УДК 681.3.01

Глибовець М. М., Іващенко С. А., Крусь О. О.

РОЗРОБКА СИСТЕМИ УПРАВЛІННЯ ВИЩОГО НАВЧАЛЬНОГО ЗАКЛАДУ

У статті описано основні принципи розробки, використані при побудові автоматизованої системи управління навчальним закладом. Розглянуто основні принципи архітектурних рішень, вимоги до вибраного інструментарію, нестандартні архітектурні рішення та поєднання інструментальних засобів, методи забезпечення нефункциональных вимог, захисту та розширюваності системи.

Мотивація досліджень

Метою науково-дослідної роботи було створення прототипу типової автоматизованої системи управління навчальним закладом (АСУНЗ), що забезпечує виконання стандартних функцій керування навчальним процесом та допоміжними структурними підрозділами: створення внутрішньої інформаційної мережі університету (Intranet), інформаційних порталів факультетів та основних структурних підрозділів університету із забезпеченням прозорого віддаленого доступу до інформаційно-довідкових та навчальних матеріалів засобами Інтернету, порталів навчально-методичного відділу та відділу кадрів із забезпеченням «дружнього» інтерфейсу з віддаленим доступом, який дає можливість зручного ведення облікової інформації, складання розкладів занять, розрахунків навантаження викладачів, запису студентів на вибіркові курси, ведення та обліку оцінок студентів за всі види навчальних робіт, документообігу, пошукових послуг тощо.

Проблема є актуальною, оскільки її розв'язання забезпечить значну економію часу на управлінські рішення, чіткість керування, інформованість керівництва, реальне підґрунтя для прийняття рішень (з можливістю використання автоматизованої експертної системи вироблення таких рішень).

Одним із завдань досліджень була розробка архітектурного рішення і реалізація платформи, які б забезпечили можливість ефективного виконання довільної функціональності згідно з вимогами, що упроваджуватимуться під час поетапної автоматизації навчального закладу.

Вибір технології

Вибір архітектури та технології розробки системи базувався на їх можливості забезпечити гнучку розробку системи протягом значного періоду часу багатьма незалежними розробниками. Обрана компонентно-орієнтована технологія дає змогу розбити систему на незалежні модулі та гнучко пов'язати їх в одній системі. Більше того, за необхідності внесення змін до будь-якого модуля у майбутньому кожен компонент може бути замінений на модифіковану версію без перекомпіляції всього коду Такий підхід не тільки покращує параметри підтримки коду, пришвидшує розробку системи, організовує вихідний код, а й надає можливість дуже чітко протестувати системні модулі, що підвищує стабільність всієї системи.

Inversion of Control

Основним підходом при побудові компонентної структури є так звана концепція Inversion of Control (IoC). Основна ідея IoC характеризується висловом: «Не клич мене, я покличу тебе». Тобто відповідальність за виклик будь-якого компонента в системі покладається не на самі компоненти, а на каркас, в якому вони поєднані. Таким чином, компоненти не повинні «знати» про інші компоненти для своєї роботи, а також як їх знайти та викликати. Каркас, що реалізує IoC-контейнер, сам визначає потребу компонентів одне в одному і надає необхідний компонент під час виконання.

У fi] надано детальнішу класифікацію IoC та описано патерн Dependency Injection (DI), що точніше характеризує принцип, який пов'язує компоненти в каркасі на основі інтерфейсів, відділяючи реалізацію компонентів від їх конфігурації, таким чином реалізуючи незалежність

компонентів і можливість їх легкої заміни всередині системи.

