УДК 004.451.7:004.42

Корнілов Іван Станіславович
асистент кафедри комп'ютерних наук,
Національний університет біоресурсів і природокористування України
ORCID: https://orcid.org/0009-0009-5598-2690
E-mail: i.kornilov@nubip.edu.ua

Вайганг Ганна Олександрівна
кандидат технічних наук, доцент кафедри комп'ютерних наук,
Національний університет біоресурсів і природокористування України
ORCID: https://orcid.org/0000-0002-2082-2322
E-mail: weigang.ganna@nubip.edu.ua

SOLID ЯК СИСТЕМА КОНСТРУКТИВНИХ ОБМЕЖЕНЬ У ПРОЄКТУВАННІ АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Анотація. *Стаття подає SOLID як систему конструктивних обмежень, що дисциплінує ступені свободи дизайну й переводить еволюцію програмних систем у керований процес. Пояснюється зв'язок принципів із фундаментом ООП, із шаблонами проектування та з архітектурними стилями Розкриваються неочевидні ефекти: передчасні абстракції при OCP, прихована конфігураційна зв'язність при DIP/DI, «class explosion» і фрагментація відповідальностей при SRP/ISP, семантичні порушення LSP, що не фіксуються сигнатурами. Запропоновано операційний підхід до валідації рішень через метрики, контрактні тести і контрольні пороги введення абстракцій, а також наведено практичні протоколи прийняття рішень.*

Ключові слова: *SOLID, об'єктно-орієнтоване програмування, архітектура програмного забезпечення, дизайн-патерни, когезія, зв'язність, метрики якості коду, інверсія залежностей (DI/IoC).*

Вступ. У промисловій розробці головною проблемою стає не початкове створення функціоналу, а довготривала здатність системи змінюватися без множинних регресій і неконтрольованого зростання складності. Програмні системи з часом деградують через надмірну зв'язність і слабку когезію. Це зумовлює дефектність, високу вартість змін і «крихкість» релізів. Емпіричні дослідження зв'язують класичні метрики (CBO, LCOM, тощо) із дефектопридатністю та супроводжуваністю програмного коду [1, 2, 4, 16].

Просте декларування п'яти принципів SOLID не гарантує бажаного ефекту: результат залежить від того, як саме принципи інтегруються з базовими механізмами ООП, якими патернами реалізуються точки розширень системи, і чи підтримується все це вимірюваними критеріями якості. По суті SOLID – це угода про те, що в системі слід фіксувати стабільні інваріанти, локалізувати варіативність та уникати зайвої зв'язаності між модулем, його оточенням і каналами доставки. SRP і ISP керують масштабом відповідальності, OCP визначає, де допустимо розширювати без модифікації ядра, LSP забезпечує безпечний поліморфізм, а DIP переносить залежності на рівень абстракцій. У поєднанні з шаблонами та архітектурними стилями це створює «каркас еволюційності», у межах якого нові вимоги реалізуються через додавання, а не переписування [11, 13].

Ключовим є перехід від гасел до операційності: спостерігати за змінами, виділяти інваріанти, описувати контракти, вводити абстракції лише за наявності реальних альтернатив, а рішення валідувати метриками та тестами. Саме так SOLID стає не стилістикою, а інженерною процедурою, яку можна перевіряти, порівнювати, відкотити чи посилити [17, 18].

Огляд літератури. Набір інженерних підходів, подібних за духом до SOLID, давно використовується й поза IT. Ці підходи дисциплінують проектні рішення та роблять системи придатними до еволюції. Ідея проста: розбивати складне на модулі з чіткими межами, мінімізувати небажані залежності, фіксувати стабільні інтерфейси між компонентами та відокремлювати те, що часто змінюється, від того, що має лишатися стабільним. Такі

DOI:10.31548/itees.2025.01.020
ISSN 3083-7502

обмеження зменшують ризики, скорочують цикл змін і дозволяють масштабувати продуктову лінійку без зростання складності.

В автомобілебудуванні це проявляється у платформенній інженерії та уніфікації вузлів: різні моделі збираються на спільних платформах, використовуючи стандартизовані підрамники, вузли підвіски, електропакети, блоки керування. Стабільні механічні та електричні інтерфейси дають змогу випускати «ревізії» із мінімальною переробкою несучих елементів, а зміни концентрувати в периферійних модулях – в обшивці, освітленні, мультимедійних системах, калібруваннях ПЗ. Перевикористання компонентів знижує NRE-витрати, полегшує логістику та спрощує післяпродажне обслуговування завдяки взаємозамінності і сумісності ревізій [21, 22]. У будівництві аналогічну роль відіграють модульні системи та підхід DfMA: заводська збірка блоків з уніфікованими інтерфейсами скорочує помилки при монтажі, спрощує сертифікацію та дозволяє керувати життєвим циклом будівлі як набором взаємопов'язаних, але незалежно керованих підмоделей [23].

