МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРНІГІВСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНОЛОГІЧНИЙ УНІВЕРСИТЕТ

ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ ПРОГРАМНИХ СИСТЕМ

МЕТОДИЧНІ ВКАЗІВКИ
з курсового проектування
для студентів спеціальності 123 – «Комп'ютерна інженерія»

Обговорено і рекомендовано
на засіданні кафедри
інформаційних та комп'ютерних
систем
Протокол № 2
від 27 вересня 2017 р.

ЧЕРНІГІВ ЧНТУ 2018

Технології проектування програмних систем. Методичні вказівки з курсового проектування для студентів спеціальності 123 – «Комп'ютерна інженерія». / Укл.: Казимир В.В., Пріла О.А. – Чернігів: ЧНТУ, 2018. - 23 с., укр. мовою.

Укладачі: КАЗИМИР ВОЛОДИМИР ВІКТОРОВИЧ, доктор технічних наук, професор кафедри інформаційних та комп'ютерних систем
ПРІЛА ОЛЬГА АНАТОЛІЇВНА, доцент кафедри інформаційних та комп'ютерних систем

Відповідальний за випуск: ЗАЙЦЕВ СЕРГІЙ ВАСИЛЬОВИЧ, завідувач кафедри інформаційних та комп'ютерних систем, доктор технічних наук, доцент

Рецензент: НІКІТЕНКО ЄВГЕНІЙ ВАСИЛЬОВИЧ, кандидат фізико-математичних наук, доцент кафедри інформаційних та комп'ютерних систем Чернігівського державного технологічного університету

Зміст

Вступ

Курс «Технології проектування програмних систем» присвячений вивченню сучасних підходів до проектування і реалізації корпоративних програмних систем (КПС), що характеризуються складністю бізнес-логіки, а також високими вимогами до рівня надійності, продуктивності, розширюваності, масштабованості, безпеки та захищеності.

Особлива увага приділяється особливостям застосування класичних патернів проектування та усталених практик, які використовуються для розробки КПС.

Метою виконання курсового проекту є отримання практичних навичок проектування і розробки програмних систем рівня підприємства, використовуючи загальноприйняті стандарти і підходи, а також сучасні бібліотеки та інструментальні засоби розробки КПС.

У методичних вказівках наведені вимоги до архітектурних особливостей розроблюваного в рамках курсового проекту додатку без орієнтації на конкретні інструментальні засоби і бібліотеки. Представлені вимоги до організації командної роботи над проектом. Курс орієнтований на використання специфікації Java EE 7, а також agile-принципів командної розробки програмного забезпечення.

Методичні вказівки можуть використовуватися при виконанні дипломної роботи і проходженні переддипломної практики.

1 Поняття корпоративних програмних систем. Вимоги до них

Різновидів програмних додатків надзвичайно багато і кожна ситуація висуває особливі проблеми, що відрізняються власним рівнем складності.

В даному курсі ми будемо говорити про розробку корпоративних додатків (enterprise application).

Деякі інтерпретують поняття «корпоративне застосування» як синонім терміна «велика система». Однак важливо розуміти, що не всі подібні програми насправді «великі», хоча їх внесок в діяльність корпорації може бути досить вагомим. Нерідко значущість системи далеко перевищує її розмір. Крім того, загальною практикою вважається перетворення «великого» проекту в «менший» за рахунок спрощення прийнятої архітектури і обмеження безлічі реалізованих функцій.

Корпоративні додатки мають ряд особливостей, пов’язаних з підходами щодо решення виникаючих проблем: вони істотно відрізняються від вбудованих систем, систем керування, телекомунікаційних додатків, програмних продуктів для персональних комп'ютерів і т.д. Прикладом може бути порівняння корпоративної системи з додатками, що розробляються для сфери телекомунікацій. З певної точки зору корпоративні програми набагато простіше, ніж телекомунікаційне програмне забезпечення, проблеми забезпечення багатопоточного функціонування і тісної інтеграції апаратних і програмних компонентів. Але в усьому іншому завдання КПС істотно складніші. Корпоративні додатки найчастіше мають справу з даними великого обсягу і безліччю бізнес-правил. У числі близьких аналогів поняття «додаток» можна назвати термін «інформаційна система».

Отже, основні аспекти розробки корпоративних додатків.

1) Корпоративні програми зазвичай мають на увазі необхідність довготривалого (іноді протягом десятиліть) зберігання даних. Дані часто здатні "пережити" кілька поколінь прикладних програм, призначених для їх обробки, апаратних засобів, операційних систем і компіляторів. У продовження цього терміну структура даних може піддаватися численним змінам з метою збереження нових порцій інформації без будь-якого впливу на старі. У разі внесення змін до програмно-апаратне забезпечення, дані не знищуються, а переносяться в нове середовище. Даних, з якими має справу корпоративне додаток, як правило, буває багато і завдання маніпуляції цими даними виростає в одну з основних функцій програми.

2) Безліч користувачів звертаються до даних паралельно. Як правило, їх кількість не перевищує сотні, але для веб-додатків цей показник зростає на кілька порядків. Користувачам повинна бути надана можливість одночасної роботи з одним елементом даних такими способами, які виключають ймовірність виникнення помилок. Більшість завдань з управління паралельними транзакціями вирішуються на рівні диспетчера транзакцій СУБД, проте управління рівнем ізоляції транзакцій виконується на рівні

прикладної системи. При цьому потрібно враховувати, що високий рівень ізоляції транзакцій підвищує надійність інформаційної системи програми, усуває ймовірність виникнення всіляких конфліктних ситуацій, проте значно знижує продуктивність системи.

