**Практична робота №3.** Архітектура програмного забезпечення. Визначення складових елементів архітектури ПЗ.

**Тема:** Розробка переліку складових елементів архітектури ПЗ

**Мета:** Навчитись визначати основні аспекти архітектурних рішень для розробки програмного забезпечення

**КОНТРОЛЬНІ ПИТАННЯ:**

1. В чому основна ідея архітектури ПЗ?

2. Визначте складові елементи архітектури ПЗ

3.Які основні питання для розробника архітектури программного продукту?

4. Які архітектурні стилі/парадигми вам відомі?

**Завдання:**

1. Ознайомитися з теоретичною частиною
2. Визначте для вашого проекту (над яким працюєте на останніх лабораторних та практичних роботах) наступні питання:

* В якому оточенні буде працювати програмний продукт (ПП), розроблений за вашим проектом?
* Визначте зацікавлених осіб вашого ПП.
* З яких компонентів буде складатися ваш майбутній ПП?
* Які артефакти ви будете використовувати, щоб розкрити архітектуру вашого ПП?
* Як ви будете представляти основні взаємодіючі елементи архітектури вашого ПП – структуру, поведінку, стиль?
* Як користувач буде використовувати ПП?
* Як ПП буде розгортатися і обслуговуватися при експлуатації?
* Які висуваються вимоги до таких атрибутів ПП, як: якість, безпека, продуктивність, можливість паралельної обробки, інтернаціоналізація і конфігурація?
* Як спроектувати ПП, щоб він залишався гнучким і зручним в обслуговуванні протягом усього періоду експлуатації?
* Які основні архітектурні елементи, які можуть впливати на ПП зараз або після його розгортання?

1. Робота повинна бути виконана до ГОСТ 19.106-78 на аркушах формату А4 відповідно до вимог оформлення лабораторних і практичних робіт (титульний лист, номер роботи, мета, завдання, назва власного проекту, його предметна область). При оформленні використовувати MS Office.
2. Оформлену відповідно до встановлених вимог роботу та здати на перевірку викладачеві, надіславши електронною поштою на адресу [t.i.lumpova@gmail.com](mailto:t.i.lumpova@gmail.com). Якщо викладач знаходить помилки чи неточності, він може повернути роботу на доопрацювання.

Файл з роботою повинен мати назву в такому форматі:

**KPZ <Номер групи><Номер лекції / лабораторної> [літера позначення типу роботи L – лекція, R – лабораторна]<Прізвище англійською>**.. Наприклад, **KPZ4101R**buts.doc.

Тему в заголовку листа записати

**KPZ<Номер групи>-ПР<Номер практичної>-<Прізвище>**

**Строк виконання цієї роботи ІПЗ-41 –**

**ІПЗ-42 – 13.10.2021**

Всі запитання, що виникнуть, надсилайте на електронну адресу викладача, Тему в заголовку листа записати

**KPZ <Номер групи>-Запитання-<Прізвище >**.

**ТЕОРЕТИЧНІ ВІДОМОСТІ**

**Архітектура N-рівневої системи**

ПП складається з деякої впорядкованої сукупності програмних підсистем, названих рівнями (шарами), такий, що:

* на кожному рівні (шарі) нічого не відомо про властивості (і навіть існуванні) подальших (вищих) шарів;
* кожен рівень (шар) може взаємодіяти по управлінню (звертатися до компонентів) з безпосередньо попереднім (нижчим) шаром через заздалегідь певний інтерфейс, нічого не знаючи про внутрішню будову всіх попередніх шарів;
* кожен рівень (шар) має в своєму розпорядженні певні ресурси, які він або приховує від інших шарів, або надає безпосередньо подальшому шару (через вказаний інтерфейс) деякі їх абстракції.

В n-рівневій програмній системі кожен рівень може реалізувати деяку абстракцію даних. Зв'язки між рівнями обмежені передачею значень параметрів звернення кожного рівня до суміжного нижнього рівня і видачею результатів цього звернення від нижнього до верхнього. Тут неприпустимо використання глобальних даних між рівнями.