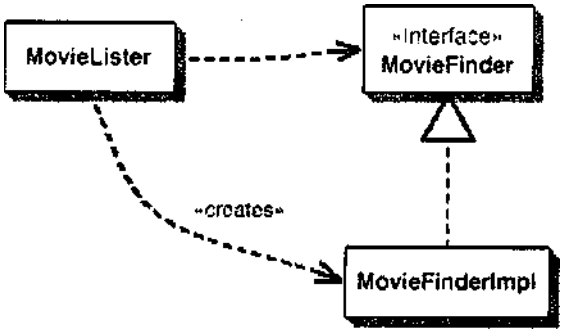

Рис. 1. Залежності при звичайній побудові класів

Для прикладу наводимо задачу fi 1, що вирішує DI (рис. 1). Існує клас MovieLister, який для своєї роботи потребує компонент, що реалізує інтерфейс MovieFinder, тобто він зачежить від конкретної реалізації MovieFinderlmpl. Задача патерна DI полягає в тому, щоб прибрати цю залежність, тобто надати можливість вибирати ту реалізацію інтерфейсу MovieFinder, яка необхідна різним компонентам.

Основна ідея DI полягає у використанні окремого об'єкта Assembler, який надасть класу MovieLister значення конкретної реалізації MovieFinder-інтерфейсу, прибираючи залежність MovieLister від конкретної реалізації та залишивши прив'язку до інтерфейсу MovieFinder (що важливо для збереження типів).

Spring Framework

Залежно від типу надання об'єктом Assembler інформації про конкретну реалізацію, розрізняють три форми DI: Constructor Injection (вставка через конструктор), Setter Injection (вставкачерез поля класу) та Interface Injection (вставка через інтерфейс). Всі три форми описано в 11]. При розробці АСУН 3 було використано каркас Spring Framework [2], який оснований на підході IoC і використовує вставку через спеціальні методи класу — сеттери, через які клас отримує конкретні дані ззовні. Spring реалізує IoC-контейнер. який використовує XML-файли для зберігання конфігурації кожного компонента (або інші методи зберігання) і за допомогою якої встановлює (injects) необхідні параметри компонентам.

Застосований підхід дає змогу будувати компоненти з чітко визначеним інтерфейсом, які не залежать одне від одного, а їх поєднання в одне ціле відбувається задопомогою окремої конфігурації. Для цього підходу характерно, шо компоненти не залежать від Spring-каркасу, а сам каркас виконує службові функції, наприклад, читає конфігураційні файли та встановлює необхідні значення.

Отже, реалізація Spring-каркасу має багато переваг: він надає «легкий» контейнер '2EE-застосуванням, використовує зв'язування компонентів за інтерфейсами, не зобов'язує застосування використовувати Spring і залежати від нього, підвищує можтавість якісного тестування кінцевого продукту за допомогою unit-тестування. Слід зазначити, що Spring не реалізує і не намагається реалізувати конкретні підкаркаси, наприклад роботи з базами даних, забезпечення трансакцій, ведення журналу,- він просто надає методи для їх зв'язування, отже, не лімітує розробників у використанні найбільш зручних і корисних для проекту технологій.

Архітектура

Цикли документообігу

Основною метафорою при розробці архітектури було розбиття роботи навчального закладу на цикли документообігу. Розроблена архітекту-

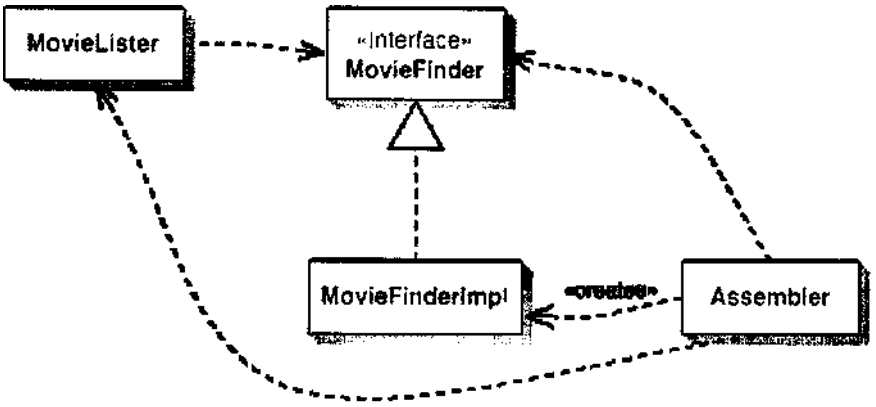

Рис. 2. Залежності при використанні DI

ра відображає реальні цикли документообігу в навчальному закладі й продовжує їх на проміжках, які можна автоматизувати. Основою для обробки є віртуальні об'єкти моделі даних, які наближено відображають реальні об'єкти. Наближення вибрано таким чином, щоб об'єкти були якомога простішими для системи, забезпечивши збереження їх інформаційних характеристик, які мають бути враховані при автоматизації.

