МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КПІ»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

## Курсова робота

## з дисципліни «Об’єктно-орієнтоване програмування»

на тему

**Шаблони проектування в ООП. Синтаксичний аналізатор коду мов С/С++**

Виконав студент

II курсу КП-01

Крамаренко Олексій Андрійович

Залікова книжка КП-0110

Керівник роботи

ст.викладач, к.т.н. Заболотня Т.М.

Оцінка

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**ЗМІСТ**

**ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ………3**

**ВСТУП……………………………………………………………………………….4**

1. **Структурна організація програмного забезпечення………...…….……5**

**1.1 Модулі програми………………………………………………………...….5**

**1.2 Діаграма класів……………………………………………………...…….11**

1. **Програмна реалізація за допомогою шаблонів проектування…….........................................................................................12**

**2.1 Обґрунтування вибору та опис шаблонів проектування для програмної реалізації синтаксичного аналізатору програм на мовах С/C++……………………………………………………………………………12**

**2.3 Результати роботи, тести…………………………………...…………….17**

**ВИСНОВКИ………………………………..…………………….………………..19**

**СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ……………….………………...20**

**ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ**

Парсер(Parser) — об’єкт, що розбиває текст на Токени.

Токен(Token) — одиниця інформації синтаксичного аналізатора.

Інтерпретатор(Interpretor) — об’єкт, що перетворює Токени на лексеми.

Лексема(Lexem) — абстрактна одиниця інформації Лексера.

Лексер(Lexer) — об’єкт, що перетворює лексеми на правила.

Правило(Rule) — одиниця інформації для Обхідника, що перетворює її на команди та виконує їх.

Закон(Law) – сукупність правил, за допомогою яких Обхідник виконує команди над правилами.

Обхідник(Walker) — останній в ланцюжку об’єкт, що виконує команди, які є результатом роботи програми.

В подальшому на діаграмах класів **жирним** шрифтом будуть позначенні статичні поля або методи. *Курсивом* – віртуальні методи.

**Вступ**

Дана курсова робота присвячена розробці програмного забезпечення для аналізу початкового коду програм, на мові С/С++. Дана тематика обрана для виконання курсової роботи тому, що результати абстрагування об’єктів у цій предметній галузі дозволяють застосовувати всі вивчені принципи та методи об’єктно-орієнтованого проектування для створення програмного забезпечення (ПЗ), зокрема шаблони проектування.

*Об’єктом* дослідження є процес аналізу початкових кодів.

*Метою роботи* є розроблення ПЗ для аналізу початкових кодів з використанням шаблонів проектування.

Для досягнення визначеної мети необхідно виконати такі *завдання*:

* Абстрагувати об’єкти предметної галузі;
* Розробити структурну організацію ПЗ за допомогою застосування основних принципів ООП та шаблонів проектування;
* Визначити та описати функціональні характеристики програми;
* Розробити дизайн інтерфейсу користувача;
* Виконати тестування розробленого ПЗ;
* Оформити документацію з КР;

Реалізовані шаблони: Фасад, Посередник, Пристосуванець, Стан, Стратегія, Команда.

До функціональних можливостей ПЗ входить: розбір початкового коду програми на елементи, їх перетворення на лексеми, обробка лексем та створення на їх основі правил.

Для функціонування розробленої програми необхідно забезпечити на комп’ютері 50 Кб вільного дискового простору.

**1**. **СТРУКТУРНО – АЛГОРИТМІЧНА ОРГАНІЗАЦІЯ ПРОГРАМИ**

* 1. **Структурна організація програми**

**Модулі програми.**

* Parser
* Interpret
* Lexer
* OpenLexer
* ClosedLexer
* Lexem
* Walker
* Rule
* Law
* SimpleLaw
* ComplexLaw

На схемі, зображеній нижче, показаний ланцюжок обробки файлів на Правила. На ній не показані модулі Law\*, Rule, OpenLexer, ClosedLexer, бо ці модулі є об’єктами і вони не приймають участь у цьому ланцюжку.



