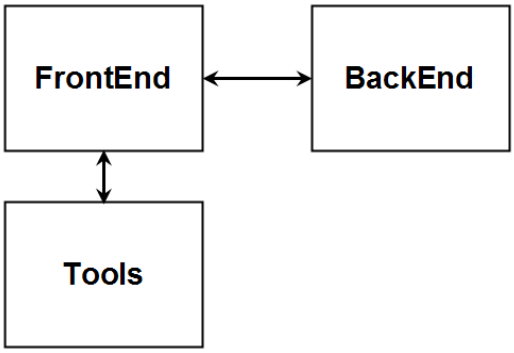
Лекція 5

Архітектура і проектування

***Анотація**: Поняття архітектури ПЗ. Точка зору і характеристики точок зору. Множинність точок зору при розробці ПЗ.*

Обговорення

Якось один менеджер пояснював основні ідеї свого достатньо крупного проекту, яким він керував. Він накреслив на дошці три кубики: ***frontend***, ***backend***, ***tools***. І сказав, що це і є головна структура проекту. І в сенсі внутрішнього устрою продукту, і в сенсі розподілу робіт в команді за трьома дистанційно рознесеним центрам розробки. Завдання backend складні, ресурсоємні, виконуються пакетно. Вони відокремлені від графічного інтерфейсу продукту (frontend), який також досить не простий. Frontend – це призначений для користувача інтерфейс: складний, такий, що параметризується, з вбудованими сервісами користувача (зокрема, браузером інформації), з можливістю персонального налагоджування. Обидві ці підсистеми взаємодіють одна з однією через добре і детально описаний програмний інтерфейс: алгоритми ***backend*** розбиті на методи, які ***frontend*** може викликати з параметрами за допомогою особливих правил, об'єднати в ланцюжок для досягнення своїх завдань. "Збоку" від всього цього знаходяться додаткові засоби – ***tools***. Вони інтегруються у frontend, не користуються методами backend, та реалізують свої завдання самостійно. Ці завдання не вимагають складної пакетної обробки, а націлені на інтерактивну взаємодію з користувачем. Для їх реалізації особливо багато уваги приділялося usability (зручності і простоті використання).



Кожна з трьох підсистем вимагала від розробників особливих навиків. У разі backend це було уміння і досвід з реалізації такого роду пакетних алгоритмів, у випадку з frontend – уміння створювати складний призначений для користувача інтерфейс, у випадку з tools вимагалось мистецтво в проектуванні і реалізації інструментів, які надають користувачам системи додаткові сервісні можливості. Для того, щоб розділити роботи таким чином, були ще і політичні аспекти. Зокрема, керівництво проекту хотіло мати процес розробки призначеного для користувача інтерфейсу поруч з собою, в одному з трьох центрів розробки, який територіально співпадав з штаб-квартирою. Вважалося, що зовнішній вигляд продукту дуже важливий для його успішного продажу і вимагає особливої уваги.

В результаті виконання проекту (а він розвивався більше 15 років, досягаючи в апогеї до 150 чоловік, які одночасно були зайняті в ньому) така чітка структура дещо змістилася: так географічно *інтерфейс* майже "переїхав" в той центр, де розроблявся backend. Але в цілому такий розподіл проекту на частини залишався багато років і був основним скелетом всієї розробки. Це і є приклад архітектури програмного проекту.

Визначення

Під **архітектурою ПЗ** слід розуміти внутрішню структуру продукту (компоненти і їх зв'язки), основи призначеного для користувача інтерфейсу продукту, а також квінтесенцію знань і рішень, що є інструментом розробки і *управління проектом*. Тобто *архітектура* – це крізна концепція або набір концепцій для подолання ентропії і хаосу, які прагнуть "проковтнути" розробку з причини складності, нематеріальності, погоджуваності і мінливості *ПЗ.* При цьому не слід розділяти продукт і проект, оскільки на практиці це, як правило, одне ціле, причому це "поєднання", якщо воно існує, є "сильною" стороною даної розробки.

Часто під архітектурою розуміють, наприклад, лише внутрішній устрій *ПЗ,* який виражений в *UML-діаграмах.* Ось жарт на тему того, що архітектуру не можна розуміти односторонньо. Одного з відомих розробників трансляторів запитали, чому в його відомому трансляторі рівно 21 відображення. Чекали почути *перерахування алгоритмічних* проблем, які у такий спосіб вдалося подолати, щось про особливу *ефективність алгоритмів*, які були організовані таким чином, і так

далі. Всіх здивувала відповідь метра. Він сказав, що саме стільки людей (тобто рівно 21), було у нього в команді розробників...

