УДК 004.4

Пахолюк Д. А.

АНАЛІЗ ВЕРТИКАЛЬНОЇ ТА ГОРИЗОНТАЛЬНОЇ МОДЕЛЕЙ ІЄРАРХІЇ УСПАДКУВАННЯ КЛАСІВ

Вінницький національний технічний університет

Анотація

У статті розглянуто переваги, недоліки та сфери застосування моделей ієрархій успадкування у розробці програмного забезпечення.

Ключові слова: семантика, успадкування, ООП, проектування, розробка.

Abstract

The article considers advantages, disadvantages and areas of application of models of inheritance hierarchies in software development.

Keywords: semantics, inheritance, OOP, design, development.

Вступ

Сучасний процес розробки програмного забезпечення є складним. Застосунки містять в собі безліч функцій, можливостей, алгоритмів маніпуляції даними, способів взаємодії з користувачем і тому подібного. І це все — тисячі й тисячі рядків коду, який треба не тільки написати так, щоб він працював, а ще й треба вміти його підтримувати. Зручний, інтуїтивно зрозумілий код робить підтримку і розширення застосунку набагато приємнішим ,зручнішим і легшим. А зручність і простота роботи програміста — це його час, його сили, а також час і прибуток замовника.

Парадигма об’єктно-орієнтованого програмування є такою популярною неспроста. Її безмежні можливості організації коду дозволяють створювати чистіший та більш розширюваний код ніж якби ми керувались лише процедурною парадигмою. успадкування є одним з найважливіших та найстаріших концепцій ООП, але неграмотне його використання лише призведе до зв’язаності та повторюваності коду, численних перевантажень членів класів та незрозумілості коду. Для уникнення таких неприємностей у цій тезі буде розглянуто принципи побудови моделі ієрархії та роль семантики у цьому процесі.

Загальний принцип застосування успадкування

Так, успадкування не є панацеєю, і тим більше ніколи не може забезпечити чистоту коду самостійно. Дуже часто воно комбінується з абстракцією, композицією та інкапсуляцією. Більш досвідчені програмісти часто цитують вираз “Надавайте перевагу композиції, а не успадкуванню”[1]. Але і застосунків, що можуть повністю обійтись без успадкування, також не існує.

Як відомо, дочірній клас завжди отримує всі члени батьківського класу. Це означає що дочірній клас отримає всю функціональність та властивості батьківського класу. І якщо нам коли-небудь прийдеться перевизначати чи приглушувати успадковані члени, то відповідальність запам’ятовування таких змін покладається саме на програміста. Середовище програмування ніяк не виділяє потреби в таких змінах, тому код при частому перевизначенні членів стає важкозрозумілим і важкозмінюваним.

З цього випливає, що частого перевизначення варто уникати. Класи мають бути відкриті для розширення, але модифікувати їх вкрай небажано. Цей принцип відповідає літері “O” в абревіатурі “SOLID” і називається “Принцип відкритості/закритості”[2]. Зазвичай для забезпечення цього класи успадковують таким чином, щоб дочірній клас відповідав більш вузькому поняттю ніж батьківський клас. Таким чином дочірні класи розширюють функціонал батьківських класів та звужують свою сферу застосування. Найсильніше такий вплив семантики відчувається на класах, що репрезентують реальні сутності, тобто містять в собі їх властивості та алгоритми обробки даних, пов’язаних з цією сутністю. Такі класи часто називають моделями даних, або просто моделями.

Правильна семантична організація допоможе програмістам краще зрозуміти структуру додатку і зробить їх працю більш ефективною.

Особливості вертикального розширення ієрархії успадкування

Як вже було сказано, при успадкуванні класи розширюють функціонал батьківських класів. При цьому вони звужують свою сферу застосування, тобто отримують певну специфічну роль. Але чимало кінцевих компонентів матимуть спільний функціонал. Повторення коду в кінцевих класах є вкрай поганою практикою, тому краще перенести спільний функціонал у клас-предок. Якщо принцип дії тієї чи іншої функції є чітко визначеним і не міняється у різних класах-нащадках, то краще його так і реалізовувати в класі-предку. Але якщо її реалізація може відрізнятись в нащадках (тобто якщо ви хочете досягнути поліморфізму), то такий метод варто зробити абстрактним. Оголошувати цей метод в абстрактному класі має сенс тоді коли він мусить імплементуватись у всіх класах нащадків і лише в них. Якщо ж така умова не дотримується, то краще визначити його в інтерфейсі.