Для забезпечення можливості роботи з базовими об'єктами системи в архітектурі узгоджено й цілісно використовується метафора документа. Документ є підсистемою реально існуючої структури даних, у межах якої можуть відбуватися зміни. Кожен документ виділяється таким чином, щоб зміни в структурі об'єктів, які входять до його складу, були атомарними, тобто не впливали на всю структуру в цілому. Документ також є об'єктом.

Кожен документ передається між прошарками системи, і для кожного прошарку достатньо працювати з документом, щоб забезпечити конкретні кроки будь-якого автоматизованого процесу. Кожному прошарку відома структура даних системи, і він може модифікувати документ у межах цієї структури.

Будь-який запит користувача виконується як проведення певної операції над документом. Можливі дії над документом визначаються циклами і підциклами документообігу (workflow cycles). Кожен цикл документообігу виконується в трансакції та трасується, поки не завершиться. Кожен серверний виклик передбачає те, що виклик проходить у якомусь конкретному циклі, який перебуває в конкретному стані.

Зміни в інформації системи здійснюються в основному через відповідні документи. Якщо необхідно внести зміни до системи, наприклад змінити прізвище студента, потрібно відкрити документ, який відповідає реєстраційній картці студента, і внести зміни. Модель захисту чітко відстежує, які операції користувач може здійснювати над документом і що потрібно для того, щоб документ міг внести зміни до діючої системи. Наприклад, якщо для зміни стану будь-якого бізнес-об'єкта потрібні додаткові санкції, то документ відповідним чином обробляється перед тим, як він змінить стан системи. Система записує зміни, які потім зберігаються у сховищі даних.

Система складається з таких основних прошарків:

1) інтерфейс користувача:
- показує певну частину стану системи користувачеві;
- обробляє дії користувача і фіксує їх у змінених документах;
- отримує документи від моделі і передає змінені документи в модель для обробки;

2) модель:
- формує базову структуру даних;
- обробляє модифіковані документи згідно з бізнес-правилами;
- отримує дані від прошарку даних і зберігає дані через нього;

3) прошарок обробки даних:
- забезпечує правильне й узгоджене зберігання даних та повернення даних на запити.

Інтерфейс користувача

На рівні інтерфейсу користувача можна змінювати документ (але не можна створювати новий), проглядати документ, додавати нові об'єкти в документ у межах структури даних. Інтерфейс може як завгодно довго утримувати документ в сесії і крок за кроком обробляти його, проте такі зміни вважаються заявкою на модифікацію і не є реальною зміною для системи. Остання не може реагувати на ці модифікації як на щось важливе (хоча користувач може отримувати повідомлення про реальні помилки і попередження про неправильність введення тощо).

Щоб не обробляти щоразу дані за подібною схемою в конкретних класах на рівні інтерфейсу, однакові принципи обробки виносяться в методи роботи з контентом документа. Якщо якась комплексна зміна чи відтворення стосується скоріше вмісту, аніж презентації даних, вона обов'язково обробляється методами контенту документа. Якщо на певному етапі потрібно активізувати зміни, документ передається на рівень моделі та бізнес-логіки.

Модель (сервіс користувача)

На рівні моделі документ обробляється як уже повністю підготовлений. Модель обробляє документ цілком. Може перевірятися узгодженість документа з іншими даними, дозвіл на модифікацію та правильність внесених або модифікованих даних. На цьому рівні може бути виявлено помилку й обробку документа може бути перервано. Якщо з певними документами пов'язані певні правила їх обробки, вони застосовуються на цьому рівні.