Отже, *архітектура* продукту є *інваріантом* проекту. Вона зустрічається і несподівано виникає в різних його частинах. Це і є аналогом "простих" природно-наукових постулатів і законів, відсутність яких в розробці *ПЗ, на думку* Фредеріка Брукса (***Фредерік Філіпс Брукс (Frederick Phillips Brooks, Jr.; народився 19 квітня, 1931) — інженер програмного забезпечення та вчений-інформатик, найбільш відомий за управління розробкою ОС System/360 для IBM, та її наступників, а пізніше за опис цього процесу в книжці Міфічний людино-місяць. Брукс отримав за своє життя багато нагород, включаючи Національну медаль технологій та інновацій США у 1985, та премію Тюрінга у 1999***), є причиною складності ПЗ (у сенсі хаосу, тобто "поганої" складності). Створювати такі структури – непроста справа, що вимагає великого мистецтва. Але саме це є *шляхом* до управління хаосом та зростаючою *ентропією,* у вигляді вимог до системи, які змінюються, для запобігання втрати розробниками ясного розуміння, яку ж саме систему вони створюють. І саме розробка таких структур надає *дійсно* творчу насолоду підчас розробки програмних систем. Добре "працюють" прості моделі, які не легко створювати. Такі моделі надають розробнику можливість розповідати про проект дуже довго, красиво його оформити та повісити на стінку.

В межах багатьох проектів не створюється оригінальна архітектура, оскільки вони є типовими і/або невеликими та ґрунтуються на готових технологіях, архітектурних зразках, моделях команди і оргструктурах проектів.

Проте, часто перед колективами, які добре себе зарекомендували в таких проектах, виникає завдання побудувати дійсно оригінальну нову архітектуру, що ґрунтується на колишніх розробках. Тут, перш за все, важливо відмітити цей перехід, усвідомити, що старі методи роботи не підходять і потрібний принципово новий *досвід*, якого, дуже часто, немає у колективу та його лідерів...

Множинність точок зору

Під час розробки архітектури ПЗ важливим є поєднання *безлічі* точок зору. ПЗ виявляється настільки складним, що його архітектуру

не можна побудувати як єдину модель: безліч окремих аспектів повинні бути представлені в архітектурі, їх зв'язки є складними і тому погано відображаються у явному вигляді. Тому корисним виявляється створення *безлічі* моделей, які спираються на різні точки зору.

Причина множинності точок зору підчас розробки ПЗ.

Уміння розглядати предмет з різних точок зору є найважливішою філософією успішної практики у разі роботи з великими об'ємами різнорідної і складної інформації. Розглянемо розробку ПЗ і те, чому там затребувані різні погляди на процес, систему і так далі.

Це відбувається, перш за все, через різні види діяльності процесу розробки ПЗ (див. **Рис.** 5.1). Під час складання функціональних *вимог* до ПЗ звертають увагу на те, яка саме функціональність повинна бути реалізованою, але при цьому опускаються принципи і деталі реалізації. На етапі проектування, навпаки, на перше місце виходять принципи реалізації ПЗ. А підчас тестування деталі реалізації знову неважливі – на ПЗ дивляться як на *чорний ящик*, що реалізовує (не важливо яким чином) деякий набір призначеної для користувача функціональності. На етапі розгортання ПЗ у замовника на ПЗ дивляться як на набір файлів, сховищ даних і так далі.

