedings of the 38th International Conference on Conceptual Modeling (ER), (Salvador, Brazil), A. H. F. Laender, B. Pernici, E.-P. Lim, and J. P. M. de Oliveira, Eds., Springer, Cham, Nov. 2019, pp. 439–447, isbn: 978-3-030-33222-8.

53. .-Y. Song, Y. Zhu, H. Ceong, and O. Thonggoom, "Methodologies for Semiautomated Conceptual Data Modeling from Requirements," in Proceedings of the 34th International Conference on Conceptual Modeling (ER), (Stockholm, Sweden), P. Johannesson, M. L. Lee, S. W. Liddle, A. L. Opdahl, and Ó. P. López, Eds., Springer, Cham, Oct. 2015, pp. 18–31, isbn: 978-3-31925263-6.

54. V. Herrero, A. Abelló, and O. Romero, "NOSQL Design for Analytical Workloads: Variability Matters," in Proceedings of the 35th International Conference on Conceptual Modeling (ER), (Gifu, Japan), I. Comyn-Wattiau, K. Tanaka, I.-Y. Song, S. Yamamoto, and M. Saeki, Eds., Springer, Cham, Nov. 2016, pp. 50–64, isbn: 978-3-319-46396-4.

55. A. Stellmann and J. Greene, Learning Agile: Understanding Scrum, XP, Lean, and Kanban. O'Reilly, 2014, isbn: 1449331920.

56. P. Loos, Datenstrukturierung in der Fertigung. Munich, Germany: R. Oldenburg Verlag, 1992, isbn: 3486222864.

57. B. Meyer, Object Oriented Software Construction, 2nd ed. Upper Saddle River, NJ: Prentice Hall, Nov. 1998, isbn: 0136291554.

58. S. Müllenbach. (Sep. 23, 2020). Ableitung vom konzeptuellen zum logischen Schema, [Online]. Available: https://ohs.informatik.hs-augsburg.de: 4443/web/cmsanw$.main?i_id=1597808695&i_anw=bine&i_dok=dbabl. abl_beg.

59. D. Baisley, M. Björkander, C. Bock, S. Cook, P. Desfray, N. Dykman, A. Ek, D. Frankel, E. Gery, Ø. Haugen, S. Iyengar, C. Kobryn, B. Møller-Pedersen, J. Odell, G. Övergaard, K. Palmkvist, G. Ramackers, J. Rumbaugh, B. Selic, T. Weigert, and T. Williams, “OMG Unified Modeling Language (OMG UML), Superstructure - Version 2.2,” Object Management Group, Tech. Rep., 2009. [Online]. Available: https://www.omg.org/spec/UML/2.2/ Superstructure/PDF.

60. K. M. Adams, “Introduction to Non-functional Requirements,” in Nonfunctional Requirements in Systems Analysis and Design. Cham: Springer International Publishing, 2015, pp. 45–72, isbn: 978-3-319-18344-2. doi: 10.1007/978-3-319-18344-2_3. [Online]. Available: https://doi.org/10.1007/978-3-319-18344-2_3.

61. J. M. Fernandes and R. J. Machado, “Requirements,” in Requirements in Engineering Projects. Cham: Springer International Publishing, 2016, pp. 45–64, isbn: 978-3-319-18597-2. doi: 10.1007/978-3-319-18597-2_3. [Online]. Available: https://doi.org/10.1007/978-3-319-18597-2_3.

62. ISO Central Secretary, “Systems and software engineering — Vocabulary,” International Organization for Standardization, Geneva, Switzerland, Standard ISO/IEC/IEEE 24765:2017, Sep. 2017. [Online]. Available: https: //www.iso.org/standard/71952.html.

63. Wikipedia contributors. (Oct. 2, 2019). Non-functional requirement. accessed 2020-02-21, [Online]. Available: https://en.wikipedia.org/wiki/ Non-functional_requirement.

64. ArangoDB Inc. (Mar. 17, 2020). ArangoDB Graphs, [Online]. Available:

65. https://www.arangodb.com/docs/stable/graphs.html.

66. M. Kolonko, “Konzeption und Implementierung einer Abstraktionsschicht fu¨r den Datenzugriff zur Integration verschiedener Systeme auf Datenbankebene,” Diploma Thesis, Augsburg University of Applied Sciences, 2008.

67. M. Schaarschmidt, F. Gessert, and N. Ritter, “Towards Automated Polyglot Persistence,” in Datenbanksysteme fu¨r Business, Technologie und Web (BTW 2015), T. Seidl, N. Ritter, H. Scho¨ning, K.-U. Sattler, T. Ha¨rder, S. Friedrich, and W. Wingerath, Eds., Bonn: Gesellschaft für Informatik e.V., 2015, pp. 73–82.

68. G. Copeland and D. Maier, “Making Smalltalk a Database System,” in SIGMOD ’84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts: Association for Computing Machinery, 1984, pp. 316–325, isbn: 0897911288. doi: 10.1145/602259.602300. [Online]. Available: https://doi.org/10.1145/602259.602300.

69. J. Melton, Querying XML : XQuery, XPath, and SQL/XML in context. Morgan Kaufmann Publishers, 2006, isbn: 9780080540160.

70. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley, 1997, isbn: 0-201-63361-2.

71. K. Eilebrecht, Pattern kompakt : Entwurfsmuster fu¨r effektive Softwareentwicklung, 5th ed. Springer Vieweg, 2019, isbn: 9783662579367.

72. D. Popel, Learning PHP data objects : a beginner’s guide to PHP data objects, database connection abstraction library for PHP 5. Birmingham, U.K.: Packt Pub., 2007, isbn: 9781847192660.

73. L. Welling, PHP and MySQL web development, 5th ed. Hoboken, NJ: AddisonWesley, 2017, isbn: 9780133038644.

74. PHP Group. (Jun. 9, 2020). PHP Data Objects, [Online]. Available: https://www.php.net/manual/en/book.pdo.php.

75. C. Baumgart, “Ableitung des konzeptuellen Datenbankschemas nach AGILA MOD+ auf das Paradigma der Dokument-Datenbanken,” Bachelor’s Thesis, Augsburg University of Applied Sciences, 2020.

76. T. Dalzell, Ed., The Routledge Dictionary of Modern American Slang and Unconventional English. 2 Park Square, Milton Park, Abingdon, Oxon OX14 4RN: Routledge, 2009, isbn: 978-0-415-37182-7.

