**Лабораторна робота №7.**  **Архітектура програмного забезпечення.Застосування шаблонів проектування.** (2 год.)

**Мета роботи**: засвоєння навичок використання шаблонів проектування для прискорення і підвищення надійності конструювання програмного забезпечення

**Завдання:**

1. Ознайомитися з теоретичною частиною
2. Розглянути приклади застосування шаблонів проектування на сайті "Патерни проектування. Каталог **C++** прикладів" <https://refactoring.guru/uk/design-patterns/cpp>
3. Запропонувати та обґрунтувати 2-3 шаблони проектування для реалізації компонентів ПЗ, що розробляється за вашим проектом.
4. Прокоментувати переваги і, якщо такі є, недоліки використаних шаблонів проектування для зазначених проектних рішень.

По закінченню лабораторну роботу потрібно здати на перевірку викладачеві, надіславши електронною поштою на адресу [t.i.lumpova@gmail.com](mailto:t.i.lumpova@gmail.com). Якщо викладач знаходить помилки чи неточності, він може повернути роботу на доопрацювання.

Файл з роботою повинен мати назву в такому форматі:

**KPZ <Номер групи><Номер лекції / лабораторної> [літера позначення типу роботи L – лекція, R – лабораторна]<Прізвище англійською>**.. Наприклад, **KPZ4101R**buts.doc.

Не копіюйте фрагментів з різних інформаційних джерел, подумайте і викладіть свою точку зору. При наявності робіт-"близнюків" відповідь буде зараховуватися першому за часом надсилання.

Тему в заголовку листа записати

**KPZ<Номер групи>-ЛР<Номер лабораторної>-<Прізвище >**

**Строк виконання цієї роботи ІПЗ- ІПЗ-41,42,43 –**

Всі запитання, що виникнуть, надсилайте на електронну адресу викладача, Тему в заголовку листа записати

**KPZ <Номер групи>-Запитання-<Прізвище >**.

**КОНТРОЛЬНІ ПИТАННЯ:**

1. Що таке шаблон проектування?
2. Які групи шаблонів вам відомі?
3. Опишіть основні характеристики шаблону FAÇADE.
4. Які переваги та недоліки, на Вашу думку, має шаблон проектування FACADE?
5. Опишіть основні характеристики шаблону Adapter.
6. Які переваги та недоліки, на Вашу думку, має шаблон проектування Adapter?

**Теоретичні відомості:**

В розробці ПЗ, шаблон проектування або паттерн – повторювана архітектурна конструкція, що представляє собою рішення проблеми проектування в рамках деякого контексту.

Зазвичай, шаблон не є кінцевим зразком, який може бути відразу перетвореним в код; це лише приклад вирішення задачі, який можна використати в різних ситуаціях. Об’єктно-орієнтовані шаблони показують відношення і взаємодію між класами або об’єктами, без визначення того, які кінцеві класи або об’єкти додатку будуть використовуватися.

«Низькорівневі» шаблони, що враховують специфіку конкретної мови програмування називаються *ідіомами*. Це хороші рішення проектування, характерні для певної мови або програмної платформи, і тому не універсальні.

На найвищому рівні беруть участь архітектурні шаблони, вони включають архітектуру усієї програмної системи.

Алгоритми по своїй суті також являються шаблонами, але не проектування, а розрахунку, саме через те, що вирішують різноманітні задачі. Головна користь кожного окремого шаблона полягає в тому, що він описує рішення цілого класу абстрактних проблем. Також той факт, що кожен шаблон має своє ім’я, полегшує дискусію про абстрактні структури даних (ADT) між розробниками. Так як вони можуть посилатися на відомі шаблони. Таким чином, за рахунок шаблонів виконується уніфікація термінології, назв модулів та елементів проекту. Правильно сформульований шаблон проектування дозволяє, відшукавши вдале рішення, користуватися ним знову і знову. Але вони і можуть пропагувати і погані стилі розробки додатків. Для подолання цих недоліків використовується рефакторинг.