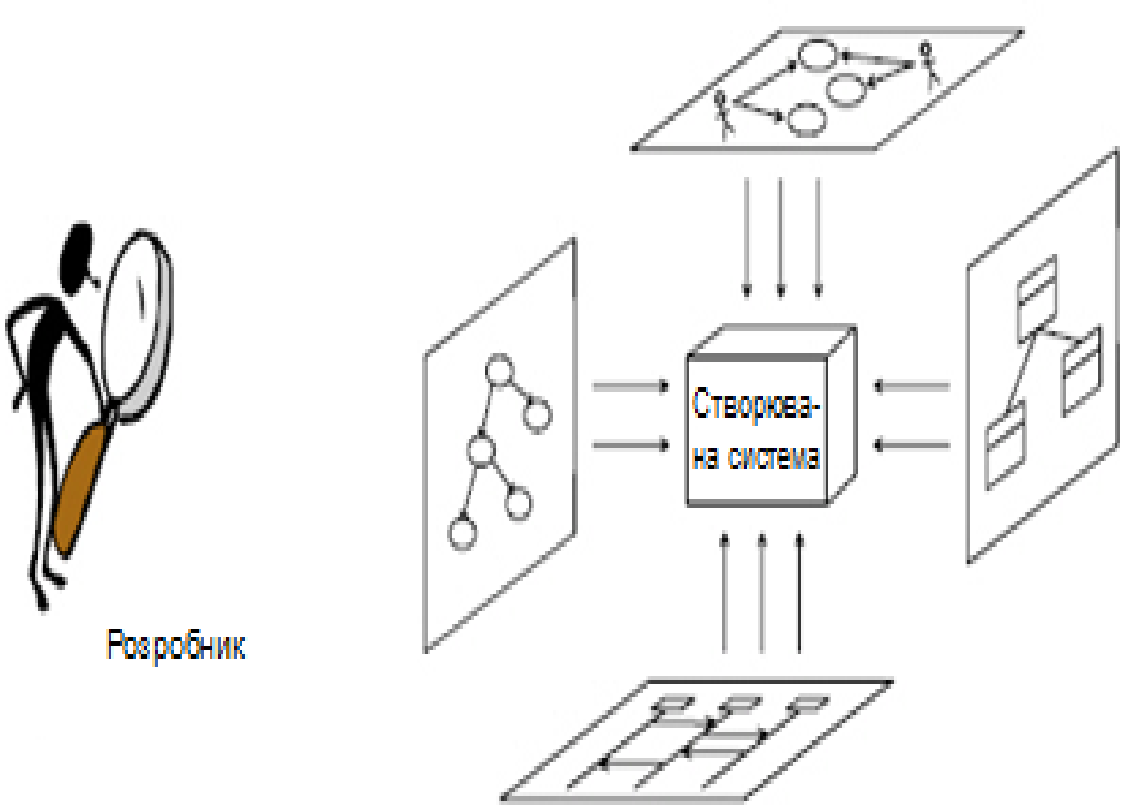


Рис. 5.1. Різні види діяльності – різні погляди на систему

Далі, у розробку/експлуатацію ПЗ залучено велика кількість дуже різних фахівців: програмісти, інженери, тестувальники, технічні письменники, менеджери, замовники, користувачі, продавці-маркетологи і так далі (див. **Рис.**5.2). Для всіх цих фахівців потрібна різна інформація про програмну систему. Уявіть, що відбудеться, якщо, наприклад, продавцеві або замовникові-непрограмістові у відповідь на прохання трохи краще ознайомитися з ПЗ ви дасте почитати програмний код.