# ДОДАТОК А

## СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

*Наукові праці, в яких опубліковані основні наукові результати:*

1. Елена Арсирий, Matthias Kolonko, Sabine Müllenbach, Борис Трофимов, Мария Глава Concept modeling for developing databases with polyglot persistence as a direction of business digitalization. *Устойчивое развитие: Международный журнал*. (Варна, Болгария) – 2020. - №2-2020. – С. 22-29. Періодиний журнал Технічного університету м. Варна, Болгарія. https://maurorg77.wixsite.com/maur-org/anotaciya

*2*. Müllenbach Sabine, Kern-Bausch Lore, Kolonko Matthias. Conceptual Modeling Language Agila Mod. *Herald of Advanced Information Technology, 2019* Vol. 2, No. 4, pp. 246-258. *Видання входить до міжнародних наукометричних баз Academia.edu, ROAD, Національна бібліотека України імені В. І. Вернадського, Djerelo, Україніка наукова, РИНЦ, Index Copernicus. https://hait.opu.ua/?fetch=articles&with=info&id=35*

3. Olena O. Arsirii., Maria G. Glava, Matthias Kolonko., Alina O. Glumenko. Information technology of supporting architectural solutions using polyglot persistence concept in learning management systems . *Applied Aspects of Information Technology*. 2020; Vol. 3 No. 2: – P 13-31 . *Видання входить до міжнародних наукометричних баз Academia.edu, ROAD, Национальная библиотека Украины имени В. И. Вернадского, Djerelo, Українікананаукова, РИНЦ, Index Copernicus* https://aait.opu.ua/?fetch=articles&with=info&id=42

4. M. Kolonko and S. Müllenbach, "Polyglot Persistence in Conceptual Modeling for Information Analysis," *2020 10th International Conference on Advanced Computer Information Technologies (ACIT)*, Deggendorf, Germany, 2020, pp. 590-594, doi:10.1109/ACIT49673.2020.9208928.

https://ieeexplore.ieee.org/document/9208928 *Видання входить до міжнародних наукометричних баз* ***Scopus, IEEE Xplore***.

*Наукові праці апробаційного характеру та праці, в яких опубліковані додаткові наукові результати:*

1. Matthias Kolonko and Sabine Müllenbach. Modern Databases – What‘s really new about them?“ In: *Proceedings of the V. Ukrainian-German conference „Informatics. Culture. Technology.“ (IKT).* (Odessa, Ukraine). Odessa National Polytechnical University, May 2017, pp. 35–37. ISBN: 978-966-2666-18-2

2. Matthias Kolonko and Sabine Müllenbach. „From E³R to AGILA MOD – A Syntax Evaluation and Advancement towards a Comprehensive Conceptual Semantically Irreducible Modeling Language.“ In: *Proceedings of Modern Information Technology 2018.* (Odessa, Ukraine). Odessa National Polytechnical University, May 2018, pp. 172–173. UDC: 004.55. ISBN: 978-617-7046-50-8

3. Matthias Kolonko and Sabine Müllenbach and Elena Arsirii and Boris Trofymov. „Extensions to the Conceptual Modeling Language AGILA MOD“. In: *Proceedings of the VI. Ukrainian-German conference „Informatics. Culture. Technology.“ (IKT).* (Odessa, Ukraine). Odessa National Polytechnical University, September 2018, pp. 38–39. UDC: 004.652 ISSN: 2522-1523

4. Matthias Kolonko and Sabine Müllenbach. „An Analysis of Identification and Integrity Properties of Database Systems“ In: *Proceedings of the VII. Ukrainian-German conference „Informatics. Culture. Technology.“ (IKT).* (Odessa, Ukraine). Odessa National Polytechnical University, September 2019, pp. 53–57. UDC: 004.652. ISSN: 2522-1523

5. Глуменко А.О., Стельмах Д.Е., Маттиас Колонко, Арсирий Е.А. Использование многовариантного хранения и мультимодельных СУБД для обработки разнообразных данных. Сучасні інформаційні технології : мат. X Міжнар. наук. конф. студентів та молодих вчених, м. Одеса, 14 – 15 травня 2020 р. / МОН України; Одес. Нац. політех. ун-т; Ін-т комп’ют. систем. Одеса : Наука і техніка, 2020. С. 118–119.

# ДОДАТОК Б

# АКТ ВПРОВАДЖЕННЯ

Hochschule Augsburg Postfach 11 06 05 D-86031 Augsburg

Hochschule
Augsburg University of
Applied Sciences

Fakultät für
Informatik

Aktenzeichen | Ihr Schreiben vom / Ihr Zeichen | Augsburg, den

Regarding: Confirmation of use of PhD research results of Matthias Kolonko

Dear Sir or Madam,
the faculty of computer science at Augsburg University of Applied Sciences hereby confirms that the research results of the currently ongoing promotion of Mr. Matthias Kolonko with the dissertation „Conceptual Models and Methods for Developing Databases with Polyglot Persistence“ is already actively used at our faculty for teaching and further research with kind approval of Mr. Kolonko. As his research directly draws on the teaching content concerning conceptual modeling in the lecture „Databases“ of our lecturer Sabine Müllenbach, she certainly integrates the results as further developed methods in this lecture. Moreover, as the thesis itself suggests, it has already been used for further research within the scope of a currently ongoing bachelor thesis of our student Christian Baumgart.
The mentioned approval of Mr. Kolonko for using his research is also attached to this document.

Kind regards,

Prof. Dr. Jürgen Scholz
Dean of Faculty of Computer Science
University of Applied Sciences Augsburg

Prof. Dr. Sabine Müllenbach
Lecturer

**Prof. Dr. rer. nat.**
**Jürgen Scholz**
Dekan
Fakultät für Informatik

Hochschule für angewandte
Wissenschaften Augsburg
University of Applied Sciences
An der Hochschule 1
D-86161 Augsburg
Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info@hs-augsburg.de
Anfahrt:
Campus am Roten Tor
Anfahrt Schülestraße
ÖPNV:
Straßenbahnlinien 2 und 3
Haltestelle Rotes Tor
Straßenbahnlinie 6
Buslinien 32, 35 und 36
Haltestelle Fachhochschule