Кожен з шаблонів допомагає розв'язати певну проблему, що виникає при об'єктно-орієнтованому аналізі, і яка виникає практично в будь-якому проекті з розробки програмного забезпечення. Однак іноді шаблони консервують громіздку і малоефективну систему понять, розроблену вузькою групою. Коли кількість шаблонів зростає, перевищуючи критичну складність, виконавці починають ігнорувати шаблони і всю систему, з ними пов'язану. Нерідко шаблонами замінюється відсутність або недоліки документації, яка складна програмному середовищі. Є думка, що сліпе застосування шаблонів з довідника, без осмислення причин і передумов виділення кожного окремого шаблону, уповільнює професійне зростання програміста. Люди, які дотримуються цієї думки, вважають, що знайомитися зі списками шаблонів треба тоді, коли "доріс" до них в професійному плані - і не раніше. Хороший критерій потрібного ступеня професіоналізму - виділення шаблонів самостійно, на підставі власного досвіду. При цьому, зрозуміло, знайомство з теорією, пов'язаної з шаблонами, корисно на будь-якому рівні професіоналізму і направляє розвиток програміста в правильну сторону. Сумніву піддається тільки використання шаблонів "за довідником". Шаблони можуть пропагувати погані стилі розробки додатків, і часто сліпо застосовуються.

*Типи шаблонів проектування*:

1. ***Твірні шаблони*** або ***породжуючі шаблони*** (англ. *Creational patterns*) – шаблони проектування, які абстрагують процес побудови об'єктів. Вони дозволяють зробити систему незалежною від способу створення, композиції і представлення об’єктів. Шаблон, який породжує класи, використовує успадкування щоб варіювати створюваний клас, а шаблон, що породжує об’єкти, делегує інстанціювання іншому об’єкту.

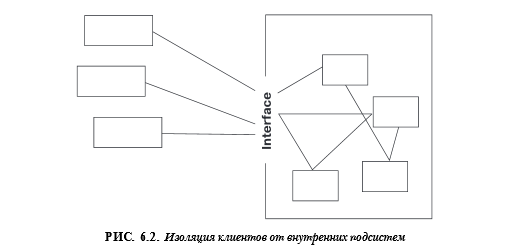
**Інстанціювати** — термін з програмування: створити конкретний об'єкт за зразком абстрактного шаблону. Визначає інтерфейс для створення об'єкта, але залишає підкласам рішення про те, який саме клас **інстанціювати**. **Інстанціювати** = **створити примірник, від** англ. instance - примірник, приклад, копія, екземпляр

1. **Структурні шаблони** – визначають різноманітні складні структури, які міняють інтерфейс вже існуючих об’єктів або його реалізацію, дозволяючи полегшити розробку і оптимізувати програму.
2. **Поведінкові шаблони** – визначають взаємодію між об’єктами, підвищуючи таким чином його гнучкість.
3. **Шаблони паралельного програмування** – використовується для більш ефективного написання багатопоточних програм, і надає готові рішення проблем синхронізації.

Також, станом на нинішній день, існує ряд інших шаблонів:

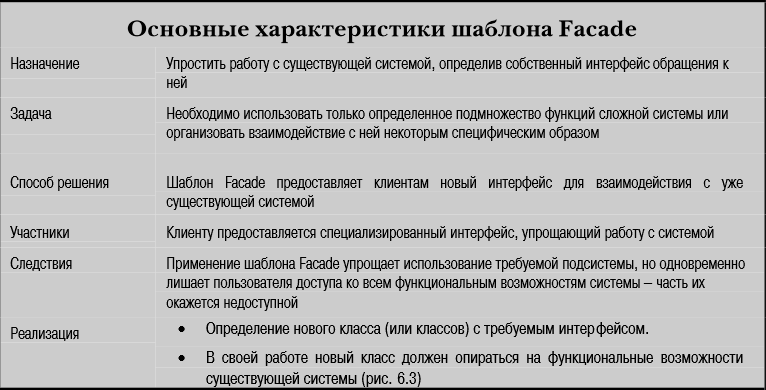
* ***Carrier Rider Mapper*** описують надання доступу до інформації що зберігається
* ***Аналітичні шаблони*** описують основний підхід для створення вимог для програмного забезпечення до початку самого процесу програмної розробки
* ***Комунікаційні шаблони*** описують процес спілкування між окремими учасниками/співробітниками організації
* ***Організаційні шаблони*** описують організаційну ієрархію підприємства/фірми
* ***Анті-паттерни*** описують, як не слід робити при розробці програм, показуючи характерні помилки в дизайні і в реалізації.

