В.Заяць, Д.Іванов
Національний університет "Львівська політехніка"

УДК 621.372

ЗАСТОСУВАННЯ ШАБЛОНІВ ПРОЕКТУВАННЯ ДЛЯ ОПТИМІЗАЦІЇ СИСТЕМИ РОЗПІЗНАВАННЯ РУКОПИСНОГО ТЕКСТУ

Описано деякі шаблони проектування і продемонстровано, як їх можна застосувати при проектування програмної системи на прикладі системи розпізнавання рукописного тексту.

The article contains overview of design patterns. Also, the article shows how patterns should be used. As example is used system of recognizing of handwritten text.

1.Вступ

Під час проектування програмних засобів довільної складності неодноразово доводиться спостерігати ситуацію, коли один і той же розробник (або група розробників) створює ту чи іншу функціональність, яка була вже реалізована раніше. Отже, виникає проблема використання коду, який вже працює, є відлагоджений і відтестований, використовувався не один раз (як часто буває в реальному житті), а багато разів. Також виникають інші проблеми: як зробити код, що призначений для повторного вживання, більш зрозумілим для користувачів, більш ефективним і надійним, якомога простим та легкозмінюваним?

Саме для таких цілей застосовуються так звані патерні проектування.

Патерн – це набір класів, який іменує, абстрагує й ідентифікує ключові аспекти структури загального розв'язку, які і дозволяють застосувати його для створення дизайну, що може бути повторно використаний. Патерн визначає класи-учасники та їх екземпляри, їх роль та відношення, а також функції. При написанні кожного патерна увага концентрується на конкретній задачі об'єктно-орієнтованого проектування.

Виходячи з наведеного вище визначення патерна, можна зробити висновок, що він має бути реалізований лише за допомогою об'єктно-орієнтованого підходу до проектування систем, а отже, застосовувати та створювати (виділяти) патерни можуть лише ті розробники, які є досить досвідченими і кваліфікованими у цій галузі. Патерн сьогодні є вершиною процесу, який називається об'єктно-орієнтованим проектуванням.

2. Приклади патернів

Наведемо декілька прикладів патернів. Хоча їх кількість постійно зростає, та все одно є незначною, тим не менше цього достатньо для розв'язання майже всіх задач, що можуть виникнути в процесі створення тієї чи іншої програмної системи. Для підтвердження цього висновку наведемо такий приклад: для того, щоб отримати

сертифікат розробника від фірми *Sun Microsystems* (яка, до речі, і запропонувала шаблонний підхід до проектування), необхідно володіти всього 23 патернами [1,2]. Однак цього достатньо, щоб створювати майже довільну систему із урахуванням усіх вимог, що висуваються у всьому світі до програмних комплексів.

Наведемо декілька прикладів патернів:

1. *Factory* (фабрика) – надає інтерфейс для створення сімейств, що є пов'язані між собою, або незалежних об'єктів, конкретні класи яких є невідомими. Інша назва цього патерну є *Kit* (Інструментарій). В основному цей патерн використовується в таких випадках:
   a. Система не повинна залежати від того, як створюються, компонуються і зображаються об'єкти, з яких вона складається;
   b. Взаємопов'язані об'єкти, що входять у сімейство, повинні використовуватися разом, і розробнику необхідно забезпечити виконання цього обмеження;
   c. Система повинна конфігуруватися одним із сімейств об'єктів, що її складають;
   d. Розробник хоче надати бібліотеку об'єктів з розкриттям лише їх інтерфейсів, але не реалізації.
2. *Memento* (зберігач) – дозволяє, не порушуючи інкапсуляцію, отримати і зберегти у зовнішній пам'яті чи на будь-якому іншому носії інформації внутрішній стан об'єкта, щоби пізніше його можна було відтворити точно в такому ж стані. Області застосування:
   a. Необхідно зберігти миттєвий відбиток стану об'єкта (або його частини) для того, щоб пізніше відновити його;
   b. Коли пряме отримання цього стану розкриває деталі реалізації і порушує інкапсуляцію і цілісність даних об'єкта.
3. *Composite* – компонує об'єкти у деревоподібні структури для наведення ієрархій "частина–ціле". Дозволяє клієнтам однаково сприймати індивідуальні і складені об'єкти. Використовується:
   a. Коли необхідно навести ієрархію об'єктів вигляду "частина–ціле".
   b. Коли необхідно, щоб клієнти однаково трактували складені і індивідуальні об'єкти.
4. *Strategy* – визначає сімейство алгоритмів, інкапсулює кожний з них і робить їх взаємозамінними. Стратегія дозволяє змінювати алгоритми незалежно від клієнтів, що їх потребують. Використовується:
   a. Коли є багато споріднених класів, шо відрізняються лише поведінкою. Стратегія дозволяє сконфігурувати клас шляхом задання однієї із можливих поведінок;
   b. Коли необхідно мати декілька різних варіантів алгоритму. Наприклад, можна визначити два варіанти алгоритму (один вимагає більше часу для опрацювання, а інший – більше пам'яті). Стратегії дозволено застосовувати, коли варіанти алгоритмів реалізовані у вигляді ієрархії класів;
   c. Коли алгоритм містить дані, про які клієнт не повинен "знати". Стратегія використовується для того, щоб закрити складні, специфічні для алгоритму структури даних;

