**1. Складні та великі системи**

**1.1 Властивості систем: емерджентність, адитивність, еквіфінальність.**

*1. Емерджентність*: Ця властивість описує явище, коли система виявляє нові властивості або поведінку, які не можна пояснити або передбачити шляхом аналізу її окремих складових частин. Емерджентність виникає зі взаємодії між компонентами системи.

*2. Адитивність*: Це властивість, коли властивості системи визначаються сумою властивостей її складових частин. В той час як це може бути правдою для деяких систем, у більш складних випадках взаємодія між складовими може призводити до емерджентних властивостей, які не можуть бути просто визначені через суму властивостей окремих елементів.

*3. Еквіфінальність*: Ця властивість вказує на те, що система може досягти того самого результату з різних початкових умов або через різні шляхи. Це означає, що існує багато можливих шляхів досягнення одного й того ж результату, або що система може змінювати свою поведінку в залежності від умов, у яких вона знаходиться.

**1.2 Відкриті та закриті системи; класифікація за призначенням, походженням, видом елементів, способом організації.**

Системи можна класифікувати за різними критеріями, включаючи призначення, походження, вид елементів та спосіб організації. Основні класифікації включають відкриті та закриті системи:

*1. Відкриті системи*: Це системи, які взаємодіють з навколишнім середовищем, обмінюючися енергією, ресурсами або інформацією. Вони відкриті для зовнішнього впливу та можуть пристосовуватися до змін у середовищі. Приклади таких систем включають економічні системи, екосистеми та соціальні системи.

*2. Закриті системи*: Це системи, які ізольовані від зовнішнього середовища і не обмінюються ним енергією або ресурсами. Вони зазвичай працюють за внутрішніми правилами та процесами і не впливають на зовнішнє середовище. Приклади таких систем включають замкнені контурні системи у фізиці та обмежені ізольовані лабораторні експерименти.

Класифікація за призначенням може включати такі типи систем, як інформаційні системи, технічні системи, соціальні системи тощо. Класифікація за походженням може враховувати структуру та походження системи, наприклад, природні чи штучні системи. Класифікація за видом елементів може визначати системи за їх складовими частинами, такими як апаратура, програмне забезпечення, люди або процеси. Класифікація за способом організації може враховувати структурні та функціональні особливості системи, такі як ієрархічна організація, мережеві структури або децентралізовані системи.

**1.3 Спільне та відмінності складних і великих систем.**

Складні і великі системи подібні у тому, що обидві можуть містити багато взаємозалежних компонентів та процесів. Ось деякі спільні риси:

*1. Складність*: Якщо мається на увазі кількість взаємодіючих компонентів та складність їх взаємодії, складні та великі системи можуть бути дуже схожими. Вони можуть включати багато елементів, що взаємодіють між собою в непередбачуваний спосіб.

*2. Величина*: Обидві можуть бути значної розмірності, як за кількістю компонентів, так і за обсягом оброблюваних даних чи об'ємом ресурсів, які вони використовують.

Відмінності між ними полягають у контексті та характеристиках:

*1. Область застосування*: Великі системи можуть включати в себе складні підсистеми, але ці системи часто фокусуються на роботі з великим обсягом даних або обробці великих потоків інформації. Складні системи можуть бути більш різноманітними за своїм призначенням і функціоналом.

*2. Взаємодія*: У великих системах взаємодія між компонентами може бути менш тісно зв'язаною, оскільки вони можуть працювати незалежно або в різних частинах системи. У складних системах компоненти частіше взаємодіють між собою та можуть бути взаємозалежними.

*3. Розуміння*: Великі системи можуть бути більш прогнозованими і менш непередбачуваними через їх масштаб та обсяг даних. Складні системи можуть мати більш складні моделі та потребувати глибшого розуміння їх взаємодії для ефективного управління ними.

**2. Моделі систем**

**2.1 Склад і структура системи; моделі типу чорної та білої скриньки.**

Склад і структура системи визначають, з яких елементів вона складається і як вони взаємодіють між собою.

*1. Склад системи*: Це сукупність всіх компонентів або частин, які утворюють систему. Наприклад, у випадку комп'ютерної системи склад може включати апаратне забезпечення, програмне забезпечення, мережеві з'єднання та користувачів.

*2. Структура системи*: Це опис того, як компоненти системи організовані та взаємодіють між собою. Структура може бути ієрархічною, мережевою, лінійною або будь-якою іншою формою, яка відображає взаємозв'язки між компонентами.

Моделі типу чорної та білої скриньки - це методи опису системи з різних точок зору:

*1. Модель чорної скриньки*: У цій моделі система розглядається як чорний ящик, про внутрішню структуру та роботу якого нічого не відомо. Замість цього, вона розглядається як чорний ящик, з якого можна отримати результати, не розглядаючи деталі внутрішнього функціонування. Наприклад, користувачі програмного забезпечення можуть використовувати його функціонал без знання його внутрішньої реалізації.

*2. Модель білої скриньки*: У цій моделі система розглядається як відкрита книга, де структура та робота всіх її компонентів є відомою. Це означає, що аналітик або інженер може досліджувати систему, знаючи всі деталі її функціонування та взаємозв'язки між компонентами.

Обидва підходи використовуються для аналізу та моделювання систем, але вони ставляться у контексті різного рівня абстракції та деталізації.

**2.2 Концептуальні, математичні і комп’ютерні моделі.**

Концептуальні, математичні і комп'ютерні моделі - це різні способи представлення реальних систем чи явищ з метою їх аналізу, розуміння та прогнозування.

*1. Концептуальні моделі*: Це уявні або графічні моделі, які описують структуру та взаємозв'язки між елементами системи без використання математичних формул або алгоритмів. Вони допомагають в зрозумінні принципів функціонування системи та визначенні ключових взаємодій.

*2. Математичні моделі*: Це формалізовані моделі, які використовують математичні рівняння, формули або алгоритми для опису системи. Вони дозволяють проводити кількісний аналіз та прогнозування поведінки системи на основі введених даних та параметрів.