# ДОДАТОК В

## JAVA IMPLEMENTATION OF THE GENERICPOLYGLOT-DATAACCESS

```
package de.hs_augsburg.agila.api.polyglot;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import javax.annotation.Nonnull;
import javax.annotation.Nullable;
import org.apache.commons.collections4.OrderedMap;
import de.hs_augsburg.agila.api.DataAccessorMap;
import de.hs_augsburg.agila.api.Entity;
import de.hs_augsburg.agila.api.GenericDataIF;
public class GenericPolyglotDataAccess
    implements GenericDataIF<PolyglotEntity> {
  protected
    final DataAccessorMap<GenericDataIF<?>> accessorMap;
  public GenericPolyglotDataAccess(
      DataAccessorMap<GenericDataIF<?>> accessorMap) {
    this.accessorMap = accessorMap;
  }
  @Override
  public Class<PolyglotEntity> getHandlingUnit() {
    return PolyglotEntity.class;
  }

  /*
   * Transactional functionalities touch
   * the particular database.
   */
    @Override
  public void beginTransaction() {}
    @Override
  public void commit() {}
    @Override
  public void rollback() {}
    /* CRUD functions */
    @Override
  @Nullable
  public <E extends PolyglotEntity>
  E getById(@Nonnull E e) {
    e.amendMissingPersistents();
     try {
```

```
    @SuppressWarnings("unchecked")
    E result = (E) e.getClass().getDeclaredConstructor()
        .newInstance();
    Map<String, Object> connectingValues = null;
    List<Entity> missing = new ArrayList<>();
     for (Entity persistent : e.getPersistentEntities()) {
      if (persistent.getKeyValues().containsValue(null)) {
        missing.add(persistent);
      } else {
        Entity received = getAccessorFor(persistent)
            .getById(persistent);
        // no entry found with all given keys
        if (received == null) return null;
        result.addPersistentEntity(received);
        if (connectingValues == null) connectingValues = e
            .getConnectingValues(persistent.getClass());
      }
    }

    if (missing.size() > 0) {
      if (connectingValues.size() == 0)
        throw new RuntimeException(
            "Couldn't find all persistent entries by key "
                + "and no connecting attributes defined! "
                + e.getClass());
              for (Entity persistent : missing) {
        e.setConnectingValues(persistent.getClass(),
            connectingValues);
        List<Entity> received = getAccessorFor(persistent)
            .getByTemplate(persistent, null);
                  // the given value combination in the key
        // attributes of PolyglotEntity instance
        // resulted in nothing found.
        if (received.size() == 0) return null;
        if (received.size() > 1)
          throw new RuntimeException(
              "Found more than one entry with given key "
              + "and connecting value combination!");
        result.addPersistentEntity(received.get(0));
      }
    }
          result.fillFromPersistentEntities();
    return result;
  } catch (InstantiationException | IllegalAccessException
      | IllegalArgumentException
      | InvocationTargetException | NoSuchMethodException
      | SecurityException e1) {
    // TODO Auto-generated catch block
    throw new RuntimeException(e1);
  }
}
```

```
  @Override
@Nonnull
public <E extends PolyglotEntity> List<E> getByTemplate(
    @Nonnull E t,
    @Nullable OrderedMap<String, Boolean> order) {
  t.amendMissingPersistents();
     // check each persistent for amount of returns and begin
  // with least amount
  List<Entity> persistents = t.getPersistentEntities();
      Long minAmount = null;
  Integer selectedPersistentKey = null;
      for (int i = 0; i < persistents.size(); i++) {
    Entity curEntity = persistents.get(i);
    Long current = getAccessorFor(curEntity)
        .countByTemplate(curEntity);
    if (minAmount == null || minAmount > current) {
      minAmount = current;
      selectedPersistentKey = i;
    }
  }
      if (selectedPersistentKey == null)
    // this could only happen if there are no persistents
    // at all!
    throw new RuntimeException(
        "No minimal amount of results found!");
  // obviously nothing found
  if (minAmount == 0) return new ArrayList<>();
      List<Entity> received = getAccessorFor(
      persistents.get(selectedPersistentKey))
          .getByTemplate(
              persistents.get(selectedPersistentKey),
              null);
      List<E> draftResult = new ArrayList<>();
  for (Entity r : received) {
    try {
      @SuppressWarnings("unchecked")
      E current = (E) t.getClass()
          .getDeclaredConstructor().newInstance();
              current.addPersistentEntity(r);
      Map<String, Object> connectingValues = t
          .getConnectingValues(r.getClass());
              for (int i = 0; i < persistents.size(); i++) {
        // this has already been received!
        if (i == selectedPersistentKey) continue;
                  t.setConnectingValues(
            persistents.get(i).getClass(),
            connectingValues);
        List<Entity> receivedConnected = getAccessorFor(
            persistents.get(i))
                .getByTemplate(persistents.get(i), null);
        // nothing found --> don't add to current
```

```
      if (receivedConnected.size() == 0) continue;
      if (receivedConnected.size() > 1)
        throw new RuntimeException(
            "Found more than one persistent Entity "
            + "despite using connection attributes!");
      current.addPersistentEntity(
          receivedConnected.get(0));
    }
            draftResult.add(current);
  } catch (InstantiationException
      | IllegalAccessException
      | IllegalArgumentException
      | InvocationTargetException
      | NoSuchMethodException | SecurityException e) {
    // TODO Auto-generated catch block
    throw new RuntimeException(e);
  }
}
    List<E> result = new ArrayList<>();
for (E draft : draftResult) {
  if (draft.getPersistentEntities().size() == t
      .getPersistentEntities().size()) {
    draft.fillFromPersistentEntities();
    result.add(draft);
  }
}
    if (order != null) {
  Collections.sort(result, new Comparator<Entity>() {
    @Override
    public int compare(Entity o1, Entity o2) {
      for (Map.Entry<String, Boolean> orderField : order
          .entrySet()) {
        final int factor = (orderField.getValue() ? 1
            : -1);
        try {
          Field f = o1
              .getFieldByName(orderField.getKey());
          Object v1 = f.get(o1);
          Object v2 = f.get(o2);
                      if (v1 == v2) continue;
          if (v1 == null && v2 != null)
            return -1 * factor;
          if (v1 != null && v2 == null)
            return 1 * factor;
                      if (v1 instanceof Comparable) {
                          @SuppressWarnings("unchecked")
            Comparable<Object> c1 =
                (Comparable<Object>) v1;
                          @SuppressWarnings("unchecked")
            Comparable<Object> c2 =
                (Comparable<Object>) v2;
```

```
                            int compared = c1.compareTo(c2);
              if (compared != 0) return compared * factor;
            }
          } catch (NoSuchFieldException
              | IllegalAccessException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            continue;
          }
        }
        return 0;
      }
    });
  }

  return result;
}
  @Override
@Nonnull
public <E extends PolyglotEntity>
long countByTemplate(@Nonnull E t) {
  return getByTemplate(t, null).size();
}
  @Override
@Nonnull
public <E extends PolyglotEntity>
  boolean checkExistenceByTemplate(@Nonnull E t) {
  return countByTemplate(t) > 0;
}
  @Override
@Nonnull
public <E extends PolyglotEntity> E insert(@Nonnull E e) {
  try {
    // check for polyglot references first
    checkPolyglotReferences(e);
          // now the insert can be handed over to the
    // corresponding persistent accessors.
    PolyglotDefinition def = e.getDefinition();
          e.amendMissingPersistents();
          Map<String, Object> connectingValues = null;
    for (Entity persistent : e.getPersistentEntities()) {
      if (connectingValues != null) {
        for (Map.Entry<String, Object> entry :
          connectingValues.entrySet()) {
          String mappedConnectingAttribute = def
              .getMappedField(entry.getKey(),
                  persistent.getClass());
          persistent
              .getFieldByName(mappedConnectingAttribute)
              .set(persistent, entry.getValue());
        }
```

```
      }
              Entity result = getAccessorFor(persistent)
          .insert(persistent);
      e.addPersistentEntity(result);
              if (connectingValues == null) {
        connectingValues = new HashMap<>();
        for (String connectingAttribute : def
            .getConnectingAttributes()) {
          String mappedConnectingAttribute = def
              .getMappedField(connectingAttribute,
                  persistent.getClass());
          connectingValues.put(connectingAttribute,
              result
                  .getFieldByName(
                      mappedConnectingAttribute)
                  .get(result));
        }
      }
    }
          e.fillFromPersistentEntities();
    return e;
  } catch (Exception ex) {
    // TODO
    throw new RuntimeException(ex);
  }
}
  @Override
@Nullable
public <E extends PolyglotEntity>
boolean update(@Nonnull E e) {
  checkPolyglotReferences(e);
  if (e.getPersistentEntities().size() == 0)
    throw new IllegalArgumentException(
        "Given PolyglotEntity did not contain " +
        "any persistents!");
  boolean updated = false;
  e.updatePersistentEntities();
  for (Entity persistent : e.getPersistentEntities()) {
    updated |= getAccessorFor(persistent)
        .update(persistent);
  }
  return updated;
}
  @Override
@Nonnull
public <E extends PolyglotEntity>
boolean delete(@Nonnull E e) {
  // check the key fields of the persistents! Only delete
  // if all necessary keys are present.
  if (e.getKeyValues().containsValue(null))
    throw new IllegalArgumentException(
```

```
        "Not all key fields have been set to delete.");
      boolean deletedAll = true;
  e.updatePersistentEntities();
  for (Entity persistent : e.getPersistentEntities()) {
    deletedAll &= getAccessorFor(persistent)
        .delete(persistent);
  }
  return deletedAll;
}
  @Override
@Nonnull
public <E extends PolyglotEntity>
boolean deleteByTemplate(@Nonnull E t) {
  boolean deletedAll = true;
  // must use getByTemplate. Using deleteByTemplate of
  // every single accessor would lead to unpredictable
  // results!
  for (PolyglotEntity i : getByTemplate(t, null)) {
    deletedAll &= delete(i);
  }
  return deletedAll;
}
  protected <E extends PolyglotEntity>
void checkPolyglotReferences(E e) {
  Map<Field, Object> mod = e.getFieldValues(true);

  List<String> fieldNames = new ArrayList<>();
  for (Field f : mod.keySet()) {
    fieldNames.add(f.getName());
  }
      Set<PolyglotReference> refCheck = new HashSet<>();
  for (PolyglotReference ref : e
      .getPolyglotReferences()) {
    for (String refFieldName : ref.fieldNames) {
      if (fieldNames.contains(refFieldName)) {
        refCheck.add(ref);
      }
    }
  }
      for (PolyglotReference ref : refCheck) {
    try {
      Entity refEntity = ref.refEntity.getConstructor()
          .newInstance();
              List<Field> keyFields = refEntity.getKeyFields();
      if (keyFields.size() != ref.fieldNames.size())
        throw new RuntimeException("Number of dey fields "
            + "and ref fields do not match!"); // TODO
              for (int i = 0; i < keyFields.size(); i++) {
        keyFields.get(i).set(refEntity,
            e.getFieldByName(ref.fieldNames.get(i))
                .get(e));
```

```
        }
                if (!getAccessorFor(refEntity)
            .checkExistenceByTemplate(refEntity)) {
          throw new RuntimeException("Values do not "
              + "reference an existing entry."); // TODO
        }
      } catch (InstantiationException
          | IllegalAccessException
          | IllegalArgumentException
          | InvocationTargetException | NoSuchFieldException
          | NoSuchMethodException | SecurityException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
      }
    }
  }
    @Nonnull
  protected <E extends Entity>
  GenericDataIF<E> getAccessorFor(@Nonnull E e) {
    @SuppressWarnings("unchecked")
    Class<E> clazz = (Class<E>) e.getClass();
    return getAccessorByClass(clazz);
  }
    @Nonnull
  protected <E extends Entity> GenericDataIF<E>
  getAccessorByClass(@Nonnull Class<E> clazz) {
    @SuppressWarnings("unchecked")
    GenericDataIF<E> accessor = (GenericDataIF<E>)
      accessorMap.getAccessor(clazz);
  return accessor;
  }
  }
```