3) Якщо обсяги даних настільки великі, в додатку повинно бути передбачено і безліч різних варіантів екранного інтерфейсу. Повинна бути надана можливість подання даних в різних формах, зручних для користувачів всіх категорій.

4) Бізнес-логіка корпоративних систем складна, насичена і характеризується постійним внесенням змін. Так, наприклад, вигідний клієнт може вимовити для себе особливі умови оплати кредиту, що відповідають термінам надходження коштів на його розрахунковий рахунок. І пару тисяч таких ось окремі випадки здатні зробити бізнес-логіку абсолютно нелогічною, а відповідне програмне додаток - заплутаним і складно піддається сприйняттю. Для прикладу, при створенні операційної системи присутня сувора логіка.

5) Якщо програма забезпечує отримання тих чи інших переваг (а такими повинні бути всі корпоративні додатки), затримка з її впровадженням в експлуатацію означає прямі фінансові втрати.

Отже, корпоративні програми найчастіше мають справу з даними великого обсягу і складними бізнес-правилами. Подібні додатки припускають необхідність відображення, обробки та збереження великих масивів (складних) даних, а також реалізації моделей бізнес-процесів, які маніпулюють цими даними.

Корпоративна програмна система – це програмна система, яка оперує великими обсягами даних за допомогою певних бізнес-правил, орієнтована на конкретний вид діяльності підприємства і від її функціонування часто залежить сама діяльність підприємства.

До числа корпоративних додатків відносяться, наприклад, системи бронювання квитків, фінансові програми, пакети програм торгового обліку, бухгалтерський облік, ведення медичних карт пацієнтів, економічне прогнозування, аналіз кредитної історії клієнтів банку, страхування, зовнішньоекономічні торговельні операції і т.п. Корпоративними додатками не є засоби обробки тексту, регулювання витрати палива в автомобільному двигуні, управління ліфтами і обладнанням телефонних станцій, автоматичного контролю хімічних процесів, а також операційні системи, компілятори, ігри і т.д.

Крім зазначених, суттєвими нефункціональними вимогами до КПС є такі:

- надійність;
- захищеність;
- масштабованість;

простота супроводу.

2 Вимоги до архітектури та структури застосування

Термін «архітектура програмної системи» не має однозначного визначення. Крім того, існує більше одного способу опису архітектури і ступінь важливості кожного з них змінюється упродовж життєвого циклу системи.

Проте, можна назвати два класичних визначення терміна «архітектура»:

1) ідентифікація головних компонент системи і способів їх взаємодії;

2) вибір рішень, які інтерпретуються як основоположні і не підлягають зміні в майбутньому.

Перший пов'язаний з поділом системи на найбільш великі складові частини, у другому випадку маються на увазі якісь конструк-тивні рішення, які після їх прийняття практично не підлягають зміні. Якщо пізніше виявляється, що щось змінити легше, ніж здавалося спочатку, це "щось" виключається з "архітектурної" категорії.

2.1 Використання архітектурних патернів

Основа паттерна (типового рішення) - підхід, досить загальний і ефективний для того, щоб забезпечити подолання періодично виникають проблем певної природи. Типові рішення - не наукові відкриття; вони великою мірою є узагальнення результатів, накопичених у відповідній області. Тому більш доречно говорити про "пошуку" типового рішення, а не про його "винахід".

Вибираючи архітектуру і варіанти проектування, слід брати до уваги особливості конкретної системи. Обраний типове рішення найчастіше вимагає модернізації для конкретного випадку.

При проектуванні архітектури розробляється повинні бути застосовані типові підходи до виділення основних компонентів додатка, архітектурних шарів і схеми взаємодії між ними.

Особливості використання шарів при проектуванні. Концепція шарів (layers) - одна з загальновживаних моделей, що використовуються розробниками програмного забезпечення для розділення складних систем на більш прості частини.

Шар вищого рівня користується службами, які надаються нижнім шаром, але той не "обізнаний" про наявність сусіднього верхнього шару. Більш того, зазвичай кожен проміжний шар "приховує" нижній шар від верхнього: наприклад, шар 4 користується послугами шару 3, який звертається до шару 2, але шар 4 не знає про існування шару 2. Не в кожної архітектурі шари настільки «непроникні», в зв'язку з цим виділяють поняття чітке / нечітке розшарування системи.

Розподілення системи на шари надає ряд переваг.

- окремий шар можна сприймати як єдине самодостатнє ціле;
- можна вибирати альтернативну реалізацію базових шарів;

- залежність між шарами можна звести до мінімуму;
- кожен шар є вдалим кандидатом на стандартизацію.

Схема розшарування має і певними недоліками:

1) шари здатні вдало инкапсулировать багато, але не всі: модифікація одного шару часом пов'язана з необхідністю внесення каскадних змін в інші шари. Класичний приклад з області корпоративних програмних додатків: поле, доданий в таблицю бази даних, може бути відтворена в графічному інтерфейсі і має знайти відповідне відображення в кожному проміжному шарі.

2) Наявність надлишкових верств нерідко знижує продуктивність системи. При переході від шару до шару моделюються суті зазвичай піддаються перетворенням з одного подання до іншого. Незважаючи на це, інкапсуляція нижележащих функцій часто дозволяє досягти вельми істотної переваги. Наприклад, оптимізація шару транзакцій зазвичай призводить до підвищення продуктивності всіх верхніх шарів.

Однак найважче при використанні архітектурних шарів - це визначення вмісту і меж відповідальності кожного шару.

Класична модель розшарування системи. Класична архітектура передбачає виділення 3 основних шару: шар уявлення (presentation), шар бізнес-логіки (предметна область, domain) і шар інтеграції (джерела даних, data source).