*3. Комп'ютерні моделі*: Це моделі, які реалізуються у вигляді програмного забезпечення та виконуються на комп'ютері. Вони використовуються для симуляції реальних процесів або систем, що дозволяє проводити експерименти та аналізувати їх поведінку в різних умовах.

Усі ці типи моделей можуть використовуватися для дослідження та аналізу складних систем у різних областях, таких як фізика, економіка, біологія, інженерія та інші. Кожен з них має свої переваги та обмеження, тому вибір конкретної моделі залежить від мети дослідження та доступних ресурсів.

**2.3 Зв’язок між системою та моделлю; ізо- та гомоморфізм.**

*1. Зв'язок між системою та моделлю*: Система - це реальний об'єкт або процес, який ми хочемо дослідити або зрозуміти. **Модель** - це спрощене або абстрактне представлення цієї системи, яке дозволяє нам аналізувати, прогнозувати або взаємодіяти з нею. Модель створюється з метою дослідження або розуміння реальної системи, і вона відображає деякі аспекти цієї системи.

*2. Ізоморфізм*: Це відображення або взаємозв'язок між двома системами або моделями, яке зберігає структуру та властивості цих систем або моделей. Ізоморфні системи або моделі можуть мати різні конкретні деталі чи елементи, але їх взаємний зв'язок залишається однаковим.

*3. Гомоморфізм*: Це відображення або взаємозв'язок між двома системами або моделями, яке зберігає лише деякі властивості цих систем або моделей. Відображення може перетворювати одні елементи або аспекти системи на інші, але не зберігає всі аспекти відношень між ними.

Отже, ізо- та гомоморфізм - це два поняття, які допомагають у з'ясуванні взаємозв'язків між системами та їх моделями, а також у визначенні того, наскільки точно модель відображає реальну систему.

**3. Інформаційні системи**

**3.1 Поняття, цілі, значення, класифікація за функціональністю, масштабом, сферою застосування.**

*1. Поняття*: Інформаційна система - це система, яка забезпечує збір, зберігання, обробку та передачу інформації з метою підтримки прийняття рішень, виконання завдань та взаємодії між користувачами.

*2. Цілі*: Основними цілями інформаційних систем є полегшення доступу до інформації, підвищення продуктивності роботи, покращення якості прийнятих рішень, автоматизація бізнес-процесів, забезпечення безпеки інформації тощо.

*3. Значення*: Інформаційні системи відіграють важливу роль у сучасному суспільстві, сприяючи вирішенню різноманітних завдань та оптимізації діяльності різних галузей.

*4. Класифікація*:

- За функціональністю: Можуть бути класифіковані на системи обробки транзакцій, системи управління базами даних, системи підтримки прийняття рішень, системи управління знаннями тощо.

- За масштабом: ІС можуть бути розглянуті як малі, середні чи великі, залежно від обсягу оброблюваної інформації та кількості користувачів.

- За сферою застосування: Інформаційні системи можуть бути адаптовані для використання в різних сферах, таких як банківська справа, охорона здоров'я, освіта, виробництво тощо.

**3.2 Забезпечення інформаційних систем: організаційне, інформаційне, математичне, програмне, технічне, лінгвістичне, методичне, правове.**

*1. Організаційне забезпечення*: Включає в себе встановлення правил та процедур, розподіл обов'язків та відповідальності, управління персоналом, планування ресурсів тощо.

*2. Інформаційне забезпечення*: Основний аспект полягає в забезпеченні доступу до потрібної інформації в потрібний час та місце, а також її якості та достовірності.

*3. Математичне забезпечення*: Включає в себе математичні методи та моделі для аналізу, моделювання та оптимізації процесів у системі.

*4. Програмне забезпечення*: Це набір програм та алгоритмів, які використовуються для роботи інформаційної системи.

*5. Технічне забезпечення*: Включає в себе апаратне забезпечення, мережеві пристрої, засоби зберігання даних та інші технічні засоби, необхідні для функціонування системи.

*6. Лінгвістичне забезпечення*: Спрямоване на розвиток та використання мовних засобів комунікації у системі, таких як інтерфейси, документація, комунікаційні протоколи тощо.

*7. Методичне забезпечення*: Включає в себе методики та процедури, що використовуються для розробки, впровадження та підтримки інформаційних систем.

*8. Правове забезпечення*: Основна мета - забезпечення відповідності діяльності системи вимогам законодавства, а також захист прав та інтересів користувачів та власників даних.

**4. Аналіз вимог**

**4.1 Класифікація вимог до програмного забезпечення, джерела та методи збирання вимог;**

Класифікація вимог до програмного забезпечення, джерела та методи збирання вимог - це важливий етап у процесі розробки програмного забезпечення:

1. Класифікація вимог:

- Функціональні вимоги: Описують функції, які має виконувати програмне забезпечення. Наприклад, операції, які повинна виконувати система або результати, які вона повинна генерувати.

- Нефункціональні вимоги: Описують властивості системи, такі як продуктивність, надійність, безпека, доступність та інші аспекти, які не визначаються конкретними функціями.

- Бізнес-вимоги: Виходять за межі технічних аспектів та описують потреби бізнесу або замовника щодо програмного забезпечення.

2. Джерела збирання вимог:

- Клієнти та користувачі: Отримання вимог від кінцевих користувачів або клієнтів, які будуть використовувати програмне забезпечення.

- Експерти: Використання фахівців у відповідних галузях для збору вимог та консультацій.

- Документація: Вивчення документів, таких як специфікації, звіти, проекти, що містять інформацію про потреби та очікування замовника.

3. Методи збирання вимог:

- Інтерв'ю: Особисті зустрічі з потенційними користувачами або замовниками для обговорення їхніх потреб та вимог.

- Анкетування: Збір інформації шляхом опитування користувачів або замовників за допомогою анкет.

- Фокус-групи: Обговорення вимог в групі людей з різними ролями та перспективами.

- Спостереження: Спостереження за реальними процесами або робочими потоками для виявлення потреб та вимог.

**4.2 Вимоги користувача (варіанти використання та історії користувачів).**

*Вимоги користувача* - це опис того, що користувачі вимагають від програмного забезпечення, як вони хочуть взаємодіяти з системою та як вони хочуть, щоб система виконувала певні завдання. Два основних аспекти вимог користувача - це варіанти використання та історії користувачів:

*1. Варіанти використання*: Це конкретні сценарії або шляхи, за якими користувачі взаємодіють з системою для досягнення своїх цілей або виконання певних завдань. Кожен варіант використання може бути описаний від початку до кінця і включати в себе послідовність кроків, які користувачі виконують для досягнення певного результату.

*2. Історії користувачів*: Це короткі описи функціональності або функціональних вимог системи з точки зору користувачів. Історії користувачів зазвичай включають в себе короткий опис функції, роль користувача, який цією функцією користується, та короткий сценарій використання.

Ці два підходи допомагають розробникам та дизайнерам зрозуміти потреби та очікування користувачів щодо системи, що дозволяє їм створити програмне забезпечення, яке відповідає їхнім потребам та надає зручний та ефективний інтерфейс користувача.

**4.3 Функціональні та нефункціональні вимоги, обмеження; структуризація функціональних вимог.**

*1. Функціональні вимоги*: Це вимоги, які описують функції та можливості, які програмне забезпечення повинно здійснювати. Це включає в себе конкретні функції, які користувачі можуть виконувати, операції, які система повинна виконувати, та результати, які вона повинна генерувати.

*2. Нефункціональні вимоги*: Це вимоги, які визначають характеристики системи, які не стосуються конкретних функцій, але впливають на її якість, продуктивність та інші аспекти. Це може включати в себе вимоги до продуктивності, надійності, безпеки, доступності та інші характеристики.

*3. Обмеження*: Це параметри або умови, які обмежують або обмежують реалізацію системи. Це може бути обмеження щодо бюджету, часу, технічних обмежень або вимог до стандартів та регулятивних вимог.

Структуризація функціональних вимог - це процес організації та класифікації функцій та можливостей системи для кращого розуміння та управління ними. Це може включати в себе створення ієрархії функцій, групування схожих функцій разом, визначення пріоритетів та інші аспекти організації функціональності. Це допомагає забезпечити систематичний підхід до визначення та управління функціональними вимогами.

**5. Проектування програмного забезпечення**

**5.1 Види проектування:**

**Структурне проектування (Structural Design)**

**Об’єктно-орієнтоване проектування (Object-Oriented Design)**

**Функціональне проектування (Functional Design)**

**Архітектурне проектування (Architectural Design)**

**Інтерфейсне проектування (Interface Design).**

*1. Структурне проектування (Structural Design)*: Це визначення структури програмного забезпечення, яка включає в себе розподіл програми на модулі або компоненти, визначення взаємозв'язків між ними та інші аспекти, що впливають на організацію коду.

*2. Об'єктно-орієнтоване проектування (Object-Oriented Design)*: Це підход до проектування програмного забезпечення, де програма розглядається як набір взаємодіючих об'єктів, які мають стан та поведінку. Об'єкти групуються в класи, а взаємодія відбувається через методи та повідомлення.

*3. Функціональне проектування (Functional Design)*: Це підход, де програмне забезпечення розглядається як функціональна система, яка складається з функцій та їх взаємозв'язків. Функції розділяються на модулі, кожен з яких виконує певні завдання.

*4. Архітектурне проектування (Architectural Design)*: Це визначення загальної структури програми або системи, включаючи компоненти, їх взаємозв'язки та взаємодію. Архітектурне проектування визначає основні аспекти системи, такі як рівні доступу, розподіленість, масштабованість тощо.

*5. Інтерфейсне проектування (Interface Design)*: Це визначення зовнішнього інтерфейсу користувача або інших систем, з якими взаємодіє програмне забезпечення. Інтерфейсний дизайн включає в себе визначення елементів інтерфейсу, їх вигляду, поведінки та взаємодії з користувачем.

**5.2 Парадигми проектування:**

**функціональна декомпозиція згори донизу,**

**архітектура, орієнтована на дані,**

**об'єктно-орієнтований аналіз та проектування,**

**подієво-керована архітектура.**

*1. Функціональна декомпозиція згори донизу*: Це методологія проектування, де система розбивається на менші функціональні компоненти, які згодом деталізуються на більш прості елементи.

*2. Архітектура, орієнтована на дані*: Це підхід, де великий акцент робиться на структуруванні даних і їх взаємозв'язку, що дозволяє ефективно організувати інформацію для подальшого використання.

*3. Об'єктно-орієнтований аналіз та проектування*: Це методологія, яка базується на визначенні об'єктів, які існують у реальному світі, та їх взаємодії для створення програмних систем.