Рис. 1 (Схема модулів програми)

* + 1. **Parser**

Парсер — перший і єдиний клас до якого може бути доступ ззовні. Він виражає собою шаблон Фасад, адже він ховає за собою реалізацію та інтерфейси усіх інших класів, що не доступні для користувача.

Парсер має багато полів, для зберігання тимчасових результатів його роботи. Такі поля як: **\_in, \_file\_names, \_interpret, \_word, \_words, \_line** є закритими(тобто доступними тільки безпосередньо з класу Parser. Вони використовуються для зберігання інформації про тільки що оброблений рядок та про файли, які потрібно оброблювати. **\_in —** це потік, що зв'язаний з поточним файлом, з якого йде читання. **\_file\_names —** масив, що має у собі імена усіх файлів, з яких має відбуватися читання та їх синтаксичний розбір. **\_interpret —** вказівник на об'єкт Інтерпретатора, котрий наступний в черзі на виконання операцій над файлом. **\_word, \_words, \_line —** мають у собі відповідно поточне слово, масив усіх слів в рядку та поточний рядок.

Парсер має у собі багато методів, для роботи с Токенами та їх розбиттями, але усі вони закриті для доступу ззовні. Для користувача доступні методи **add\_file, set\_law, set\_lexer, process\_file, process\_files.** Ці методи дають змогу подати файл до Парсера. Подати спосіб обробки правил обхідником. Подати Лексер, яким будуть потім оброблюватися Токени. Обробити конкретний файл. Та обробити усі файли, що були подані до Парсера.

У собі Парсер має покажчик на Інтерпретатор, тобто на наступний етап аналізу коду. Парсер є не тільки Фасадом, але й Посередником, адже, як буде показано далі, допомагає різним об'єктам взаємодіяти, не маючи безпосередній зв'язок між собою.

**1.1.2. Interpret**

Цей модуль перетворює Токени, що до нього поступають з Парсера та передає їх у вигляді лексем до Лексера. Від Парсера надходять прості слова, а інтерпретатору треба з них зробити лексеми — об'єкти які мають у собі не тільки слово-Токен, а ще й специфічну інформацію. Зважаючи на те, що конструктор лексем досить важкий(за рядком треба створити лексему, а для цього треба виконати багато операцій порівняння між строками), було вирішено намагатися якнайменше його використовувати. Для цього використано шаблон Пристосуванець. Спочатку йдеться пошук на вже створений об’єкт Лексема, з масиву усіх до цього все створених лексем, а вже потім, якщо його нічого не вдалося знайти — створюється новий об'єкт і додається до вже існуючого масиву лексем.

Ще цей клас виступає посередником в переданні команд між Парсером(фасадом, або головною програмою) та Walker-ом або Lexer-ом для яких ці команди визначалися. Робить він це перенаправляючи методи **set\_law** та **set\_lexer** на Лексер далі.

**1.1.3 Lexer**

Лексер, отримуючи лексеми з інтерпретатора, аналізує їх, і потім — передає далі у вигляді правил до обхідника. Через те, що лексеми аналізуються не по одній, а групами, було створено ієрархію класів, з абстрактним родичем Lexer и двома його наслідуючими класами OpenLexer та ClosedLexer. У Лексера може бути 2 стани: відкритий та закритий. Тобто такий Лексер, що все ще приймає лексеми, бо в нього недостатньо даних, щоб створити правило, та такий Лексер, що все зібрав достатньо лексем і займається їх аналізом для подальшого перетворення у правило. Ця сукупність класів відображає шаблон Стан, адже в цьому випадку у Лексера може бути ці 2 стани.

У Лексері також є 2 статичних поля **\_interpret** та **\_law**, які мають у собі відповідно носильник на єдиний для всіх Лексерів(адже Лексер дуже часто змінює свій стан) Інтерпретатор, та закон, який передасться(як шаблон Команда) до Walker-а і буде ним використовуватися у шаблоні Стратегія.