До основних функцій шару уявлення відносяться відображення інформації та інтерпретація вводяться користувачем команд з перетворенням їх у відповідні операції в контексті домену (бізнес-логіки) і джерела даних.

Шар джерела даних реалізує підмножину функцій, що забезпечують взаємодію зі сторонніми системами, які виконують завдання в інтересах додатки. Код цієї категорії не-сет відповідальність за моніторинг транзакцій, управління іншими застосування, обмін повідомленнями і т.д. Для більшості корпоративних додатків основна частина логіки джерела даних зосереджена в коді СУБД.

Логіка домену (бізнес-логіка або логіка предметної області) описує основні функції програми, призначені для досягнення поставленої перед ним мети. До таких функцій відносяться обчислення на основі введених і збережених даних, перевірка всіх елементів даних і обробка команд, що надходять від шару відображення, а також передача інформації шару джерела даних.

Модель трьох шарів - не єдина, існують альтернативні схеми "розшарування" коду.

Хоча три основних шару - відображення, бізнес-логіка і джерело даних - можна виявити в будь-якому корпоративному додатку, спосіб їх поділу залежить від ступеня складності цього додатка. Форма розшарування може бути довільною, але в будь-якому корпоративному додатку шари повинні бути ідентифіковані.

Крім того, часто присутні декілька варіантів реалізації кожного шару.

Шар сервісів в класичній моделі розшарування. При виконанні курсового проекту пропонується поділ логіки додатка від логіки домена і введення додаткового шару сервісів (або шару служб).

Проміжні шари - часто, але не свідомо корисні. Загальний підхід: якщо який-небудь з трьох базових шарів переходить розумну межу складності, модель можна поповнити додатковим шаром, які приймають на себе надлишок функцій.

Ідея виокремлення шару служб з шару предметної області заснована на підході, що передбачає можливість відмежування логіки процесу від "чистої" бізнес-логіки. Рівень служб зазвичай охоплює логіку, яка відноситься до конкретного варіанту використання системи або забезпечує взаємодію з іншими інфраструктурами (наприклад, за допомогою механізму повідомлень).

Шар служб визначає межі застосування і безліч операцій, що надається для інтерфейсних клієнтських шарів коду. Шар інкапсулює бізнес-логіку додатка, управляє транзакціями і координує реакції на дії. В даному випадку бізнес-логіку розносять по двох категоріях: логіка домену (domain logic) має справу тільки з предметною областю як такою і логіка додатки (application logic) описує сферу відповідальності додатки (скажімо, повідомляє користувачів і сторонні додатки про виконання процесу обчислення доходів). Розміщення логіки додатка в "чистих" класах домену небажано. По-перше, класи домену допускають меншу ймовірність повторного використання, якщо вони реалізують специфічну логіку програми та залежать від тих чи інших прикладних інструментальних пакетів. По-друге, змішування логіки обох категорій в контексті одних і тих же класів ускладнює можливість нової реалізації логіки додатка за допомогою специфічних інструментальних засобів, якщо необхідність такого кроку стає очевидною. З цих причин шар служб передбачає розподілення "різної" логіки по окремим верствам, що забезпечує традиційні переваги розшарування, а також велику ступінь свободи застосування класів домену в різних додатках.

Шар служб може використовуватись також для розміщення логіки управління транзакціями і забезпечення безпеки.

Найскладнішим у роботі над бізнес-логікою є, ймовірно, вибір того, що саме і як слід відносити до того чи іншого шару. Наведемо приклад. Уявіть, що в програму додається принципово відмінний шар, наприклад інтерфейс командного рядка для Web-додатки. Якщо існує якийсь набір функцій, які доведеться продублювати для здійснення задуманого, значить, тут логіка домену "перетікає" в шар уявлення. Можна сформулювати тест інакше: чи потрібно повторювати логіку при необхідності заміни реляційної бази даних XML-файлом?

Уявіть, що додаток відображає список виділених червоним кольором назв товарів, обсяги продажів яких зросли більш ніж на 10% в порівнянні з рівнем минулого місяця. Припустимо, програміст розмістив відповідну логіку безпосередньо в шарі уявлення, вирішивши тут же зіставляти рівні продажів поточного і минулого місяця і змінювати колір, якщо різниця пе-

ревищує заданий поріг. Проблема полягає в тому, що в шар кончини вводиться невластива йому логіка предметної області. Щоб належним чином розділити шари, потрібен метод бізнес-логіки, що відображає факт перевищення рівня продажів певного продукта на задану величину. Метод повинен здійснити порівняння рівнів продажів за два місяці і повернути значення булевого типу. У коді шару уявлення досить викликати цей метод і прийняти рішення про зміну кольору відображення. В цьому випадку процес розбивається на дві частини: виявлення факту, який може служити підставою для зміни кольору, і власне зміна.

Взаємодія між шарами. Крім необхідності поділу на шари, існує правило, що стосується взаємовідносин шарів: залежність бізнес-логіки і джерела даних від рівня відображення не допускається. Іншими словами, в тексті програми не повинно бути викликів функцій відображення з коду бізнес-логіки або джерела даних. Правило дозволяє спростити можливість адаптації шару уявлення або заміни його альтернативним варіантом зі збереженням основи програми. Зв'язок між бізнес-логікою і джерелом даних, однак, не настільки однозначна і багато в чому визначається вибором типових рішень для архітектури джерела даних.