SECTION C. ENTITY CLASSES FOR PROJECT “KRAS”

Common Entities

```
<?php
require_once 'model/generic/generic_data.if.php';
require_once 'model/generic/sql/sql_entity.php';
require_once 'lib/i18n_exception.inc.php';

class District extends SqlEntity {
    const TABLENAME = self::class;
    const FD_DISTRICT_ID = array("district_id",
    self::TYPE_NUMERIC,3,false);
    const FD_NAME = array("name",self::TYPE_VARCHAR,30,false);
    const FD_IN_DISTRICT = array("in_district",
    self::TYPE_NUMERIC,3,true);
    const CO_PK_ID = array(District::FD_DISTRICT_ID[0]);
    const CO_UNQ_NAME = array(District::FD_NAME[0]);
    const CO_FK_IN_DISTRICT = array(
    array(District::FD_IN_DISTRICT[0]),
    array(District::TABLENAME,District::CO_PK_ID));
```

```
    const DB_USER_ACCESS = array(true,false,false,false);
    const DEFAULT_ORDERING = array(
    District::FD_DISTRICT_ID[0] => true);
    public $district_id;
    public $name;
    public $in_district;
    public static function withId(int $districtId): self {
    $instance = new self();
    $instance->district_id = $districtId;
    return $instance;
  }
    public function validateValues(): void {
    if (! is_null($this->district_id)
        && (! settype($this->district_id, "integer")
          || $this->district_id <= 0)
        ) {
      throw new I18nException(
        'exception.db_access.wrong_id_type_int',
        $this->district_id .
        " provided - expected positive integer!",
        -55);
    }
    if ($this->in_district == $this->district_id) {
      $this->in_district = null;
    }
  }
}
class Role extends SqlEntity {
  const TABLENAME = self::class;
    const FD_ROLE_ID = array("role_id",self::TYPE_NUMERIC,5,
    false);
    const FD_ROLE_NAME = array("role_name",self::TYPE_VARCHAR,
    40,false);
    const FD_PERMISSION_KEY = array("permission_key",
    self::TYPE_NUMERIC,10,false);
    const CO_PK_ID = array(Role::FD_ROLE_ID[0]);
    const CO_UNQ_NAME = array(Role::FD_ROLE_NAME[0]);
    const DB_USER_ACCESS = array(true,false,false,false);
    const DEFAULT_ORDERING = array(Role::FD_ROLE_ID[0]=>true);
    public $role_id;
    public $role_name;
    public $permission_key;
    public static function withId(int $roleId): self {
    $instance = new self();
    $instance->role_id = $roleId;
    return $instance;
  }
    public function validateValues(): void {
    if (! is_null($this->role_id) &&
      (! settype($this->role_id, "integer") ||
      $this->role_id <= 0)) {throw new I18nException(
```

```
        'exception.db_access.wrong_id_type_int',
        $this->role_id .
        " provided - expected positive integer!", - 55);}
  }
}
class Role_District extends SqlEntity {
    const TABLENAME = self::class;
    const FD_ROLE = array("role",Role::FD_ROLE_ID[1],
    Role::FD_ROLE_ID[2],false);
    const FD_DISTRICT = array("district",
      District::FD_DISTRICT_ID[1],
      District::FD_DISTRICT_ID[2], false);
    const FD_EDIT = array("edit",self::TYPE_BOOLEAN,null,
    false,0);
    const CO_PK_ROLE_DISTRICT = array(
    Role_District::FD_ROLE[0],Role_District::FD_DISTRICT[0]);
    const CO_FK_ROLE = array(array(Role_District::FD_ROLE[0]),
    array(Role::TABLENAME,Role::CO_PK_ID),self::OD_CASCADE);
    const CO_FK_DISTRICT = array(
    array(Role_District::FD_DISTRICT[0]),
    array(District::TABLENAME,District::CO_PK_ID),
    self::OD_CASCADE);
    const DB_USER_ACCESS = array(true,false,false,false);
    public $role;
    public $district;
    public $edit;
    public static function withId(int $roleId,
      int $districtId): self {
    $instance = new self();
    $instance->role = $roleId;
    $instance->district = $districtId;
    return $instance;
  }
}
class License extends SqlEntity {
  const TABLENAME = self::class;
   const FD_LICENSE_ID = array("license_id",
    self::TYPE_VARCHAR,5,false);
    const FD_LICENSE_LABEL = array("license_label",
    self::TYPE_VARCHAR,30,false);
    const CO_PK_ID = array(License::FD_LICENSE_ID[0]);
    const DB_USER_ACCESS = array(true,false,false,false);
    const DEFAULT_ORDERING = array(
    License::FD_LICENSE_ID[0] => true);
    public $license_id;
    public $license_label;
    public static function withId(string $licenseId): self {
    $instance = new self();
    $instance->license_id = $licenseId;
    return $instance;
  }
```

```
}
abstract class SqlReferee {
    const TABLENAME = "Referee";
    const FD_USER_ID = array("user_id",self::TYPE_INTEGER,
    null,false,self::DEFAULT_SEQUENCE);
    const FD_REMARK = array("remark",self::TYPE_VARCHAR,200,
    true);
    const FD_REF_SINCE = array("ref_since",self::TYPE_DATE,
    null,true);
    const FD_LICENSE = array("license",
    License::FD_LICENSE_ID[1],License::FD_LICENSE_ID[2],
    false);
    const FD_HOMEDISTRICT = array("homedistrict",
    District::FD_DISTRICT_ID[1],District::FD_DISTRICT_ID[2],
    false);
    const FD_ICAL_UUID = array("ical_uuid",self::TYPE_UUID,
    null,true);
    const FD_RATING = array("rating",self::TYPE_VARCHAR,2,
    true);
    const FD_LAST_LOGIN = array("last_login",
    self::TYPE_TIMESTAMP,null,true);
    const FD_MODIFIED = array("modified",
    self::TYPE_AUTO_TIMESTAMP,null,false);
    const FD_MODIFIER = array("modifier",self::FD_USER_ID[1],
    self::FD_USER_ID[2],true);
    const CO_PK_ID = array(self::FD_USER_ID[0]);
    const CO_UNQ_UUID = array(self::FD_ICAL_UUID[0]);
    const CO_FK_HOMEDISTRICT = array(
    array(self::FD_HOMEDISTRICT[0]),
    array(District::TABLENAME,District::CO_PK_ID));
    const CO_FK_LICENSE = array(array(self::FD_LICENSE[0]),
    array(License::TABLENAME,License::CO_PK_ID));
    const CO_FK_MODIFIER = array(array(self::FD_MODIFIER[0]),
    array(self::TABLENAME,self::CO_PK_ID),self::OD_NULL);