Відкритий Лексер при настанні певної умови(потрібна кількість лексем отримана і вже можливо їх аналізувати єдиним чином) переходить у закритий і передає йому усі накопичені лексеми.

Закритий Лексер віддає створені на основі даних з лексем правила до Обхідника, і, маючи доступ до статичного(а значить і єдиного) поля Інтерпретатора, змінює стан його Лексера знову на відкритий. Далі він запускає функцію Обхідника для обробки за поданими правилами(Law) усіх надісланих лексем.

**1.1.4. OpenLexer**

Це стан Лексера, у якому він чекає на наступні лексеми, бо йому недостатньо даних для аналізу лексем та прийняття рішення. У цьому стані Лексер може приймати лексеми за допомогою метода **add\_lexem**, та аналізувати ситуацію методом **analize**. Ситуація аналізується після кожного виклику методу **add\_lexem**, адже можливо тільки що була додана остання потрібна лексема. Тоді при проходженні усіх перевірок у методі **analize()** Лексер може змінити стан на ClosedLexer.

**1.1.5. ClosedLexer**

****У цьому стані Лексер перебуває недовго. Лише той час, що потрібен для перетворення лексем на правила, їх передачі до обхідника та його відпрацювання. Після усіх цих завдань, Лексер знову стає відкритим. Усі минулі лексеми видаляються і він готовий знову сприймати лексеми від інтерпретатора.

* + 1. **Lexem**



Клас, що відображає саму лексему. Поки що він досить простий, і не несе в собі того важкого значення, котре може бути додане до нього далі. При подальшій зміні та збільшенню цього класу, головне триматися інтерфейсу, тоді такі зміни не потягнуть за собою зміну інших класів. Поки що лексема в собі має лише слово, що передане їх з інтерпретатора, і яке вона передасть до обхідника.

* + 1. **Walker**

****Обхідник. Клас, котрий за правилами, що йому передані Лексером, та за деякою реалізацією класу Law виконує завдання над правилами. У цьому класі реалізований шаблон Стратегія, адже різні кроки алгоритму роботи над правилами можуть бути задані різними класами, нащадками від Law.

* + 1. **Rule**

****Клас, що відображає собою правило. В загальному випадку — це може бути все, що завгодно, адже правила роботи над файлами нічим не регламентуються. У разі ж роботи з початковими кодами програм, правила являють собою перетворення одних рядків коду в інші. Щоб не сильно ускладнювати роботу, в програмі реалізовані тотожні правила. Тобто то, що не змінюють рядки, що поступають. У подальшому розвитку програми є великий потенціал, якщо розроблювати та ускладнювати саме цю частинку програми, адже саме за цими правилами виконуєтсья все перетворення у програмі.

* + 1. **Law**

****Це абстрактний клас, котрий дозволяє змінювати реалізацію роботи над правилами обхідника, не змінюючи середину цих класів. Він є базовим для 2-х класів, тобто можлива поведінка 2-х типів. Просто виводити на екран. Та виводити в файл. Змінна, що відповідає за те, як виводити інформацію, тобто що відповідає за поведінку є статичною і зберігається в Лексері. Це оптимальне місцеположення, адже с одного боку стає можливий доступ з боку Парсера, а з іншого — саме в класі Парсера відбувається роздвоєння(наслідування) на 2 інші класи, тому найкраще зберігати такий об’єкт саме тут.

**1.1.10. SimpleLaw**

****Це однин з 2-х реалізованих Законів роботи Обхідником Правилам. Цей клас, так само, як клас ComplexLaw, являє собою об’єкт команди, який передається Walker-у.