Іноді шари організують таким чином, щоб бізнес-логіка повністю приховувала джерело даних від уявлення. Найчастіше, однак, код уявлення може звертатися до джерела даних безпосередньо. Хоча такий варіант менш бездоганний з теоретичної точки зору, в практичному відношенні він нерідко більш зручний і доцільний: код уявлення може інтерпретувати команду користувача, активізувати функції джерела даних для вилучення відповідних порцій інформації з бази даних, звернутися до засобів бізнес-логіки для аналізу цієї інформації і здійснення необхідних розрахунків і потім відобразити результат на екрані.

Розміщення шарів. Вище йшлося про логічних шарах, тобто про розчленування системи на окремі частини. Подібне розділення корисно навіть тоді, коли всі верстви функціонують на одній машині. Втім, існують ситуації, в яких відмінності в поведінці системи можуть бути обумовлені принципами її фізичної організації. У більшості випадків існує тільки два варіанти розміщення і виконання компонентів корпоративних додатків - на стороні клієнта і на стороні сервері. Найчастіше найпростішим є функціонування коду всіх верств системи на сервері, при цьому максимально спрощуються процедури виправлення помилок і оновлення версій. В цьому випадку не доводиться турбуватися про внесення відповідних змін на всіх комп'ютерах, про їх сумісність з іншими додатками і синхронізації з серверними компонентами. Загальні аргументи на користь розміщення будь-яких верств на комп'ютері клієнта полягають в підвищенні швидкості реагування (responsiveness) додатки і в забезпеченні можливості локальної роботи.

Шар джерела даних краще завжди розташовувати на сервері. Виняток становить випадок, коли функції сервера дублюються в коді "дуже товстого" клієнта для забезпечення засобів локального функціонування си-

стеми. При цьому передбачається, що зміни, що вносяться до окремих джерел даних на клієнтській машині і на сервері, підлягають синхронізації за допомогою механізму реплікації.

Рішення про те, де повинен функціонувати шар уявлення, здебільшого залежить від переваг у виборі типу призначеного для користувача інтерфейсу. Застосування інтерфейса товстого клієнта автоматично тягне за собою необхідність розміщення шару відображення на клієнтській машині. Використання Web-інтерфейсу означає, що логіка уявлення зосереджена на сервері. Можливий також змішаний варіант розміщення логіки подання - частково на сервері, частково на клієнті.

Розщеплення безлічі бізнес-функцій між сервером і клієнтом виглядає як найгірше рішення, оскільки в загальному випадку ускладнює ідентифікацію того чи іншого фрагмента логіки. Основна причина, що спонукає застосовувати подібну архітектуру, може полягати в тому, що клієнту необхідно володіти тільки якоюсь частиною бізнес-логіки. Головне - ізолювати цю порцію коду в окремому модулі, які не залежачи-щем від інших частин системи. Це дасть можливість активізувати код і на комп’ютері клієнта, і на сервері, якщо така потреба виникне пізніше.

Як вже було зазначено, існують різні варіанти реалізації кожного з архітектурних шарів. Існують наступні «класичні» шаблони реалізації бізнес-логіки програми: «Сценарій транзакцій», «Модуль таблиці», «Модель предметної області (домен)».

При виборі варіанта реалізації шару сервісів додатки рекомендуються до розгляду шаблони «Інтерфейс доступу до домену», «Сценарій операції».

Повинні бути розглянуті наступні варіанти реалізації шару інтеграції «Шлюз таблиці даних», «Шлюз записи даних», «Активна запис», «Перетворювач даних (Data Mapper)».

Різні варіанти реалізації уявлення і контролера в моделі MVC також повинні бути вивчені і проаналізовані.

Класична архітектура корпоративного програми, що характеризується складністю бізнес-логіки, представлена на рисунку 1. Реалізація шару бізнес-логіки у вигляді «Моделі домену» і шару сервісів як «Інтерфейс доступу до домену» дозволяє реалізувати об'єкти бізнес-логіки «чистими», в т. ч. незалежними від реалізації шару інтеграції.

При проектуванні і розробці компонентів додатка повинен бути застосований принцип інверсії контролю і (Inversion of Control, IoC), що дозволяє знизити рівень зв'язності компонентів програми.