Прошарок обробки даних

Документ обробляється одним із трьох можливих способів: створення, збереження, відтво-

рення. Кожна операція реалізується окремо для кожного типу контенту документа. Для обробки кожного документа часто необхідно знати не лише те, що міститься в ньому (часто документи можуть охоплювати досить великі структури даних, навіть рекурсивно пов'язані в потенційно нескінченні ланцюги структур), а й те, що саме з усього контенту документа змінено (або потрібно створити). Для цього існують фільтри, які передаються операціям разом із документами.

Фільтр є структурою функціональних об'єктів, що накладається на контент (можливий контент) документа для формування обмеженого контенту. При застосуванні в операціях відтворення документа фільтр вказує, яку частину з усього доступного для документа контенту очікують інші прошарки для обробки. Проте фільтр не слід плутати з природним обмеженням контенту документа — воно враховується незалежно від фільтрів. При застосуванні в операціях збереження фільтр вказує, в якій частині контенту документа відбулися зміни, щоб не було необхідності обробляти весь контент задля збереження операційних ресурсів.

Фыьтри

Фільтр утворюється простими операціями з атомарних фільтрів. Кожен атомарний фільтр формується на рівні прошарку обробки даних і передається на запит через рівень моделі на рівень інтерфейсу користувача. Об'єкти інтерфейсу можуть отримати будь-яку кількість атомарних фільтрів у будь-який момент і сформувати з них будь-яку структуру, використовуючи доступні базові операції. Цю структуру вони передають на обробку якдодаток до документа. На наступному етапі фільтр передається на рівень роботи зданими, якому відомо, як його застосувати для формування або збереження об'єкта.

Нестандартні архітектурні рішення

Неконтрольовані модифікації

У системі часто спостерігається явище, коли кілька ролей можуть змінювати деякі дані відповідно до доступної наданий момент інформації про них, але загальний контроль заданими насправді має обмежена кількість ролей або лише одна роль. З одного боку, не можна заборонити іншим ролям змінювати критичні дані в системі (бо вони можуть застаріти), а з іншого — змінені критичні дані можуть стати помилковими, оскільки роль, яка має над ними контроль, не відстежує зміни.

У такій ситуації використовується механізм фіксування даних у сталих документах. Окрім динамічної структури даних у базі, контролююча роль може зберегти (зафіксувати) певну частину даних в окремому документі поряд зі стандартним збереженням.

Я кщо пізніше хтось вносить зміни до даних, контролююча роль може звірити зафіксовані дані з реальними і виправити або реальні дані (скоректувати помилки), або зафіксовані дані (затвердити зміни). Отже, будь-які модифікації можуть бути піддані послідовному контролю над ними.

Передача параметрів через фільтр

Часто виникає необхідність на рівні обробки даних знайти деякі значення чітко визначених властивостей об'єктів, що використовуються для фільтрації. Наприклад, вибираються всі викладачі одного факультету, а значення цього факультету потрібно використати ще в інших операціях. Шукати ці значення в графі фільтру не зовсім зручно і не зовсім правильно, як і передавати такі параметри додатково як особливу FilterFeature, оскільки це додає зайву функціональність у фільтр і ускладнює як обробку, так і реалізацію контенту документа. Тому для цього використовують фільтрування за прикладом (example). Сам фільтр розуміє, що треба відібрати елементи за рівністю встановлених непустих значень, а інші структури рівня даних можуть знаходити сталі значення властивостей із прикладу і використовувати їх. Такий підхід вважатиметься найвдалішим.

Ієрархічні форми та заповнення даних

Для заповнення даних у складних ієрархічних структурах використовують ієрархічні форми з аналогічною структурою компонентів (приблизно кожен компонент відповідає певному типу даних у структурі). Основною була вибрана стратегія централізованого доступу/модифікації даних у структурі. Всі компоненти інтерфейсу забезпечують лише інформацію про шлях до певного параметру в структурі й дають змогу отримувати або змінювати параметр відповідно до цього шляху. Базовий компонент повинен обробляти задані шляхи і виконувати операції зі структурою об'єктів. Це дає можливість гнучко працювати зі структурою даних у будь-який спосіб.