Антіпаттерни - класи, найбільш часто впроваджуваних поганих рішень проблеми, вони використовуються для розпізнання при вивченні непрацюючої системи. Вони діляться на групи: управління розробкою ПЗ; розробки ПЗ; в ООП; в програмуванні; методологічні; управління конфігурацією; організаційні; соціальні; «жартівливі».

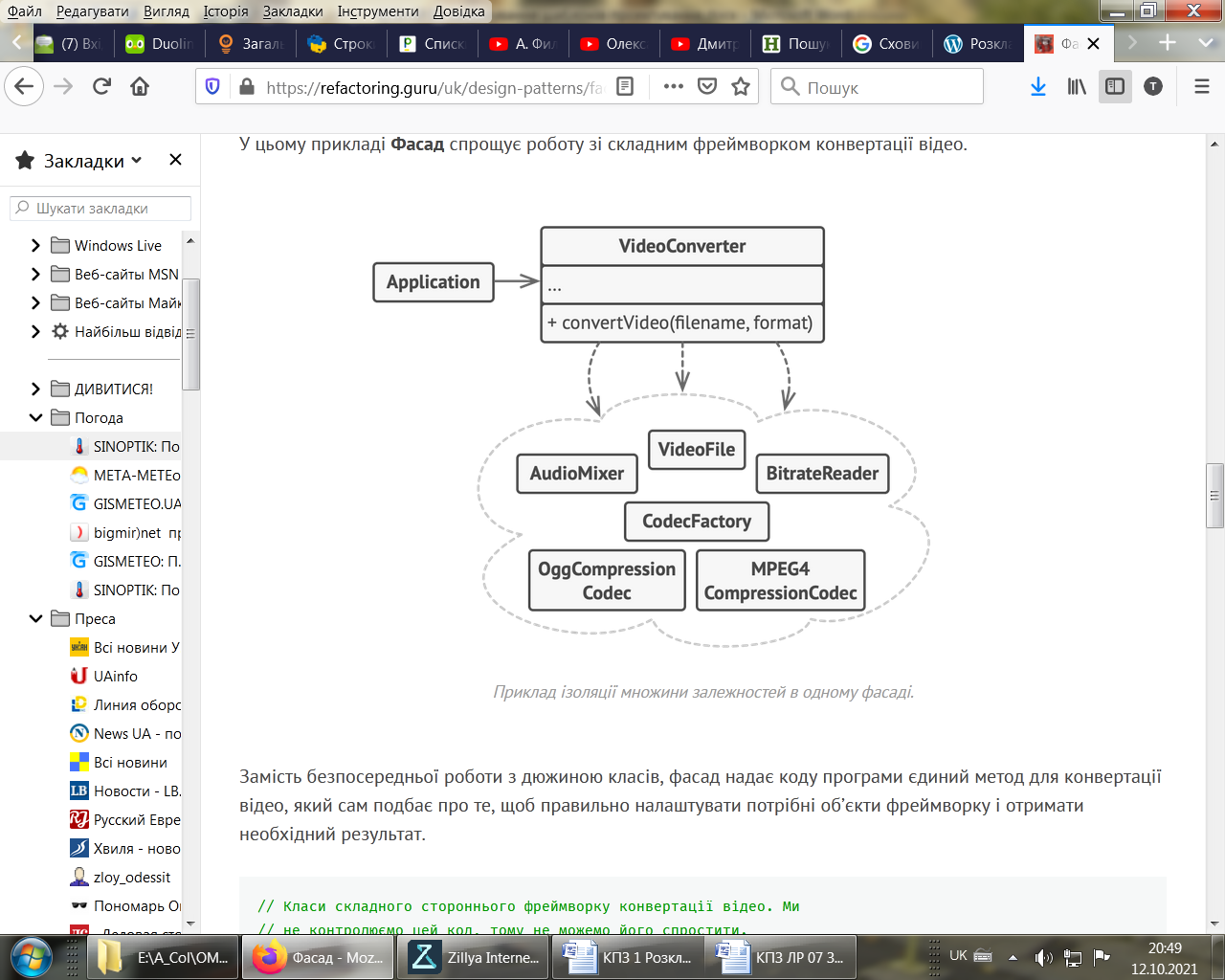
**Проектування проекту з використанням шаблону Façade.**