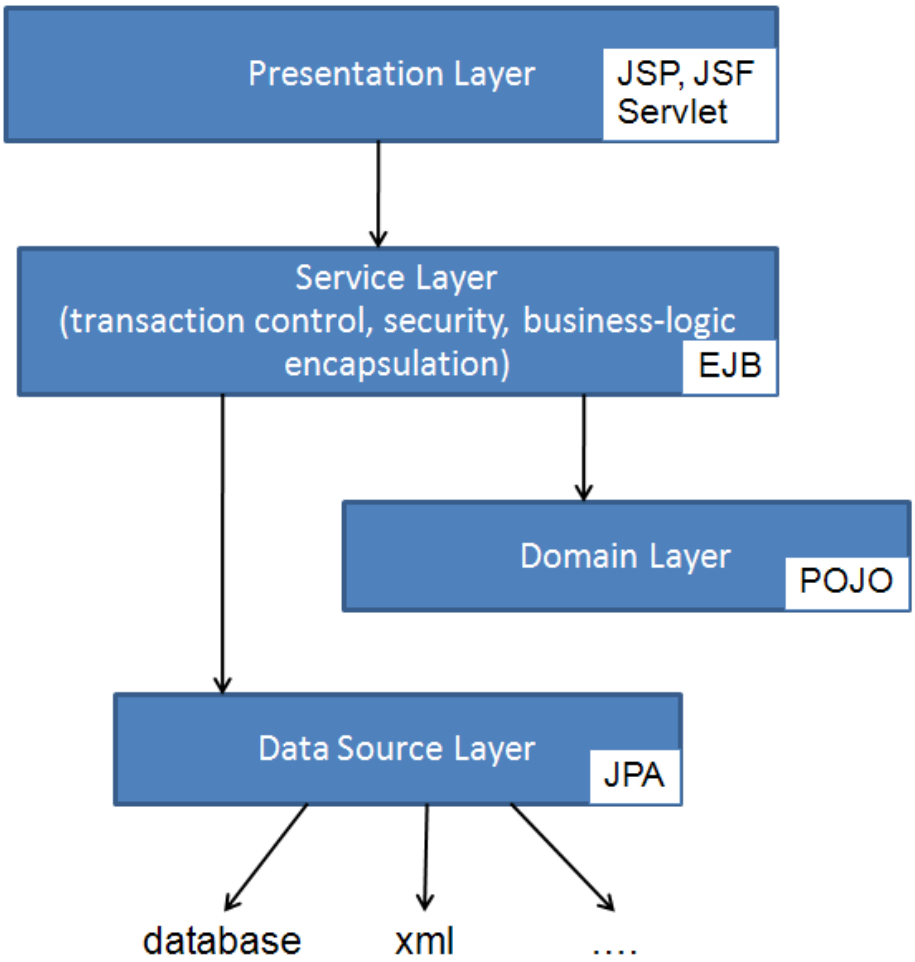

Рисунок 1 – Шар сервісів в класичній архітектурі корпоративних застосувань

2.2 Вертикальна та горизонтальна декомпозиція системи

Архітектура застосування повинна відображати не тільки вертикальну, але і горизонтальну декомпозицію системи. Під вертикальною декомпозицією мається на увазі виділення архітектурних шарів, в той час як під горизонтальною - виділення функціональних-них модулів в рамках одного архітектурного шару. Вертикальна декомпозиція не відображує функціональних особливостей і предметної області застосування, в більшості випадків стандартна для багатьох корпоративних додатків.

2.3 Багатомодульна структура застосування

Компоненти бізнес-логіки, також як і компоненти шару передуявлення, повинні бути представлені окремими модулями, які можуть бути незалежно розгорнуті на сервері додатків. Подібна організація структури додатка дозволяє повторне використання компонентів за допомогою інтеграції в інші додатки. Крім того, бізнес-логіка може бути представлена кількома модулями, які визначені функціональної складової. Таким чином забезпечується масштабованість додатки шляхом розгортання компонентів бізнес логіки на декількох серверах.

Специфікація JavaEE7 пропонує наступну структуру розкладання (рисунок 2):

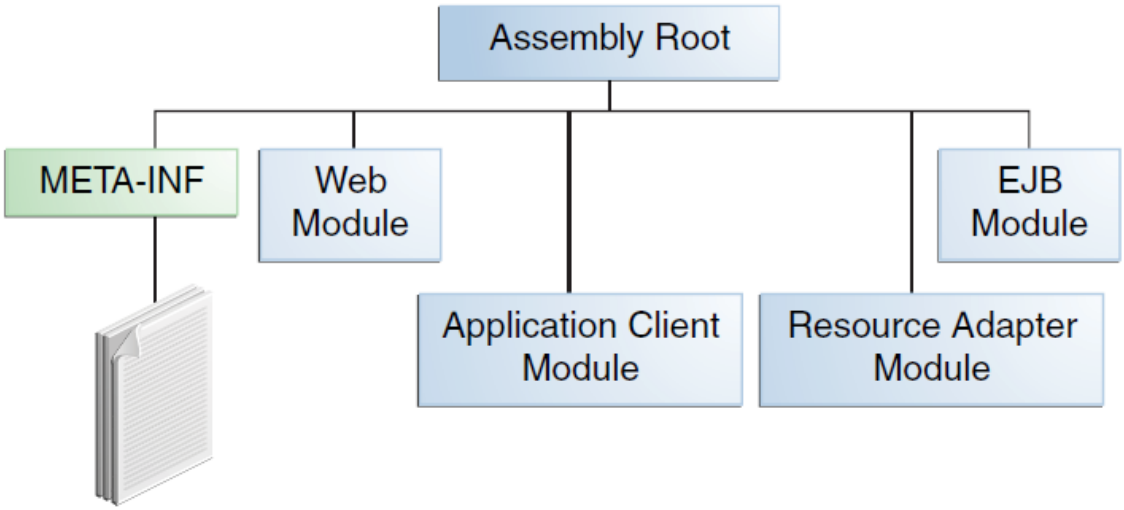

Рисунок 2 – Структура багатомодульного застосування згідно специфікації JavaEE 7

Специфікація Java EE 7 вводить такі типи модулей:

- ejb-модулі, упаковуються в формат .jar;
- web-модулі, упаковуються в jar-файли з розширенням .war (web-архів);
- модулі клієнтської програми, упаковуються в jar-файли з розширенням .jar;
- модулі адаптера ресурсів, представляють собою реалізацію архітектури Java EE Connector для певної EIS (Enterprise information system, інформаційна система корпоративного додатка). Упаковуються в jar-файли з розширенням .rar (resource adapter archive)

Також вводиться поняття пакета Enterprise Archive (ear) - формат файлів, який використовується для упаковки одного або більше модулів в одному архіві, щоб розгортання різних модулів на сервері додатків відбувалося одночасно й узгоджено. В каталозі META-INF пакету містяться дескриптори розгортання кожного з модулів.

2.4 Параметри продуктивності веб-застосування

Багато архітектурних рішення безпосередньо пов'язані з аспектами продуктивності (performance) системи.

Нижче наведені параметри, що характеризують продуктивність системи.

Час відгуку (response time) - час обробки запиту.

Швидкість реагування (responsiveness) - швидкість підтвердження запиту. Інтерактивна система, що володіє нормальним часом відгуку, але не відрізняється високою швидкістю реагування, характеризується низькою ефективністю використання. Якщо, перш ніж прийняти черговий запит, система повинна повністю завершити обробку поточного, параметри

часу відгуку і швидкості реагування, по суті, збігаються. Якщо ж система здатна підтвердити отримання запиту раніше, її швидкість реагування вище. Наприклад, застосування динамічного індикатора стану процесу копіювання підвищує швидкість реагування екранного інтерфейсу, хоча ніяк не позначається на значенні часу відгуку.

Час затримки (latency) - мінімальний інтервал часу до отримання будь-якого відгуку (навіть якщо від системи більш нічого не потрібно). Зазначений чинник нерегульований розробником, проте при проектуванні і розробці додатків враховувати його потрібно шляхом мінімізації кількості віддалених викликів. Останнє реалізується за допомогою «Об'єкту передачі даних» (Data Transfer Object).

Пропускна здатність (throughput) - кількість даних (операцій), що передаються (виконуваних) в одиницю часу. Якщо, наприклад, тестується процедура копіювання файлу, пропускна здатність може вимірюватися числом байтів в секунду. У корпоративних додатках зазвичай мірою продуктивності служить число транзакцій в секунду (transactions per second - tps), однак такий підхід неоднозначний - транзакції розрізняються за ступенем складності. Для конкретної системи необхідно розглядати суміш "типових" транзакцій.

У контексті розглянутих термінів під продуктивністю можна розуміти один з двох параметрів - час відгуку або пропускну здатність, зокрема той, який більшою мірою відповідає конкретній ситуації. Іноді буває важко судити про продуктивність, якщо, наприклад, використання деякого рішення підвищує пропускну здатність, одночасно збільшуючи час відгуку. З точки зору користувача, значення швидкості реагування може виявитися більш важливим, ніж час відгуку, так що поліпшення швидкості реагування ціною втрати пропускної здатності або зростання часу відгуку цілком здатне підвищити продуктивність.

Завантаження (load) - значення, що визначає ступінь "тиску" на систему і вимірюється кількістю запитів, що одночасних надходять до системи. Параметр завантаження зазвичай слугує контекстом для подання інших функціональних характеристик, подібних часу відгуку, наприклад «час відгуку на запит становить 0,5 секунди для 10 користувачів і 2 секунди для 20 користувачів».

Чутливість до завантаження (load sensitivity) – вираз, що задає залежність часу відгуку від завантаження. Припустимо, що система А має час відгуку, що дорівнює 0,5 секунди для 10-20 користувачів, а система В - часом відгуку в 0,2 се¬кунди для 10 користувачів та 2 секунди для 20 користувачів. Отже, що система А має меншою чутливістю до завантаження, ніж система В. Також використовується термін погіршення (degradation), щоб підкреслити факт меншою стійкості параметрів системи В.

Ефективність (efficiency) - питома продуктивність в перерахунку на одну одиницю ресурсу. Наприклад, система з двома процесорами, здатна виконати 30 tps, більш ефективна в порівнянні з системою, оснащеної чотирма аналогічними процесорами та володіє продуктивністю в 40 tps.

Потужність (capacity) - найбільше значення пропускної здатності або завантаження. Це може бути як абсолютний максимум, так і деяке число, при якому велічина продуктивності все ще перевершує заданий прийнятний поріг.

Здатність до масштабування (scalability) - властивість, що характеризує поведінку системи при додаванні ресурсів (зазвичай апаратних). Масштабованою прийнято вважати систему, продуктивність якої зростає пропорційно обсягу використовуваних ресурсів (наприклад, удвічі при подвоєнні кількості серверів). Вертикальне масштабування (vertical scalability, scaling up) – це збільшення потужності окремого сервера (наприклад, за рахунок збільшення обсягу оперативної пам'яті). Горизонтальне масштабування (horizontal scalability, scaling out) - це нарощування потенціалу системи шляхом додавання нових серверів.

Здатність додатку до масштабування багато в чому визначається архітектурними особливостями. При проектуванні і реалізації корпоративних додатків слід використовувати принципи і технології розробки розподілених додатків. В рамках курсу пропонується використання EJB-технології та веб-сервісів.

Обов'язковою вимогою є реалізація реплікації бази даних (можливо, декількох) додатки з метою підвищення надійності і продуктивності.

Оцінка часу відгуку програми і чутливості до навантаження повинна бути проведена засобами навантажувального тестування.

3 Вимоги до підсистеми безпеки застосування

Згідно Java EE 7 виділяють наступні характеристики безпеки застосування.

- Аутентифікація (Authentication) - процес ідентифікації взаємодіючих об'єктів, таких як клієнт або сервер; перевірка їх достовірності.
- Авторизація (Authorization, access control) - процес підтвердження прав користувача (або іншого об'єкта) на виконання тих чи інших дій.
- Цілісність даних (Data integrity) - кошти забезпечення неможливості підміни пересилаються по мережі даних «третьою особою» (чи не джерелом інформації). Одержувач даних повинен мати можливість виявлення і скасування прийому повідомлень, які були змінені після відправ-ки.
- Конфіденційність (Confidentiality data privacy) - забезпечення доступності захищеної інформації тільки для авторизованих користувачів.
- QoS (Quality of Service) - кошти забезпечення необхідного рівня якості обслуговування.
- Облік (Accounting, Auditing) виконуваних операцій з доступу до захищених ресурсів додатки з метою оцінки ефективності політики безпеки і використовуваних механізмів.