Економічний ефект у всіх галузях однаковий: модульність і стандартизовані інтерфейси знижують ризики, прискорюють ревізії і здешевлюють обслуговування завдяки взаємозамінності і сумісності ревізій. Іншими словами, зменшується вартість адаптації й експлуатації, незалежно від того, йдеться про програмні системи, автомобільні платформи чи будівельні об'єкти [21, 22, 23]. У такому контексті SOLID доречно розглядати як систему конструктивних обмежень: локальні правила, які стримують необачні проектні рішення, забезпечуючи глобальні властивості – модульність, передбачуваність змін і керовану еволюцію. Багаторічні дослідження підтверджують зв'язок когезії та зв'язності із дефектопридатністю та супроводжуваністю програмного коду. Класичний набір ОО-метрик (CBO, LCOM, RFC, WMC) валідовано як індикатори якості, а альтернативні підходи, такі як динамічне або концептуальне зв'язування, доповнюють статичний аналіз для виявлення ризиків [1,2,4,5,6,7,10].

Систематичні огляди щодо шаблонів проєктування показують неоднорідний вплив: у контекстах, де патерни справді інкапсулюють варіативність і залишають інтерфейси прозорими, поліпшуються показники підтримуваності [11, 13]. Для DIP/DI окреслено як позитивні ефекти такі, як зменшення жорсткої зв'язаності, а також можливість легкого тестування компонентів системи, так і ризики конфігураційної зв'язності через контейнери, коли граф залежностей стає непрозорим. Для LSP пропонуються формальні специфікації передумов, післяумов та інваріантів, які роблять підстановність перевірюваною у ієрархіях класів [17, 18].

Типи зв'язності та когезія: що саме контролює SOLID

SOLID адресує різні прояви зв'язності. Структурна зв'язність відображається у статичному графі залежностей і вимірюється метриками на кшталт CBO, RFC, WMC; когезія класів вимірюється LCOM та його поліпшеними варіантами [1,2,16,4]. Концептуальна зв'язність аналізує близькість тематики за текстовими артефактами та ідентифікує приховані тематичні кластери [6,7,10]. Логічна або еволюційна зв'язність (рис. 1) спирається на історію змін і спільні коміти, виявляючи компоненти, які еволюціонують і потребують спільної уваги [8,9,10].

На практиці поєднання статичних, динамічних та семантичних підходів дає найкращий ефект: статичні метрики окреслюють «гарячі зони», динамічні – виявляють залежності під час виконання (навігація подіями, виклики), семантичні – підсвічують латентні зв'язки за даними репозиторію, трекера задач і текстових артефактів [3,5,6,8,9,10,20]. Динамічне зчеплення особливо корисне у подієвих, плагінних та мікросервісних архітектурах [5,19].

Мапування принципів SOLID на проблеми зв'язності

SRP (принцип єдиної відповідальності) – відповідає за підвищення когезії та зменшення логічних «швів». SRP зменшує тематичну «розмитість» класів, підвищуючи концептуальну когезію (на кшталт C3/CCC). Емпірично когезія класів корелює з якістю та дефектністю [6]. Якщо класу не можливо дати коротке призначення класу, то цей клас потрібно декомпозувати.