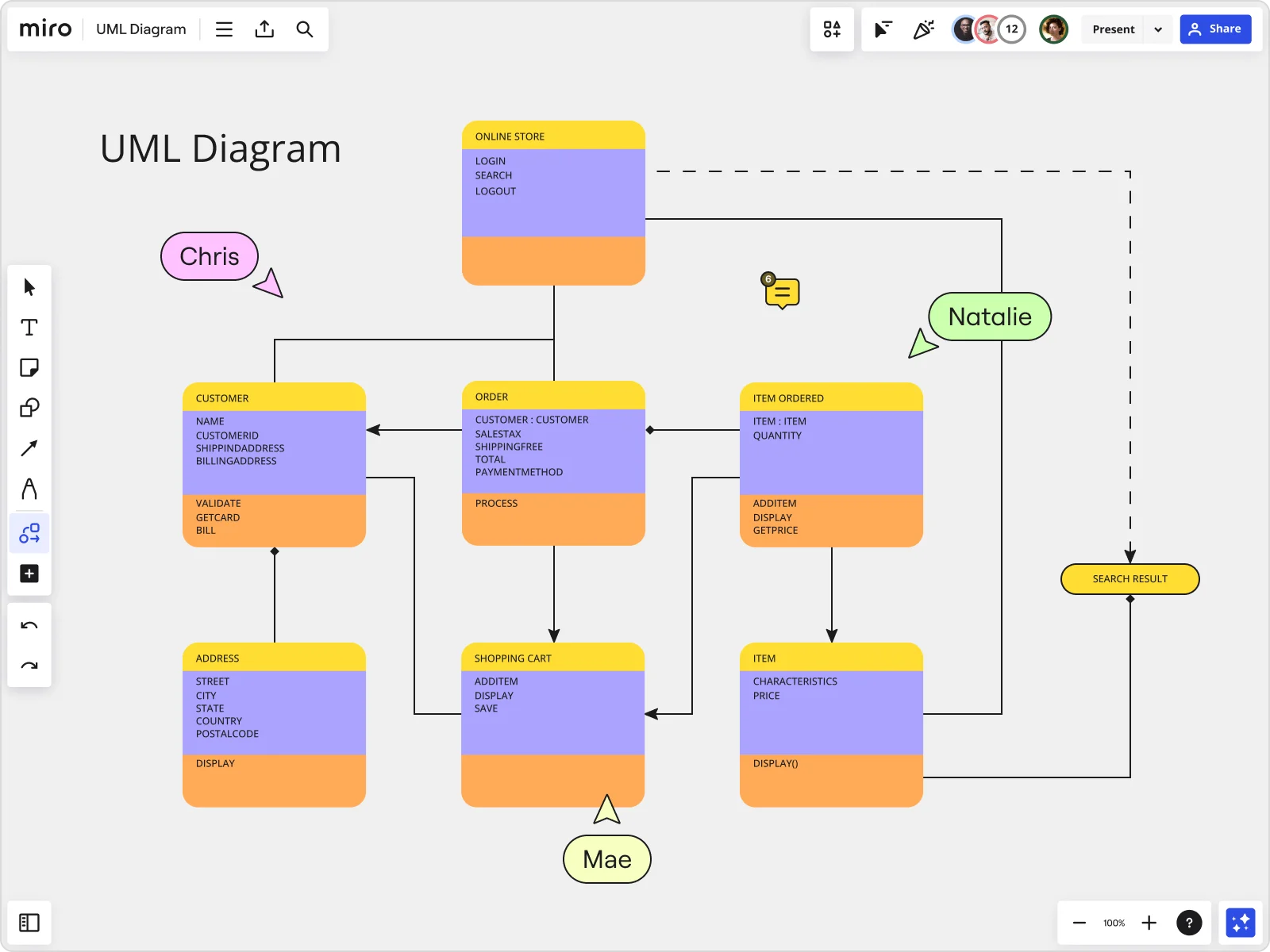
*4. Подієво-керована архітектура*: Це архітектурний підхід, де система реагує на події, які виникають в середовищі, і відповідає на них відповідним чином, забезпечуючи таким чином більш гнучку та реактивну поведінку.

**5.3 Ідентифікація класів предметної області. UML- діаграми ієрархії класів: моделювання підсистем,**

**класів та зв’язків між ними.**

*Ідентифікація класів предметної області* - це процес визначення основних об'єктів або сутностей, які існують у системі, та їх характеристик. У реальному світі це можуть бути такі речі, як користувачі, товари, замовлення тощо.

*UML-діаграми* ієрархії класів використовуються для візуалізації структури класів у системі, їхніх взаємозв'язків та спадкування. Моделювання підсистем, класів та зв’язків між ними дозволяє розкрити архітектурну структуру програмної системи на високому рівні. Підсистеми представляють собою логічні групи класів, які виконують певні функції або обслуговують певні частини системи. Класи представляють об'єкти, які мають спільні властивості та поведінку, а зв’язки між класами показують взаємодію між об'єктами.

**5.4 Проектування сценаріїв реалізації варіантів використання на основі UML-діаграм послідовностей та комунікації.**

Проектування сценаріїв реалізації варіантів використання на основі UML-діаграм послідовностей та комунікації - це процес визначення та візуалізація послідовностей дій, які відбуваються між об'єктами системи під час виконання певного варіанту використання.

UML-діаграми послідовностей дозволяють показати порядок виконання дій між об'єктами системи в часі, зокрема, які методи об'єктів викликаються та в якій послідовності. Комунікаційні діаграми, з іншого боку, показують взаємодію між об'єктами в рамках певного сценарію використання, уточнюючи деталі про передачу повідомлень та відповіді між ними.

Іншими словами, це метод розробки програмних систем, який дозволяє визначити, як саме об'єкти системи взаємодіють між собою для досягнення конкретної функціональності.