Підсистема безпеки програми повинна бути реалізована з використанням JAAS або іншої реалізації стандарту системи інформаційної безпеки PAM.

Повинна бути проведена настройка реалма веб-контейнера, що містить колекцію користувачів і груп користувачів, з подальшою прив'язкою ролей додатки до конкретних користувачам або групам. Як реалма пропонується використання сервера LDAP.

Повинні бути передбачені механізми захищеної передачі даних для відповідних ресурсів веб-додатки. Доброю практикою є використання захищених протоколів передачі даних не для всіх ресурсів програми, оскільки процеси шифрування / дешифрування є обчислювально ресурсоємними.

4 Автоматизація процесів тестування, збірки та розгортання проекту

Реалізація кожного компонента додатка повинна супроводжуватися відповідними автоматизованими модульними тестами.

При проектуванні тестів важливим є забезпечення незалежного тестування компонентів програми та уникнути непрямого тестування сторонніх компонентів. Наприклад, з метою уникнення непрямого тестування шару інтеграції при тестуванні шару сервісів, рекомендовано використання mock-об'єктів з метою імітації інформаційних ресурсів програми.

Принцип інверсії контролю забезпечує незалежність компонентів і спрощує процес тестування, оскільки дозволяє легко підміняти реальну реалізацію сервісів за допомогою заглушок або mock-об'єктів.

Крім модульних, повинні бути сплановані і реалізовані інтеграційні тести.

Для оцінки параметрів продуктивності додатка, має бути проведено тестування навантаження.

Для написання модульних і інтеграційних тестів рекомендується використання бібліотеки JUnit 4, для проведення навантажувального тестування - інструментарій Apache JMeter.

Обов'язковою вимогою є організація автоматичного складання і розгортання програми з використанням будь-якої з існуючих систем збирання незалежно від середовища розробки проекту і середовища оточення. Збірка проекту повинна включати етап автоматичного запуску модульних і інтеграційних тестів.

В якості системи зборки java-додатків рекомендується система Apache Maven.

5 Організація командної роботи над проектом

Склад команди, розподіл ролей. Курсовий проект виконується групами по 3-4 людини з виділенням наступних ролей в команді: менеджер, розробник і розробник тестів (тестувальник).

Оскільки проект носить навчальний характер, завдання учасників, що мають різні ролі в проекті, будуть перетинатися. Наприклад, тестувальник повинен бути зайнятий не тільки в плануванні та написанні тестів, але і в розробці модулів системи. При цьому повинен залишатися пріоритет завдань, характерних для певної ролі.

Розподіл робіт повинно бути виконано таким чином, щоб кожен член команди взяв участь в розробці / тестуванні компонентів (модулів) кожного шару додатки.

Завдання менеджера проекту:

- проектування архітектури додатку;
- планування робіт, розподіл і призначення завдань;
- проектування користувальницького інтерфейсу;
- формування фінальної документації;
- розробка і тестування модулів системи (20%).

Завдання розробника:

- проектування;
- розробка модулів системи (60%);
- модульне тестування власних модулів;
- документування.

Завдання тестувальника:

- Проектування і реалізація інтеграційних тестів;
- Тестування інтерфейсу користувача;
- Планування і проведення навантажувальних тестів;
- Розробка модулів системи (20%).

Використання системи управління проектами. Фіксація, призначення та контроль виконання завдань проекту повинні виконуватися централізовано, з використанням системи управління проектами. Можливе використання як публічних зовнішніх сервісів спільної розробки проектів, так і кафедральний сервіс, заснований на системі управління проектами Redmine.

Використання системи управління версіями. Обов'язковою умовою командної роботи над проектом є використання сховища проекту.

Системи безперервної інтеграції (continuous integration). Основний принцип безперервної інтеграції - часта і швидка збірка і тестування програми, в ідеалі при кожному внесенні змін до проекту. Існують системи автоматизації процесу складання і тестування програми, що дозволяють визначити періодичність або умова запуску процесу. Використання в процесі розробки курсового проекту принципів безперервної інтеграції є необов'язковим.

6 Структура пояснювальної записки

Пояснювальна записка повинна бути оформлена відповідно до СОККР.

Розширене технічне завдання. Крім стандартних вимог до структури технічного завдання, необхідно подати:

1) розширені функціональні вимоги до додатка з визначенням ролей користувачів програми та варіантів використання системи для кожної ролі. У разі корпоративної програмної системи кількість призначених для користувача ролей має бути вище трьох;

2) нефункціональні вимоги до системи і механізми їх забезпечення;

3) вимоги до підсистеми безпеки;

4) передбачувані технології та інструментальні засоби розробки системи;

5) склад команди, розподіл ролей і визначення зон відповідальності кожного учасника.

Основна частина пояснювальної записки повинна відображати наступні аспекти проектування і реалізації програми:

- аналіз предметної області, визначення функціональних і не функціональних вимог до системи;
- проектування інтерфейсів користувачів додатка;
- проектування архітектури додатку, вибір архітектурних рішень вибір технологій і засобів реалізації КПС;
- планування і розподіл завдань проекту;
- проектування і реалізація бізнес-логіки програми: використовувані патерни реалізації, використовувані технології, листинги доменних об'єктів з описом, листинги модульних і інтеграційних тестів, результати тестування, листинги файлів конфігурації;
- проектування і реалізація шару інтеграції (структура аналогічна попередній);
- проектування і реалізація шару уявлення (структура аналогічна попередній);
- реалізація підсистеми безпеки;
- механізми забезпечення продуктивності програми та результати тестування.