Створення такої розмежованості реалізації функціоналу дозволяє нам сконструювати таку ієрархію успадкування, яку дуже просто розширяти і рефакторити. Як приклад вдалого застосування такого прийому можна навести структуру фреймворку Yii 2 на PHP[3]. Тут є базовий клас BaseObject, від якого розширюються всі інші класи. У ньому інкапсульована загальна логіка маніпулювання об’єктами: розпізнавання аксесорів і мутаторів та перевірка на доступність членів класу. Клас Component прямо розширює BaseObject та інкапсулює в собі можливості повноцінного компоненту фреймворку (“події” та “поведінки”. Вони не реалізовані в самому ядрі PHP і є особливістю Yii 2). Клас Model, як ви вже здогадуєтесь, інкапсулює можливості моделі даних. Варто звернути увагу на те, що такі класи складають кістяк нашого застосунку, але вони не мають нічого спільного з конкретною бізнес-логікою. Вони існують для того, щоб забезпечувати нащадки загальним функціоналом, але сама реалізація алгоритму обробки даних має бути саме в дочірньому класі. Таке рішення запобігає перевизначенню методів та робить структуру додатку більш розширюваною та переносимою. Спрощена UML-діаграма цієї кончтрукції зображена на рисунку 1.

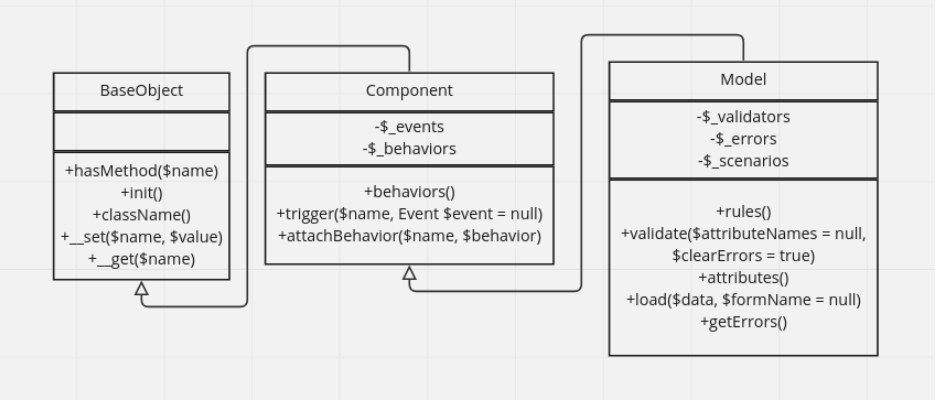

Рисунок 1 – Діаграма фрагменту кістяку Yii2

Не зважаючи на те, що в цьому прикладі класи не є абстрактними, функціонально вони виконують роль абстракцій. Така невідповідність зумовлена тим, що багато членів класу, що іноді перевизначаються в класах-нащадках, дуже часто цього перевизначення не потребують. Наприклад, метод Model::rules() використовується для визначення правил валідації, тому різні конкретні сутності мають мати власну імплементацію методу rules(), але відсутність конкретної імплементації не є смертельною. А якщо валідація атрибутів моделі вам взагалі не потрібна, то цей метод перевизначати нема сенсу.

Роль горизонтального розширення ієрархії успадкування у досягненні поліморфізму

Для ефективної підтримки коду необхідно щоб логіку компонентів можна було легко міняти. Модульність коду і поліморфізм є поширеними прийомами програмування. Модульність забезпечує

розподілення і самостійність роботи компонентів, а поліморфізм дозволяє легко замінювати модулі без шкоди для роботи коду. Для поліморфізму необхідно щоб взаємозамінні класи мали один тип даних і різні методи реалізації функціоналу. Найчастіше для цього використовують абстрактні класи або інтерфейси, в залежності від знаходження взаємозамінних класів на ієрархічному дереві.

Як приклад вдалого використання абстрактного класу для досягнення поліморфізму можна навести іншу частину фреймворку Yii 2. А саме різноманітність класів ActiveRecord. У фреймворк за замовчуванням вбудований абстрактний клас BaseActiveRecord (розширює Model). Функція нащадків цього класу — маніпулювання базами даних. Так, вбудований клас yii\db\ActiveRecord дає можливість легко взаємодіяти з релятивними базами даних. Дочірні класи ActiveRecord відповідають таблицям в базі даних, їх об'єкти — записам в БД, атрибути об'єктів — значення відповідних полів таблиці[4]. Але реляційна модель бази даних не завжди є найкращим варіантом, тому для фреймворку існують розширення, що дають можливість з такою ж легкістю взаємодіяти з іншими моделями баз даних. Ієрархія класів ActiveRecord продемонстрована на рисунку 2.

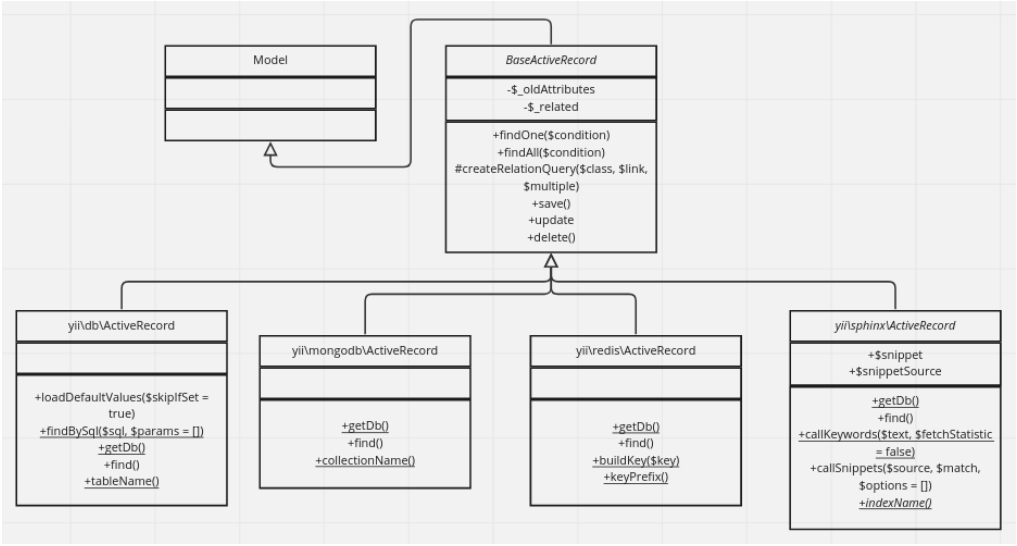

Рисунок 2 – Діаграма, що демонструє поліморфізм класів ActiveRecord

Оскільки решта фреймворку цілком може задовольнити свої потреби у передачі даних класом Model, то ми можемо використовувати будь-які класи ActiveRecord і відповідні їм бази даних. При цьому робота інших компонентів залишиться незмінною, а наш додаток не буде залежати від бази даних.

Висновки

Таким чином було розглянуто найпоширеніші прийоми організації ієрархії успадкування. Також продемонстровано важливість комбінації успадкування з іншими прийомами ООП для створення надійної, зрозумілої та легкорозширюваної архітектури додатків.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Composition over inheritance: веб-сайт URL: https://en.wikipedia.org/wiki/Composition_over_inheritance
2. Open/closed principle: веб-сайт URL: https://en.wikipedia.org/wiki/Open%E2%80%93closed_principle
3. Yii PHP framework: веб-сайт URL: https://www.yiiframework.com/
4. Active record pattern : веб-сайт URL: https://en.wikipedia.org/wiki/Active_record_pattern

Пахолюк Дмитро Анатолійович – студент групи 6ПІ-22б, Факультет інформаційних технологій та комп'ютерної інженерії, Вінницький національний технічний університет, Вінниця, e-mail: dmytro0pakholiuk@gmail.com

Dmytro Pakholiuk - student of 6PI-22b group, Faculty of Information Technologies and Computer Engineering, Vinnytsia National Technical University, Vinnytsia, e-mail: dmytro0pakholiuk@gmail.com