Цей підхід застосовується, коли використовується тільки частина всіх можливостей складної системи або коли взаємодія з нею здійснюється деяким специфічним чином. Шаблон проектування Facade спрощує використання складної системи, окремої частини системи або забезпечує звернення до системи деяким специфічним чином. Ми маємо складну систему, і нам потрібно використовувати тільки якусь її частину (окремий модуль). В результаті застосування шаблону Facade ми отримаємо нову, більш просту у використанні систему, яка буде точно відповідати нашим потребам. На наданій вище схемі показано ізоляцію клієнтів від внутрішніх підсистем.

Основна частина роботи як і раніше буде виконуватися вихідною системою. Шаблон Facade надає лише колекцію методів, простих у розумінні та використанні. Ці методи звертаються до основної системи для реалізації знову певних функцій зовнішньої системи.



Приклад з <https://refactoring.guru/uk/design-patterns/cpp>



Замість безпосередньої роботи з дюжиною класів, фасад надає коду програми єдиний метод для конвертації відео, який сам подбає про те, щоб правильно налаштувати потрібні об’єкти фреймворку і отримати необхідний результат.

Псевдокод.

// Класи складного стороннього фреймворку конвертації відео.

// Ми не контролюємо цей код, тому не можемо його спростити.

class VideoFile

// ...

class OggCompressionCodec

// ...

class MPEG4CompressionCodec

// ...

class CodecFactory

// ...

class BitrateReader

// ...

class AudioMixer

// ...

// Замість цього, ми створюємо Фасад — простий інтерфейс для

// роботи зі складним фреймворком. Фасад не має всієї

// функціональності фреймворку, але приховує його складність від

// клієнтів.

class VideoConverter is

method convert(filename, format):File is

file = new VideoFile(filename)

sourceCodec = new CodecFactory.extract(file)

if (format == "mp4")

destinationCodec = new MPEG4CompressionCodec()

else

destinationCodec = new OggCompressionCodec()

buffer = BitrateReader.read(filename, sourceCodec)

result = BitrateReader.convert(buffer, destinationCodec)

result = (new AudioMixer()).fix(result)

return new File(result)

// Програма не залежить від складного фреймворку конвертації

// відео. До речі, якщо ви раптом вирішите змінити фреймворк,

// вам потрібно буде переписати тільки клас фасаду.

class Application is

method main() is

convertor = new VideoConverter()

mp4 = convertor.convert("funny-cats-video.ogg", "mp4")

mp4.save()

Цей приклад показує структуру патерна **Фасад**, а саме — з яких класів він складається, які ролі ці класи виконують і як вони взаємодіють один з одним.

**main.cc: Приклад структури патерна**

/\*\*

\* The Subsystem can accept requests either from the facade or client directly.

\* In any case, to the Subsystem, the Facade is yet another client, and it's not

\* a part of the Subsystem.

\*/

class Subsystem1 {

public:

std::string Operation1() const {

return "Subsystem1: Ready!\n";

}

// ...

std::string OperationN() const {

return "Subsystem1: Go!\n";

}

};

/\*\*

\* Some facades can work with multiple subsystems at the same time.