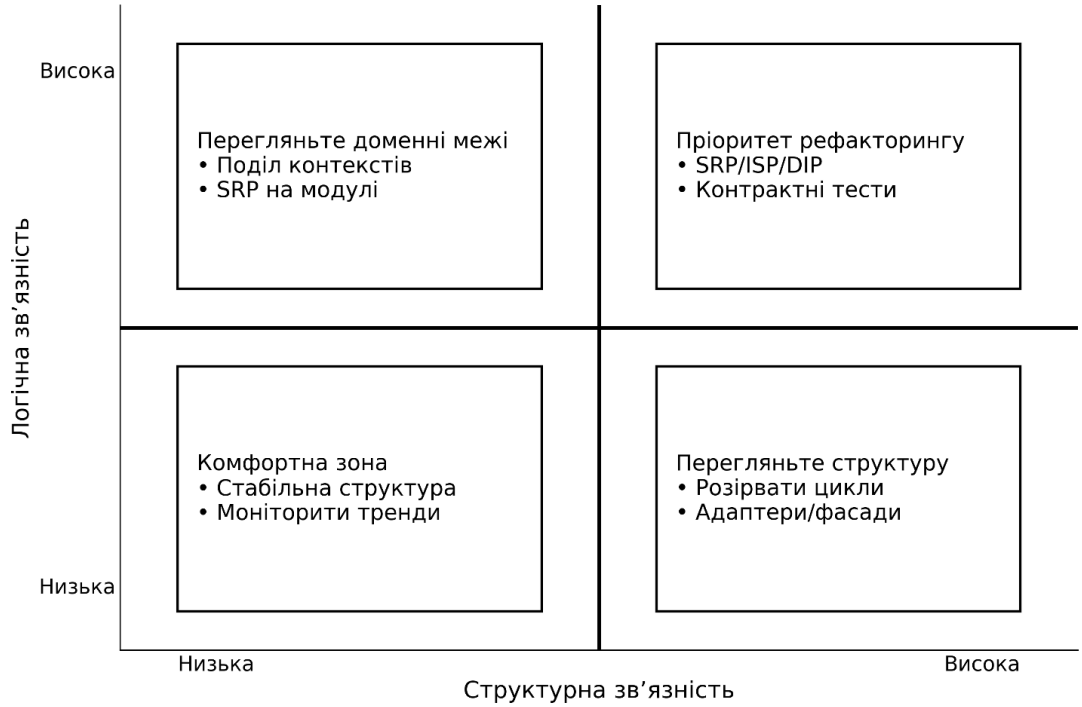

Рисунок 1 – Матриця зв'язностей: структурна та логічна

OCP (принцип відкритості до розширення) – відповідає за визначення контрольованих місць зміни коду, щоб при можливій зміні вимог не потребувалася модифікація основних (концептуальних) частин системи. Масштаб застосування принципу повинен збільшуватися еволюційно: не потрібно намагатися «вгадати» майбутнє передчасними абстракціями, але при цьому потрібно враховувати можливі ризики зміни функціональних вимог. Відповідно в тих місцях коду, де ризик концептуальних змін логіки низький, але високий ризик зміни реалізації контрактів, потрібно визначити точки розширення. Дослідження еволюції архітектурних запахів (циклічні залежності, God Object, Hub-like Dependency) зв'язують їх із зростанням змін коду і зусиль супроводу [14].

LSP (принцип підстановки Liskov) – відповідає за поведінкові контракти в ієрархії наслідування класів. LSP – формалізація поведінкової сумісності через поведінкове підтипування. Порушення принципу призводять до прихованих дефектів, які важко виявити статично, але які сильно впливають на очікувані результати виконання системи. Концепція походить із робіт з поведінкового підтипування та проектування за контрактами [17], [18]. Для унеможливлення порушення принципу мають бути перевірки інваріантів, передумов та постумови у тестах, а також контрактні перевірки на рівні інтерфейсів. Важливо враховувати що принцип відповідає лише за очікувану поведінку, яка продиктована контрактом (сигнатурою), інваріантами, передумовами, постумовами в реалізації класів, а також документаціїєю. Будь-яке порушення очікуваної поведінки призведе до непередбачуваних наслідків роботи системи, включаючи можливі проблеми безпеки.

ISP (принцип розділення інтерфейсів) – відповідає за обмеження області використання компоненту з точки зору викликаючої сторони коду. ISP знижує «нав'язування» зайвих залежностей клієнтам, що зменшує логічне зчеплення між несуміжними змінами. В даному випадку клієнт (викликаючий код) визначає, які логіку виокремити з інтерфейс. Принцип ISP дещо перекликається з принципом SRP, але не стосується ніяким чином самої реалізації класів. Принцип ISP керується лише контрактами і потребами клієнтського коду.

DIP (принцип інверсії залежностей) – фундаментальний код має максимально залежати від абстракцій, в свою чергу абстрації не повинні залежати від деталей реалізації. DIP спрямовує залежності на абстракції, зменшуючи структурну зв'язність модулів і полегшуючи тестування. Водночас надмірне використання DI породжує конфігураційні залежності.

DOI:10.31548/itees.2025.01.020
ISSN 3083-7502

Впровадження DI має супроводжуватися архітектурною дисципліною й автоматизованими перевірками [17].

Гексагональна архітектура

Гексагональна архітектура (порти та адаптери) розділяє систему на «ядро» з бізнес-логікою та «зовнішній світ», пов'язуючи їх через порти – явні точки взаємодії, які реалізуються змінними адаптерами (рис. 2). Таке компонентно-конекторне подання робить інтерфейси першокласними елементами моделі, спрямовує залежності до абстракцій (узгоджується з DIP/ISP), полегшує підміну технологій і забезпечує тестування ядра в ізоляції від БД/мережі/GUI. Формальне трактування «портів» як інтерфейсних точок компонентів і способів їх моделювання розвинуто в літературі з архітектурного моделювання, що на практиці прямо підтримує ідеї гексагональної схеми про стабільні межі та змінні адаптери.

В індустріальних мікросервісах ці принципи проявляються через чіткі межі API, інверсію залежностей на стиках сервісів і контрактне тестування адаптерів – підходи, які емпірично пов'язують із кращою керованістю змін, спостережуваністю та підтримуваністю систем. Таким чином, «порти й адаптери» слугують інженерними обмеженнями, що дозволяють локалізувати варіативність і безболісно еволюціонувати систему, не торкаючись бізнес-ядра.