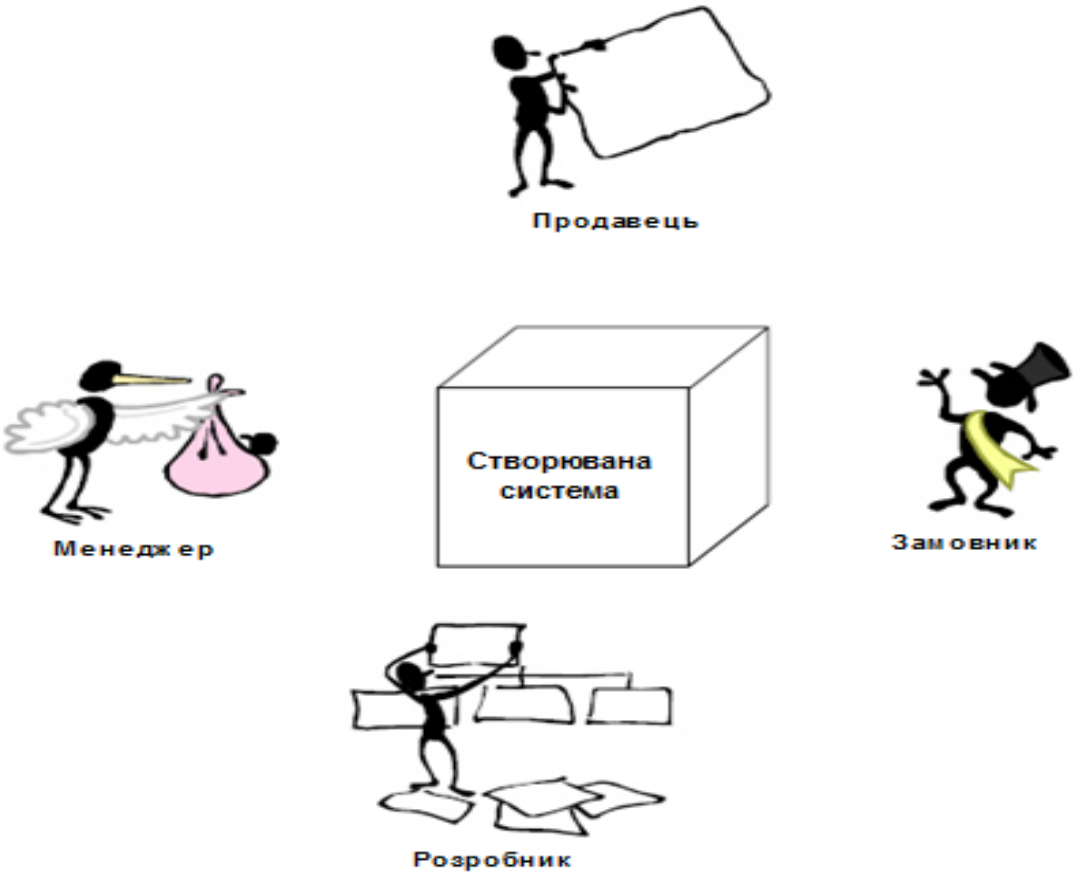


Рис. 5.2. Різні фахівці – різні погляди на систему

Множинність точок зору походить також від того, що немає єдиних стандартів і норм розробки ПЗ. Тобто розробка набагато у чому є "stateofart" («стан мистецтва»). Часто доводиться винаходити нову точку зору моделювання відповідно до ситуації – щоб саме цей експерт тебе зрозумів, щоб саме ці особливості системи були підкреслені. Часто тут – як в лотереї: створюється декілька описів системи з різних точок зору, який-небудь з описів виявляється вдалим і саме його надалі всі використовують.

Отже, різні види діяльності в процесі розробки ПЗ, різні категорії фахівців, що задіяні в програмному проекті, і унікальність кожної конкретної ситуації підчас розробки – все це призводить до створення і використання різних моделей, які виконані з різних точок зору.

Точка зору (***view point***) – ***це певний погляд на систему, який здійснюється для виконання якогось певного завдання ким-небудь з учасників проекту***. Точку зору слід ясно усвідомлювати на

етапі створення візуальних моделей, наприклад, *варіантів використання*. Важливо розуміти, що модель може бути у кожному конкретному випадку своя. Найважливішими характеристиками точки зору моделювання є **мета** (навіщо створюється модель) і **цільова аудиторія** (тобто, для кого вона призначається).

Важливим питанням, на яке потрібно чесно собі відповісти на самому початку моделювання, – це «Навіщо ви використовуєте діаграми?» (зокрема, UML). Це і є визначення мети моделювання. Бо так створювати моделі правильно? І всі проблеми (навіть ті, про які нічого ще не відомо) чарівним чином зникнуть? Дуже часто, наприклад, підчас створення моделі випадків використання присутня саме така "мета" моделювання. А потім виявляється, що жодні проблеми не "вилікувалися", а навпаки, виникли нові (наприклад, створені нами діаграми ніхто не розуміє і не приймає). Навіть сам аналітик відчуває, що діаграми вийшли якісь дивні.

А може все відбувається зовсім не так. Наприклад, аналітик дійсно задався метою виявити вимоги до системи – не нав'язати своє власне бачення іншим, а з'ясувати потрібну інформацію, змоделювати і викласти її доступно. Для цього він і використовує діаграми випадків використання. Йому важливо, щоб майбутні користувачі системи могли брати участь в цьому процесі, діаграми малюються для них, вони зрозумілі і не надмірні. І ці ж діаграми структурують і прояснюють інформацію для самого аналітика.