Як приклад розглянемо використання такої архітектури для побудови операційної системи. Таку архітектуру застосував Дейкстра при побудові операційної системи THE. Ця операційна система складається з чотирьох шарів (див. рис.1).

|  |  |  |
| --- | --- | --- |
| Комп'ютер | Прикладні програми |  |
| 3: Управління вхідними і вихідними потоками даних |
| 2: Забезпечення зв'язку з консоллю оператора |
| 1: Управління пам'яттю |
| 0: Диспетчеризація і синхронізація процесів |

Рис. 1. Архітектура операційної системи THE.

На нульовому шарі проводиться обробка всіх переривань і виділення центрального процесора програмам (процесам) в пакетному режимі. Тільки цей рівень обізнаний про мультипрограмні аспекти системи. На першому шарі здійснюється управління сторінковою організацією пам'яті. Всім вищестоящим шарам надається віртуальна безперервна (не сторінкова) пам'ять. На другому шарі здійснюється зв'язок з консоллю (пультом управління) оператора. Тільки цей шар знає технічні характеристики консолі. На третьому шарі здійснюється буферизація вхідних і вихідних потоків даних і реалізуються так звані абстрактні канали введення і виводу, так що прикладні програми не знають технічних характеристик пристроїв введення і виводу.

## Багатошарова архітектура

Терміни шар (layer) і рівень(tier) часто змішують. Однак -

• шар позначає логічне розділення функціональності,

• рівень позначає фізичне розгортання на різних системах.

Багаторівнева архітектура забезпечує угрупування зв'язаної функціональності застосування в різних шарах, що вибудовуються вертикально, поверх один одного. Функціональність кожного шару об'єднана загальною роллю або відповідальністю. Шари слабо зв'язані, і між ними здійснюється явний обмін даними. Правильне розділення застосування на шари допомагає підтримувати строге розділення функціональності, що у свою чергу, забезпечує гнучкість, а також зручність і простоту обслуговування.

Багатошарова архітектура описана як перевернута піраміда повторного використання, в якій кожен шар агрегує відповідальності і абстракції рівня, розташованого безпосередньо під ним. При строгому розділенні на шари компоненти одного шару можуть взаємодіяти тільки з компонентами того ж шару або компонентами шаруючи, розташованого прямо під даним шаром. Вільніше розділення на шари дозволяє компонентам шари взаємодіяти з компонентами того ж і всіх шарів, що пролягають нижче.

Шари застосування можуть розміщуватися фізично на одному комп'ютері (на одному рівні) або бути розподілені по різних комп'ютерах (n-рівнів), і зв'язок між компонентами різних рівнів здійснюється через строго певні інтерфейси. Наприклад, типове Веб-сервер-застосування складається з шару уявлення (функціональність, пов'язана з UI), бізнес-шар (обробка бізнес-правил) і шаруючи даних (функціональність, пов'язана з доступом до даним, часто практично повністю реалізовується за допомогою високорівневих інфраструктур доступу до даним).

Загальні принципи проектування з використанням багатошарової архітектури:

* Абстракція. Багатошарова архітектура представляє систему як єдине ціле, забезпечуючи при цьому достатньо деталей для розуміння ролей і відповідальностей окремих шарів і відносин між ними.
* Інкапсуляція. Під час проектування немає необхідності робити які-небудь припущення про типи даних, методи і властивості або реалізацію, оскільки всі ці деталі приховані в рамках шару.
* Чітко певні функціональні шари. Розділення функціональності між шарами дуже чітке. Верхні шари, такі як шар уявлення, посилають команди нижнім шарам, таким як бізнес-шар і шар даних, і можуть реагувати на події, що виникають в цих шарах, забезпечуючи можливість передачі даних між шарами вгору і вниз.
* Висока зв'язність. Чітко певні межі відповідальності для кожного шару і гарантоване включення в шар тільки функціональності, безпосередньо пов'язаній з його завданнями, допоможе забезпечити максимальну зв'язність в рамках шару.
* Можливість повторного використання. Відсутність залежностей між нижніми і верхніми шарами забезпечує потенційну можливість їх повторного використання в інших сценаріях.
* Слабке скріплення. Для забезпечення слабкого скріплення між шарами зв'язок між ними грунтується на абстракції і подіях.

Прикладами багатошарових застосувань можуть служити бізнес-додатки (line-of-business, LOB), такі як системи бухгалтерського обліку і управління замовниками; Веб-сервера-застосування і Веб-сайти підприємств; настільні або смарт-клієнти підприємств з централізованими серверами застосувань для розміщення бізнес-логіки.