\*/

class Subsystem2 {

public:

std::string Operation1() const {

return "Subsystem2: Get ready!\n";

}

// ...

std::string OperationZ() const {

return "Subsystem2: Fire!\n";

}

};

/\*\*

\* The Facade class provides a simple interface to the complex logic of one or

\* several subsystems. The Facade delegates the client requests to the

\* appropriate objects within the subsystem. The Facade is also responsible for

\* managing their lifecycle. All of this shields the client from the undesired

\* complexity of the subsystem.

\*/

class Facade {

protected:

Subsystem1 \*subsystem1\_;

Subsystem2 \*subsystem2\_;

/\*\*

\* Depending on your application's needs, you can provide the Facade with

\* existing subsystem objects or force the Facade to create them on its own.

\*/

public:

/\*\*

\* In this case we will delegate the memory ownership to Facade Class

\*/

Facade(

Subsystem1 \*subsystem1 = nullptr,

Subsystem2 \*subsystem2 = nullptr) {

this->subsystem1\_ = subsystem1 ?: new Subsystem1;

this->subsystem2\_ = subsystem2 ?: new Subsystem2;

}

~Facade() {

delete subsystem1\_;

delete subsystem2\_;

}

/\*\*

\* The Facade's methods are convenient shortcuts to the sophisticated

\* functionality of the subsystems. However, clients get only to a fraction of

\* a subsystem's capabilities.

\*/

std::string Operation() {

std::string result = "Facade initializes subsystems:\n";

result += this->subsystem1\_->Operation1();

result += this->subsystem2\_->Operation1();

result += "Facade orders subsystems to perform the action:\n";

result += this->subsystem1\_->OperationN();

result += this->subsystem2\_->OperationZ();

return result;

}

};

/\*\*

\* The client code works with complex subsystems through a simple interface

\* provided by the Facade. When a facade manages the lifecycle of the subsystem,

\* the client might not even know about the existence of the subsystem. This

\* approach lets you keep the complexity under control.

\*/

void ClientCode(Facade \*facade) {

// ...

std::cout << facade->Operation();

// ...

}

/\*\*

\* The client code may have some of the subsystem's objects already created. In

\* this case, it might be worthwhile to initialize the Facade with these objects

\* instead of letting the Facade create new instances.

\*/

int main() {

Subsystem1 \*subsystem1 = new Subsystem1;

Subsystem2 \*subsystem2 = new Subsystem2;

Facade \*facade = new Facade(subsystem1, subsystem2);

ClientCode(facade);

delete facade;

return 0;

}

**Output.txt: Результат виконання**

Facade initializes subsystems:

Subsystem1: Ready!

Subsystem2: Get ready!

Facade orders subsystems to perform the action:

Subsystem1: Go!

Subsystem2: Fire!

**Проектування проекту з використанням шаблону Adapter.**

Найпростіший спосіб зрозуміти призначення шаблону Adapter - це розглянути його застосування на прикладі. Припустимо, існують такі вимоги.

• Створити класи для представлення в системі точок, ліній і квадратів, кожен з яких матиме метод display (відобразити).

• Як використовують їх об'єкти не повинні знати, з яким саме елементом вони мають справу в дійсності - з точкою, лінією або квадратом. Клієнтським об'єктам досить знати, що вони отримали доступ до об'єкта, який представляє одну із зазначених фігур. Іншими словами, необхідно представити ці конкретні фігури за допомогою концепції більш високого порядку - назвемо її зображуваної фігурою. Тепер розширимо наш приклад, уявивши собі ще одну близьку ситуацію.