    const DB_USER_ACCESS = array(true,true,true,true);
    const DEFAULT_ORDERING = array(
      self::FD_SURNAME[0]   => true,
      self::FD_FIRSTNAME[0] => true);
    public $user_id;
    public $remark;
    public $ref_since;
    public $license;
    public $homedistrict;
    public $ical_uuid;
    public $rating;
    public $last_login;
    public $modified;
    public $modifier;
}
class District_Referee extends SqlEntity {
    const TABLENAME = self::class;
```

```
    const FD_DISTRICT = array("district",
    District::FD_DISTRICT_ID[1],District::FD_DISTRICT_ID[2],
    false);
    const FD_REFEREE = array("referee",
    SqlReferee::FD_USER_ID[1],SqlReferee::FD_USER_ID[2],
    false);
    const FD_EVALUATOR = array("evaluator",self::TYPE_BOOLEAN,
   null,false,0);
    const CO_PK_DISTRICT_REF = array(
    District_Referee::FD_DISTRICT[0],
    District_Referee::FD_REFEREE[0]);
    const CO_FK_DISTRICT = array(
    array(District_Referee::FD_DISTRICT[0]),
    array(District::TABLENAME,District::CO_PK_ID),
    self::OD_CASCADE);
    const CO_FK_REFEREE = array(
    array(District_Referee::FD_REFEREE[0]),
    array(SqlReferee::TABLENAME,SqlReferee::CO_PK_ID),
    self::OD_CASCADE);
    const DB_USER_ACCESS = array(true,true,true,true);
    public $district;
    public $referee;
    public $evaluator;
    public static function withId(int $districtId,
      int $userId): self {
    $instance = new self();
    $instance->district = $districtId;
    $instance->referee = $userId;
    return $instance;
  }
}
class Referee_Role extends SqlEntity {
    const TABLENAME = self::class;
    const FD_REFEREE = array("referee",
    SqlReferee::FD_USER_ID[1],SqlReferee::FD_USER_ID[2],
    false);
    const FD_ROLE = array("role",Role::FD_ROLE_ID[1],
    Role::FD_ROLE_ID[2],false);
    const FD_DEPUTY = array("deputy",self::TYPE_BOOLEAN,null,
    false,0);
    const CO_PK_REF_ROLE = array(Referee_Role::FD_REFEREE[0],
    Referee_Role::FD_ROLE[0]);
    const CO_FK_REFEREE = array(
    array(Referee_Role::FD_REFEREE[0]),
    array(SqlReferee::TABLENAME,SqlReferee::CO_PK_ID),
    self::OD_CASCADE);
    const CO_FK_ROLE = array(array(Referee_Role::FD_ROLE[0]),
    array(Role::TABLENAME,Role::CO_PK_ID),self::OD_CASCADE);
    const DB_USER_ACCESS = array(true,true,true,true);
    public $referee;
```

```
  public $role;
   public $deputy;
   public static function withId(int $userId,
      int $roleId): self {
    $instance = new self();
    $instance->referee = $userId;
    $instance->role = $roleId;
    return $instance;
  }
}
class Referee_License_History extends SqlEntity {
    const TABLENAME = self::class;
    const FD_REFEREE = array("referee",
    SqlReferee::FD_USER_ID[1],SqlReferee::FD_USER_ID[2],
    false);
    const FD_LICENSE_DATE = array("license_date",
    self::TYPE_DATE,null,false);
    const FD_LICENSE = array("license",
    License::FD_LICENSE_ID[1],License::FD_LICENSE_ID[2],
    false);
    const FD_REMARK = array("remark",self::TYPE_VARCHAR,100,
    true);
    const CO_PK_REF_DATE = array(
    Referee_License_History::FD_REFEREE[0],
    Referee_License_History::FD_LICENSE_DATE[0]);
    const CO_FK_REFEREE = array(
    array(Referee_License_History::FD_REFEREE[0]),
    array(SqlReferee::TABLENAME,SqlReferee::CO_PK_ID),
    self::OD_CASCADE);
    const DB_USER_ACCESS = array(true,true,true,true);
    const DEFAULT_ORDERING = array(
    Referee_License_History::FD_LICENSE_DATE[0] => true);
    public $referee;
    public $license_date;
    public $license;
    public $remark;
    public static function withId(int $userId,
      string $date): self {
    $instance = new self();
    $instance->referee = $userId;
    $instance->license_date = $date;
    return $instance;
  }
}
class Session extends SqlEntity {
    const TABLENAME = self::class;
    const FD_REFEREE = array("referee",
    SqlReferee::FD_USER_ID[1],SqlReferee::FD_USER_ID[2],
    false);
    const FD_LAST = array("last",self::TYPE_TIMESTAMP,null,
    false);
```

```
    const CO_PK_REFEREE = array(Session::FD_REFEREE[0]);
    const CO_FK_REFEREE = array(array(Session::FD_REFEREE[0]),
    array(SqlReferee::TABLENAME,SqlReferee::CO_PK_ID),
    self::OD_CASCADE);
    const DB_USER_ACCESS = array(true,true,true,true);
    public $referee;
    public $last;
    public static function withId(int $userId): self {
    $instance = new self();
    $instance->referee = $userId;
    return $instance;
  }
}
class Event extends SqlEntity {
    const TABLENAME = self::class;
    const FD_EVENT_ID = array("event_id",self::TYPE_INTEGER,
    null,false,self::DEFAULT_SEQUENCE);
    const FD_DISTRICT = array("district",
    District::FD_DISTRICT_ID[1],District::FD_DISTRICT_ID[2],
    false);
    const FD_EVENT_DATE = array("event_date",self::TYPE_DATE,
    null,false);
    const FD_EVENT_DAYS = array("event_days",
    self::TYPE_NUMERIC,2,false,1);
    const FD_VENUE=array("venue",self::TYPE_VARCHAR,80,true);
    const FD_NAME = array("name",self::TYPE_VARCHAR,80,false);
    const FD_MAT_AMOUNT = array("mat_amount",
    self::TYPE_NUMERIC,2,true);
    const FD_REMARK = array("remark",self::TYPE_VARCHAR,200,
    true);
    const FD_TRAINING = array("training",self::TYPE_BOOLEAN,
    null,true);
    const FD_ANNOUNCEMENT_URL = array("announcement_url",
    self::TYPE_VARCHAR,250,true);
    const FD_SENT=array("sent",self::TYPE_BOOLEAN,1,false,0);
    const FD_CREATED = array("created",
    self::TYPE_DEFAULT_TIMESTAMP,null,false);
    const FD_CREATOR = array("creator",
    SqlReferee::FD_USER_ID[1],SqlReferee::FD_USER_ID[2],
    true);
    const FD_MODIFIED = array("modified",
    self::TYPE_AUTO_TIMESTAMP,null,false);
    const FD_MODIFIER = array("modifier",
    SqlReferee::FD_USER_ID[1],SqlReferee::FD_USER_ID[2],