Багатошарова архітектура підтримується рядом шаблонів проектування. Наприклад, під назвою Separated Presentation (Відділення уявлення) об'єднується ряд шаблонів, що розділяють взаємодію користувача з UI, уявлення, бізнес-логіку і дані застосування, з якими працює користувач. Відділення уявлення дозволяє створювати UI в графічних дизайнерах, тоді як розробники пишуть код, що управляє. Таке розділення функціональності на ролі підвищує можливість тестування поведінки окремих ролей.

Можливість застосування багатошарової архітектури необхідно розглянути, якщо у вашому розпорядженні є вже готові рівні, відповідні для повторного використання в інших застосування; якщо є застосування, що надають відповідні бізнес-процеси через інтерфейси сервісів; або якщо створюється складне застосування і попереднє проектування вимагає розділення, щоб групи могли зосередитися на різних ділянках функціональності. Багатошарова архітектура також буде доречна, якщо застосування повинне підтримувати різні типи клієнтів і різні пристрої, або якщо потрібно реалізувати складні і/або бізнес-правила, що настроюються, і процеси.

**Компонентна архітектура**

Компонентна архітектура описує підхід до проектування і розробки систем з використанням методів проектування програмного забезпечення. Основна увага в цьому випадку приділяється розкладанню дизайну на окремі функціональні або логічні компоненти, певні інтерфейси, що надають чітко, містять методи, події і властивості. В даному випадку забезпечується вищий рівень абстракції, чим при об'єктно-орієнтованій розробці, і не відбувається концентрації уваги на таких питаннях, як протоколи зв'язку або загальний стан.

Компоненти повинні володіти наступними якостями:

* Повторне використання. Як правило, компоненти проектуються із забезпеченням можливості їх повторного використання в різних сценаріях різних застосувань. Проте деякі компоненти створюються спеціально для конкретного завдання.
* Взаємозамінність. Компоненти можуть без зусиль замінюватися іншими подібними компонентами.
* Незалежність від контексту. Компоненти проектуються для роботи в різних середовищах і контекстах. Спеціальні відомості, такі як дані про стан, повинні не включатися або витягуватися компонентом, а передаватися в нього.
* Розширюваність. Компонент може розширювати існуючі компоненти для забезпечення нової поведінки.
* Інкапсуляція. Компоненти надають інтерфейси, що дозволяють іншій стороні використовувати їх функціональність, не розкриваючи при цьому деталі внутрішніх процесів або внутрішні змінні або стан.
* Незалежність. Компоненти проектуються з мінімальними залежностями від інших компонентів. Таким чином, компоненти можуть бути розгорнені в будь-якому відповідному середовищі без впливу на інші компоненти або системи.

Зазвичай в застосуваннях використовуються компоненти призначеного для користувача інтерфейсу (їх часто називають елементами управління), такі як таблиці і кнопки, а також допоміжні або службові компоненти, що надають певний набір функцій, використовуваних в інших компонентах. До іншого типу поширених компонентів відносяться ресурсоємні компоненти, доступ до яких здійснюється нечасто, і активація яких виконується «точно вчасно» (just-in-time, JIT) (зазвичай використовується в сценаріях з видаленими або розподіленими компонентами); і компоненти з чергою, виклики методів яких можуть виконуватися асинхронно за рахунок застосування черги повідомлень, для зберігання і пересилки.

Компоненти залежать від платформи:

• Об'єктна модель програмних компонентів (component object model, COM) в Windows

• Об'єктна модель розподілених програмних компонентів (distributed component object model, DCOM) в Windows

• Загальна архітектура брокера об'єктних запитів (Common Object Request Broker Architecture, CORBA)

* Серверні компоненти Java (Enterprise JavaBeans, EJB) на інших платформах.

Використовувана компонентна архітектура описує механізми розміщення компонентів і їх інтерфейсів, передачі повідомлень або команд між компонентами і, в деяких випадках, збереження стану.