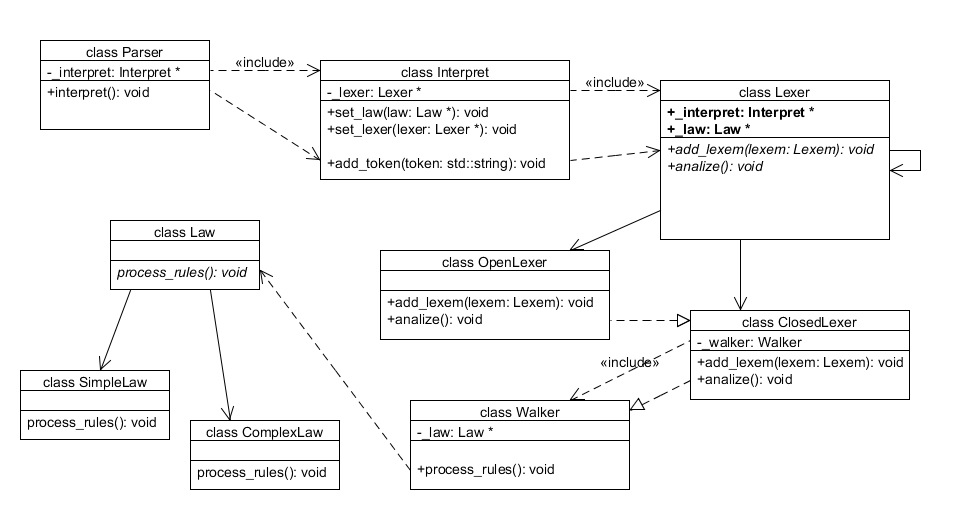
За алгоритмом, який інкапсулює цей клас, Правила просто виводяться на монітор.

Саме цей клас використовується Walker-ом за замовченням, якщо навмисно його не замінювати.

**1.1.11. ComplexLaw**

Цей клас так само, як і клас SimpleLaw просто реалізує алгоритм обробки Законів Обхідником.

* 1. **Діаграми класів**

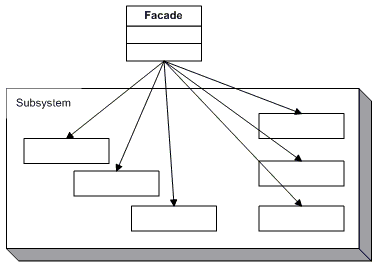


**2. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ЗА ДОПОМОГОЮ ШАБЛОНІВ ПРОЕКТУВАННЯ**

**2.1. Обґрунтування вибору та опис шаблонів проектування для програмної реалізації синтаксичного аналізатору.**

**2.1.1. Фасад + Посередник**

**Структура.** Фасад.

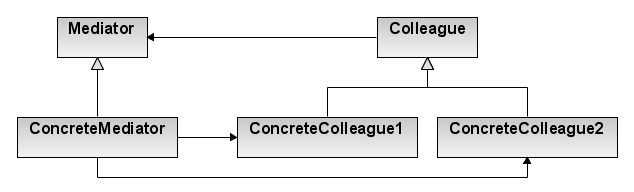
Фасад – шаблон проектування, що спрощує доступ до декількох класів, або об’єктів, використовуючи для цього один інший об’єкт.

У даній курсовій для цього використаний клас Parser, який інкапсулює у собі доступ до всіх інших класів(Interpret, Lexer, Walker). Це спрощує використання усієї ієрархії класів для користувача так як йому потрібно знати лише інтерфейс Фасаду, для користування усією системою.

**Обґрунтування використання шаблону.**

Велику систему класів корисно інкапсулювати в одному класі, для спрощення її використання. Вся робота програми для користувача зводиться до додавання файлів до Парсера(тут – це назва самої програми, а не одного конкретного класу) та їх подальший синтаксичний розбір. Тому було створено відповідний інтерфейс для Парсера і реалізовано його. Зараз через парсер користувач може делегувати роботу усієї програми, тому це найкраще місце для шаблону Фасад.