Вміст вихідних файлів проекту в додатку виносити не слід. У додатках можуть бути приведені результати тестування, конфігураційні файли та ін.

7 Порядок захисту та оцінювання роботи над проектом

Для захисту курсової роботи кожна команда повинна надати презентацію проекту, а також заповнений протокол захисту курсового проекту (див. Додаток А). Порядок презентації проекту представлений нижче.

Підготовча робота для демонстрації проекту:

- завантажити вихідні файли проекту зі сховищ;
- продемонструвати структуру каталогів проекту (вона повинна відповідати декомпозиції проекту, обов'язкова наявність файлів: readme, authors, copyright);
- продемонструвати автоматичну збірку і розгортання проекту з використанням будь-якої з існуючих систем збирання незалежно від середовища розробки проекту. Збірка проекту повинна включати етап автоматичного запуску тестів.

В ході презентації проекту необхідно відобразити такі аспекти:

- призначення розробленої системи, її основні функції;
- уявити архітектуру системи, що використовуються архітектурні рішення і патерни;
- розподіл завдань за проектом між учасниками команди, визначення ролей і функцій кожного учасника;
- реалізація системи і використовувані при розробці технології (кожен учасник описує підсистеми, розробкою яких він займався);
- демонстрація системи на основі сценаріїв тестування (3 основних робочих сценарію);
- демонстрація працездатності реплікації бази даних (основної бази даних або бази даних користувачів);
- надати і оцінити результати навантажувального тестування програми;
- організація підсистеми захисту програми.

Супровід презентації слайдами вітається.

Кожен учасник проекту отримує індивідуальну оцінку. Початкова оцінка студента - це оцінка проекту відповідно до пред'явленими вище вимогами. Внесок кожного учасника в проект оцінюється відповідно до наступних критеріїв:

- активність роботи з репозиторієм проекту (загальна кількість змін, внесених кожним з учасників проекту, має бути відповідним);
- якість реалізації і презентації своєї частини робіт за проектом;
- якість оформлення документації. Звіт команда оформляє в одному екземплярі;
- володіння і знання вихідного коду всього проекту (командна розробка передбачає колективне володіння кодом, тобто кожен з учасників повинен орієнтуватися в початкових кодах всіх модулів проекту);
- лідер проекту може підвищувати або знижувати до 10 балів для всіх учасників проекту, з огляду на індивідуальний внесок кожного в проект.

Приклади можливих питань: «яким чином поле X через шар бізнес-логіки потрапляє з БД на екран користувача?»; «Поясніть тестовий метод для тієї чи іншої функції»; «Яким чином перевіряються права користувача на виконання тієї чи іншої дії в системі?» і т.д.

Додаток А - Протокол захисту курсового проекту

Протокол захисту курсового проекту з дисципліни «ТППС»

дата захисту ___.___.____

Склад команди (ПІБ, група) _____
Тема роботи: _____

Відповідність програмного проекту вимогам, затвердженим в ТЗ _____
Загальна оцінка проекта _____
Індивідуальні оцінки _____

Критерії оцінювання проекта

Управління конфігурацією проекту
• наявність вихідних файлів проекту в репозиторії;
• правильна структура каталогів проекту;
• автоматизована збірка, запуск тестів і розгортання програми.
Управління проектом
• розподіл завдань, планування;
• документація по проекту.
Технічна сторона реалізації
• розподіл архітектури за рівнями;
• реалізація підсистеми безпеки з використанням JAAS (налаштування реалма веб-контейнера, конфігураційні файли)
(В разі налаштування LDAP):
• структура LDAP-дерева;
• наявність файлів конфігурації для налаштування LDAP-сервера;
• налаштування реплікації БД;
Презентація проекту (бажано супровід слайдами)
• розгортання серверів додатку;
• опис розробленої системи (призначення, функції);
• реалізація системи, технології, що використовувалися при розробці;
• демонстрація системи на основі сценаріїв використання.

Критерії індивідуального оцінювання учасників проекта

• активність роботи з репозиторієм проекту;
• якість реалізації і презентації своєї частини робіт за проектом;
• якість оформлення звіту;
• володіння вихідним кодом всього проекту;
• оцінювання лідером команди роботи учасника проекту.

Рекомендована література

1. Структура и оформление квалификационных и курсовых работ. Методические указания для студентов профессионального напраления «Компьютерная инженерия» / Сост. А.И. Вервейко, С.А. Нестеренко, Е.В. Никитенко. – Чернигов: ЧГТУ, 2001. – 34 с.
2. Фаулер, М. Архитектура корпоративных программных приложений: пер. с англ. – М.: Издательский дом «Вильямс», 2006. – 544 с.
3. Eric Jendrock, Ricardo Cervera-Navarro, Ian Evans and others. The Java EE 7 Tutorial. – August 2013.
4. Hudson Continuous Integration [Электронный ресурс]. – Режим доступа: http://hudson-ci.org/.
5. JMeter – Apache JMeter [Электронный ресурс]. – Режим доступа: http://jakarta.apache.org/jmeter.
6. JUnit.org Resources for Test Driven Development [Электронный ресурс]. – Режим доступа: http://www.junit.org.
7. Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern [Электронный ресурс]. – Режим доступа: URL: http://martinfowler.com/articles/injection.html.
8. Redmine – Overview [Электронный ресурс]. – Режим доступа: http://www.redmine.org.
9. Vincent Massol, Ted Husted. JUnit In Action. Manning. – Special Sales Department, 2004. – P. 386.
10. Welcome to Apache Maven [Электронный ресурс]. – Режим доступа: http://maven.apache.org.