    true);
    const CO_PK_ID = array(Event::FD_EVENT_ID[0]);
    const CO_UNQ_DIST_DATE_NAME = array(Event::FD_DISTRICT[0],
    Event::FD_EVENT_DATE[0],Event::FD_NAME[0]);
    const CO_FK_DISTRICT = array(array(Event::FD_DISTRICT[0]),
    array(District::TABLENAME,District::CO_PK_ID));
    const CO_FK_CREATOR = array(array(Event::FD_CREATOR[0]),
```

```
    array(SqlReferee::TABLENAME,SqlReferee::CO_PK_ID),
    self::OD_NULL);
    const CO_FK_MODIFIER = array(array(Event::FD_MODIFIER[0]),
    array(SqlReferee::TABLENAME,SqlReferee::CO_PK_ID),
    self::OD_NULL);
    const CO_CK_DAYS = self::FD_EVENT_DAYS[0] . " > 0";
    const DB_USER_ACCESS = array(true,true,true,true);
    const LEAGUE_VENUE = "liga";
    public $event_id;
    public $district;
    public $event_date;
    public $event_days;
    public $venue;
    public $name;
    public $mat_amount;
    public $remark;
    public $training;
    public $announcement_url;
    public $sent;
    public $created;
    public $creator;
    public $modified;
    public $modifier;
    public static function withId(int $eventId): self {
    $instance = new self();
    $instance->event_id = $eventId;
    return $instance;
  }
    public function validateValues(): void {
    if (isset($this->event_days)
        && !is_null($this->event_days)
        && $this->event_days < 1){
      $this->event_days = 1;
    }
  }
}
class Team extends SqlEntity {
  const TABLENAME = self::class;
   const FD_TEAM_ID = array("team_id",self::TYPE_INTEGER,
    null,false,self::DEFAULT_SEQUENCE);
   const FD_TEAM_NAME = array("team_name",self::TYPE_VARCHAR,
    50,false);
    const FD_DISTRICT = array("district",
    District::FD_DISTRICT_ID[1],District::FD_DISTRICT_ID[2],
    false);
    const CO_PK_ID = array(Team::FD_TEAM_ID[0]);
    const CO_UNQ_NAME_DISTRICT = array(Team::FD_TEAM_NAME[0],
    Team::FD_DISTRICT[0]);
    const CO_FK_DISTRICT = array(array(Team::FD_DISTRICT[0]),
    array(District::TABLENAME,District::CO_PK_ID));
    const DB_USER_ACCESS = array(true,true,true,true);
```

```
    const DEFAULT_ORDERING = array(
    Team::FD_TEAM_NAME[0] => true);
    public $team_id;
    public $team_name;
    public $district;
    public static function withId(int $teamId): self {
    $instance = new self();
    $instance->team_id = $teamId;
    return $instance;
  }
}
class Event_Team extends SqlEntity {
  const TABLENAME = self::class;
  const FD_EVENT = array("event",Event::FD_EVENT_ID[1],
    Event::FD_EVENT_ID[2],false);
    const FD_TEAM = array("team",Team::FD_TEAM_ID[1],
    Team::FD_TEAM_ID[2],false);
    const CO_PK_EVENT_TEAM = array(Event_Team::FD_EVENT[0],
    Event_Team::FD_TEAM[0]);
    const CO_FK_EVENT = array(array(Event_Team::FD_EVENT[0]),
    array(Event::TABLENAME,Event::CO_PK_ID),
    self::OD_CASCADE);
    const CO_FK_TEAM = array(array(Event_Team::FD_TEAM[0]),
    array(Team::TABLENAME,Team::CO_PK_ID));
    const DB_USER_ACCESS = array(true,true,true,true);
    public $team;
    public $event;
   public static function withId(int $teamId,
      int $eventId): self {
    $instance = new self();
    $instance->team = $teamId;
    $instance->event = $eventId;
    return $instance;
  }
}
class Event_Referee extends SqlEntity {
  const TABLENAME = self::class;
    const FD_EVENT = array("event",Event::FD_EVENT_ID[1],
    Event::FD_EVENT_ID[2],false);
    const FD_REFEREE = array("referee",
    SqlReferee::FD_USER_ID[1],SqlReferee::FD_USER_ID[2],
    false);
    const FD_TEAM = array("team",Team::FD_TEAM_ID[1],
    Team::FD_TEAM_ID[2],true);
    const FD_MARK = array("mark",self::TYPE_NUMERIC,3.2,true);
    const FD_ASSIGNED = array("assigned",self::TYPE_BOOLEAN,
    null,true);
    const FD_CANCELED = array("canceled",self::TYPE_BOOLEAN,
    null,true);
    const FD_REFEREE_IN_CHARGE = array("referee_in_charge",
    self::TYPE_BOOLEAN,null,true);
```

```
const FD_EVALUATOR = array("evaluator",self::TYPE_BOOLEAN,
null,true);
const FD_SCALE = array("scale",self::TYPE_BOOLEAN,null,
true);
const FD_CARPOOL = array("carpool",self::TYPE_NUMERIC,2,
true);
const FD_CREATED = array("created",
self::TYPE_DEFAULT_TIMESTAMP,null,false);
const FD_CREATOR = array("creator",
SqlReferee::FD_USER_ID[1],SqlReferee::FD_USER_ID[2],
true);
const FD_MODIFIED = array("modified",
self::TYPE_AUTO_TIMESTAMP,null,false);
const FD_MODIFIER = array("modifier",
SqlReferee::FD_USER_ID[1],SqlReferee::FD_USER_ID[2],
true);
const CO_PK_EVENT_REFEREE = array(
Event_Referee::FD_EVENT[0],Event_Referee::FD_REFEREE[0]);
const CO_FK_EVENT = array(
array(Event_Referee::FD_EVENT[0]),
array(Event::TABLENAME,Event::CO_PK_ID),
self::OD_CASCADE);
const CO_FK_REFEREE = array(
array(Event_Referee::FD_REFEREE[0]),
array(SqlReferee::TABLENAME,SqlReferee::CO_PK_ID),
self::OD_CASCADE);
const CO_FK_EVENT_TEAM = array(
array(Event_Referee::FD_EVENT[0],
  Event_Referee::FD_TEAM[0]),
array(Event_Team::TABLENAME,
  Event_Team::CO_PK_EVENT_TEAM));
const CO_FK_CREATOR = array(
array(Event_Referee::FD_CREATOR[0]),
array(SqlReferee::TABLENAME,SqlReferee::CO_PK_ID),
self::OD_NULL);
const CO_FK_MODIFIER = array(
array(Event_Referee::FD_MODIFIER[0]),
array(SqlReferee::TABLENAME,SqlReferee::CO_PK_ID),
self::OD_NULL);
const DB_USER_ACCESS = array(true,true,true,true);
public $event;
public $referee;
public $team;
public $mark;
public $assigned;
public $canceled;
public $referee_in_charge;
public $evaluator;
public $scale;
public $carpool;
public $created;
```

```
    public $creator;
    public $modified;
    public $modifier;
    public static function withId(int $eventId,
      int $userId): self {
    $instance = new self();
    $instance->event = $eventId;
    $instance->referee = $userId;
    return $instance;
  }
}
```