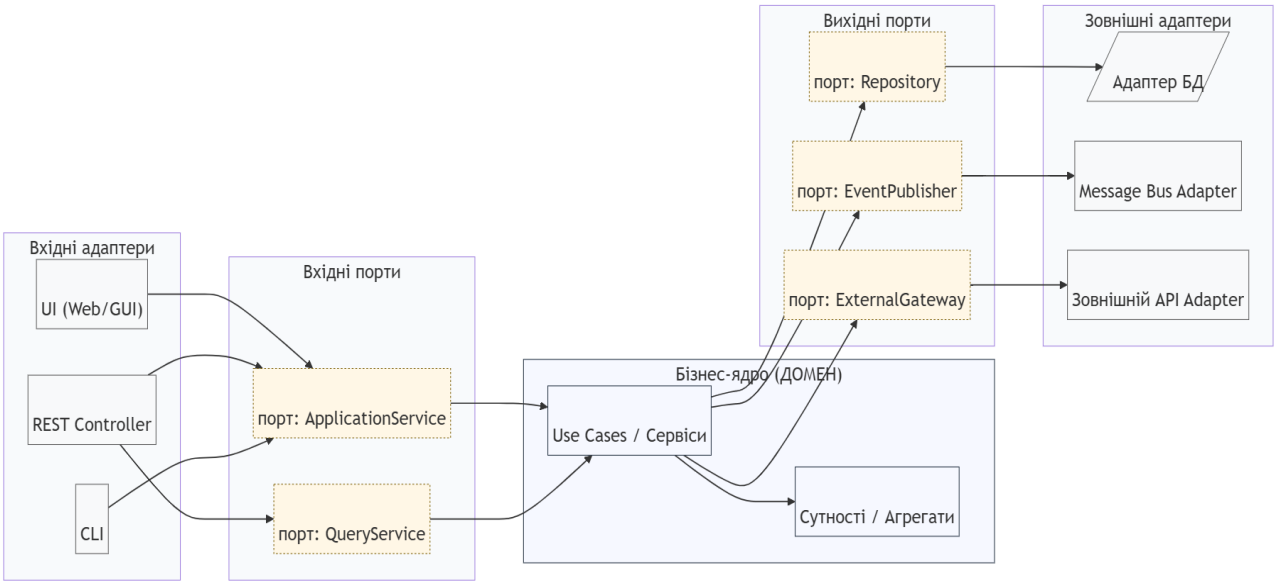

Рисунок 2 – Архітектура портів та адаптерів

Кількісні індикатори якості ОО-дизайну: CBO, LCOM, RFC, WMC.

CBO (Coupling Between Objects) – це скільки інших класів «чіпляє» один клас. Чим більше зовнішніх залежностей, тим важче щось змінювати: будь-яка правка в сусідньому модулі може зламати ваш. Орієнтир простий: менше зв'язків – менше сюрпризів. Якщо CBO завеликий, винесіть інтеграції за інтерфейси/адаптери, приберіть зайві «короткі шляхи» до чужих модулів, розірвіть циклічні залежності.

LCOM (Lack of Cohesion in Methods) показує, чи робить клас одну справу, чи має декілька відповідальностей. Якщо методи працюють з різними полями і майже не перетинаються – когезія низька (LCOM висока). Це ознака, що клас змішував кілька ролей. Рішення: розділити на менші класи або сервіси з чіткими завданнями, а спільні речі залишити в окремих допоміжних об'єктах.

RFC (Response For a Class) – скільки різних методів може виконатися, коли викликається один публічний метод класу. Високий RFC означає багато можливих шляхів виконання, складніше зрозуміти поведінку і покрити тестами. Щоб зменшити RFC, потрібно спростити логіку, прибрати зайві виклики до зовнішніх сервісів із «тонких» місць, перенести складні сценарії у спеціалізовані компоненти.

WMC (Weighted Methods per Class) – «вага» класу за складністю його методів. Якщо існує кілька методів із великою кількістю умов/гілок – WMC росте, код важче читати й тестувати. Рішення: ділити довгі методи на менші, виносити варіативну поведінку у стратегії/політики, прибирати дублювання.

Мета статті полягає у перетворенні принципів SOLID із декларативного набору правил у відтворювану інженерну процедуру, що забезпечує керовану еволюцію програмних систем шляхом кількісного опису впливу SOLID на когезію, зв'язність і стабільність архітектури, а також перевірки його застосування за допомогою метрик, контрактних тестів і архітектурних порогів. Це передбачає інтеграцію п'яти принципів (SRP, OCP, LSP, ISP, DIP) із базовими механізмами об'єктно-орієнтованого програмування, шаблонами проєктування та архітектурними стилями типу «Порти й адаптери», що дозволяє перейти від декларативного застосування SOLID до вимірюваного контролю архітектурної складності через побудову профілів метрик (CBO, LCOM, RFC, WMC), опис поведінкових контрактів та встановлення кількісних порогів абстракцій, забезпечуючи відтворюваність рішень, уникнення передчасних узагальнень і баланс між стабільністю та варіативністю системи.

Для досягнення поставленої мети в роботі вирішуються такі науково-прикладні завдання:

1. Проаналізувати фундаментальні принципи SOLID у контексті їх взаємозв'язку з об'єктно-орієнтованим програмуванням, архітектурними патернами та метриками якості програмного дизайну.
2. Розробити методологію кількісної оцінки впливу принципів SOLID на архітектурну складність із використанням метрик CBO, LCOM, RFC та WMC.
3. Побудувати операційний цикл застосування SOLID, що охоплює етапи спостереження, формування інваріантів, вибору патернів, формалізації контрактів, інверсії залежностей і корекції результатів.
4. Обґрунтувати ефективність гексагональної архітектури (Ports & Adapters) як практичної реалізації принципів DIP, ISP та SRP для підвищення модульності, керованості й відтворюваності програмних систем.

Очікуваним результатом реалізації зазначених завдань є формування відтворюваної методики застосування принципів SOLID як системи конструктивних обмежень, що забезпечує кількісно контрольовану еволюцію програмних архітектур, підвищує їхню стабільність, модульність і якість супроводження.

Методологічне обґрунтування. Перший крок – спостереження: збір профілю метрик (CBO, LCOM, RFC, WMC) для цільових модулів, побудова карт семантичної близькості та матриць змін коду за релізами [1,2,4,16]. Графіки, таблиці й діаграми на цьому етапі фіксують «гарячі зони», де зміни концентруються, і групи файлів, що еволюціонують разом.

Другий крок – формування інваріантів: визначення стабільних властивостей домену й зовнішніх контрактів, що не мають залежати від інфраструктури. Інваріанти описуються короткими специфікаціями й підкріплюються тестами, які в подальшому виконують роль «сигналізаторів» при рефакторингу.

Третій крок – вибір носія варіативності (патернів проектування): для поведінкових альтернатив – Стратегія; для фабрикації залежно від контексту – Фабрика/Абстрактна Фабрика; для ізоляції різних реалізацій інтерфейсів і стабілізації API – Міст і Адаптер; для поступового нарощування поведінки – Декоратор; для спрощення взаємодії з підсистемами – Фасад; для конструювання складних об'єктів – Білдер; для уніфікації фіксованої послідовності кроків із варіативними деталями – Шаблонний метод. На архітектурному рівні варіативність відмежовується через Порти та Адаптери та модульну архітектуру.

Четвертий крок – контракти: докладний опис передумов, післяумов та інваріантів для базових типів і їх підтипів. Для забезпечення LSP усі підтипи проходять однакові «контрактні тести», а нестабільні ієрархії обмежуються запаковуванням класу на рівні визначення або замінюються композицією.

DOI:10.31548/itees.2025.01.020
ISSN 3083-7502

П’ятий крок – інверсія залежностей із контролем прозорості: зв’язки встановлюються через абстракції у composition-root, забороняються приховані сервіс-локатори, граф контейнера перевіряється автоматично на цикли, дублікати і невпроваджені сервіси. Для кількісної оцінки пропонується узагальнений показник зв’язності (1), що враховує ін’єктовані залежності та розмір графа конфігурації [3, 5, 19, 20]:

$$DCBO = CBOstruct + \alpha \times CBOinjected + \beta \times |Econtainer|, \quad (1)$$

де α, β вагові коефіцієнти, які обираються емпірично за історією змін;

CBOstruct – структурне зчеплення класу з іншими типами, що видно зі статичного коду (виклики методів, поля, параметри/результати методів, наслідування/агрегація, generic-аргументи тощо);

CBOinjected – кількість *ін’єктованих* залежностей (через конструктор / властивість / метод), тобто портів, які підмінюються DI. Це «м’якші» зв’язки, тому ми зважуємо їх коефіцієнтом α;

|*Econtainer*| – кількість *конфігураційних ребер* DI-контейнера, що стосуються цього класу (зв’язки «порт-адаптер», фабричні реєстрації, тощо). Це «інфраструктурне» зчеплення, яке додає непрямі залежності, тому його зважуємо β.

Шостий крок – вимірювання та корекція: порівняння «до/після» за метриками, аналіз локальності змін при додаванні альтернатив, ревізія точок варіації [5,8,9,10]. Для OCP запроваджується поріг: абстракція вводиться лише за наявності принаймні двох реальних альтернатив і очікуваного приросту варіативності; для LSP – заборона нових перевизначень без контрактних тестів; для ISP – поділ інтерфейсів погоджується з реальними ролями клієнтів і спостереженнями змін коду; для DIP – обов’язкова валідація графа залежностей і звітність про DCBO.

Додатково рекомендується процесна дисципліна: ADR-записи для кожної точки варіації, семантичне версіонування адаптерів і плагінів, контроль глибин ієрархій, регулярна візуалізація показників у таблицях і графіках.

Результати та обговорення. Результати проведеного дослідження демонструють практичну значущість системного підходу до застосування принципів SOLID у поєднанні з метриками якості програмного коду. Узагальнення емпіричних спостережень і кількісних показників підтвердило, що збалансоване використання принципів дає змогу зменшити структурну зв’язність, підвищити когезію та стабільність архітектури без надмірного ускладнення її компонентної структури. Виявлені залежності між показниками метрик і типами архітектурних порушень свідчать про те, що SOLID виконує роль регулятора еволюційної динаміки системи, а його операційне застосування може бути формалізоване через об’єктивно вимірювані параметри. Практика показує, що SRP і ISP дієві тоді, коли рольова модель інтерфейсів і «осьова» відповідальність класів збігаються з реальними сценаріями використання. Некоректний масштаб призводить до «class explosion» або до «God object». Антипатерни SOLID узагальнено у табл. 1.

Таблиця 1 – Антипатерни SOLID

Антипатерн	Порушений принцип	Симптоми (метрики/ознаки)	Рефакторинги
God Class	SRP/ISP	Високі LCOM, fan-in/out	Виділення підмодулів, фасадів; SRP-декомпозиція
Cyclic Dependency	OCP/DIP	Цикли в графі, високе CBO	Розірвання циклів, введення портів/адаптерів
Feature Envy	SRP	Методи працюють з «чужими» даними	Переміщення методів, редизайн доменних меж
Hub-like Dependency	DIP	Один вузол із надмірним зв’язками	Розподіл відповідальностей, використання брокерів подій

Закономірність еволюції архітектури з ростом кількості релізів (рис. 3) така, що чим більше релізів, тим складніше змінювати код системи, якщо присутні типові архітектурні «запахи». Керувати балансом допомагає аналіз змін коду: методи й модулі, що часто змінюються разом, зазвичай належать до однієї відповідальності й мають бути зближені, а ті, що не перетинаються за сценаріями, варто розвести в різні контракти [11,12,18].

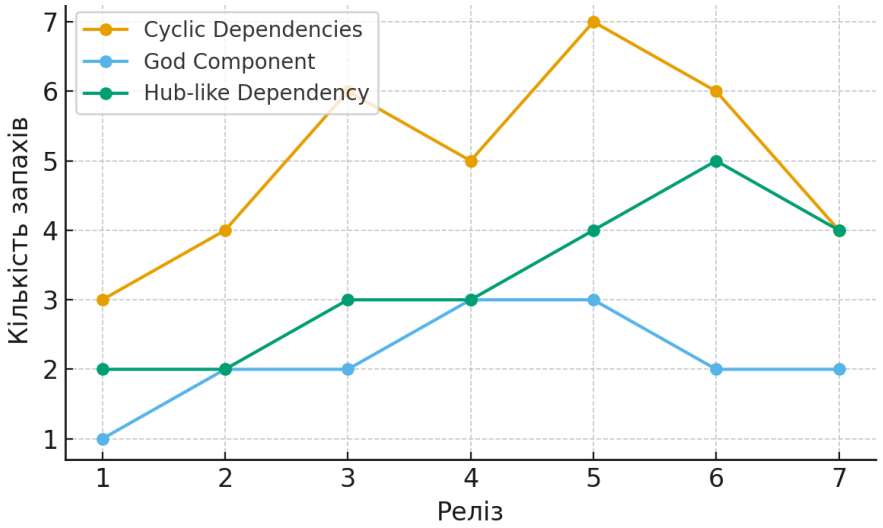

Рисунок 3 – Еволюція архітектурних «запахів» за релізами

OCP приносить вигоду, коли потік альтернатив достатній, щоб компенсувати вартість обслуговування абстракцій. Успішні кейси характеризуються стабільними інваріантами ядра, невеликими й чіткими інтерфейсами, прозорими точками розширення [11,13,12,14].

LSP часто порушується не на рівні сигнатур, а на рівні семантики: змінюються класи винятків, дозволені діапазони значень, політика обробки null, інваріанти стану. Такі зміни непомітні для компілятора, але руйнують підстановність класів-наслідників. Контрактні тести та обмеження успадкування там, де природних підтипів не передбачено, різко знижують ризик регресій [17,18].

DIP у поєднанні з DI зменшує жорстку зв'язаність і покращує замінність компонентів у тестах, проте створює ризики конфігураційної зв'язності: графи контейнера ускладняються, життєві цикли стають неочевидними, а трасування – дорогим. Протидією є composition-root, автоматична валідація графа, явні модулі, ліміти на «магічні» прив'язки та регулярний контроль DCBO. Результати узагальнено у табл. 2.