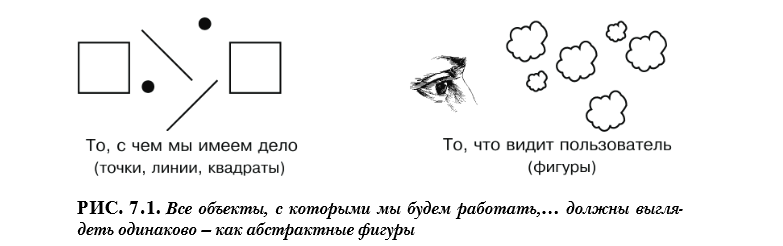
• Необхідно використовувати підпрограму або метод, написаний кимось іншим, оскільки в ньому реалізовані саме ті функції, які нам потрібні.

• При цьому включити готову підпрограму безпосередньо в створювану програму неможливо.

• Інтерфейс підпрограми або спосіб її виклику в коді не відповідають тим умовам, в яких вона буде використовуватися. Інакше кажучи, хоча в системі представлені точки, лінії і квадрати, нам потрібно, щоб все виглядало так, ніби в ній існують лише деякі абстрактні фігури.

• Це дозволить об'єктам-клієнтам працювати з будь-якими типами об'єктів-даних одним і тим же чином, не беручи до уваги існуючі між ними відмінності (рис. 2).

• Крім того, з'являється можливість згодом додавати в систему нові види фігур, не вносячи жодних змін в ті об'єкти, які будуть їх використовувати.



|  |  |
| --- | --- |
| *Те, з чим ми працюємо (точки, лінії, квадрати)* | *Те, що бачить користувач (фігури)* |

Рис.2 Всі об’єкти, з якими ми будемо працювати, повинні виглядати однаково – як абстрактні фігури

Тут використовується принцип поліморфізму; тобто в системі присутні об'єкти-дані різних типів, але використовують їх об'єкти-клієнти повинні взаємодіяти з ними одним і тим же чином. У результаті об'єкт-клієнт може просто вказати об'єкту-даному (незалежно від того, представляє він точку, лінію або квадрат), що необхідно виконати ту чи іншу дію - наприклад, відобразити себе на екрані або, навпаки, видалити з екрану своє зображення. У цьому випадку кожен об'єкт, що представляє точку, лінію або квадрат, повинен буде нести повну відповідальність за коректне виконання необхідних операцій в повній відповідності зі своїм типом. Для вирішення поставленого завдання створимо клас Shape (фігура), а потім визначимо похідні від нього класи, що представляють точки (Point), лінії (Line) і квадрати

(Square) - як показано на рис. 3.

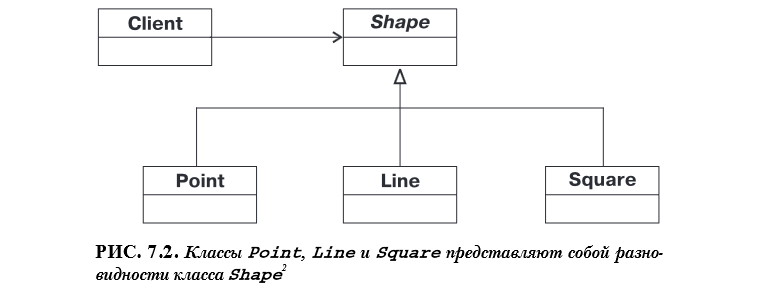


Рис.3. Класи Point, Line, Square є різновидом класу Shape

Насамперед необхідно визначити специфічну поведінку, яке повинен демонструвати клас Shape. Для вирішення цього завдання слід описати у ньому інтерфейс виклику методів, відповідальних за поведінку, а потім реалізувати ці методи в кожному з породжених класів. Поведінка, яка має демонструвати клас Shape, передбачає наступні методи (рис. 4).

• Отримати дані про становище об'єкта Shape (метод setLocation).

• Повідомити дані про становище об'єкта Shape (метод getLocation).

• Показати представлену об'єктом фігуру на екрані (метод display).

• Зафарбувати зображення фігури вказаним кольором (метод fill).

• Встановити колір зафарбовування фігури (метод setColor).

• Видалити зображення фігури з екрана (метод undisplay).