Основні переваги компонентної архітектури:

* Простота розгортання. Існуючі версії компонентів можуть замінюватися новими сумісними версіями, не роблячи впливу на інші компоненти або систему в цілому.
* Менша вартість. Використання компонентів сторонніх виробників дозволяє розподіляти витрати на розробку і обслуговування.
* Простота розробки. Для забезпечення заданої функціональності компоненти реалізують широко відомі інтерфейси, що дозволяє вести розробку без впливу на інші частини системи.
* Можливість повторного використання. Застосування багато разів використовуваних компонентів означає можливість розподілу витрат на розробку і обслуговування між декількома застосуваннями або системами.
* Спрощення з технічної точки зору. Компоненти спрощують систему через використання контейнера компонентів і його сервісів. Як приклади сервісів, що надаються контейнером, можна привести активацію компонентів, управління життєвим циклом, організацію черги викликів методів, обробку подій і транзакції.

**Об'єктно-орієнтована архітектура**

Об'єктно-орієнтована архітектура – це парадигма проектування, заснована на розділенні ПЗ або окремої системи на самостійні, придатні для повторного використання, об'єкти, кожен з яких містить дані і поведінку, що відносяться до цього об'єкту.

При об'єктно-орієнтованому проектуванні система розглядається не як набір підпрограм і процедурних команд, а як набори взаємодіючих об'єктів. Об'єкти відособлені, незалежні і слабо зв'язані; обмін даними між ними відбувається через інтерфейси шляхом виклику методів і властивостей інших об'єктів і відправки/прийому повідомлень.

Основними принципами об'єктно-орієнтованого архітектурного стилю є:

* Абстракція. Дозволяє перетворити складну операцію в узагальнення, що зберігає основні характеристики операції. Наприклад, абстрактний інтерфейс може бути широко відомим описом, що підтримує операції доступу до даним через використання простих методів, таких як Get (Отримати) і Update (Відновити). Інша форма абстракції – метадані, використовувані для забезпечення зіставлення двох форматів структурованих даних.
* Композиція. Об'єкти можуть бути утворені іншими об'єктами і за бажанням можуть приховувати ці внутрішні об'єкти від інших класів або надавати їх як прості інтерфейси.
* Наслідування. Об'єкти можуть успадковуватися від інших об'єктів і використовувати функціональність базового об'єкту або перевизначати її для реалізації нової поведінки. Більш того, спадкоємство спрощує обслуговування і оновлення, оскільки зміни, що вносяться до базового об'єкту, автоматично розповсюджуються на всі успадковані від нього об'єкти.
* Інкапсуляція. Об'єкти надають функціональність тільки через методи, властивості і події і приховують внутрішні деталі, такі як стан і змінні, від інших об'єктів. Це спрощує оновлення або заміну об'єктів і дозволяє виконувати ці операції без впливу на інші об'єкти і код, потрібно лише забезпечити сумісні інтерфейси.
* Поліморфізм. Дозволяє перевизначати поведінку базового типу, що підтримує операції в застосуванні, шляхом реалізації нових типів, які є взаємозамінними для існуючого об'єкту.
* Відділення. Об'єкти можуть бути відокремлені від споживача шляхом визначення абстрактного інтерфейсу, що реалізовується об'єктом і зрозумілого споживачеві. Це дозволяє забезпечувати альтернативні реалізації, не роблячи впливу на споживачів інтерфейсу.

Зазвичай, об'єктно-орієнтований стиль використовується для опису об'єктної моделі, що підтримує складні наукові або фінансові операції, або описи об'єктів, що представляють реальні артефакти наочної області (такі як покупець або замовлення).

До основних переваг об'єктно-орієнтованої архітектури відносяться:

* Зрозумілість. Забезпечується ближча відповідність застосування реальним об'єктам, що робить його зрозумілішим.
* Можливість повторного використання. Забезпечується можливість повторного використання через поліморфізм і абстракцію.
* Тестованість. Забезпечується покращена тестованість через інкапсуляцію. Розширюваність, інкапсуляція, поліморфізм і абстракція гарантують, що зміни в уявленні даних не вплинуть на інтерфейси, що надаються об'єктами, що могло б обмежити можливості зв'язку і взаємодії з іншими об'єктами.
* Висока зв'язність. Розміщуючи в об'єкті тільки функціонально близькі методи і функції і використовуючи для різних наборів функцій різні об'єкти, можна досягти високого рівня зв'язності.