Таблиця 2 – Принципи SOLID, метрики та тактики рефакторингу

Принцип SOLID	Тип зв'язності / когезії	Релевантні метрики	Тактики рефакторингу
SRP	Концептуальна когезія	C3/CCC, LCOM	Декомпозиція класів, перейменування, виокремлення сервісів
OCP	Керовані точки варіації	Fan-in/out, цикли	Використання стратегій, модульної архітектури, ADR для варіацій
LSP	Поведінкова сумісність	Покриття контрактними тестами	Проектування за контрактами, property-based тест
ISP	Аферентна зв'язність	Fan-in, інтерфейси/ клієнти	Клієнтоорієнтовані інтерфейси, використання фасадів
DIP	Міжмодульна залежність	Граф залежностей, CBO	Порти/адаптери; використання ін'єкції залежностей

DOI:10.31548/itees.2025.01.020
ISSN 3083-7502

На архітектурному рівні порти та адаптери, а також плагінна архітектура масштабують принципи: ядро утримує інваріанти домену, а адаптери інкапсулюють інфраструктуру й канали взаємодії. Завдяки цьому зміни у спосіб доставки (GUI/API/CLI/черги) не вимагають модифікації бізнес-логіки. Вартість такого підходу – чітка контрактна дисципліна, тестування контрактів, каталогізація портів і контроль версій інтерфейсів.

Практичний протокол ухвалення рішень зводиться до простих правил. Спочатку – спостерігати гарячі точки й кластери змін коду, а потім – вводити абстракції у місцях реальної варіативності. Кожен підтип має бути з явно зафіксованим контрактом, кожна ін'єкція залежності – видима у composition-root, кожен інтерфейс – під реального клієнта. Графіки та таблиці стають інструментами комунікації: вони доводять або спростовують гіпотези про користь конкретних рішень, а діаграми демонструють рамки відповідальності та межі компонентів.

Висновки. SOLID працює як інженерна мова компромісів між стабільністю та варіативністю програмних систем, поєднуючи концептуальні принципи ООП із вимірюваними показниками якості архітектури. Його практичне значення полягає у створенні дисципліни проектування, що дозволяє підтримувати сталість логічних інваріантів системи, локалізувати варіативність і зменшувати ризики деградації структури коду під час еволюції продукту. У результаті дослідження визначено, що ефективність SOLID підвищується за умов використання формалізованих метрик (CBO, LCOM, RFC, WMC), контрактних тестів і архітектурних порогів, які дають змогу виявляти зони підвищеної складності та обґрунтовувати доцільність введення абстракцій.

Дотримання процесних порогів для OCP, поведінкової дисципліни для LSP, контрольованої інверсії залежностей для DIP/DI та рольового поділу для ISP забезпечує керовану еволюцію системи без втрати якості. Виявлені закономірності дозволяють розглядати SOLID як систему конструктивних обмежень, що створює баланс між гнучкістю і передбачуваністю змін. Використання гексагональної архітектури, патернів варіативності та системи метрик формує відтворюваний підхід до підтримки архітектурної цілісності, що має значний потенціал для подальших досліджень у сфері автоматизованого аналізу та валідації програмних рішень.

Список використаних джерел

1. Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6), 476–493. https://doi.org/10.1109/32.295895.
2. Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering, 22(10), 751–761. https://doi.org/10.1109/32.489317.
3. Briand, L. C., Daly, J. W., & Wüst, J. (1999). A unified framework for coupling measurement in object-oriented systems. IEEE Transactions on Software Engineering, 25(1), 91–121. https://doi.org/10.1109/32.748920.
4. Zhou, Y., & Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. IEEE Transactions on Software Engineering, 32(10), 771–789. https://doi.org/10.1109/TSE.2006.102
5. Arisholm, E., Briand, L. C., & Føyen, A. (2004). Dynamic coupling measurement for object-oriented software. IEEE Transactions on Software Engineering, 30(8), 491–506. https://doi.org/10.1109/TSE.2004.41.
6. Marcus, A., Poshyvanyk, D., & Ferenc, R. (2008). Using the conceptual cohesion of classes for fault prediction. IEEE Transactions on Software Engineering, 34(2), 287–300. https://doi.org/10.1109/TSE.2007.70768.
7. Revelle, M., Gethers, M. & Poshyvanyk, D. (2011). Using structural and textual information to capture feature coupling in object-oriented software. Empir Software Eng, 16, 773–811. https://doi.org/10.1007/s10664-011-9159-7.

