## Компонентне програмування

**Component-based software engineering**, або компонентне програмування (далі КП) – це стиль розробки ПЗ, що спрямований на створення програмної системи з компонентів, слабо пов’язаних між собою та призначених для багаторазового використання.

[https://en.wikipedia.org/wiki/Component-based\_software\_engineering]

Такий підхід забезпечує швидші цикли розробки, скорочує час виходу на ринок і підвищує якість програмного забезпечення. КП має кілька переваг. По-перше, воно дає змогу розробляти програмні системи модульно, що полегшує їх управління та супровід. Завдяки модульним компонентам розробники можуть зосереджуватися на окремих функціях, що забезпечує ефективніший процес розробки. Крім того, КП сприяє повторному використанню коду, зменшуючи надмірність і підвищуючи загальну ефективність розробки. Використовуючи перевірені й протестовані компоненти, розробники можуть заощадити час і зусилля на створення програмного забезпечення з нуля. До того ж, КП сприяє кращій співпраці між командами розробки, оскільки вони можуть зосередитися на конкретних компонентах, мінімізуючи конфлікти під час інтеграції коду.

[https://wesoftyou.com/outsourcing/component-based-software-engineering explained/]

## Принципи компонентного програмування

### Відокремлення інтерфейсу від реалізації

Фундаментальний принцип компонентного програмування полягає в тому, що базовою одиницею в застосунку є бінарно-сумісний (binary-compatible) інтерфейс. Інтерфейс забезпечує абстрактне визначення сервісу між клієнтом і об’єктом. Цей принцип протиставляється об'єктно-орієнтованому підходу, в якому в центрі перебуває саме об'єкт, а не його інтерфейс. Інтерфейс забезпечує взаємодію між клієнтом та сервісом. Кожен сервіс реалізує інтерфейс у свій, незалежний спосіб. Інтерфейс реалізується бінарним компонентом, який є «чорною скринькою» і повністю інкапсулює свою внутрішню логіку.

Щоб використати компонент, клієнт має знати лише визначення інтерфейсу та мати доступ до бінарного компонента, який цей інтерфейс реалізує. Такий рівень опосередкування між клієнтом і об’єктом дозволяє замінити одну реалізацію інтерфейсу іншою без змін у клієнтському коді. Клієнт не потребує перекомпіляції для використання нової версії. Інколи навіть не потрібно перезапускати застосунок. Якщо інтерфейс залишається незмінним, об’єкти, що його реалізують, можуть еволюціонувати, і нові версії можна впроваджувати плавно й легко. Для реалізації функціональності, описаної інтерфейсом, у середині компонента використовують традиційні об'єктно-орієнтовані методи, але отримані ієрархії класів зазвичай простіші та легші в супроводі.

[https://www.oreilly.com/library/view/programming-net components/0596102070/ch01s03.html]

### Бінарна сумісність між клієнтом і сервером

Ще одним ключовим принципом компонентного програмування є бінарна сумісність між клієнтом і сервером. У традиційному об'єктно-орієнтованому програмуванні всі учасники – як клієнти, так і сервери – повинні бути частиною єдиного монолітного застосунку.

Натомість компонентне програмування ґрунтується на пакуванні коду в компоненти – тобто бінарні будівельні блоки. Зміни в коді компонента залишаються локалізованими всередині цього бінарного блоку; немає потреби перекомпілювати та повторно розгортати клієнтські застосунки. Однак можливість заміни та підключення нових бінарних версій сервера передбачає бінарну сумісність між клієнтом і сервером, тобто клієнтський код під час виконання повинен взаємодіяти з точками входу компонента, які точно відповідають його очікуванням щодо розміщення в пам’яті.

Ця бінарна сумісність є основою контракту між компонентом і клієнтом. Поки нова версія компонента дотримується цього контракту, клієнтський код залишатиметься незмінним і працездатним.

### Незалежність від мови програмування

На відміну від традиційного об'єктно-орієнтованого програмування, у компонентному програмуванні сервер розробляється незалежно від клієнта. Оскільки клієнт взаємодіє із сервером лише під час виконання, єдиним зв’язком між ними є бінарна сумісність.

Звідси випливає важливий наслідок: мова програмування, якою реалізовано клієнт і сервер, не повинна впливати на їхню здатність взаємодіяти під час виконання.