d. Коли в класі визначено багато поведінок, які подано розгалуженими умовними операторами. У такому випадку простіше перенести код з гілок в окремі класи стратегії.

5. *Mediator* (посередник) – визначає об'єкт, що інкапсулює спосіб взаємодії декількох об'єктів. Посередник забезпечує слабку зв'язаність системи, позбавляючи при цьому систему від необхідності явно робити посилання один на одного, що дає можливість незалежно змінювати взаємодію між ними. Застосовується:
   a. Коли є об'єкти, зв'язки між якими є складними і чітко визначеними. Взаємозалежності, що при цьому виникають, є неструктурованими і складними для розуміння;
   b. Коли неможливо повторно використати об'єкти, оскільки він обмінюється інформацією з багатьма іншими об'єктами;
   c. Поведінка, що розподілена між декількома класами, повинна піддаватися налаштуванню без породження великої кількості підкласів.

3. Застосування патернів для оптимізації системи розпізнавання рукописного тексту

Розглянемо патерн проектування *State Machine* [1] на прикладі системи розпізнавання рукописного тексту, алгоритм побудови якої наведено в роботі [3].

Патерн *State Machine* не був наведений вище, але він є одним із ключових патернів (належить до патернів поведінки). *State Machine* дозволяє чітко розділити весь процес роботи програми на певні етапи і спостерігати за тим, який етап після якого і при якій умові має виконуватися. Він чітко визначає, в яких станах може перебувати система та які події спричиняють зміну стану системи. Фактично цей патерн розроблений для того, щоб розділити великі частини коду на дрібніші, внаслідок чого код стає зрозумілим для його розробників та користувачів.

Шаблон *State Machine* маніпулює такими поняттями:

1. Стан – може бути початковим (лише один в системі), кінцевим та декілька робочих (їх може бути довільна кількість);
2. Подія – команда, після виконання якої система може перейти до іншого стану; Події також можуть містити:
   a. Умову – після її виконання здійснюється подія.
   b. Перехід – набір команд, який виконують після здійснення події при зміні стану системи.

На діаграмі (за специфікацією мови *UML – Unified Modeling Language*) події позначаються так: подія [умова] / перехід.

Отже, спробуємо навести у вигляді патерну *State Machine* процес розпізнавання рукописного тексту (для спрощення приклад буде грунтуватися на процесі розпізнавання тільки однієї літери рукописного тексту).

Спочатку необхідно виділити стани, в яких може перебувати система:

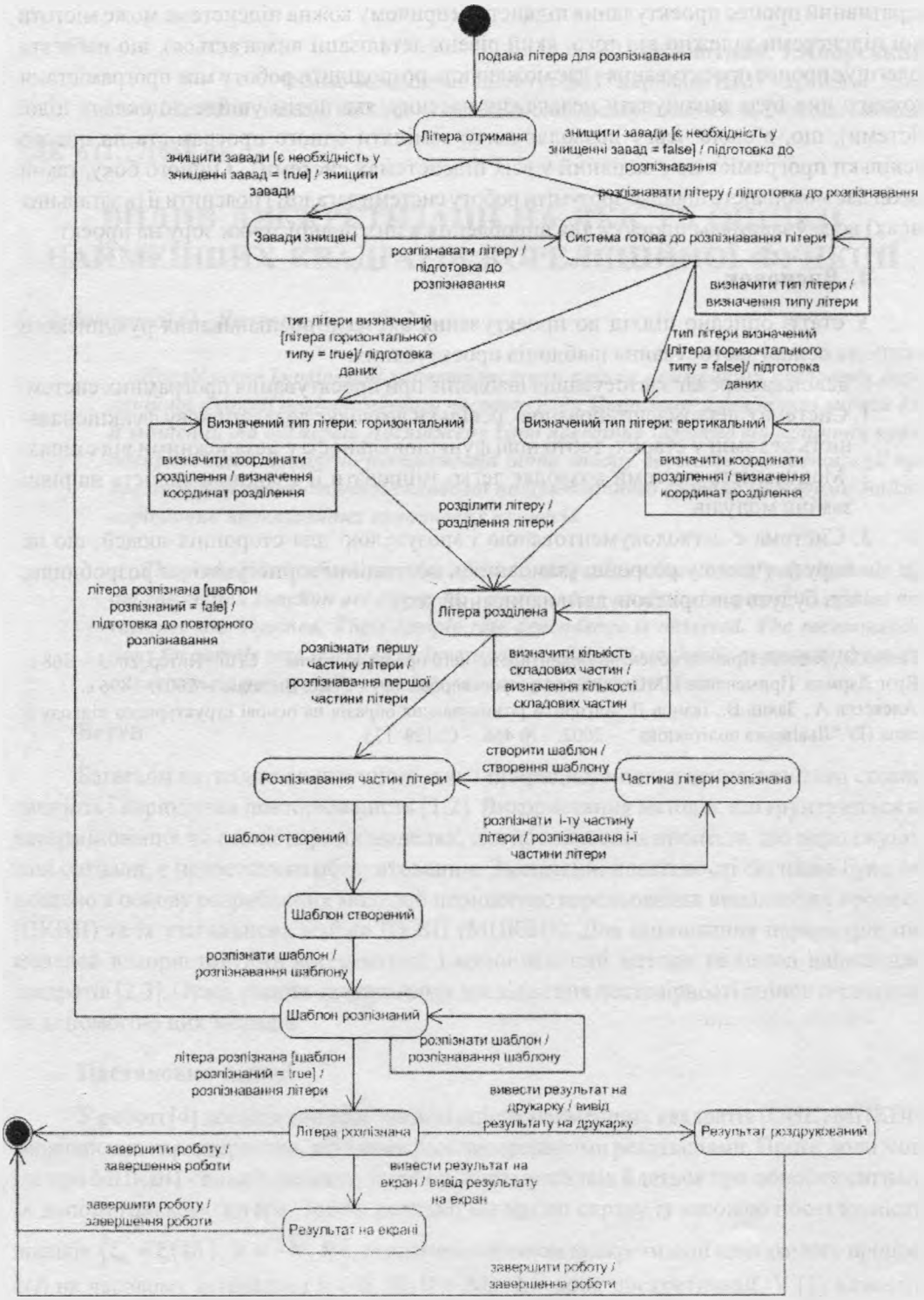
1. "*Початковий стан*" – стан, коли ще не було зроблено ніяких подій. Система знаходиться в очікуванні;
2. Коли система отримала на вхід літеру, вона переходить до стану "*Літера отримана*".