Припустимо, що в систему необхідно включити новий тип об'єктів класу Shape, призначений для представлення кіл (не забувайте, що вимоги постійно змінюються!). Для цієї мети створимо новий клас, Circle (Коло), який представлятиме в системі кола. Реалізуємо клас Circle як похідний від класу Shape, що дозволить скористатися перевагами його поліморфної поведінки.

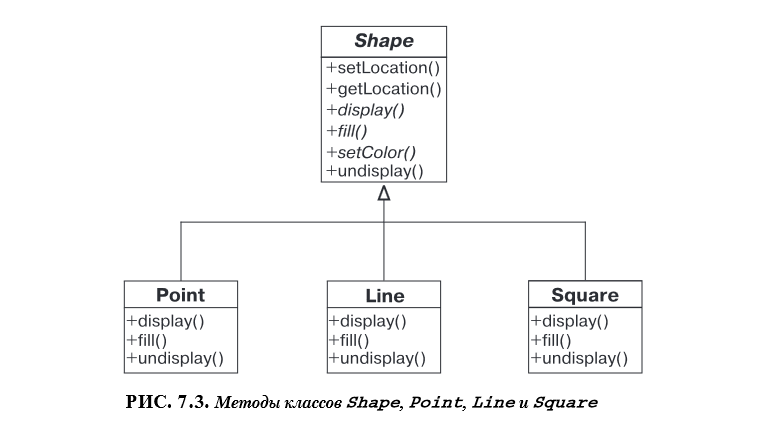


Рис.4. Методи класів Shape, Point, Line, Square

Тепер необхідно написати методи display, fill і Undisplay для класу Circle. Це завдання може виявитися досить складною. Після недовгих пошуків альтернативних рішень (що завжди повинен робити кожен хороший програміст), з'ясувалося, що вже існує описаний в системі клас XXCircle, призначений для роботи з колами (рис. 5), але методи цього класу мають такі імена: displayIt, fillIt, undisplayIt.

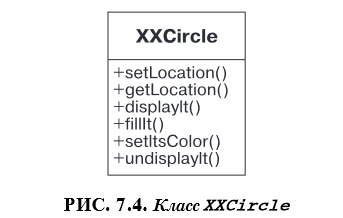
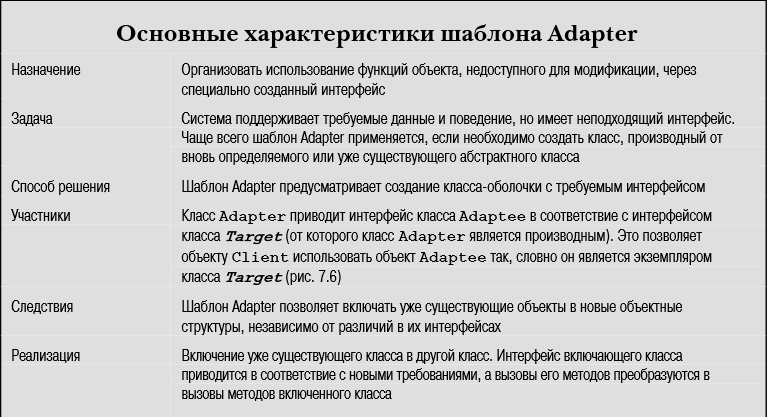