**5.5 Основні патерни проектування:**

**MVC,**

**Abstract Factory,**

**Facade,**

**Decorator,**

**Flyweight,**

**Visitor,**

**Observer,**

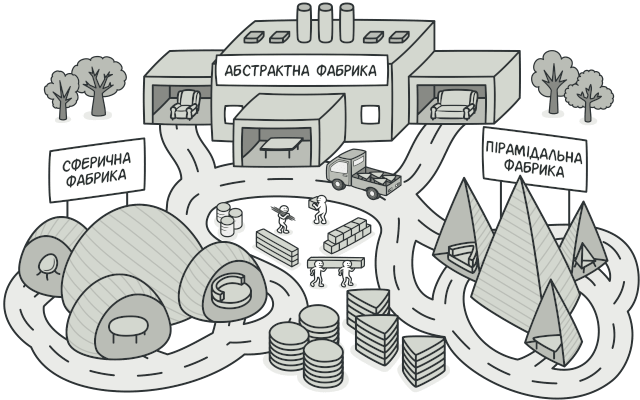
**Proxy,**

**Strategy,**

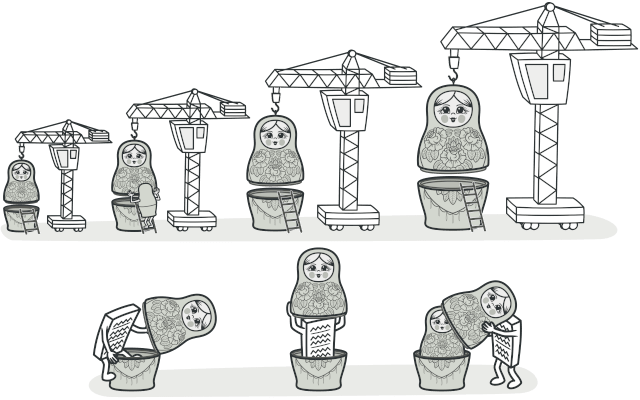
**Chain of Responsibility.**

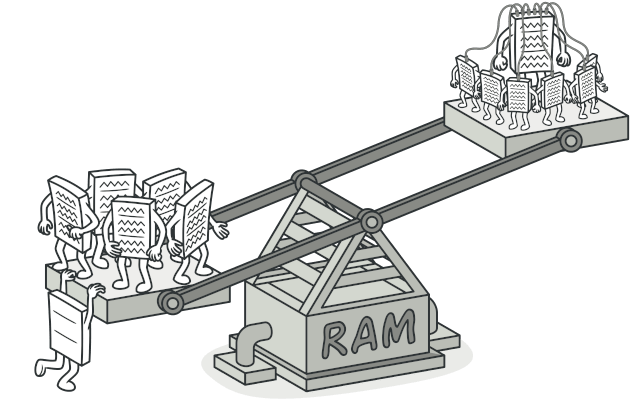
*1. MVC (Model-View-Controller)*: Розділення програми на три компоненти - модель (дані), представлення (відображення) та контролер (логіка управління), що дозволяє покращити модульність та обслуговуваність програмного забезпечення.

Цей шаблон передбачає поділ системи на три взаємопов'язані частини: модель даних, вигляд (інтерфейс користувача) та модуль керування. Застосовується для відокремлення даних (моделі) від інтерфейсу користувача (вигляду) так, щоб зміни інтерфейсу користувача мінімально впливали на роботу з даними, а зміни в моделі даних могли здійснюватися без змін інтерфейсу користувача.

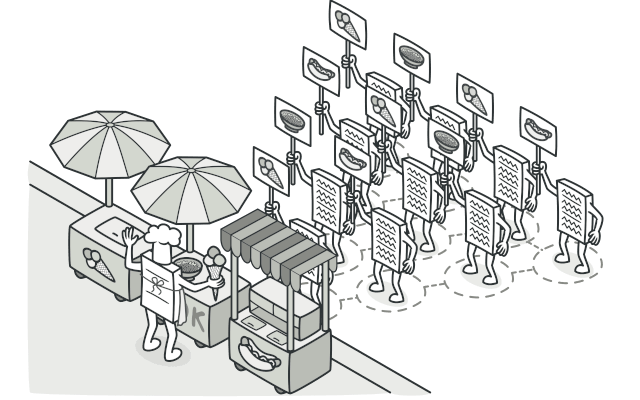
*2. Abstract Factory*: Паттерн, який надає інтерфейс для створення сімейств пов'язаних або залежних об'єктів без вказівки їх конкретних класів.

*3. Facade*: Паттерн, який надає простий інтерфейс до складного підсистеми, що допомагає спростити взаємодію з нею.

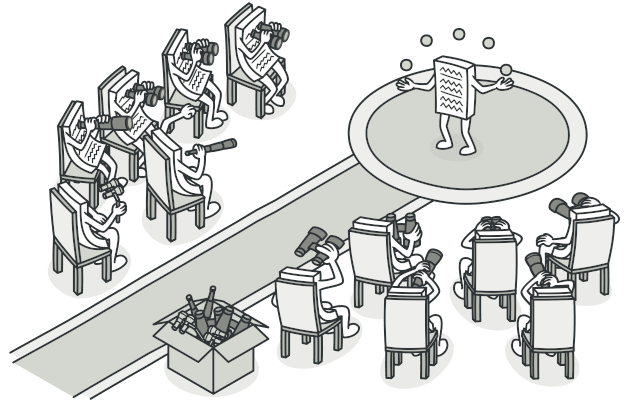
*4. Decorator*: Паттерн, який дозволяє додавати новий функціонал об'єктам, не змінюючи їхнього основного класу.



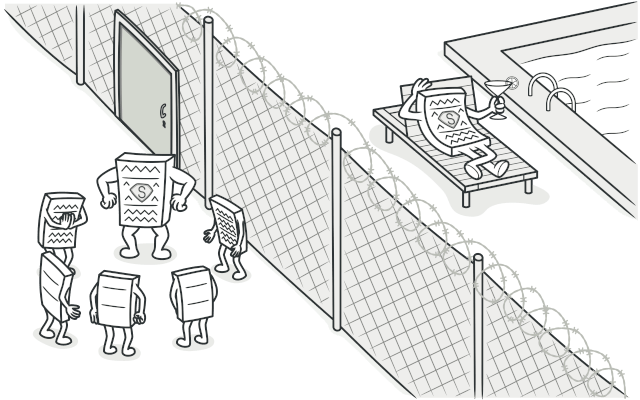
*5. Flyweight*: Паттерн, який дозволяє ефективно використовувати пам'ять шляхом зменшення кількості об'єктів у системі.