Подібних сюжетів на практиці відбувається безліч. Тут важливо розуміти, що мета моделі – це не якесь гіпотетичне завдання типу "опису архітектури, тому що так потрібно, так правильно", а цільова аудиторія – це не абстракція типу "люди, які хочуть ознайомитися з ПЗ". І те і інше – щось дуже конкретне, що реально існує в проекті або поряд з ним. Адже розробники ПЗ не можуть дозволити собі за гроші замовника створювати щось на всі століття і для всіх народів. І мета моделювання, і аудиторія, яка працюватиме з діаграмами, завжди існують, важливо лише ясно розуміти, які вони.

Існує корисний практичний прийом для орієнтації на цільову аудиторію, для якої призначена створювана вами модель. Можна вибрати одного представника такої аудиторії – конкретну і відому вам людину – і створювати діаграми, які зрозумілі саме йому. При цьому важливо не обговорювати надмірно з ним ваші моделі, оскільки це може створити додатковий контекст, якого інші користувачі моделей

будуть позбавлені. Корисно уявляти собі цю людину підчас роботи з моделями – його реакції, питання, здивування і ін. І, виходячи з цього, корегувати, виправляти створене. І, звичайно ж, корисно перевірити свої припущення, коли показуєш йому, що вийшло.

Крім того, важливо, щоб точка зору була "живою", а не вигадувалася аналітиком або бездумно копіювалася з книжок і присвячених UML-тренінгів. Непомітно для себе аналітик може придумати свій власний проект, своїх власних користувачів системи, замовника і так далі. Тобто аналітик поволі, нав'язує самому собі певне сприйняття реально існуючих людей, завдань, сильно спотворюючи реальне положення справ. І саме в контексті цієї уявної ситуації він створює свої моделі. Але ж реальні люди, реальні ситуації володіють своєрідністю, великим діапазоном варіативності. Відповідно, аналітик повинен володіти гнучкістю свідомості, великим діапазоном техніки, а також чуйністю і щирим прагненням до того, щоб зробити кожен конкретний проект, де він бере участь, більш гармонійним та адекватним.

Мова UML

Часто поняття архітектури сильно звужують, і розуміють під ним лише опис основних, важливих аспектів ПЗ, які створюються, наприклад, архітектором підчас розробки проекту системи. Для цих цілей використовується мова моделювання UML (Unified Modeling Language).

Ця мова є підсумком розвитку засобів схематичного опису програмних систем, які розвивалися з блок-схем, запропонованих ще Джоном фон Нейманом в кінці 40-х років (***Джон фон Нейман*** *(John von Neumann),* ***Нейман Янош Лайош*** *(угор. Neumann János Lajos),* ***Йоганн фон Нойман*** *(нім. Johann von Neumann) нар. 28 грудня 1903 — пом. 8 лютого 1957) — американський математик угорського походження, що зробив значний вклад у квантову фізику, функціональний аналіз, теорію множин, інформатику, економічні науки та в інші численні розділи знання. Він став засновником теорії ігор разом із Оскаром Морґенштерном у 1944 році. Розробив архітектуру (так звану «архітектуру фон Неймана»), яка використовується в усіх сучасних комп'ютерах.*). Він припускав, що ці схеми стануть високорівневою мовою введення алгоритмів в обчислювальні машини, але еволюція мов програмування пішла шляхом розвитку текстових мов. Проте, блок-схеми набули поширення

при специфікації і документуванні ПЗ, були стандартизовані, але широкого практичного застосування не отримали. В кінці 60-х років, у зв'язку з пошуком нових засобів розробки ПЗ, народженням програмної інженерії і загальним розвитком в галузі проектування і розробки штучних систем з'явився термін структурний аналіз (structure data analysis) систем. Термін був введений вченим з MIT, Дугласом Россом, який також запропонував діаграмний метод аналізу і проектування великих штучних систем (***Дуглас Тейлор «Дуги» Росс*** *(21 грудня 1929 - 31 грудень 2007) американський вчений піонер IT, голова SOFTECH Inc. Найбільш відомий як автор терміну CAD (САПР) для автоматизованого проектування, і вважається батьком APT (Automatically Programmed Tools – Засобів автоматичного програмування) мови для управлінням виробництвом з цифровим управлінням*). Метод отримав назву SADT (Structured Analisysand Design Technique), став основою серії військових стандартів США серії IDEF і широко розповсюдився в індустрії. Проте, діаграмна мова в SADT була дуже скромною – набір блоків і зв'язків між ними, з підтримкою декомпозиції блоків. У 70-х роках, у зв'язку з масовим виходом ПЗ на вільний ринок (тобто програмні системи почали створюватися не лише у військовій галузі, для крупного бізнесу, але також для середнього і малого бізнесу) структурний аналіз став бурхливо еволюціонувати – набір діаграм збагатився діаграмами станів і переходів, суть-зв'язків, потоків даних і так далі. З розвитком об’єктно-орієнованих засобів розробки (кінець 80-х – середина 90-х) структурний аналіз перетворився на об'єктно-орієнтований аналіз і проектування. З'явилася велика кількість методологій, поступово склалася єдина мова моделювання, яка і була закріплена в стандарті UML. Відбулося це в 1997 році.