Видалення і зміни сущностей

Видаленню підлягають тільки сутності, які не пов'язані з іншими сутностями. Якщо наявний зв'язок певної сутності з іншими сутностями, тоді користувач не повинен мати можливості видалити сутність, доки не будуть видалені пов'язані сутності, або отримає повідомлення про помилку при спробі видалення. Змінювати дозволяється все, що не заборонено правилами документа, який забезпечує зміни.

Протоколювання (ведення журналу змін)

У системі ведеться запис виконаних операцій і змін даних. Усі зміни зведені до конкретного значення поля таблиці даних і поділені натри категорії: додавання поля, модифікація поля і видалення поля. Будь-яка зміна стану бази даних може вважатися очікуваною, неочікуваною або неконтрольованою. Якщо запис для зміни протоколюється, вона є очікуваною і стан бази даних після і до зміни може бути пізніше відтворений за журналом. Якщо запис певних змін принципово не ведеться, такі зміни вважаються неконтрольованими. Всі інші зміни є неочікуваними і можуть бути наслідком збоїв, зламу або помилкою імплементації системи.

Протоколювання змін ведеться для всіх простих типів полів. Для бінарних даних протоколювання має місце, тільки якщо їх розмір не перевищує певного визначеного розміру. Для текстових полів ведеться запис лише перших 1000 символів: вважається, що вони є найінформативнішими із всього контенту поля.

Реалізація

Основним каркасом у реалізації описаної архітектури є Spring Framework. Для реалізації конкретних прошарків або функціональних частин були застосовані додаткові каркаси.

Прошарок роботи з даними (Hibernate)

Одним з основних є прошарок роботи з даними. При виборі технології реалізації прошарку велику увагу було приділено незалежності від системи, в якій дані зберігаються, можливості чітко зіставляти бізнес-об'єкти системи з даними і швидкістю роботи. На момент вибору технології найкращою системою підтримки зберігання об'єктів і їх зв'язків, а також пошуку таких об'єктів, став Hibernate [3]. Він дає змогу моделювати наслідування, асоціацію і композицію об'єктів на реляційних базах даних, а також ефективно шукати об'єкти за допомогою як стандартного SQL, так і специфічного HQL (Hibernate Query Language). Істотною перевагою останнього є те, що запити на пошук об'єктів задаються в SQL-подібному вигляді й виконуються досить швидко.

Можливості Hibernate надають значних переваг для загальної системи, особливо при зберіганні таких об'єктів, як конкретні види документів, шо можуть бути дуже великими і відображатися на величезну кількість таблиць реляційної бази даних. У межах Hibernate можна просто викликати об'єкт, змінити необхідні параметри як будь-якого класу Java та записати модифіковані дані. Hibernate сам вирішує,