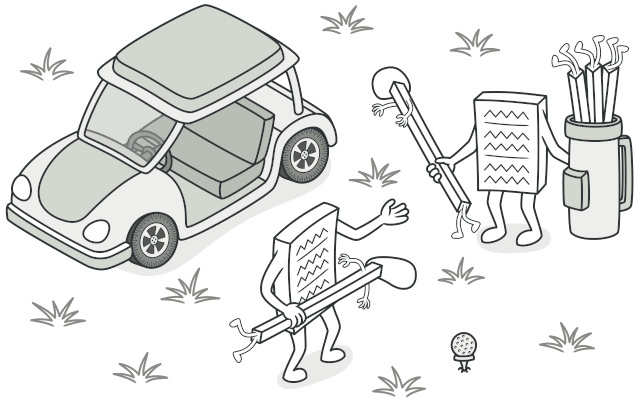


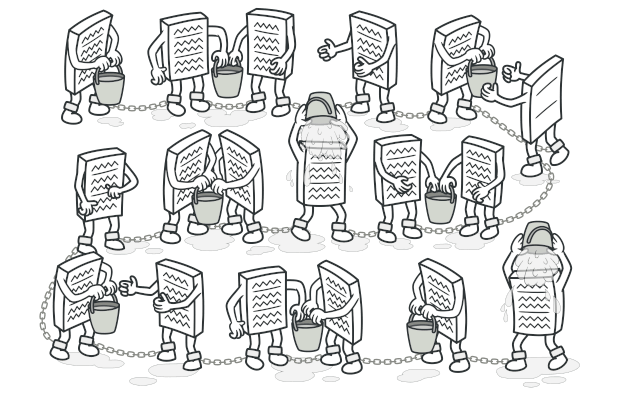
*6. Visitor*: Паттерн, який дозволяє додавати нові операції до об'єктів без зміни їхньої структури.



*7. Observer*: Паттерн, що визначає залежність одного або більше об'єктів від стану іншого об'єкта та автоматично повідомляє їх про будь-які зміни.

*8. Proxy*: Паттерн, який дозволяє контролювати доступ до об'єкта, додаючи додатковий рівень індирекції.

*9. Strategy*: Паттерн, що дозволяє обирати один з декількох алгоритмів або стратегій під час виконання  програми.



*10. Chain of Responsibility*: Паттерн, що дозволяє передавати запити вздовж ланцюжка обробників, кожен з яких може обробляти запит або передавати його далі по ланцюжку.

**6. Реалізація програмного забезпечення**

**6.1 Вимоги до оформлення коду: стиль, розбиття на структуровані одиниці, найменування змінних,**

**класів, об’єктів.**

"Вимоги до оформлення коду" — це набір правил і рекомендацій, що визначає, як код повинен бути структурований та оформлений. Це включає в себе стиль програмування (наприклад, використання відступів, регулярних пробілів тощо), розбиття коду на структуровані одиниці (наприклад, функції, класи), а також чітке і консистентне найменування змінних, класів і об'єктів. Виконання цих вимог допомагає підтримувати код зрозумілим, читабельним і легко змінюваним, сприяє його якості та довготривалій підтримці.

**6.2 Засоби автоматичної генерації програмного коду.**

"Засоби автоматичної генерації програмного коду" це інструменти, які допомагають створювати частини програмного коду автоматично, на основі певних шаблонів, правил або конфігурацій. Ці засоби можуть використовуватися для швидкої генерації шаблонних фрагментів коду, автоматичного створення коду на основі моделей даних або специфікацій, а також для генерації тестових скриптів або документації. Вони дозволяють прискорити процес розробки, зменшити кількість ручно написаного коду і зменшити його вразливість до помилок.

Засоби автоматичної генерації програмного коду можуть бути використані у різних областях програмування. Ось кілька прикладів:

1. Генератор коду для ORM (Об'єктно-реляційне відображення): Такі інструменти, як Django ORM в Python або Entity Framework в .NET, можуть автоматично генерувати класи, які відображають таблиці бази даних в об'єкти програмного коду і навпаки.

2. Шаблонізатори коду: У багатьох мовах програмування існують інструменти, які дозволяють генерувати шаблонні фрагменти коду, наприклад, використовуючи шаблони Mustache або Jinja в Python.

3. Конструктори коду для інтерфейсів: Сучасні IDE (середовища розробки), такі як Visual Studio або IntelliJ IDEA, можуть автоматично створювати каркаси коду для різних типів файлів, таких як класи, інтерфейси, тести тощо, що відповідають вибраним шаблонам.

4. Генератори тестів: Інструменти, як PyTest для Python або JUnit для Java, можуть автоматично генерувати тестові скрипти на основі анотацій або специфікацій коду.

5. Фреймворки для генерації документації: Інструменти, такі як Swagger для API або Sphinx для Python, можуть автоматично генерувати документацію на основі анотацій або коментарів у програмному коді.

**6.3 Налагодження: точки зупинки (Breakpoints), спостереження за змінними (Variable Watch),**

**виведення на консоль (Console Output), налагоджувач (Debugger), аналізатори коду (Code Analyzers).**

"Налагодження" в програмуванні - це процес виявлення та виправлення помилок у програмному коді. Ось коротка інформація про різні аспекти налагодження:

1. Точки зупинки (Breakpoints): Це місця у програмному коді, де ви хочете зупинити виконання програми для аналізу стану системи. Коли досягається точка зупинки, виконання програми призупиняється, і ви можете перевірити значення змінних, стек викликів тощо.

2. Спостереження за змінними (Variable Watch): Це можливість стежити за значеннями конкретних змінних під час виконання програми. Ви можете встановити спостереження за певними змінними і переглядати їх значення на кожному кроці виконання.

3. Виведення на консоль (Console Output): Це метод відображення повідомлень або значень змінних на консолі під час виконання програми. Ви можете використовувати це для відладки, виводячи важливі значення змінних або повідомлення про поточний стан програми.

4. Налагоджувач (Debugger): Це інструмент, який дозволяє контролювати виконання програми, дозволяючи вам встановлювати точки зупинки, спостерігати за змінними, виконувати програму по кроці тощо. Він забезпечує інтерактивний підхід до відладки.

5. Аналізатори коду (Code Analyzers): Це інструменти, які автоматично аналізують програмний код для виявлення потенційних помилок або недоліків. Вони можуть виявляти проблеми зі стилем коду, недоліки безпеки, потенційні помилки у виконанні тощо.