8. Ajienka, N., & Capiluppi, A. (2017). Understanding the interplay between logical and structural coupling of software classes. Journal of Systems and Software, 134, 120–137. https://doi.org/10.1016/j.jss.2017.08.042.
9. Ajienka, N., Capiluppi, A., & Counsell, S. (2018). An empirical study on the interplay between semantic coupling and co-change. Empirical Software Engineering, 23(4), 1799–1836. https://doi.org/10.1007/s10664-017-9569-2.
10. Kagdi, H., Gethers, M. & Poshyvanyk, D. (2013). Integrating conceptual and logical couplings for change impact analysis in software. Empir Software Eng, 18, 933–969. https://doi.org/10.1007/s10664-012-9233-9.
11. Ampatzoglou, A., Frantzeskou, G., & Stamelos, I. (2012). A methodology to assess the impact of design patterns on software quality. Information and Software Technology, 54(4), 331-346. https://doi.org/10.1016/j.infsof.2011.10.006.
12. Yamashita, A., & Moonen, L. (2013). AiOLoS: A model for assessing organizational learning in software development organizations, 55(11), 1904-1924. https://doi.org/10.1016/j.infsof.2013.05.004.
13. Alfadel, M., Aljasser, K., & Alshayeb M. (2020). Empirical study of the relationship between design patterns and code smells. PLoS ONE, 15(4): e0231731. https://doi.org/10.1371/journal.pone.0231731.
14. Gnoyke, P., Schulze, S., & Krüger, J. (2024). Evolution patterns of software-architecture smells. Journal of Systems and Software, 213, 112170. https://doi.org/10.1016/j.jss.2024.112170.
15. Arisholm, E., Sjøberg, D. I. K., & Jørgensen, M. (2001). Assessing the changeability of two object-oriented design alternatives: A controlled experiment. Empirical Software Engineering, 6(3), 231–277. https://doi.org/10.1023/A:1011439416657.
16. Chae, H. S., Kwon, Y. R., & Bae, D. H. (2004). Improving cohesion metrics for classes. IEEE Transactions on Software Engineering, 30(8), 548–564. https://doi.org/10.1109/TSE.2004.88
17. Liskov, B., & Wing, J. (1994). A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems, 16(6), 1811–1841. https://doi.org/10.1145/197320.197383
18. Meyer, B. (1992). Applying "design by contract." Computer, 25(10), 40–51. https://doi.org/10.1109/2.161279.
19. Waseem, M., Liang, P., Shahin, M., Di Salle, A., & Márquez, G. (2021). Design, monitoring, and testing of microservices systems: The practitioners' perspective. Journal of Systems and Software, 182, 111061. https://doi.org/10.1016/j.jss.2021.111061.
20. Fregnan, E., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., & Lucia, A. D. (2019). On nonlinear Schrödinger equations with attractive inverse-power potentials. Information and Software Technology, 107, 159-178. https://doi.org/10.48550/arXiv.1903.04636.
21. Lampón, J. F., Cabanelas, P., & González-Benito, J. (2017). The impact of modular platforms on automobile manufacturing networks. Production Planning & Control, 28(4), 335–348. https://doi.org/10.1080/09537287.2017.1287442.
22. Pandremenos, J., Paralikas, J., Salonitis, K., & Chryssolouris, G. (2009). Modularity concepts for the automotive industry: A critical review. CIRP Journal of Manufacturing Science and Technology, 1(3), 148–152. https://doi.org/10.1016/j.cirpj.2008.09.012.
23. Bao, Z., Laovisutthichai, V., Tan, T., Wang, Q., & Lu, W. (2022). Design for manufacture and assembly (DfMA) enablers for offsite interior design and construction. Building Research & Information, 50(3), 325–338. https://doi.org/10.1080/09613218.2021.1966734.

Kornilov Ivan
Assistant, Department of Computer Science,
National University of Life and Environmental Sciences of Ukraine
ORCID: https://orcid.org/0009-0009-5598-2690
E-mail: i.kornilov@nubip.edu.ua

DOI:10.31548/itees.2025.01.020
ISSN 3083-7502

Weigang Ganna
Candidate of Engineering Sciences, Associate Professor, Associate Professor of the Department of Computer Science,
National University of Life and Environmental Sciences of Ukraine
ORCID: https://orcid.org/0000-0002-2082-2322
E-mail: weigang.ganna@nubip.edu.ua

SOLID AS A SYSTEM OF CONSTRUCTIVE CONSTRAINTS IN SOFTWARE ARCHITECTURE DESIGN

Abstract. *The article presents SOLID as a system of constructive constraints that disciplines the degrees of freedom in design and transforms the evolution of software systems into a controlled process. It explains the relationship of the principles with the foundations of object-oriented programming, design patterns, and architectural styles. Non-obvious effects are examined, including premature abstractions under OCP, hidden configuration coupling under DIP/DI, class explosion and responsibility fragmentation under SRP/ISP, and semantic violations of LSP that are not captured by type signatures. An operational approach to decision validation is proposed through metrics, contract-based tests, and threshold controls for introducing abstractions, along with practical decision-making protocols.*

Keywords: *SOLID; Object-Oriented Programming; Software Architecture; Design Patterns; Cohesion; Coupling; Code Quality Metrics; Dependency Inversion (DI/IoC).*