## Сервісно-орієнтована архітектура

Сервісно-орієнтована архітектура (Service-oriented architecture, SOA) забезпечує можливість надавати функціональність застосування у вигляді набору сервісів і створювати застосування, що використовують програмні сервіси. Сервіси слабо зв'язані, тому що використовують засновані на стандартах інтерфейси, які можуть бути викликані, опубліковані і виявлені.

Основне завдання сервісів в SOA – надання схеми і взаємодії із застосуванням за допомогою повідомлень через інтерфейси, областю дії яких є застосування, а не компонент або об'єкт. Не слід розглядати SOA-сервіс як компонентний постачальник сервісів.

SOA-архітектура може забезпечити упаковку бізнес-процесів в сервіси, що підтримують можливість взаємодії і що використовують для передачі інформації широкий діапазон протоколів і форматів даних. Клієнти і інші сервіси можуть виконувати доступ до локальних сервісів, що виконуються на тому ж рівні, або до видалених сервісів по мережі.

Основними принципами архітектурного стилю SOA є:

* Сервіси автономні. Обслуговування, розробка, розгортання і контроль версій кожного сервісу відбувається незалежно від інших.
* Сервіси можуть бути розподілені. Сервіси можуть розміщуватися в будь-якому місці мережі, локально або видалено, якщо мережа підтримує необхідні протоколи зв'язку.
* Сервіси слабо зв'язані. Кожен сервіс абсолютно не залежить від останніх і може бути замінений або оновлений без впливу на застосування, що його використовують, за умови надання сумісного інтерфейсу.
* Сервіси спільно використовують схему і контракт, але не клас. При обміні даними сервіси спільно використовують контракти і схеми, але не внутрішні класи.
* Сумісність заснована на політиці. Політика, в даному випадку, означає опис характеристик, таких як транспорт, протокол і безпека.

Типові сервісно-орієнтовані ПП забезпечують сумісне використання інформації, виконання багатоетапних процесів (системи резервування і онлайн- магазини), надання спеціальних галузевих даних або сервісів між організаціями і створення складених застосувань, які об'єднують дані з багатьох джерел.

Основними перевагами SOA-архітектури є:

* Узгодження наочних областей. Повторне використання загальних сервісів із стандартними інтерфейсами розширює технологічні і бізнес-можливості, а також скорочує вартість.
* Абстракція. Сервіси є автономними, доступ до них здійснюється по формальному контракту, що забезпечує слабке скріплення і абстракцію.
* Можливість виявлення. Сервіси можуть надавати описи, що дозволяє іншим застосуванням і сервісам виявляти їх і автоматично визначати інтерфейс.
* Можливість взаємодії. Оскільки протоколи і формати даних базуються на галузевих стандартах, постачальник і споживач сервісу можуть створюватися і розгортатися на різних платформах.
* Раціоналізація. Сервіси забезпечують певну функціональність, усуваючи необхідність її дублювання в застосуваннях.

**Контроль архітектури ПП**

Для контролю архітектури ПП використовується суміжний контроль і ручна імітація.

Суміжний контроль архітектури ПЗ зверху - це її контроль розробниками зовнішнього опису: розробниками специфікації якості і розробниками функціональної специфікації. Суміжний контроль архітектури ПЗ знизу - це її контроль потенційними розробниками програмних підсистем, що входять до складу ПЗ відповідно до розробленої архітектури.

Ручна імітація архітектури ПЗ проводиться аналогічно ручній імітації функціональної специфікації, тільки метою цього контролю є перевірка взаємодії між програмними підсистемами. Так само як і у разі ручної імітації функціональної специфікації ПЗ повинні бути спочатку підготовлені тести. Потім група розробників повинна для кожного такого тесту імітувати роботу кожної програмної підсистеми, що входить до складу ПЗ. При цьому роботу кожної підсистеми імітує один який-небудь розробник (не автор архітектури), ретельно виконуючи всі взаємодії цієї підсистеми з іншими підсистемами (точніше, з розробниками, що їх імітують) відповідно до розробленої архітектури ПЗ. Тим самим забезпечується імітаційне функціонування ПЗ в цілому в рамках архітектури, що перевіряється.

## Архітектурні стилі (шаблони) ПП

Архітектурний стиль (архітектурний шаблон) – це набір принципів (високорівнева схема), що забезпечує абстрактну інфраструктуру для великого сімейства систем.

Архітектурний стиль покращує секціонування і сприяє повторному використанню дизайну завдяки забезпеченню вирішень проблем, які часто зустрічаються.

Архітектурні стилі і шаблони можна розглядати як набір принципів, що формують ПП.

Архітектура ПЗ зазвичай містить декілька видів, які аналогічні різним типам креслень будинків у будівництві. В онтології, встановленої ANSI/IEEE 1471-2000, види є екземплярами точки зору, де точки зору існують для опису архітектури з точки зору заданої множини зацікавлених осіб.

У якості стандарту "для моделювання програмних систем (і не тільки)" було створено мову UML.

**Уніфікована мова моделювання UML**

Unified Modeling Language, скорочено UML, застосовується на різних етапах розробки програмного забезпечення (ПЗ). Мова UML є спільноцільовою мовою візуального моделювання, яку розроблено для специфікації, візуалізації, проектування та документування компонентів програмного забезпечення, бізнес-процесів та інших систем. Мова UML одночасно є простим і потужним засобом моделювання, який може бути ефективно використаний для побудови концептуальних, логічних та графічних моделей складних систем самого різного цільового призначення. Ця мова увібрала в себе найкращі якості методів програмної інженерії, які з успіхом використовувалися впродовж останніх років при моделюванні великих і складних систем.

Мова UML заснована на деякому числі базових понять, які можуть бути вивчені й застосовані більшістю програмістів і розробників, знайомих з методами об'єктно-орієнтованого аналізу і проектування. При цьому базові поняття можуть комбінуватися і розширюватися таким чином, що фахівці об'єктного моделювання отримують можливість самостійно розробляти моделі великих та складних систем у самих різних областях додатків.

Конструктивне використання мови UML ґрунтується на розумінні загальних принципів моделювання складних систем та особливостей процесу об'єктно-орієнтованого аналізу і проектування зокрема. Вибір засобів для побудови моделей складних систем зумовлює ті завдання, які можуть бути вирішені з використанням даних моделей. При цьому одним з основних принципів побудови моделей складних систем є принцип абстрагування, який наказує включати в модель тільки ті аспекти проектованої системи, які мають безпосереднє відношення до виконання системою своїх функцій або свого цільового призначення. При цьому всі другорядні деталі опускаються, щоб надмірно не ускладнювати процес аналізу та дослідження отриманої моделі.

Іншим принципом побудови моделей складних систем є принцип багатомодельності. Цей принцип є твердженням про те, що жодна єдина модель не може з достатнім ступенем адекватності описувати різні аспекти складної системи. Стосовно до методології ООАП це означає, що достатньо повно модель складної системи допускає деяке число взаємопов'язаних уявлень (views), кожне з яких адекватно відображає певний аспект поведінки або структури системи. При цьому найбільш загальними уявленнями складної системи прийнято вважати статичне і динамічне представлення, які у свою чергу можуть підрозділятися на інші більш приватні уявлення. Феномен складної системи якраз і полягає в тому, що ніяке її єдине подання не є достатнім для адекватного вираження всіх особливостей модельованої системи.

Ще одним принципом прикладного системного аналізу є принцип ієрархічної побудови моделей складних систем. Цей принцип наказує розглядати процес побудови моделі на різних рівнях абстрагування або деталізації в рамках фіксованих уявлень.

**Шаблони проектування**

Шаблони проектування (паттерн, англ. design pattern) - це багато разів застосовувана архітектурна конструкція, що надає рішення загальної проблеми проектування в рамках конкретного контексту й описує значимість цього рішення. Паттерн не є закінченим зразком проекту, який може бути прямо перетворений в код.

Це опис або зразок для того, як вирішити завдання таким чином, щоб це можна було використовувати в різних ситуаціях. Об'єктно-орієнтовані шаблони часто показують відносини і взаємодії між класами або об'єктами, без визначення того, які кінцеві класи чи об'єкти додатку будуть використовуватися.

Алгоритми за своєю суттю також є шаблонами, але не проектування, а обчислення, так як вирішують обчислювальні завдання. Шаблони проектування незалежні від застосовуваної мови програмування.

Паттерни - це стандартні рішення типових завдань, що виникають в об'єктно-орієнтованому проектуванні, якийсь набір готових рішень, придатних для більшості завдань.

Розглянемо перші 23 паттерни, з яких все почалося і які зараз повинен знати кожен програміст. Вони розділені на три групи (для кожного паттерну наводиться англійська назва з книги GoF ).

*"Твірні шаблони".* У цій групі зібрані паттерни, що описують різні способи створення об'єктів. Перш за все це "**Фабричний метод**" (Factory Method), прийом визначення інтерфейсу створення об'єктів, при цьому обраний клас втілюється у підкласах. Шаблон "**Абстрактна фабрика**" (Abstract Factory) визначає інтерфейс для створення сімейств, пов'язаних між собою або незалежних об'єктів, конкретні класи яких невідомі. За допомогою шаблону "**Будівельник**" (Builder) можна відокремити процес конструювання складного об'єкту від його конкретного уявлення і при цьому використовувати один і той же процес для створення різних уявлень. "**Прототип**" (Prototype) описує види розроблюваних об'єктів за допомогою прототипу і створює нові шляхом його копіювання. Застосування шаблону **"Одинак**" (Singleton) гарантує, що деякий клас може мати тільки один примірник (і надає глобальну точку доступу до нього).

*"Структурні паттерни".* У цій групі зібрані паттерни, які дозволяють змінювати структуру взаємодії класів. "**Адаптер**" (Adapter) дозволяє адаптувати інтерфейс класу до конкретної ситуації, засобами шаблону. "**Міст**" (Bridge) можна відокремити інтерфейс класу і його реалізацію, "**Компонувальник**" (Composite) об'єднує об'єкти в деревовидну структуру для представлення ієрархії від приватного до цілого. Компонувальник дозволяє клієнтам одноманітно звертатися до окремих об'єктів і груп об'єктів. Паттерн "**Оформлювач**" (Decorator, також відомий як Wrapper, "**Оболонка**") дозволяє динамічно додавати нову поведінку до об'єкта, "**Фасад**" (Facade) - приховати складність системи шляхом зведення всіх можливих зовнішніх викликів до одного об'єкту, що делегує їх відповідним об'єктам системи. Шаблон "**Пристосуванець**" (Flyweight) використовується для полегшення роботи з великим числом дрібних об'єктів, а "**Заступник**" (Proxy) - контролювати доступ до об'єкта, перехоплюючи усі виклики до нього.

У групі *"Паттерни поведінки"* зібрані шаблони, відповідальні за реалізацію поведінки об'єктів. "**Ланцюжок відповідей**" (Chain of Response) дозволяє пропустити запит через ланцюжок об'єктів, "**Команда"** (Command) інкапсулює команду в об'єкт, "**Інтерпретатор**" (Interpreter) дозволяє створити спільне декларативне рішення для часто змінюваних умов завдання. У шаблоні "**Ітератор**" (Iterator) організовується послідовний доступ до колекції, "**Посередник**" (Mediator) визначає спрощений механізм взаємодії класів, "**Нагадування"** (Memento) задає принципи, що дозволяють записувати і відновлювати внутрішній стан об'єкта. Засобами шаблону "**Спостерігач**" (Observer) можна повідомляти про зміни безлічі об'єктів, "**Стан**" (State) - змінювати поведінку об'єкта при зміні його стану. "**Стратегія**" (Strategy) інкапсулює алгоритм усередині класу.

Паттерн "Шаблонний метод" (Template Method) виділяє конкретні кроки в алгоритмі й спирається на підкласи для їх реалізації. Засобами паттерну "**Відвідувач**" (Visitor) до класу додаються нові операції без його зміни.

Головна користь кожного окремого шаблону полягає у тому, що він описує рішення цілого класу абстрактних проблем. Також той факт, що кожен шаблон має своє ім'я, полегшує дискусію про абстрактні структури даних (ADT) між розробниками, так як вони можуть посилатися на відомі шаблони. Таким чином, за рахунок шаблонів проводиться уніфікація термінології, назв модулів і елементів проекту.

Правильно сформульований шаблон проектування дозволяє, відшукавши вдале рішення, користуватися ним знову і знову.