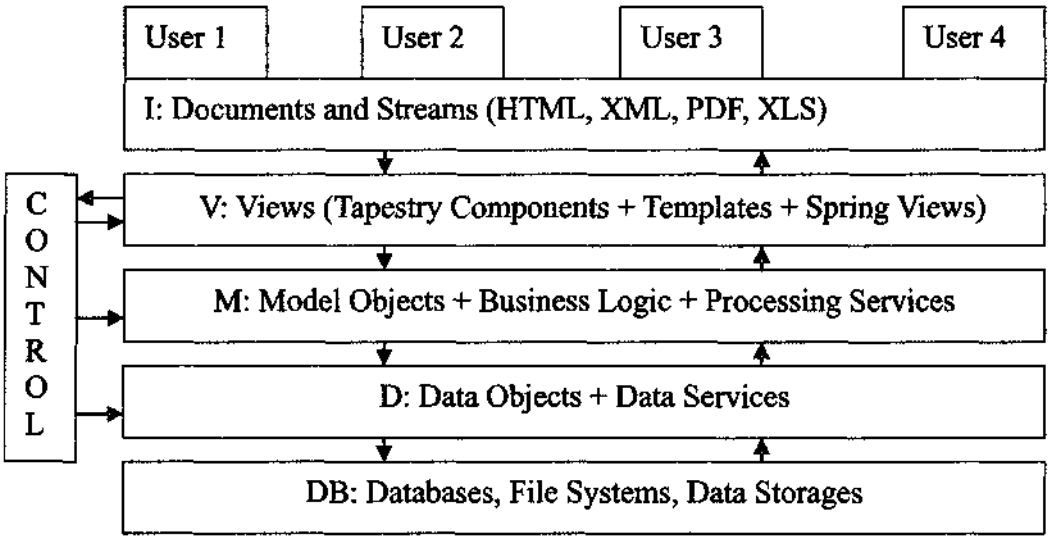

Рис. 3. Загальна модель системи

які реляції треба змінити, причому сама зміна робиться в контексті трансакції.

Як уже було зазначено, Spring Framework не реалізує конкретні засоби роботи з даними, але дає можливість використовувати будь-яку реалізацію, якою в нашому випадку є Hibernate. Таким чином, об'єкти, шо зберігаються (persistent), поєднуються задопомогою патерна DI, шо дає змогу, в свою чергу, конфігурувати такі об'єкти через зовнішні конфігураційні файли, зберігати SQL/HQL запити у зовнішніх конфігураціях тощо. Отже, об'єкти прошарку даних є повністю незалежними від інших об 'єктів, проте легко з ними пов'язуються.

Інтерфейс користувача

При розробці архітектури використовувався патерн MVC (Model-View-Control), що дало змогу досягнути незалежності від представлення будь-яких даних. Наприклад, одні й ті самі дані можуть бути відображені HTML-сторінкою, PDF-документом або SVG-діаграмою.

Основною концепцією при побудові інтерфейсу користувача було створення тонкого клієнта — веб-застосування, що надає такі переваги до системи АСУ НЗ: доступ з будь-якого місця Інтернету, відсутність необхідності інсталювати систему на робочих станціях, одночас - надоступність змін одразу всім користувачам, невибагливість до програмного забезпечення клієнтів та потужності клієнтських комп'ютерів, безкоштовність платформи для клієнтів, легкість інтерфейсів у використанні. Незважаючи на це, архітектура підтримує й інші види інтерфейсів (наприклад, повноцінний графічний інтерфейс або інтерфейс для мобільних пристроїв), і вони можуть бути використані поряд зі стандартним веб-інтерфейсом у разі потреби.

При розробці тонкого клієнта було вибрано каркас Tapestry [4], основною відмінністю якого порівняно зі схожими розробками (JSP, Velocity) є чітка компонентна структура і використання параметрів JavaBeans для задания конфігурації веб-компонентів. Це забезпечує низку переваг: чітке розділення представлення від коду, можчивість повторного використання всіх компонентів у Tapestry в будь-якому місці веб-застосування; підвищення можливості якісного тестування компонент-представления, що в умовах інших каркасів є майже недосяжним; додаткові переваги у захисті, а також можливість роздільної роботи команди програмістів та технічних дизайнерів.

Використання JavaBeans і компонентного підходу як основного в Tapestry дуже вдало інтегрується в каркас IoC Spring Framework, а отже, ми отримуємо додаткові переваги від їх поєднання. Як і з іншими компонентами, веб-компоненти Tapestry є незалежними і конфігуруються окремими конфігураціями, які потім виставляються контейнером Spring.

Каркас зв 'язування

Використавши Spring як основний каркас системи, що реалізує принцип IoC, ми змогли природно зв'язати додаткові каркаси роботи з даними, а також каркаси представлення. Як видно з рис. 3, різні прошарки системи передають інформацію в обох напрямах через відкриті інтерфейси, виділяється чіткий прошарок бізнес-логіки, і презентаційний шар напряму ніколи не працює з прошарком даних. Слід зазначити , що всі прошарки даних працюють уніфіковано, оскільки взаємодія проходить через визначений інтерфейс документа (Document), проте кожен прошарок обробляє тип контенту документа згідно з власними зобов'язаннями та правами.