### Прозорість розташування

Компонентний застосунок містить кілька бінарних компонентів. Ці компоненти можуть перебувати в одному процесі, у різних процесах на одній машині або на різних машинах у мережі. З появою вебсервісів компоненти тепер також можуть бути розміщені через Інтернет.

Базова компонентна технологія має забезпечувати клієнту прозорість розташування, що дозволяє клієнтському коду не залежати від фактичного розташування об'єктів, якими він користується. Прозорість розташування означає, що в коді клієнта немає нічого, що вказує на місце виконання об’єктів. Той самий клієнтський код має бути здатен обробляти всі варіанти розташування об’єкта:

* у тому самому процесі;
* у різних процесах на тій самій машині;
* на різних машинах у тій самій локальній мережі;
* або навіть на різних машинах через Інтернет.

### Керування одночасним доступом

Розробник компонента не може заздалегідь знати всі можливі способи його використання, особливо – чи будуть до нього одночасно звертатися кілька потоків. Тож слід заздалегідь підготуватись до використання компонента в багатопотоковому середовищі.

Одним із варіантів є реалізація механізму синхронізації всередині самого компонента. Однак такий підхід має два недоліки:

* Може призвести до взаємоблокування (deadlock) – за наявності власного синхронізаційного замка в кожного компонента застосунку, може виникнути ситуація, коли два компоненти з різних потоків намагаються звернутися один до одного.
* Неефективне використання системних ресурсів – якщо всі компоненти в застосунку будуть доступні лише одному потоку.

Щоб уникнути цих проблем, базова компонентна технологія повинна надавати сервіс керування одночасним доступом – спосіб, за допомогою якого компоненти можуть брати участь у загальній для всього застосунку системі синхронізації, навіть якщо самі компоненти були розроблені окремо.

Крім того, базова технологія має дозволяти як компонентам, так і клієнтам реалізовувати власні рішення синхронізації для більш точного контролю і кращої продуктивності.

### Підтримка контролю версій (Versioning)

Компонентне програмування повинно дозволяти клієнтам і компонентам розвиватися незалежно один від одного. Розробники компонентів мають мати змогу випускати нові версії (або просто виправлення) існуючих компонентів без впливу на вже існуючі клієнтські застосунки. Розробники клієнтських застосунків мають мати можливість випускати нові версії своїх програм і очікувати їхню сумісність зі старішими версіями компонентів.

Базова компонентна технологія повинна підтримувати контроль версій, що дозволяє компоненту розвиватися в різних напрямках і дозволяє одночасно розміщувати на одній машині або навіть в одному клієнтському процесі кілька версій одного й того самого компонента. Технологія також має виявляти несумісності якомога раніше і повідомляти про них клієнта.

### Безпека на основі компонентів

У компонентно-орієнтованому програмуванні компоненти розробляються окремо від клієнтських застосунків, які їх використовують. Розробники компонентів не можуть знати, як саме клієнтський застосунок або кінцевий користувач скористається їхньою роботою. Аналогічно, клієнтський застосунок не може знати, чи взаємодіє він із зловмисним компонентом, який зловживає наданими клієнтом правами доступу. Крім того, навіть якщо і клієнт, і компонент не мають злих намірів, кінцевий користувач застосунку все одно може спробувати зламати систему або випадково завдати шкоди.

Щоб зменшити ці ризики, компонентна технологія має надавати безпекову інфраструктуру для роботи з такими ситуаціями, без жорсткого зв’язку між компонентами та клієнтськими застосунками. Вимоги до безпеки, політики та події (наприклад, додавання нових користувачів) є одними з найбільш мінливих аспектів життєвого циклу застосунку, і політики безпеки відрізняються в різних застосунках і для різних клієнтів.

Продуктивна компонентна технологія повинна забезпечувати мінімум безпекових політик у самих компонентах і мінімум безпекової логіки в коді. Вона також має дозволяти системним адміністраторам налаштовувати і керувати політикою безпеки застосунку без необхідності змінювати код.

## Single Page Application

Односторінковий додаток (Single Page Application) – це веб-додаток, який завантажує лише один HTML-документ, а далі змінює вміст сторінки динамічно через JavaScript (наприклад, за допомогою Fetch API), коли потрібно показати інший контент.