3. Після очищення літери від завад (при необхідності усунення завад) система переходить або до стану "*Завади знищені*" (з цього стану система перейде за окремою командою до стану "*Система готова до розпізнавання літери*". Стан "*Завади знищені*" виділений для того, щоб розробник мав можливість написати свою певну логіку після того, як завади будуть знищені. Якщо цієї логіки не потрібно, то стан вважається рудиментним і підлягає знищенню), або до стану "*Система готова до розпізнавання літери*".
4. Після визначення типу розпізнаваної літери система переходить або до стану "*Визначений тип літери: горизонтальний*" або "*Визначений тип літери: вертикальний*". Можна було б об'єднати ці два стани в один (наприклад, "*Визначений тип літери*"), але це б внесло у системи додаткові перевірки, що не є бажаним при застосуванні патерна *State Machine* (який, власне, і призначений для мінімізації кількості блоків *if ... then ... else*). Крім того, якщо в майбутньому створиться ще один тип літери, система не буде значно змінюватися, оскільки до неї буде додано один стан (новий код), а старий код не зміниться.
5. Після стану визначення типу літери система переходить до стану "*Літера розділена*", де і будуть визначатися координати розділювальної лінії.
6. Після цього система вже готова для розпізнавання кожної частини розділеної літери. Отже, система перейде до стану "*Розпізнавання частин літери*".
7. Після розпізнавання окремих частин літери система створить шаблон розпізнаваної літери і перейде до стану "*Шаблон створений*".
8. Потім система має розпізнати шаблон і визначити за ним, яку із літер було подано на розпізнавання. Отже, необхідно визначити ще один стан: "*Шаблон розпізнаний*". Якщо нема потреби додавати певну логіку, що обробляє подію розпізнавання шаблону, то можна відразу перевести систему у стан "*Літера розпізнана*" ("*Або літера не розпізнана*"). Цей стан є останнім і може містити переходи у початковий ("*Літера отримана*") або у кінцевий ("*Вивід інформації на екран, на друк та ін.*", "*Завершення роботи*" та інше.).

Зробимо спробу зобразити те, що написано вище, за допомогою діаграми станів, яка буде відображена умовними позначеннями мови проектування *UML*. На рисунку зображено діаграму *State Machine* для процесу розпізнавання літери.

Наведена схема дає змогу побачити, в яких станах може перебувати система, які події можуть виникнути в системі, яка реакція її на ці події. Таке відображення процесу опрацювання літери дає можливість визначити, які класи будуть брати у ньому участь, які функції (методи) мають бути визначені для коректної роботи. Також за допомогою схеми можна без зайвих труднощів долучати елементи (як стани, так і події), причому відразу стає зрозумілим, до якого методу чи класу необхідно внести певні корективи. Тобто, патерн *State Machine* дає нам можливість оцінити роботу системи загалом без зосередження на реалізації.

Наведений приклад продемонстрував, як можна відобразити поведінку системи, а патерн Mediator (посередник) дає можливість зробити акцент на взаємодії таких систем. Отже, при наявності детального опису роботи кожної підсистеми можна описати роботу системи загалом. Отже, проектування складних систем перетворюється в



Діаграма State Machine процесу розпізнавання літери

ітеративний процес проектування підсистем (причому кожна підсистема може містити свої підсистеми залежно від того, який рівень деталізації вимагається), що набагато полегшує процес проектування і дає можливість розподілити роботу між програмістами (кожен з них буде виконувати незалежну частину, яка потім увійде до складу цілої системи), що, в свою чергу дозволяє легко замінять одного програміста на іншого (оскільки програміст не є задіяний у всіх підсистемах системи). З іншого боку, такий підхід дає можливість швидко зрозуміти роботу системи загалом і пояснити її (в загальних рисах) всім учасникам проекту для вироблення в них певної точки зору на проект.

4. Висновок

У статті описано підхід до проектування системи розпізнавання рукописного тексту на основі застосування шаблонів проектування.

Є декілька переваг застосування шаблонів при проектування програмних систем:

1. Система є легкомасштабованою, оскільки дозволяє додавати нову функціональність без змін у старих, тобто нові функціональності є незалежними від старих.
2. Модульність системи дозволяє легко змінювати її функціональність на рівні заміни модулів.
3. Система є легкодокументованою і зрозумілою для сторонніх людей, що не беруть участь у розробці (замовники, потенційні користувачі чи розробники, які будуть використовувати написаний код).

1. Гамма Э., Хелм Р. Приемы объектно-ориентированного проектирования. - СПб.: Питер, 2003. – 368 с.
2. Крэг Ларман. Применение UML и шаблонов проектирования. – СПб.: Вильямс. – 2003. – 496 с.
3. Алексєєв А., Заяць В., Іванов Д. Алгоритм розпізнавання образів на основі структурного підходу // Вісник НУ "Львівська політехніка": – 2002. – № 468. – С. 129–133.