Така уніфікована робота практично не застосовується в інших подібних системах АСУНЗ. Зазвичай шари систем працюють неуніфіковано, шар інтерфейсу викликає різні методи сервісів, а сервіси — потрібні їм методи шару роботи з даними. Такий архітектурний підхід знижує контроль над виконанням операцій, зумовлює розсіювання подібних функцій за багатьма класами і збільшує можливість помилок при обробці даних у різних контекстах.

Новизна розробленої архітектури полягає у тому, що поточний контекст задається типом контенту документа. Такий підхід забезпечує виконання важливої вимоги — робота з даними завжди відбувається відповідно до контексту, в якому вони збираються, обробляються і зберігаються.

Нефункціональні та інші вимоги

При побудові архітектури, а також реалізації системи велику увагу було приділено нефункціональним вимогам, архітектурні рішення для підтримки яких є ще однією перевагою АСУНЗ порівняно з подібними розробками.

Захист

У системі реалізовано модель захисту, що базується на ролі користувача. Кожна роль має

набір службових обов'язків (duties), які вона може виконувати. Кожен фізичний користувач може відігравати різні ролі у системі, тобто мати різні рівні доступу до різних частин системи. Наприклад, якщо користувач є і викладачем, і керівником магістерської програми одночасно, він зможе використовувати функціональністьсистеми, яка призначена для обох ролей.

Модель захисту функціонально розширює модель захисту платформи Java. Реалізується метод аутентифікації, шо використовується за замовчанням. Система легко підтримує й інші методи, наприклад використання сертифікатів (Kerberos, X. 509). Сесія підтримується за рахунок захищених cookies.

На рівні доступу до бази даних система захищена від прямого доступу до даних (доступ здійснюється тільки через проміжний шар). Кожному *об'єкту* дії користувача (вищезгадане duty) встановлюється необхідний доступ до бази даних на час виконання дії. Це означає, що відповідний користувач встановлюється не тільки для певної категорії дій бази даних з окремими правами, а навіть для заданої ситуації (яка дія, який користувач, у яких умовах виконується дія, що він виконував до того чи який стан системи на момент виконання дії). Використовується деперсоналізація користувача при доступі до даних. Доступ відбувається через використання попередньо визначеного набору користувачів у самій базі даних.

Застосовується також і непрямий захист системи. Використання журналу внесення змін і протоколювання всіх без винятку операцій у системі дають змогу чітко відстежувати намагання втрутитись у систему і повертати видалені або спотворені дані, лімітують навмисну зміну даних та несанкціонований доступ.

Розширюваність

Побудована на компонентному підході із застосуванням принципу IoC, система без принципових обмежень може бути розширена до рівня автоматизації будь-якого навчального закладу або будь-якої установи, де документообіг відіграє важливу роль, без обмеження функціональності поточних доробок. Найвразливішими до подібних змін є тільки інтерфейси користувача (вони можуть суттєво відрізнятися для кожного нового застосування системи) та специфічні для окремого випадку використання бізнес-об'єкти.

Структура бази даних та структура класів повністю відповідає реальним об'єктам світу і не вносить суттєвих обмежень на майбутню розширюваність системи. Основні ймовірні ускладнення системи можуть скоріше зумовити додавання нових структур і зміни ролі існуючих, аніж їх повну заміну і повторну розробку.

Підхід до обробки інформації в системі (через цикли документообігу) є стандартним для будь-якої системи документообігу, а тому система пристосована до розширення для підтримки більшості адміністративних функцій.

Масштабованість

Застосування гнучкої архітектури дає можливість розподіляти частини системи, будуючи розподілене застосування, що особливо важливо за великої кількості користувачів та інформації, що зберігається в системі. У такому випадку вдається вирішити проблеми піків навантаження, встановити пріоритети в обробці інформації та забезпечити безперервне функціонування системи. Опис наявних інтерфейсів у небінарному форматі може відкрити доступ до їх використання для будь-якої іншої системи за допомогою, наприклад, технології веб-сервісів.

Реалізацію цієї системи було протестовано в поточному варіанті щодо робіт із кількістю інформації, що в 10 разів перевищує заплановану завантаженість. При поточному режимі користування системою вона може працювати на одному сервері із задовільним часом відгуку.