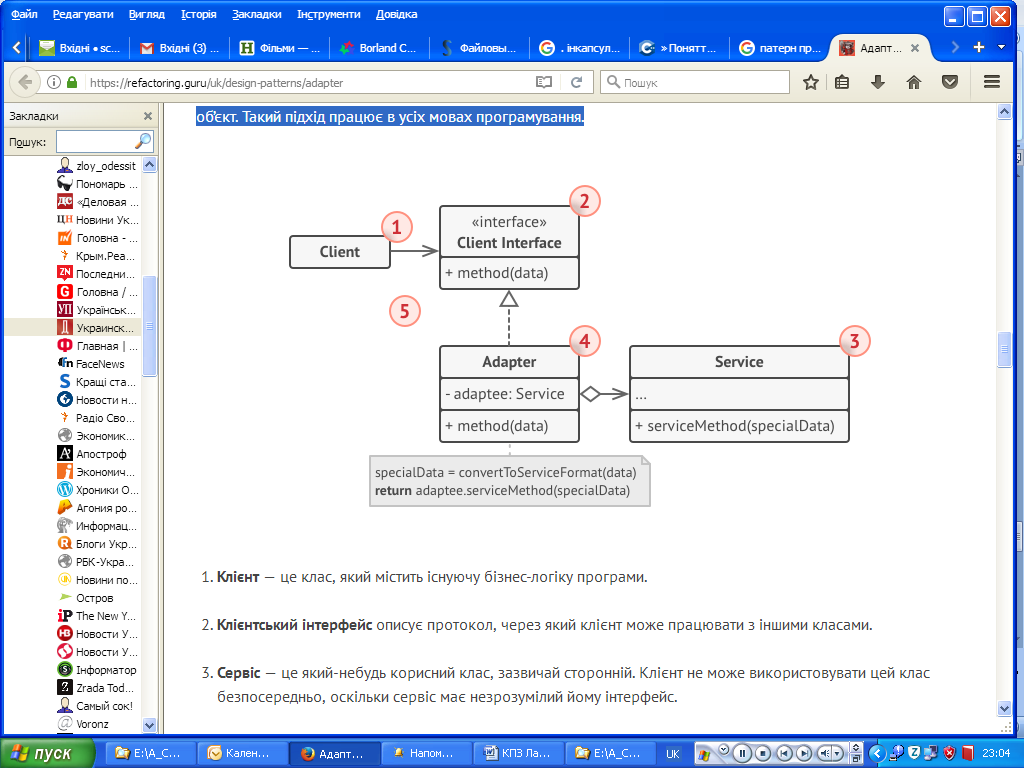
Рис.5 Клас XXCircle

В результаті, безпосередньо використовувати клас XXCircle не можна, оскільки бажано зберегти поліморфну поведінку, реалізоване в класі Shape, але цьому перешкоджають наступні моменти.

• Клас Shape і клас XXCircle включають методи з різними іменами і різними списками параметрів.

• Клас XXCircle повинен не тільки мати співпадаючі імена методів, а й обов'язково бути похідним від класу Shape.



**Адаптер об’єктів** - реалізація використовує агрегацію: об’єкт адаптера «загортає», тобто містить посилання на службовий об’єкт. Такий підхід працює в усіх мовах програмування.

1. **Клієнт** — це клас, який містить існуючу бізнес-логіку програми.
2. **Клієнтський інтерфейс** описує протокол, через який клієнт може працювати з іншими класами.
3. **Сервіс** — це який-небудь корисний клас, зазвичай сторонній. Клієнт не може використовувати цей клас безпосередньо, оскільки сервіс має незрозумілий йому інтерфейс.
4. **Адаптер** — це клас, який може одночасно працювати і з клієнтом, і з сервісом. Він реалізує клієнтський інтерфейс і містить посилання на об’єкт сервісу. Адаптер отримує виклики від клієнта через методи клієнтського інтерфейсу, а потім конвертує їх у виклики методів загорнутого об’єкта в потрібному форматі.
5. Працюючи з адаптером через інтерфейс, клієнт не прив’язується до конкретного класу адаптера. Завдяки цьому ви можете додавати до програми нові види адаптерів, незалежно від клієнтського коду. Це може стати в нагоді, якщо інтерфейс сервісу раптом зміниться, наприклад, після виходу нової версії сторонньої бібліотеки.

#### Адаптер класів - реалізація базується на спадкуванні: адаптер успадковує обидва інтерфейси одночасно. Такий підхід можливий тільки в мовах, які підтримують множинне спадкування, наприклад у C++.

#### 1. - ****Адаптер класів**** не потребує вкладеного об’єкта, тому що він може одночасно успадкувати й частину існуючого класу, й частину класу сервісу.