C.2. Entities for Pure SQL Setup

```
<?php
require_once 'model/entities/entities_common.php';
class Belt extends SqlEntity {
  const TABLENAME = Belt::class;
   const FD_BELT_ID = array("belt_id",self::TYPE_VARCHAR,7,
    false);
   const FD_BELT_LABEL = array("belt_label",
    self::TYPE_VARCHAR,30,false);
   const CO_PK_ID = array(self::FD_BELT_ID[0]);
   const DB_USER_ACCESS = array(true,false,false,false);
   const DEFAULT_ORDERING = array(
    self::FD_BELT_LABEL[0] => true,
    self::FD_BELT_ID[0] => true);
   public $belt_id;
   public $belt_label;
   public static function withId(string $beltId): self {
    $instance = new self();
    $instance->belt_id = $beltId;
    return $instance;
  }
}
class Club extends SqlEntity {
  const TABLENAME = Club::class;
  const FD_MEMBERSHIPNO = array("membershipno",
    self::TYPE_NUMERIC,6,false);
  const FD_REGION = array("region",self::TYPE_VARCHAR,20,
    false);
  const FD_CLUB_NAME = array("club_name",self::TYPE_VARCHAR,
    50,false);
  const CO_PK_MEMBERSHIPNO = array(self::FD_MEMBERSHIPNO[0]);
  const CO_UNQ_REGION_NAME = array(self::FD_REGION[0],
    self::FD_CLUB_NAME[0]);
  const DB_USER_ACCESS = array(true,false,false,false);
  const DEFAULT_ORDERING = array(self::FD_REGION[0] => true,
    self::FD_CLUB_NAME[0] => true);
  public $membershipno;
  public $region;
  public $club_name;
  public static function withId(int $membershipno): self {
```

```
    $instance = new self();
    $instance->membershipno = $membershipno;
    return $instance;
  }
  public function validateValues(): void {
    if (! is_null($this->membershipno) &&
      (! settype($this->membershipno, "integer") ||
      $this->membershipno <= 0)) {throw new I18nException(
        'exception.db_access.wrong_id_type_int',
        $this->membershipno .
        " provided - expected positive integer!", -55);}
  }
}
class Referee extends SqlReferee {
  const FD_USERNAME = array("username",self::TYPE_VARCHAR,
    20,false);
  const FD_PASSWORD = array("password",self::TYPE_VARCHAR,
    100,false);
   const FD_SURNAME = array("surname",self::TYPE_VARCHAR,50,
    false);
   const FD_FIRSTNAME = array("firstname",self::TYPE_VARCHAR,
    50,false);
   const FD_STREET = array("street",self::TYPE_VARCHAR,100,
    true);
   const FD_ZIP = array("zip",self::TYPE_CHAR,5,true);
   const FD_CITY = array("city",self::TYPE_VARCHAR,100,true);
   const FD_PHONE_PRIVATE = array("phone_private",
    self::TYPE_VARCHAR,30,true);
   const FD_PHONE_BUSINESS = array("phone_business",
    self::TYPE_VARCHAR,30,true);
   const FD_PHONE_MOBILE = array("phone_mobile",
    self::TYPE_VARCHAR,30,true);
   const FD_EMAIL = array("email",self::TYPE_VARCHAR,200,
    false);
   const FD_FAX = array("fax",self::TYPE_VARCHAR,30,true);
   const FD_CLUB = array("club",Club::FD_MEMBERSHIPNO[1],
    Club::FD_MEMBERSHIPNO[2],false,0);
   const FD_VERIFIED = array("verified",self::TYPE_BOOLEAN,
    null,false);
   const FD_BIRTHDATE = array("birthdate",self::TYPE_DATE,
    null,false);
   const FD_BELT = array("belt",Belt::FD_BELT_ID[1],
    Belt::FD_BELT_ID[2],false);
   const FD_PASSNO = array("passno",self::TYPE_VARCHAR,10,
    true);
   const CO_UNQ_USERNAME = array(self::FD_USERNAME[0]);
   const CO_UNQ_NAME_BIRTH = array(self::FD_SURNAME[0],
    self::FD_FIRSTNAME[0],self::FD_BIRTHDATE[0]);
   const CO_UNQ_PASSNR = array(self::FD_PASSNO[0]);
   const CO_FK_BELT = array(array(self::FD_BELT[0]),
    array(Belt::TABLENAME,Belt::CO_PK_ID));
```

```
   const CO_FK_CLUB = array(array(self::FD_CLUB[0]),
    array(Club::TABLENAME,Club::CO_PK_MEMBERSHIPNO));
   public $username;
    public $password;
    public $surname;
    public $firstname;
    public $street;
    public $zip;
    public $city;
    public $phone_private;
    public $phone_business;
    public $phone_mobile;
    public $email;
    public $fax;
    public $club;
    public $verified;
    public $birthdate;
    public $belt;
    public $passno;
}
```