Можливість тестування системи

Використання Spring Framework, чітке виділення інтерфейсів та підхід до уніфікованого інтерфейсу Document дає змогу покрити практично всю критичну функціональність unit-тестами за допомогою JUnit або подібних систем. Крім того, використовуючи Tapestry-каркас, ми досягай високого рівня можливості тестувати компоненти презентаційногорівня, шо в комплексі дає можливість повністю застосувати підхід паралельного тестування (test-driven approach) [5] при розробці системи. Такий підхід не тільки забезпечує стабільність системи в цілому, а й полегшує її розвиток і підтримку в майбутньому.

Функціональність системи

Першою реалізацією вищеописаної архітектури стала автоматизована система управління Магістеріуму Національного університету

«Києво-Могилянська академія» (АСУМ). Система складається з ряду автоматизованих робочих місць: секретаря приймальної комісії, методиста з навчально-методичної роботи, методиста управління навчальним процесом, керівника Магістеріуму, керівника магістерських програм, студента, викладача. Останні два АРМи надають тільки лімітовані можливості, проте планується їх розширення у майбутньому.

АСУМ повністю автоматизує процес роботи приймальної комісії, у тому числі реєстрацію, спрощену реєстрацію випускників НаУКМА, внесення результатів вступних випробувань, зарахування до складу студентів, генерацію звітів, а також всіх документів, що використовуються в процесі роботи, зокрема залікових відомостей, наказів про зарахування тощо.

Використовуючи АРМ, методисти Магістеріуму мають можливість працювати зі студентами та викладачами (застосовуючи розширені види пошуку), створювати групи, працювати з навчальними планами, вносити оцінки протягом сесій, генерувати всі необхідні звіти та документи, що використовуються у навчальному процесі.

Дуже потужною функцією АСУМ є автоматизація процесу складання розкладу, що, як відомо, є NP-повною великої розмірності. Втім, застосування генетичних алгоритмів надає можливість у більшості випадків розв'язати це завдання за прийнятний час. Підхід, застосований при розробці АСУМ, описано у 16].

Висновки і подальші дослідження

У результаті проведення науково-дослідної роботи було розроблено архітектуру, що враховує останні досягнення у побудові сучасних застосувань. Для її побудови було застосовано такі стандартні рішення: використання уніфікованого інтерфейсу для передачі даних між прошарками системи; визначення контексту документа згідно з його вмістом та обов'язками прошарку, що його обробляє. Розширене використання патерну IoC дало змогу створити гнучку, стабільну та зручну платформу для розробки АСУНЗ. Прикладом реалізації архітектури стала розробка системи автоматизованого управління Магістеріуму НаУКМА. Надалі планується уточнення результатів роботи та побудова на основі розробленої платформи системи управління НаУКМА.

1. *Martin Fowler.* Inversion of Control and Ihe Dependency Injection pattern (http://www.martinfowler.com/articles/ injection .html)
2. Spring Framework (http://www.springframework.org/)
3. Hibernate — Relational Persistence For Idiomatic Java (http:/ /www.hibernate.org/)
4. Tapestry (http://jakarta.apache.org/tapestry/)
. *Scott W. Ambler.* Introduction to Test Driven Development (http://www.agiledata.org/essays/tdd.html)
. *Медвідь С. О.* Комбінований компетентний паралельний генетичний алгоритм та його застосування для задачі побудови розкладів // Проблеми програмування.- Спеціальний випуск.- 2004.- № 2-3.- С. 261.

M. M. Glybovets, O. O. Krus', S. A. Ivashchenko

DEVELOPING THE SYSTEM FOR MANAGING EDUCATIONAL INSTITUTION

Ihe article includes description of the main development concepts used in design and implementation of the common system for managing educational institutions. The document describes the main architectural decisions, requirements to chosen tools and technologies, specific design decisions and combinations of tools andjrameworks, methods of satisfying non-functional requirements, ensuring security and extensibility ofthe system.