**6.4 Керування конфігурацією та версіями програмного забезпечення.**

"Керування конфігурацією та версіями програмного забезпечення" - це процес відстеження та контролю за змінами в програмному коді, документації, налаштуваннях та інших артефактах розробки протягом їх життєвого циклу. Основні аспекти цього процесу включають:

1. Керування версіями: Це відстеження та контроль за різними версіями програмного забезпечення. Зазвичай це виконується за допомогою систем керування версіями, таких як Git, SVN або Mercurial.

2. Керування залежностями: Це керування залежностями програмного забезпечення, які використовуються у вашому проекті. Це може включати в себе відстеження версій залежностей та автоматизоване вирішення конфліктів між ними.

3. Керування конфігурацією: Це забезпечення стабільності конфігурацій програмного забезпечення на різних етапах розробки та впровадження. Це може включати в себе управління налаштуваннями середовищ, конфігураційними файлами тощо.

4. Автоматизація збирання та розгортання: Це використання інструментів автоматизації, таких як Jenkins, Ansible або Docker, для автоматизації процесів збирання, тестування та розгортання програмного забезпечення з різними версіями та конфігураціями.

5. Аудит і відстеження змін: Це відстеження та аудит змін, які вносяться до програмного коду та інших артефактів розробки, щоб забезпечити відповідність процесу розробки стандартам та вимогам.

**6.5 Постійна інтеграція/постійне впровадження (Continuous Integration/Continuous Delivery).**

Постійна інтеграція (CI) та постійне впровадження (CD) - це практики розробки програмного забезпечення, спрямовані на автоматизацію та полегшення процесу розробки, тестування та розгортання програмного забезпечення. Основні принципи цих практик включають:

1. Постійна інтеграція (CI): Це практика, в рамках якої розробники регулярно об'єднують свій код у спільний репозиторій, після чого автоматично проводяться процеси збирання, тестування та перевірки якості. Мета - виявлення і виправлення помилок якнайшвидше.

2. Постійне впровадження (CD): Це практика, в рамках якої зміни в програмному коді автоматично впроваджуються в реальне середовище безпеки, тестування та виробництва. Мета - забезпечення швидкого, автоматизованого та безпечного впровадження змін.

Разом, CI/CD створюють автоматизований шлях від написання коду до його впровадження в продуктивному середовищі, допомагаючи зменшити час циклу розробки, полегшити процес тестування та забезпечити стабільність та надійність програмного забезпечення.

**7. Забезпечення якості: спільне та відмінності процесів тестування, верифікації, валідації**

**7.1 Тестування методами білої та чорної скрині.**

Тестування методами білої та чорної скрині - це два підходи до тестування програмного забезпечення, які використовуються для виявлення помилок і перевірки функціональності системи.

1. Тестування білої скрині (White Box Testing): Цей метод тестування включає в себе аналіз внутрішньої структури програмного коду. Тестувальник має доступ до джерелного коду програми і виконує тести, використовуючи знання про внутрішні алгоритми, структури даних та логіку програми. Мета - перевірити правильність роботи окремих функцій та модулів програми.

2. Тестування чорної скрині (Black Box Testing): У цьому методі тестування тестувальник не має доступу до внутрішньої структури програми і працює тільки зі специфікацією та зовнішнім інтерфейсом програми. Він виконує тести, не знаючи деталей реалізації, тільки за допомогою вхідних та вихідних даних. Мета - перевірити функціональність та здатність програми задовольняти вимоги користувача без урахування внутрішньої реалізації.

Обидва методи мають свої переваги та недоліки і використовуються в залежності від потреб проекту та специфіки програмного забезпечення.

**7.2 Рівні тестування: модульний, системний, валідаційний.**

Рівні тестування - це різні етапи та підходи до процесу тестування програмного забезпечення, які використовуються для виявлення та виправлення помилок на різних рівнях системи. Основні рівні тестування включають:

1. Модульне тестування (Unit Testing): Це перевірка окремих модулів або функцій програмного коду на коректність їх роботи. Тестувальники пишуть та виконують тести для кожного модуля незалежно від інших частин системи.

2. Системне тестування (System Testing): Це тестування всієї системи як цілісної одиниці, коли всі модулі об'єднуються та перевіряються на відповідність вимогам та очікуванням користувача. Мета - впевнитися, що система в цілому працює правильно.

3. Валідаційне тестування (Validation Testing): Це останній етап тестування, де перевіряється, чи відповідає розроблене програмне забезпечення специфікаціям та потребам користувачів. Валідаційні тести зазвичай виконуються на реальних або схожих з реальними даними та умовами використання.

Ці різні рівні тестування використовуються в комплексі для забезпечення якості програмного забезпечення та виявлення помилок на різних етапах його розробки та впровадження.

**7.3 Розробка через тестування (Test-driven-development).**

Розробка через тестування (Test-driven development, TDD) - це методологія розробки програмного забезпечення, в якій розробники спочатку пишуть тести для очікуваної функціональності, а потім пишуть код, який проходить ці тести.

Основні етапи процесу розробки через тестування включають:

1. Написання тесту: Розробники спочатку створюють тест, який описує очікувану поведінку частини програмного коду.

2. Виконання тесту: Тест запускається, і, оскільки ще немає коду, який би реалізував вимоги тесту, він провалюється.

3. Написання коду: Розробники пишуть код, який вирішує проблему, викликану тестом.

4. Покращення коду: Після написання коду тест запускається знову. Якщо він успішний, це означає, що новий код успішно проходить тест.

5. Рефакторинг коду: Після успішного пройдення тестів розробники можуть відредагувати код, щоб покращити його структуру, продуктивність або читабельність.

Таким чином, TDD дозволяє розробникам зосередитися на функціональності, що вирішується, та гарантує, що весь написаний код має покриття тестами. Це сприяє покращенню якості програмного забезпечення, зменшенню кількості помилок та полегшенню процесу рефакторингу.