З тих пір вийшло декілька версій стандарту UML. Поточна версія UML 2.1.

Види діаграм

"Скелетом" UML є діаграмна структура. Кожен вид діаграм є типом моделей, що реалізовує певну точку зору на програмну систему. Види діаграм не є строго обов'язковими в UML – їх можна перемішувати, створювати свої власні види діаграм. Проте, стандартні види діаграм є певним надбанням програмної інженерії, оскільки відображають досвід багатьох дослідників і практиків.

Структурні діаграми:

- **діаграми класів** (***class diagrams***) призначені для моделювання структури об'єктно-орієнтованих додатків класів, їх атрибутів і заголовків методів, спадкоємства, а також зв'язків класів один з одним;
- **діаграми компонент** (***component diagrams***) використовуються при моделюванні компонентної структури розподілених додатків; усередині кожна компонента може бути реалізована за допомогою безлічі класів;
- **діаграми об'єктів** (***object diagrams***) застосовуються для моделювання фрагментів працюючої системи і відображають саме ті фрагменти, що реально існують (runtime) в екземплярах класів, та значення їх атрибутів;
- **діаграми композитних структур** (***composite structure diagrams***) використовуються для моделювання складових структурних елементів моделей – кооперацій, композитних компонент і т.д.;
- **діаграми розгортання** (***deployment diagrams***) призначені для моделювання апаратної частини системи, з якою ПЗ безпосередньо зв'язано (розміщено або взаємодіє);
- **діаграми пакетів** (***package diagrams***) служать для розбиття об'ємних моделей на складові частини, а також (традиційно) для групування класів модельованого ПЗ, коли їх дуже багато.

Поведінкові діаграми:

- **діаграми активності** (***activity diagrams***) використовуються для специфікації бізнес-процесів, які ПЗ, що розробляється, повинно автоматизувати, а також для опису складних алгоритмів;
- **діаграми випадків використання** (***use case diagrams***) призначені для з’ясування і погодження вимог з замовником, користувачами і експертами наочної галузі;
- **діаграми кінцевих автоматів** (***state machine diagrams***) застосовуються для визначення поведінки реактивних систем;
- **діаграми взаємодій** (***interaction diagrams***):
    - **діаграми послідовностей** (***sequence diagrams***) використовуються для моделювання часових аспектів внутрішніх і зовнішніх протоколів ПЗ;
    - **діаграми схем взаємодії** (***interaction overview diagrams***) служать для організації ієрархії діаграм послідовностей;

- **діаграми комунікацій** (***communication diagrams***) є аналогом діаграм послідовностей, але іншим чином зображені (у звичній графовій манері);
- **часові діаграми** (***timing diagrams***) є різновидом діаграм послідовностей і дозволяють в наочній формі показувати внутрішню динаміку взаємодії деякого набору компонент системи.

Приклади.

Центральним видом діаграм є ***діаграми класів*** (Рис. 5.3).