**Структура.** Посередник.



Єдиний об’єкт посередник використовується іншими класами для взаємодії між собою. Вони навіть не знають інтерфейси своїх сусідів, вони не здатні взаємодіяти між собою без посередника. Цей єдиний клас допомагає всім іншим передавати данні та визивати методи один одного не посилаючись безпосередньо один на одного.

**Обґрунтування використання шаблону**

Загалом структура класів у програмі являє собою ланцюжок, але є необхідність інколи мати доступ з одного класу до іншого, наприклад, щоб змінювати стан Лексера, або змінювати значення внутрішніх полів інших класів. Було вирішено не нагромаджувати цю взаємодію між усіма класами, а, за допомогою шаблону Посередник, вивести усю цю взаємодію у єдиний клас, загальний для всіх. Було обрано як базовий для Посередника клас – Парсер, адже він ще й Фасад, а цей шаблон дуже схожий за типом з Посередником.

**Опис зв’язки шаблонів**:

Обидва шаблони сконцентровані всередині класу Parser. У цьому класі сконцентровані 2 шаблони проектування: Фасад та Посередник. Вони досить схожі між собою, адже вони обидва працюють з багатьма класами і спрощують структуру доступу.

**2.1.2. Стратегія + Команда**

**Структура.** Стратегія.



У структурі Стратегія приймають участь Обхідник, та Закони. Закони унаслідуванні від абстрактного класу Law, з різними реалізаціями функції **process\_rules(rule: std::vector<Rule>)**. Для того, щоб легко було змінювати реакцію Обхідника можна передавати йому різну реалізацію алгоритмів, що містять у собі класи Law.

Цей шаблон був використаний у зв’язку з тим, що була необхідність легкої зміни алгоритму(стратегії) обходу Обхідника, для зміни мову, на яку можуть транслюватися Лексеми з початкових кодів.

**Структура.** Команда.

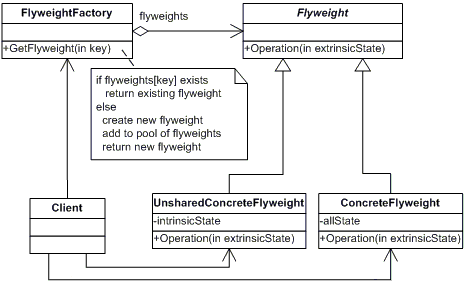
Цей шаблон інкапсулює в собі дію та її параметри. Ієрархія класів Law, SimpleLaw, ClosedLaw як раз реалізує ці можливості у зв’язці з шаблоном Стратегія. У відповідності до структури шаблону Команда, клас Law має у собі абстрактний метод **process\_rules()**, реалізуючи який, декількома різними алгоритмами є можливість змінювати поведінку Обхідника, у зв’язку з різними об’єктами Команди.

Шаблон Команда був використаний для того, щоб відокремити інтерфейс роботи Walker-а, від різних конкретних реалізацій його роботи. Реалізації зберігаються у об’єктах типу Law(та конкретних реалізацій цього абстрактного класу).

**Опис зв’язки шаблонів**:

Стратегія використовує об’єкт Команди, бо Стратегія має змінювати алгоритм в залежності від отриманих Команд, а об’єкт Команда має у собі конкретну реалізацію алгоритму, достатньо для роботи Стратегії.

**2.1.3 Пристосуванець**

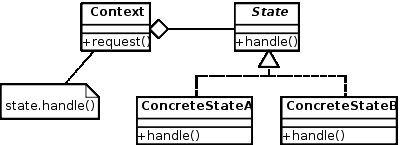


Пристосуванець створений, щоб не створювати зайвих сутностей. Для цього усі вони зберігаються в одному пулі, в якому спочатку й йдеться пошук об’єкту. Тільки за умови, що він не знайдений, визивається конструктор створення нового об’єкту.