Це дозволяє користувачам працювати з сайтом без повного перезавантаження сторінок, що покращує швидкість і робить взаємодію більш плавною та динамічною. Водночас таке рішення має і недоліки: складніше оптимізувати SEO, складніше підтримувати стан додатку, реалізувати навігацію і ефективний моніторинг продуктивності.

[https://developer.mozilla.org/en-US/docs/Glossary/SPA]

## Переваги SPA

SPA-рішення мають багато переваг як з точки зору клієнтського досвіду, так і з точки зору внутрішньої розробки. Покращена продуктивність додатків, узгодженість, скорочення часу розробки та зниження витрат на інфраструктуру дозволяють командам розробників працювати більш ефективно.

### Одноразове завантаження

Після першого завантаження сторінки сервер більше не надсилає HTML. Користувач отримує “оболонку” сторінки, а браузер відображає інтерфейс. Далі, при навігації, SPA запитує лише дані, а браузер оновлює потрібні частини сторінки без повного перезавантаження. Сервер не витрачає ресурси на рендеринг HTML під час кожного переходу, що знижує навантаження на сервер. Це дуже зручно для сайтів з великою кількістю переходів і шаблонів.

### Швидка розробка фронтенду

Разом із швидшим часом виконання, SPA також дозволяють розробникам швидше створювати фронтенд сайтів. Це пов’язано з роз’єднаною архітектурою SPA, а саме відокремленням бекенд-сервісів від фронтенд-відображення.

Багато критично важливих бізнес-функцій на бекенді змінюються досить рідко на відміну від їх відображення. Відокремлюючи логіку і дані бекенду від того, як вони презентуються, вони перетворюються на сервіси (див. розділ \_), розробники можуть створювати багато різних фронтенд-рішень для показу і використання цих сервісів.

З такою роз’єднаною структурою розробники можуть створювати, розгортати і експериментувати з фронтендом повністю незалежно від технології, що лежить в основі бекенду. Вони проектують, як саме хочуть, щоб виглядав і відчувався користувацький досвід, а потім отримують контент, дані і функціональність через ці сервіси

### Покращений користувацький досвід

Чим більше функціональностей створюється у вигляді модульних сервісів (архітектура мікросервісів), які можна оновлювати незалежно, тим легше експериментувати з тим, як вони відображаються та використовуються. SPA-застосунки, можуть без проблем працювати з бекенд-сервісами, розробленими різними мовами.

Фреймворки SPA чудово підходять для тестування таких сервісів з метою створення привабливого, динамічного та навіть анімованого користувацького досвіду.

[https://www.bloomreach.com/en/blog/what-is-a-single-page-application]

## Недоліки SPA

Будь-яка технологія, включаючи має свої недоліки. У випадку зі SPA це перш за все проблеми пов’язані із сприйняттям SPA-застосунків браузерами, які повинні сприймати різні стани застосунку, як окремі вебсторінки, що може не завжди здійснюватися коректно.

### Проблеми з SEO

SPА традиційно стикаються з труднощами пошукової оптимізації (SEO). Оскільки більшість контенту завантажується динамічно за допомогою JavaScript, пошуковим системам може бути складно індексувати такий контент, що може негативно вплинути на рейтинг сайту. Втім, існують обхідні шляхи й техніки для покращення SEO в SPA.

### Час початкового завантаження

Хоча SPА забезпечують швидкий та чутливий інтерфейс після завантаження, сам цей процес може тривати довше, ніж у традиційних сайтів. Це пов’язано з тим, що вся програма з усіма ресурсами завантажується одразу. Це може бути проблемою для користувачів із повільним Інтернет-з’єднанням.

### Історія браузера і закладки

SPА використовують маршрутизацію на стороні клієнта, тому історія переглядів і закладки браузера можуть працювати не так, як очікується. Користувачам може бути складно ділитися посиланнями або використовувати історію переглядів.

### Складна розробка

Розробка SPА зазвичай складніша, ніж створення традиційного сайту. Потрібно управляти маршрутизацією на клієнті, станом додатка та асинхронним отриманням даних. Це ускладнює розробку й може спричинити триваліший процес налагодження.

### Проблеми безпеки

SPА можуть бути вразливими до деяких типів атак, наприклад, міжсайтового скриптингу (XSS – Cross-Site Scripting), якщо не вжити належних заходів безпеки. Розробникам слід особливо уважно ставитися до захисту таких додатків від потенційних загроз.

[https://medium.com/@VAISHAK\_CP/the-pros-and-cons-of-single-page-applications-spas-06d8a662a149]

## React

React (також React.js або ReactJS) – це фронтенд JavaScript-бібліотека з відкритим кодом, яка має на меті зробити створення інтерфейсів на основі компонентів більш «плавним». Її підтримують компанія Meta (раніше Facebook) та спільнота окремих розробників і компаній.

React можна використовувати для розробки односторінкових, мобільних або серверно-рендерених застосунків за допомогою таких фреймворків, як Next.js і Remix. Оскільки React відповідає лише за інтерфейс користувача та рендеринг компонентів у DOM, застосунки на React зазвичай покладаються на сторонні бібліотеки для маршрутизації та іншої клієнтської функціональності.

Таким чином React активно використовується для створення Single Page Application в рамках концепції компонентного програмування. Варто зазначити, що ключовою перевагою React є повторний рендеринг лише тих частин сторінки, які змінилися, а отже уникання непотрібного оновлення незмінних DOM-елементів.

## Особливості React

### Декларативність

React дотримується декларативної парадигми програмування. Розробники створюють views для кожного стану застосунку, а React оновлює та рендерить компоненти, коли змінюються дані. Це контрастує з імперативним стилем програмування.

### Компоненти

Код React складається з сутностей, які називаються компонентами (див. розділ \_). Ці компоненти модульні й можуть бути повторно використані. Застосунки на React зазвичай складаються з багатьох рівнів компонентів. Вони рендеряться в кореневий елемент DOM за допомогою бібліотеки React DOM. Під час рендерингу значення передаються між компонентами через props (властивості). Внутрішні значення компонента називаються його станом (state). Два основних способи оголошення компонентів у React – це функціональні та класові компоненти.

### Функціональні компоненти

Функціональні компоненти оголошуються за допомогою функції (звичайної або стрілочної), яка приймає один аргумент props і повертає JSX. Починаючи з React v16.8, функціональні компоненти можуть використовувати стан завдяки хуку (див. розділ \_) useState.

### React Hooks

16 лютого 2019 року був випущений React 16.8, який представив React Hooks – функції, що дозволяють "підключатися" до стану та життєвого циклу з функціональних компонентів. Хуки не працюють у класах – вони дозволяють користуватись можливостями React без класів.

Правила хуків

1. Хуки викликаються лише на верхньому рівні функцій – не в циклах/умова.
2. Хуки викликаються тільки в React-функціях – не в звичайних JavaScript-функціях.

### Серверні компоненти (Server Components)

Це функціональні компоненти, які виконуються виключно на сервері. Вперше ідею представили в доповіді "Data Fetching with Server Components". На відміну від Server Side Rendering, вони не передають JavaScript-код клієнту. Вони не мають доступу до хуків, але можуть бути асинхронними, що дозволяє виконувати запити прямо в них.

### Класові компоненти

Оголошуються за допомогою класів ES6. Вони мають ту ж логіку, що й функціональні компоненти, але для роботи зі станом і життєвим циклом використовують методи життєвого циклу з базового класу React.Component.

Запровадження хуків у React 16.8 дозволило керувати станом та життєвим циклом у функціональних компонентах, зменшивши потребу в класових компонентах. Це відповідає загальній тенденції до функціонального програмування та модульного дизайну.

### Маршрутизація (Routing)

React не має вбудованої підтримки маршрутизації. Це бібліотека для побудови інтерфейсів, а не повноцінний фреймворк. Для маршрутизації використовують сторонні бібліотеки, які дозволяють створювати маршрути, керувати навігацією й обробляти зміни в URL.

### Віртуальний DOM

React використовує віртуальний DOM – кеш структури DOM у пам’яті. React обчислює різницю між попереднім і новим станом та ефективно оновлює DOM браузера. Цей процес називається "узгодженням" (reconciliation). Це дозволяє писати код так, ніби вся сторінка рендериться при кожній зміні, але React оновлює лише ті компоненти, які насправді змінились, що значно підвищує продуктивність.

[https://en.wikipedia.org/wiki/React\_(software)#Function\_components]

# Книжковий бізнес в Україні