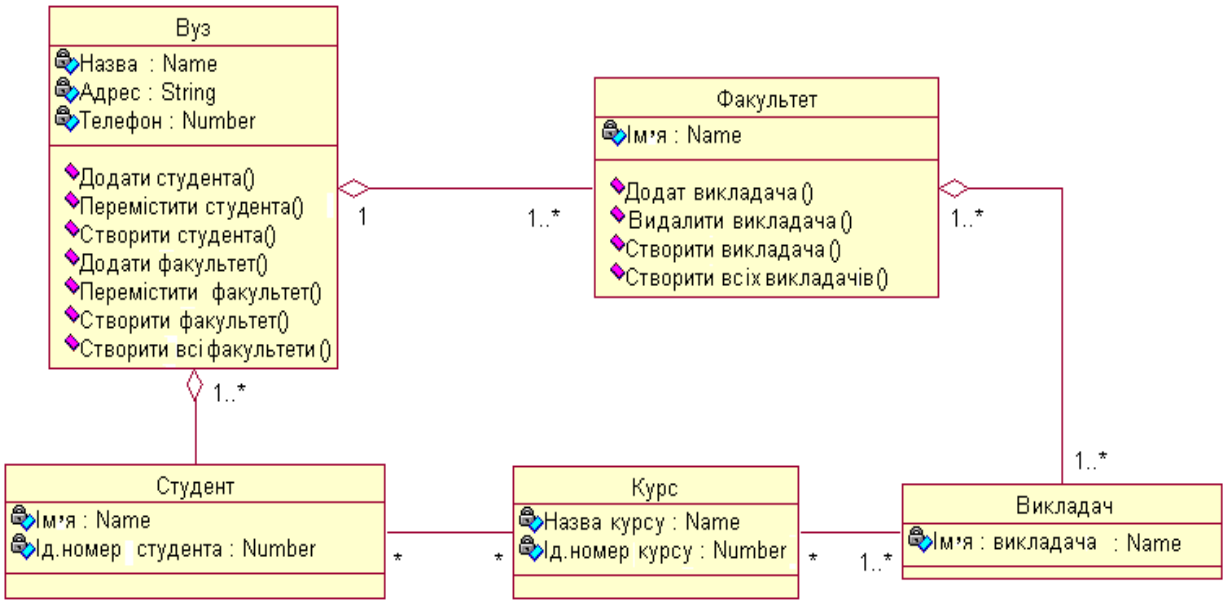


Рис. 5.3. Приклад діаграми класів

Ще один вид структурних діаграм – ***діаграми розгортання***, приклад наведений нижче.

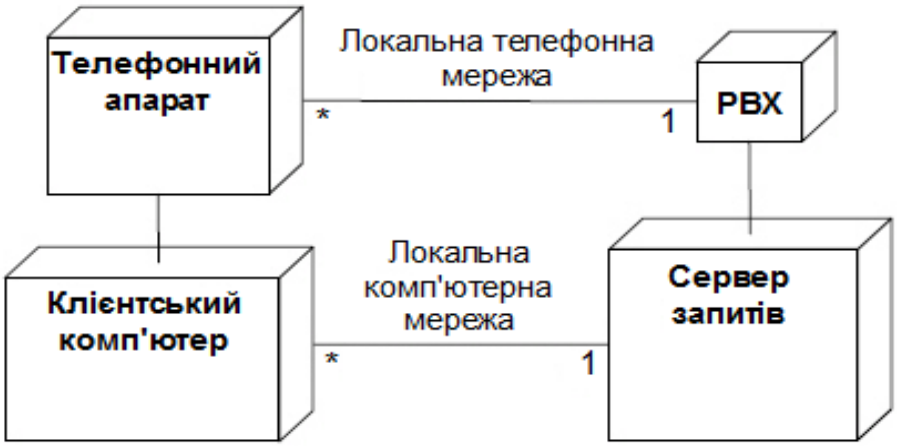


Рис. 5.4. Приклад діаграми розгортання

Відзначимо також ще один важливий вид діаграм UML – ***діаграми компонент*** (приклад на **Рис**. 5.5).

Цікавий також варіант діаграм композитних структур – складні компоненти для систем реального часу і телекомунікації. Приклад наведено на **Рис**. 5.6.