**7.4 Додаткові техніки верифікації та валідації:**

**інспекція коду,**

**перевірка на відповідність стандартам і вимогам,**

**оцінювання зручності використання та користувацького досвіду,**

**перевірка продуктивності та масштабованості.**

Додаткові техніки верифікації та валідації допомагають перевірити програмне забезпечення на відповідність вимогам, стандартам та очікуванням користувачів. Ось коротка інформація про кожну з них:

1. Інспекція коду: Це метод, в рамках якого команда програмістів аналізує програмний код для виявлення потенційних помилок, відхилень від стандартів коду та можливих покращень.

2. Перевірка на відповідність стандартам і вимогам: Це перевірка програмного забезпечення на відповідність встановленим стандартам, наприклад, вимогам безпеки, регуляторним вимогам або іншим внутрішнім або зовнішнім стандартам.

3. Оцінювання зручності використання та користувацького досвіду: Це перевірка того, наскільки легко та зручно користуватися програмним забезпеченням для кінцевих користувачів. Вона включає в себе тестування інтерфейсу користувача, навігації та загального враження від використання програми.

4. Перевірка продуктивності та масштабованості: Це тестування програмного забезпечення для оцінки його продуктивності, швидкодії та можливостей масштабування при обробці великого обсягу даних або при високому навантаженні.

Ці техніки допомагають забезпечити якість та надійність програмного забезпечення, а також забезпечують задоволення від його використання для кінцевих користувачів.

**8. Командна робота, підходи до розробки програмного забезпечення (ПЗ)**

**8.1 Класичні моделі розробки ПЗ: каскадна (водоспадна), ітераційна, інкрементна.**

Класичні моделі розробки програмного забезпечення (ПЗ) - це різні підходи до організації та керування процесом розробки ПЗ. Ось короткий огляд кожної з цих моделей:

1. Каскадна (Waterfall) модель: Це лінійний процес розробки, в якому кожен етап розробки відбувається послідовно, починаючи з визначення вимог і закінчуючи реалізацією, тестуванням та впровадженням. Кожен етап включає в себе створення відповідних документів. Ця модель підходить для проектів з чітко визначеними вимогами і функціональністю.

2. Ітераційна модель: Це модель, в якій процес розробки розділяється на ітерації або цикли. Кожен цикл включає в себе фази розробки, такі як аналіз, проектування, реалізація та тестування, і завершується готовим до використання фрагментом програмного забезпечення. Кожна наступна ітерація вдосконалює функціональність системи.

3. Інкрементна модель: Це модель, в якій система розробляється шляхом послідовного додавання функціональних блоків або інкрементів до базового продукту. Кожен інкремент додає новий функціонал, що покращує здатність системи до виконання її функцій. Ця модель часто використовується для швидкої поставки мінімально необхідного функціоналу.

Кожна з цих моделей має свої переваги та недоліки і може бути використана в залежності від потреб проекту та його характеристик.

**8.2 Промислові технології розробки ПЗ: RUP, MSF, Agile, Scrum, Extreme Programming (XP), Kanban.**

Промислові технології розробки програмного забезпечення - це набір методологій, рамок та підходів, які використовуються для організації та керування процесом розробки ПЗ. Ось короткий огляд деяких з них:

1. RUP (Rational Unified Process): Це циклічний процес розробки, який акцентує на роботі з ризиками та важливістю визначення вимог. RUP включає в себе чотири фази: визначення, проектування, реалізація та тестування.

2. MSF (Microsoft Solutions Framework): Це набір практик та методів розробки програмного забезпечення, розроблений компанією Microsoft. Він орієнтований на розробку та впровадження корпоративних програмних продуктів.

3. Agile: Це набір принципів та цінностей, які ставлять перед собою мету створити гнучку та адаптивну культуру розробки. Agile надає перевагу інтерактивності, співпраці з клієнтом та здатності швидко реагувати на зміни.

4. Scrum: Це ітеративний фреймворк для керування проектами, який надає структуру для роботи в команді, планування ітерацій та взаємодії зі зацікавленими сторонами.

5. Extreme Programming (XP): Це методологія розробки програмного забезпечення, яка покликана покращити якість програмного коду та сприяти швидкому впровадженню змін. XP акцентує на комунікації, спільному власністю коду та тестуванні.

6. Kanban: Це методологія, яка спрямована на оптимізацію потоку роботи. Вона базується на візуальному представленні завдань та використанні обмеження робочих процесів для досягнення більшої ефективності.

Ці технології використовуються в індустрії розробки ПЗ для полегшення та оптимізації процесів розробки, а також для досягнення більшої ефективності та якості у продукті.

**8.3 Ролі та обов'язки у команді проекту, переваги командної роботи, ризики та складність такої співпраці.**

У команді проекту кожен має свої ролі та обов'язки, спрямовані на досягнення загальної мети. Керівник визначає напрямок, аналізує результати та координує дії. Спеціалісти виконують конкретні завдання згідно своєї експертизи. Переваги командної роботи включають різноманітність підходів, швидкість вирішення завдань та підтримку з колег. Однак, існують ризики, такі як непорозуміння, конфлікти та втрата спрямованості, які можуть ускладнити співпрацю та затягнути виконання проекту.

**8.4 Основні етапи планування і виконання IT проекту. Життєвий цикл IT проекту.**

Основні етапи планування і виконання IT проекту включають:

1. Ініціація: Визначення потреби в проекті, встановлення мети та обґрунтування його доцільності.

2. Планування: Визначення обсягу, складання графіка, розподіл ресурсів, оцінка ризиків і складання плану дій.

3. Виконання: Реалізація плану, розробка програмного забезпечення, тестування, впровадження.

4. Контроль і моніторинг: Стеження за прогресом проекту, виявлення відхилень від плану та прийняття корективних заходів.

5. Завершення: Передача результатів замовнику, оцінка ефективності, архівування даних та звітність.

Життєвий цикл IT проекту відображає послідовність фаз, через які проект пройшов від початку до завершення, включаючи планування, розробку, впровадження та підтримку. Цей цикл може варіюватися в залежності від методології управління проектами та конкретних вимог проекту.