C.3. Entities for Polyglot Setup

```
<?php
require_once 'model/entities/entities_common.php';
class Belt extends LdapEntity {
  const BASE_DN = "";
  const OBJECT_CLASSES = array();
  const FD_BELT_ID = array("belt_id","belt_id",false,
    self::TYPE_UNKNOWN);
  const FD_BELT_LABEL = array("belt_label","belt_label",
    false,self::TYPE_UNKNOWN);
  const RDN_FIELD = self::FD_BELT_ID[0];
   public $belt_id;
   public $belt_label;
   public static function withId(string $beltId): self {
    $instance = new self();
    $instance->belt_id = $beltId;
    return $instance;
  }
}
class Region extends LdapEntity {
  const BASE_DN = "ou=regions";
  const OBJECT_CLASSES = array("region");
  const FD_REGION_NAME = array("region_name","name",false,
    self::TYPE_UNKNOWN);
  const RDN_FIELD = self::FD_REGION_NAME[0];
  public $region_name;
}
class Club extends LdapEntity {
  const BASE_DN = "ou=clubs";
  const OBJECT_CLASSES = array("club");
  const FD_MEMBERSHIPNO = array("membershipno",
```

```
    "membershipno",false,self::TYPE_UNKNOWN);
  const FD_REGION = array("region","region",false,
    self::TYPE_REFERENCE_DN,Region::class);
  const FD_CLUB_NAME = array("club_name","club_name",false,
    self::TYPE_UNKNOWN);
  const RDN_FIELD = self::FD_MEMBERSHIPNO[0];
  public $membershipno;
  public $region;
  public $club_name;
  public static function withId(int $membershipno): self {
    $instance = new self();
    $instance->membershipno = $membershipno;
    return $instance;
  }
  public function validateValues(): void {
    if (! is_null($this->membershipno) &&
      (! settype($this->membershipno, "integer") ||
      $this->membershipno <= 0)) {throw new I18nException(
        'exception.db_access.wrong_id_type_int',
        $this->membershipno .
        " provided - expected positive integer!", -55);}
  }
}
class SqlRefereePart extends SqlReferee {
   const FD_DN = array("dn",self::TYPE_VARCHAR,100,false);
    const CO_UNQ_DN = array(self::FD_DN[0]);
    public $dn;
}
class LdapRefereePart extends LdapEntity {
  const CN_FORMAT = '%1$s %2$s';
  const DISPLAY_NAME_FORMAT = '%2$s, %1$s';
  const BASE_DN = "ou=users";
  const OBJECT_CLASSES = array("mailUser","inetOrgPerson",
    "judoPerson");
  const MEMBER_OF = array(
    "cn=kras-user,ou=groups,dc=judoverband-bayern,dc=de");
  // field definition syntax: entity_attribute, attribute
  // name, type, optional
  const FD_USERNAME = array("username","uid",false,
    self::TYPE_UNKNOWN);
  const FD_PASSWORD = array("password","Password",false,
    self::TYPE_PASSWORD);
  const FD_SURNAME = array("surname","sn",false,
    self::TYPE_UNKNOWN);
  const FD_FIRSTNAME = array("firstname","givenName",false,
    self::TYPE_UNKNOWN);
  const FD_BIRTHDATE = array("birthdate","BirthDate",false,
    self::TYPE_UNKNOWN);
  const FD_STREET = array("street","street",true,
    self::TYPE_UNKNOWN);
  const FD_ZIP = array("zip","postalCode",true,
```

```
  self::TYPE_UNKNOWN);
const FD_CITY = array("city","l",true,self::TYPE_UNKNOWN);
const FD_PHONE_PRIVATE = array("phone_private",
  "homePhone",true,self::TYPE_UNKNOWN);
const FD_PHONE_BUSINESS = array("phone_business",
  "Telephone",true,self::TYPE_UNKNOWN);
const FD_PHONE_MOBILE = array("phone_mobile","mobile",
  true,self::TYPE_UNKNOWN);
const FD_EMAIL = array("email","Email",false,
  self::TYPE_UNKNOWN);
  const FD_FAX = array("fax","Fax",true,self::TYPE_UNKNOWN);
  const FD_CLUB = array("club","Club",false,
  self::TYPE_REFERENCE_DN,Club::class);
  const FD_VERIFIED = array("verified","verified",false,
  self::TYPE_UNKNOWN);
  const FD_BELT = array("belt","belt",false,
  self::TYPE_REFERENCE_DN,Belt::class);
  const FD_PASSNO = array("passno","passno",true,
  self::TYPE_UNKNOWN);
  const FD_CN = array("cn","cn",false,self::TYPE_UNKNOWN);
  const FD_DISPLAY_NAME = array("displayName","displayName",
  false,self::TYPE_UNKNOWN);
  const RDN_FIELD = self::FD_USERNAME[0];
  public $username;
  public $password;
  public $surname;
  public $firstname;
  public $birthdate;
  public $street;
  public $zip;
  public $city;
  public $phone_private;
  public $phone_business;
  public $phone_mobile;
  public $email;
  public $fax;
  public $club;
  public $verified;
  public $belt;
  public $passno;
  public $cn;
  public $displayName;
  /* questionable if necessary at all in LDAP */
public $modified;
  public $modifier;
  public function getModifications() {
  $mods = parent::getModifications();
    // CN and DisplayName are built automatically and must
  // not be changed by hand!
  unset(
    $mods[
```

```
        self::FD_CN[
          LdapEntity::FIELD_DEF_POS_NAME
        ]
      ]
    );
    unset(
      $mods[
        self::FD_DISPLAY_NAME[
          LdapEntity::FIELD_DEF_POS_NAME
        ]
      ]
    );
      // update common name and display name if firstname or
    // surname have been changed.
    if (array_key_exists(self::FD_FIRSTNAME[
          LdapEntity::FIELD_DEF_POS_NAME], $mods)
        || array_key_exists(self::FD_SURNAME[
          LdapEntity::FIELD_DEF_POS_NAME], $mods)) {
      $mods[self::FD_CN[LdapEntity::FIELD_DEF_POS_NAME]] =
        sprintf(self::CN_FORMAT,
            $this->firstname, $this->surname);
      $mods[self::FD_DISPLAY_NAME[
          LdapEntity::FIELD_DEF_POS_NAME]] =
        sprintf(self::DISPLAY_NAME_FORMAT,
            $this->firstname, $this->surname);
    }
  }
}
class Referee extends PolyglotEntity {
  const CONNECTING_ATTRS = array("dn");
    // Field mappings rename empty as all attribute names
  // share the name.
  const FIELD_MAPPING = array(
    SqlRefereePart::class => array(),
    LdapRefereePart::class => array());
    public $user_id;
    public $username;
    public $password;
    public $surname;
    public $firstname;
    public $street;
    public $zip;
    public $city;
    public $phone_private;
    public $phone_business;
    public $phone_mobile;
    public $email;
    public $fax;
    public $club;
    public $verified;
    public $remark;
```

```
    public $ref_since;
    public $license;
    public $birthdate;
    public $homedistrict;
    public $belt;
    public $passno;
    public $ical_uuid;
    public $rating;
    public $last_login;
    public $modified;
    public $modifier;
    public static function withId(int $userId): self {
    $instance = new self();
    $instance->user_id = $userId;
    return $instance;
  }
  public static function withUsername(
      string $username): self {
    $instance = new self();
    $instance->username = $username;
    return $instance;
  }
  public static function withUuid(string $uuid): self {
    $instance = new self();
    $instance->ical_uuid = $uuid;
    return $instance;
  }
}
```