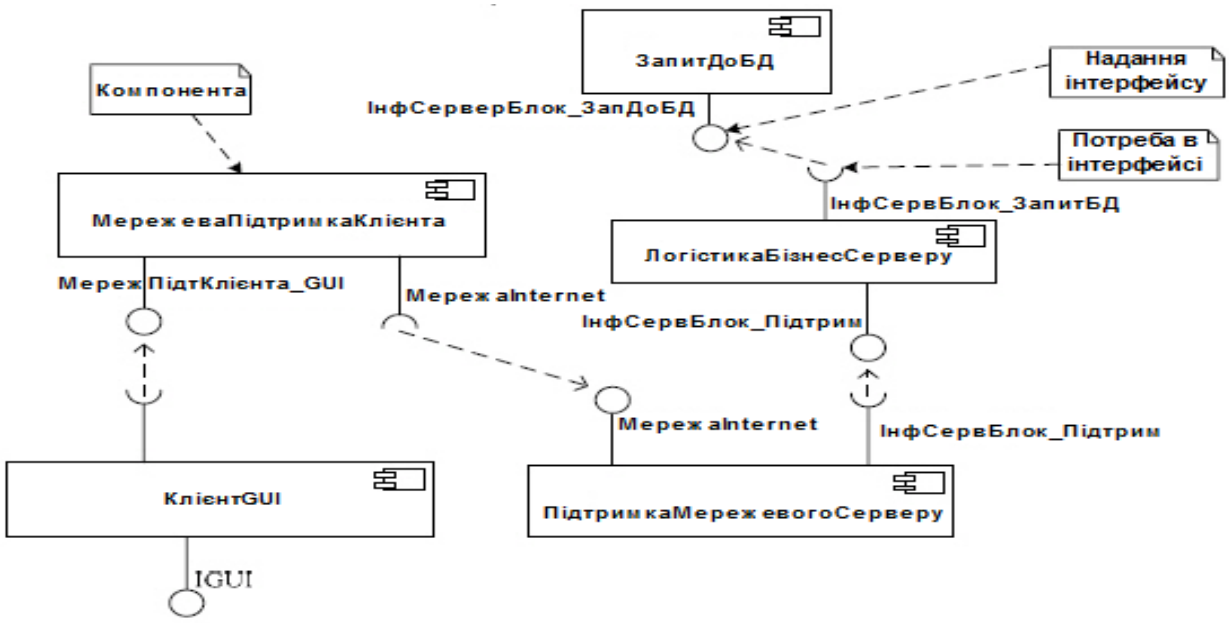


Рис. 5.5. Приклад діаграми компонент

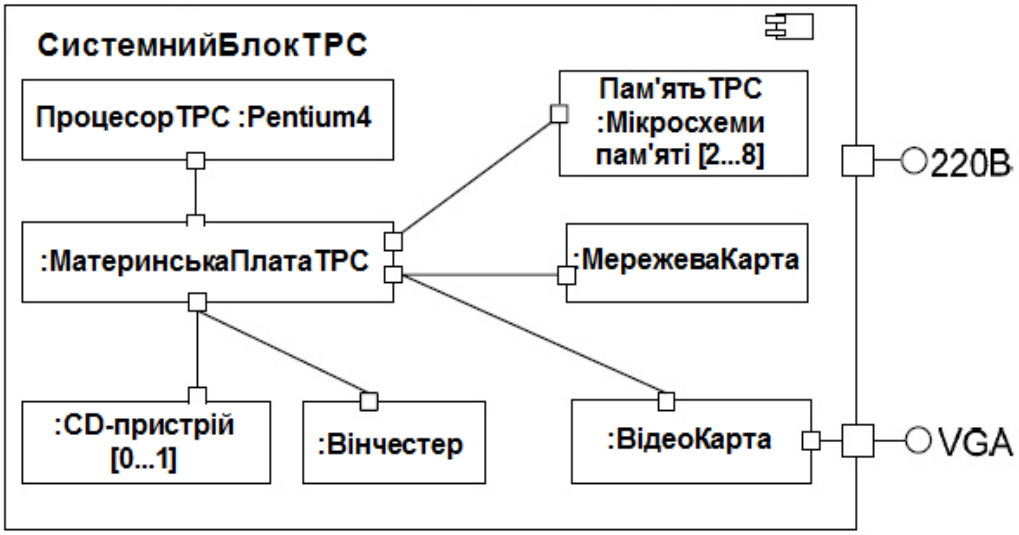


Рис. 5.6. Приклад діаграми композитних структур

Нижче наводяться приклади використання поведінкових діаграм UML. **Діаграми кінцевих автоматів** (приклад на **Рис**. 5.7).дозволяють створювати вичерпні специфікації поведінки телекомунікаційних, подіє-керованих алгоритмів і автоматично генерувати за цими описами програмний код. Приклад такої діаграми для класу «Підключення» наведений нижче.

Рис. 5.7. Приклад діаграм кінцевих автоматів

Ще один важливий вид діаграм – **діаграми послідовностей** (приклад на **Рис**. 5.8). Вони дозволяють задавати головні гілки складних телекомунікаційних алгоритмів, а також малювати ланцюжки викликів для об'єктно-орієнтованих застосувань, які програмуються в термінах об'єктів, але проектуються часто в термінах ланцюжків викликів. Приклад наведений нижче.

Рис. 5.8. Приклад діаграм послідовностей