**Обґрунтування використання шаблону**

Об’єкти класу Lexem мають дуже важкий конструктор, тому було вирішено знизити кількість їх викликів якнайбільше. Для цього був використаний шаблон Пристосуванець, що зменшує кількість викликів конструктора. Цей шаблон реалізований в класі Interpret.

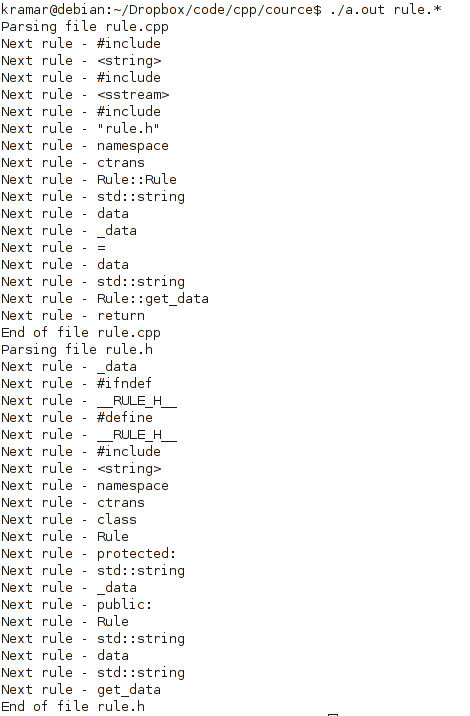
**2.1.4 Стан**



Цей шаблон використовується, коли об’єкт може мати декілька станів, і повинен мати можливість змінювати свій стан, не змінюючи інтерфейсу. У даній курсовій роботі клас Lexer має декілька станів(OpenLexer, ClosedLexer). Спочатку об’єкт ініціалізований станом OpenLexer, у якому він отримує Лексеми і постійно їх аналізує на завершеність. Якщо список закінчений, то Лексер змінює свій стан на ClosedLexer, у якому він передає Лексеми у вигляді Правил до Обхідника.

**2.2 Результати роботи**

Програма написана для використанні у командному рядку. Програма чекає на вході імена декількох файлів, вони усі будуть подані до Парсера, який потім інтерпретує зміст кожного з них.



При поданні неіснуючого імені файлу або ім’я бінарного файлу, то ця частина не буде виведена на екран, а буде виведена інформація про те, що трапилася помилка.



**ВИСНОВКИ**

Метою даної курсової роботи було розроблення програмного забезпечення для симуляції керування садом з використанням шаблонів проектування. Підставою для розроблення стало завдання на виконання курсової роботи з дисципліни «Об’єктно – орієнтоване програмування» студентами II курсу кафедри ПЗКС НТУУ «КПІ».

Для досягнення поставленої мети, виконано всі цілі, що стояли переді мною для успішної розробки даного програмного забезпечення. Реалізовано всі вимоги до програмного продукту, програмного та апаратного забезпечення, створено відповідну документацію.

Розроблене програмне забезпечення дозволяє синтаксично розбирати початкові коди на мовах C/C++.

Програму створено на основі шаблонів проектування: зокрема до структури розробленого програмного забезпечення входить реалізація семи шаблонів у тісних зв’язках, які належать до різних груп шаблонів проектування (структурних, породжувальних та поведінкових).

Для розроблення програмного забезпечення використані мова програмування С++.

Подальшою перспективою дослідження даної тематики є розробка даного програмного забезпечення зі збільшеними можливостями парсінгу, додання шаблону проектування Інтерпретатор, додання ще одну реалізацію абстрактного класу Law, для транслювання Лексем в правила на інших мовах, можливо, і на синтаксично правильну англійську мову.

**СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ**

1. [http://uk.wikipedia.org](http://uk.wikipedia.org/)
2. [http://msdn.microsoft.com](http://msdn.microsoft.com/)
3. [http](http://sourcemaking.com/design_patterns/observer/cpp/3)://sourcemaking.com/design\_patterns/
4. Э. Гамма Р. Хелм Р. Джонсон Дж. Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования».