**Лекція 30. Рефакторінг та тестування класів**

***Поняття рефакторингу***

Рефакторинг – це процес зміни системи програмного забезпечення таким чином, щоб він не змінював зовнішню поведінку коду, але покращував його внутрішню структуру. Якщо розглядати процес розробки програмного забезпечення, то можна сказати, що людині-програмісту майже неможливо за один раз написати такий програмний код, який і буде успішним кінцевим результатом. Завжди знайдеться щось, що потрібно виправити або покращити.

Як виявляється, при використанні об'єктно-орієнтованого підходу для розробки програмних систем необхідність виконувати рефакторинг виникає достатньо часто. Практика використання ООП у багатьох проектах свідчить про це. У 2000 році Мартин Фаулер зі співавторами написав книгу “Refactoring: Improving the Design of Existing Code” – Рефакторинг: покращення дизайну існуючого коду [2]. У цій книзі були зібрані та систематизовані методи та прийоми рефакторингу для ООП. Про поняття рефакторингу Мартин Фаулер пише наступне: «Це дисциплінований спосіб очищення коду, який мінімізує ймовірність появи помилок. По суті, коли ви виконуєте рефакторинг, ви покращуєте дизайн коду після його написання. «Покращення дизайну після його написання». У нашому поточному розумінні розробки програмного забезпечення ми вважаємо, що ми проектуємо, а потім кодуємо.

Хороший дизайн на першому місці, а кодування – на другому. З часом код буде змінюватися, і цілісність системи, її структура відповідно до цього дизайну поступово зникає. Код повільно переходить від розробки до хакерства» [2].

Можна виділити такі важливі аспекти рефакторингу:

- Рефакторинг складається з дрібних простих кроків. Проте, сумарний ефект таких невеличких змін може радикально покращити проект.

- Проектування безперервно відбувається впродовж розробки, а не завершується цілком на початку. При реалізації системи стає зрозуміло, як можна покращити проект. В результаті такої взаємодії якість проекту буде високою

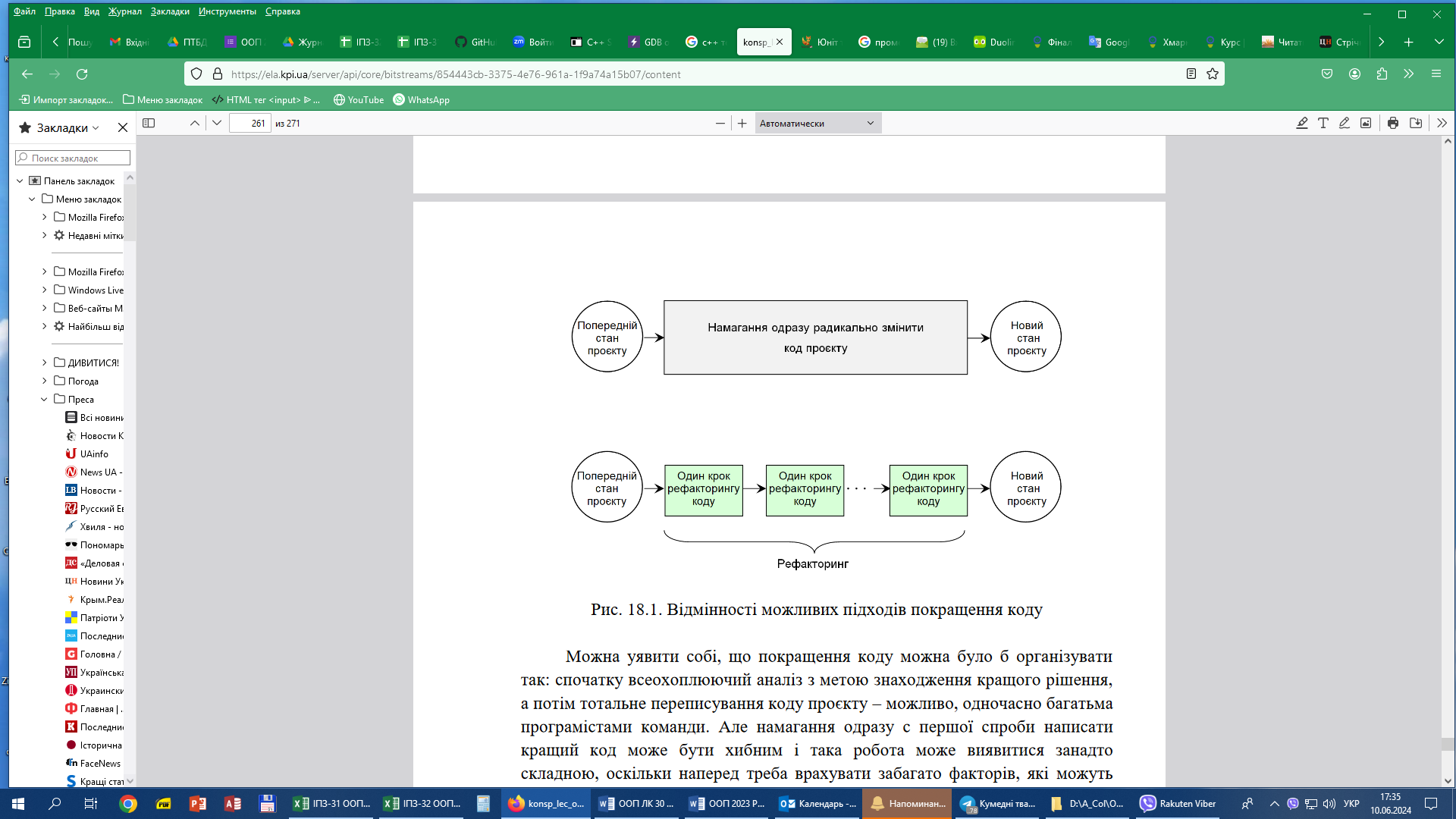


Рисунок 1 - Відмінності можливих підходів покращення коду

Можна уявити собі, що покращення коду можна було б організувати так: спочатку всеохоплюючий аналіз з метою знаходження кращого рішення, а потім тотальне переписування коду проєкту – можливо, одночасно багатьма програмістами команди. Але намагання одразу с першої спроби написати кращий код може бути хибним і така робота може виявитися занадто складною, оскільки наперед треба врахувати забагато факторів, які можуть бути невизначеними.

На відміну цього рефакторинг є послідовним цілеспрямованим виконанням дрібних кроків покращення коду. Чому саме дрібними кроками? Тому що часто необхідність виконання певного кроку не є очевидною від початку, і тільки у процесі змін може стати зрозуміло, що трансформувати у коді далі. Також може бути з’ясовано, що у результаті певного кроку виникають деякі непередбачувані проблеми – тоді варто повернутися на крок назад. Такий підхід по суті є процесом послідовного наближення до оптимального рішення.

Коли потрібно виконувати рефакторинг? У книзі [2] наведено перелік ознак того, що з программним кодом вже щось негаразд (мовою оригіналу “Bad Smells in Code“).

Ознаки проблемного коду, які можуть стати причиною рефакторингу:

- Дублювання коду

- Довгий метод

- Великий клас

- Довгий список параметрів

- Дивергентність змін. Клас часто змінюється різними способами з різних причин

- «Заздрість» методів – коли метод більше звертається до іншого класу, відмінному від того, членом якого він є оголошеним

- Неструктуровані дані

- Оператор switch. Наявність цього оператора може вказувати на те, що його треба замінити поліморфізмом

- Паралельні ієрархії успадкування

- «Ледачий клас» – клас, який робить недостатньо, щоб виправдати своє існування, слід усунути

- Умоглядна, спекулятивна загальність. Якщо є абстрактні класи, які не роблять багато. Методи з невикористаними параметрами повинні підлягати видаленню параметра. Спекулятивну загальність можна помітити, коли єдиними користувачами методу чи класу є тестові випадки

- Клас даних. Це класи, які мають поля, методи отримання та встановлення для полів (функції get та set) і нічого більше. Такі класи є лише примітивними власниками даних, якими маніпулюють зазвичай занадто часто

- «Відмова від заповіту». Підкласи успадковують усі методи та дані від базових класів. Але, якщо вони не потребують усього від них – то це означає, що ієрархія успадкування неправильна

- Інші ознаки.

Варто відзначити, що деякою мірою має місто парність категорій. Так, зокрема, з одного боку потрібно орієнтуватися на абстракції, але з іншого боку, якщо оголошено забагато абстрактних класів, які не роблять багато – то це теж погано. Неструктуровані дані є ознакою поганого коду, але наявність класів даних, у яких є тільки поля, а у якості методів присутні тільки сетери та гетери – є теж сумнівним дизайном.

Розглянемо докладніше деякі з методів рефакторингу.

*Заміна switch поліморфізмом*

Якщо ми у програмному коді десь бачимо switch(щось), то це “щось” треба зробити класом з віртуальним методом і оголосити похідні класи відповідно варіантам вибору. У похідних класах визначити програмний код метода для кожного з варіантів, а замість switch записати поліморфний виклик метода.

І усе це не тому, що оператор switch є поганим – зовсім ні, він працює дуже швидко і у багатьох випадках є незамінним. Не треба вважати, що будь що треба обходитися без нього (подібно до того як колись закликали зовсім не використовувати оператор goto). Але з точки зору ООП, якщо у деякому програмному коді switch записано для вибору альтернатив для різних типів варіантів, то якщо множина варіантів може з часом розширюватися, змінюватися, то замість переписування у клієнтському коді тіла оператора switch краще представити змінну множину варіантів класами Нехай маємо наступний фрагмент програмного коду

int type; //type = ID1,ID2,ID3, ...

. . .

type = ID2;

. . .

switch (type)

{

case ID1:

DoSomething1();

break;

case ID2:

DoSomething2();

break;

case ID3:

DoSomething3();

break;

. . .

}

Для того, щоб прибрати конструкцію switch(type), спочатку замість “type” оголосимо базовий клас ObjType і низку похідних класів

class ObjType

{

public:

virtual void DoSomething1(void) = 0;

};

class ObjType1 : public ObjType

{

public:

void DoSomething(void);

};

class ObjType2 : public ObjType

{

public:

void DoSomething(void);

};

class ObjType3 : public ObjType

{

public:

void DoSomething(void);

};

А тепер запишемо клієнтський код – вже без switch

ObjType\* p;

. . .

p = new ObjType2;

. . .

p->DoSomething(); //поліморфізм замість switch

Приклад виконання рефакторингу

Нехай маємо наступний програмний код.

Клієнтська частина – використовуємо об'єкти класу MyClass:

MyClass mc;

char tmpstr[256];

//. . .

mc.readstring(tmpstr, "oldfile.log");

mc.method(tmpstr, OLD);

//. . .

mc.readstring(tmpstr, "currentfile.log");

mc.method(tmpstr, PRESENT);

//. . .

mc.readstring(tmpstr, "futurefile.log");

mc.method(tmpstr, FUTURE);

//. . .

Клас MyClass оголошується та визначається у файлах модуля module1

Файл module1.h

enum TimeType

{

OLD = 1,

PRESENT = 2,

FUTURE = 3

};

class MyClass

{

private:

int value;

public:

int getvalue();

void readstring(char \*dest, char \*fname);

int method(char \*src, int vartype);

};

Файл module1.cpp

#include "module1.h"

int MyClass::getvalue()

{

return value;

}

void MyClass::readstring(char \*dest, char \*fname)

{

//. . .

strcpy(dest, "start\tmessage\tend");

//. . .

}

int MyClass::method(char \*src, int vartype)

{

int res = 0;

char \*pos;

if (src == 0) return 0;

pos = strchr(src, '\t');

if (pos)

value = strlen(pos);

//. . .

switch (vartype)

{

case OLD:

res = 10;

break;

case PRESENT:

res = 100 + value;

break;

case FUTURE:

res = 1000 + value;

break;

default : break;

}

//. . .

return res;

}

Як здається, у метода method занадто великий текст. Крім того, конструкція на основі switch викликає сумніви у плані можливого розширення типів.

1-й крок рефакторингу: виділення метода (Extract Method) у

програмному коді method. Частину коду виділимо в окрему функцію count()

class MyClass

{

private:

int value;

int count(int vartype);

public:

int getvalue();

void readstring(char \*dest,char \*fname);

int method(char \*src, int vartype);

};

int MyClass::count(int vartype)

{

int res = 0;

switch (vartype)

{

case OLD:

res = 10;

break;

case PRESENT:

res = 100 + value;

break;

case FUTURE:

res = 1000 + value;

break;

default : break;

}

return res;

}

int MyClass::method(char \*src, int vartype)

{

int res;

char \*pos;

if (src == 0) return 0;

pos = strchr(src, '\t');

if (pos)

value = strlen(pos);

//. . .

res = count(vartype);

//. . .

return res;

}

2-й крок рефакторингу: заміна switch поліморфізмом. Оголосимо базовий клас

class TimeBase

{

public:

virtual int count(int value) = 0;

};

Похідні класи

class OldTime : public TimeBase

{

public:

int count(int value)

{

return 10;

}

};

class PresentTime : public TimeBase

{

public:

int count(int value)

{

return 100 + value;

}

};

class FutureTime : public TimeBase

{

public:

int count(int value)

{

return 1000 + value;

}

};

Відповідно змінюємо method класу MyClass

class MyClass

{

private:

int value;

public:

int getvalue();

void readstring(char \*dest, char \*fname);

int method(char \*src, TimeBase \*tm);

};

int MyClass::method(char \*src, TimeBase \*tm)

{

int res;

char \*pos;

if (src == 0) return 0;

pos = strchr(src, '\t');

if (pos)

value = strlen(pos);

//. . .

res = tm->count(value);

delete tm;

//. . .

return res;

}

У коді клієнта класів MyClass і XXXTime буде використовуватися композиція цих класів

MyClass mc;

char tmpstr[256];

//. . .

mc.readstring(tmpstr, "oldfile.log");

mc.method(tmpstr, new OldTime);

//. . .

mc.readstring(tmpstr, "currentfile.log");

mc.method(tmpstr, new PresentTime);

//. . .

mc.readstring(tmpstr, "futurefile.log");

mc.method(tmpstr, new FutureTime);

//. . .

Як здається, рефакторинг можна продовжити. Так, наприклад, виглядає доцільним додаткове структурування з метою уникнення фактично дублювання коду для ланцюжків викликів пар методів readstring та method. Зокрема, якщо ці методи викликаються тільки парою, то можна їх перенести у закриті члени, а публічним методом класу MyClas зробити такий:

int MyClass::func(char \*fname, TimeBase \*tm)

{

char tmpstr[256];

mc.readstring(tmpstr, fname);

return mc.method(tmpstr, tm);

}

***Місце рефакторингу у процесі розробки програмного забезпечення***

Типовий процес розробки програмного забезпечення можна представити як послідовність багатьох ітерацій, які можуть повторюватися

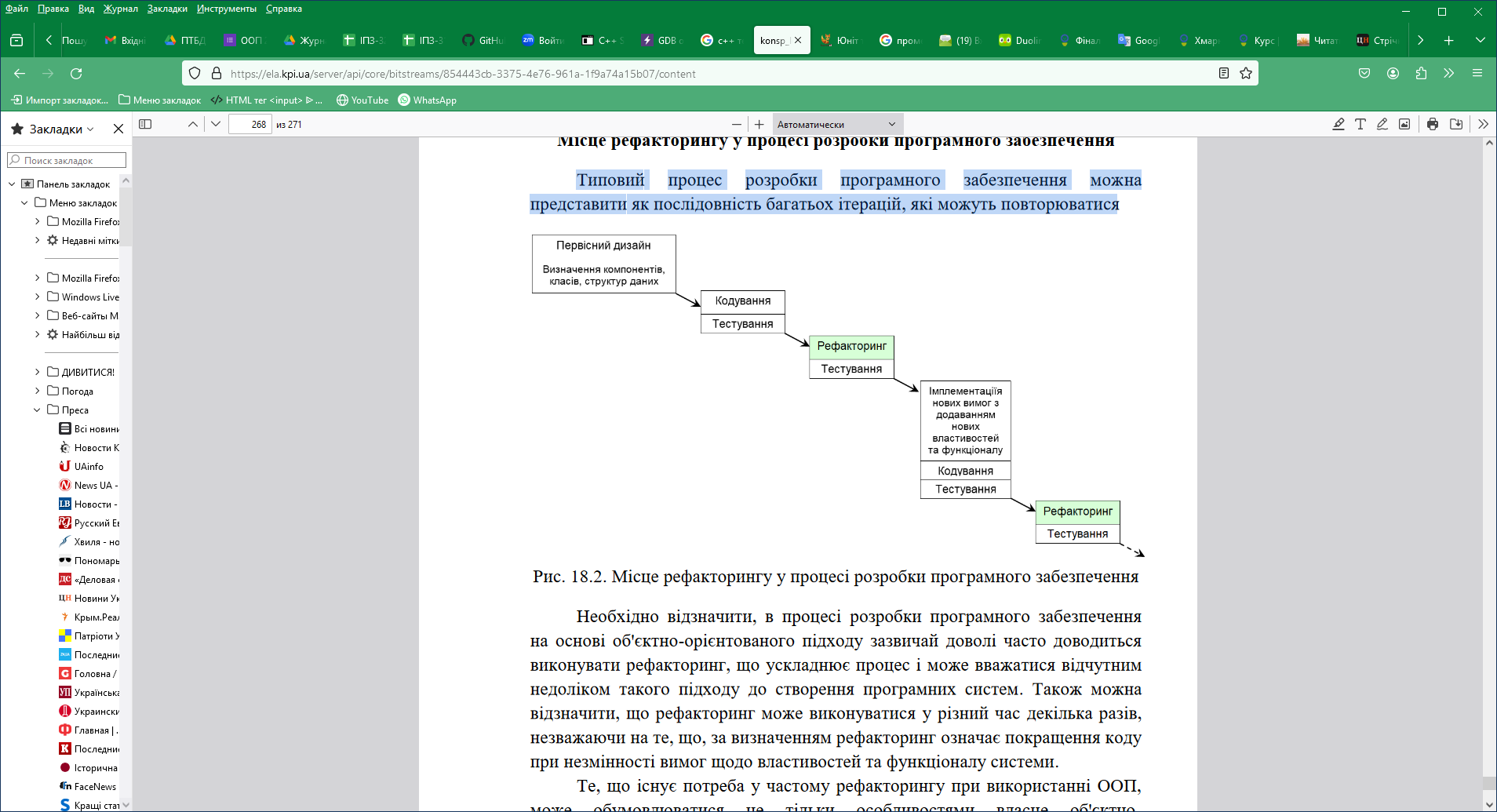


Рисунок 2 - Місце рефакторингу у процесі розробки програмного забезпечення

Необхідно відзначити, в процесі розробки програмного забезпечення на основі об'єктно-орієнтованого підходу зазвичай доволі часто доводиться виконувати рефакторинг, що ускладнює процес і може вважатися відчутним недоліком такого підходу до створення програмних систем. Також можна відзначити, що рефакторинг може виконуватися у різний час декілька разів, незважаючи на те, що, за визначенням рефакторинг означає покращення коду при незмінності вимог щодо властивостей та функціоналу системи. Те, що існує потреба у частому рефакторингу при використанні ООП, може обумовлюватися не тільки особливостями власне об'єктно-орієнтованого підходу, а й значною мірою це може визначатися людським фактором в інженерії програмного забезпечення.

**Unit тести**

Ці тести є основою для перевірки невеликих, ізольованих частин коду – так званих “юнітів”.

Unit тести – це автоматичні тести, які перевіряють невеликі частини коду, такі як функції або методи, ізольовано від решти системи. Вони дають змогу розробникам переконатися, що кожна частина коду працює правильно і відповідає очікуваній поведінці. Вони пишуться мовою програмування і можуть бути запущені автоматично для швидкої перевірки коду при кожній його зміні.

Термін “юніт” у контексті теми позначає мінімально можливу частину коду, яку можна протестувати ізольовано. Це може бути функція, метод або навіть окремий рядок коду. Кожен “юніт” має бути протестований окремо від решти коду, щоб переконатися в його коректності. Такий підхід дає змогу виявити помилки та проблеми в коді на ранніх стадіях розробки.

*Важливість юніт-тестів*

Давайте уявимо собі, що у вас є шматочок коду, який ви щойно написали. Ви думаєте, що він працює правильно, але як ви можете бути впевнені? Ось де на допомогу приходять юніт-тести. Вони дозволяють вам перевірити, що ваш код виконує свої функції так, як задумано.

Вони допомагають виявити помилки і дефекти в коді на ранніх етапах розробки. Це дає змогу заощадити час і зусилля, які в іншому разі могли б бути витрачені на пошук і виправлення проблем у коді, що вже склався. Крім того, юніт-тести допомагають запобігти появі нових помилок під час внесення змін до коду в майбутньому. Коли ви їх пишете, то декомпозуєте ваш код на окремі “юніти” і перевіряєте кожен із них окремо. Це допомагає виявити проблеми в окремих компонентах коду і переконатися, що вони працюють правильно в ізоляції.

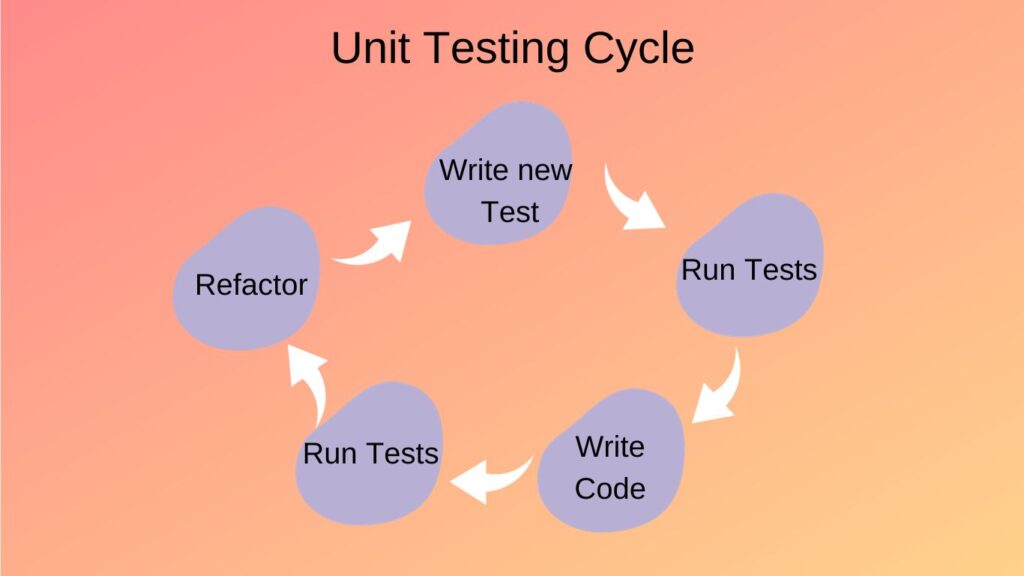
Юніт-тести також сприяють підвищенню надійності коду. Якщо у вас є хороший набір тестів, які перевіряють різні сценарії використання, ви можете бути впевнені, що ваш код буде працювати стабільно і передбачувано навіть у складних ситуаціях.

Крім того, юніт-тести слугують документацією до коду – це коли ви описуєте очікувану поведінку вашого коду в явному вигляді. Це робить ваш код зрозумілішим і полегшує його використання та підтримку для інших розробників, які можуть працювати з ним у майбутньому.

***Процес написання***

Написання юніт-тестів включає кілька кроків:

* Визначення тестованих “юнітів”. Виберіть функції, методи або класи, які ви хочете протестувати. Визначте, які частини вашого коду мають бути покриті тестами.

 Розробка тестових випадків. Визначте різні сценарії використання і стани, які мають бути протестовані. Напишіть тестові випадки для кожного сценарію, де ви перевіряєте очікувані результати та поведінку коду.

* Написання тестового коду. Напишіть код для кожного тестового випадку, який перевіряє очікуваний результат. Використовуйте спеціальні фреймворки та бібліотеки для тестування, щоб спростити процес написання та виконання тестів.
* Запуск тестів. Запустіть усі написані тести та перевірте їхні результати. Переконайтеся, що всі тести пройшли успішно і код працює правильно.
* Аналіз результатів і виправлення помилок. Вивчіть результати тестів і зверніть увагу на будь-які помилки або несподівану поведінку. Виправте проблеми у вашому коді та перезапустіть тести для перевірки виправлень.

***Як правильно писати юніт тести?***

Тут слід дотримуватися кількох принципів:

* **Напишіть тести одразу.** Краще писати тести разом із кодом або якомога раніше в процесі розробки. Це допомагає запобігти появі помилок і спростити виявлення проблем.
* **Будьте зрозумілі та специфічні.** Називайте ваші тестові випадки та функції таким чином, щоб було зрозуміло, що вони перевіряють. Зробіть кожен тестовий випадок максимально специфічним і фокусуйтеся на конкретних аспектах коду.
* **Використовуйте асерти.** Використовуйте твердження (assertions) для перевірки очікуваних результатів. Переконайтеся, що ваш код повертає очікувані значення і поводиться так, як ви припускаєте.
* **Підтримуйте свої тести.** Оновлюйте ваші тести під час внесення змін до коду. Переконайтеся, що вони все ще актуальні та продовжують перевіряти правильну поведінку вашого коду.
* **Запускайте тести регулярно**, щоб переконатися, що вони все ще працюють. Це допоможе вам швидко виявити будь-які проблеми у вашому коді.

Дотримуючись цих кроків і принципів, ви зможете створювати надійні та ефективні тести для вашого коду.

Приклади

1. ***Приклад для функції додавання*:**

Припустимо, у вас є функція add(a, b), яка додає два числа. Ви можете написати наступний юніт-тест для перевірки її роботи:

def test\_add():

    result = add(2, 3)

    assert result == 5, "Очікуваний результат: 5"

У цьому тесті ми викликаємо функцію add(2, 3) і порівнюємо її результат з очікуваним значенням 5. Якщо результат не збігається з очікуваним, тест завершиться з помилкою.

1. **Приклад для класу:**

Припустимо, у вас є клас Calculator, який має метод multiply(a, b), що виконує множення двох чисел. Ви можете написати такий юніт-тест для перевірки його роботи:

def test\_multiply():

    calculator = Calculator()

    result = calculator.multiply(2, 3)

    assert result == 6, "Очікуваний результат: 6"

У цьому тесті ми створюємо екземпляр класу Calculator, викликаємо його метод multiply(2, 3) і порівнюємо результат з очікуваним значенням 6.

Але виникає питання, як інтерпретувати результати? Вони зазвичай представлені у вигляді звітів або виведення на екран. Коли ви запускаєте тести, вони можуть показати таку інформацію:

* **Пройдені тести:** якщо все пройшло успішно, ви побачите повідомлення про проходження тесту або його статус “пройдено”. Це означає, що код, який було протестовано, працює так, як очікувалося.
* **Невдалі тести:** якщо ж не все гладко, ви побачите повідомлення про невдачу тесту або його статус “не пройдено”. Це означає, що було виявлено проблему в коді, яку потрібно виправити.
* **Зведення результатів:** наприкінці виконання всіх тестів ви можете побачити зведення результатів, яке показує загальну кількість пройдених і невдалих тестів. Це допомагає вам зрозуміти, наскільки добре ваш код пройшов тестування.

Контрольні запитання

1. Що таке рефакторинг?

2. Які ознаки поганого коду?

3. Які ви знаєте методи рефакторингу?

4. Коли треба замінювати оператор switch поліморфізмом?

5. Яким чином та як часто треба виконувати рефакторинг?

1. Патерни проектування. REFACTORING GURU [Online] Available from: <https://refactoring.guru/uk/design-patterns>
2. Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts Refactoring: Improving the Design of Existing Code. Addison Wesley Professional, Second Edition, 2018, 448 p.
3. Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern [Online] Available from: <https://www.martinfowler.com/articles/injection.html#FormsOfDependencyInjection>
4. Martin Fowler. UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd Edition, 2003, 208 p.
5. Об'єктно-орієнтоване програмування: конспект лекцій [Електронний ресурс]: навч. посіб.для студ. освітньої програми «Інженерія програмного забезпечення комп'ютерних систем» спеціальності 121 «Інженерія програмного забезпечення» / Порєв В.М.; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 4,8 МБайт). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 271 с. – Назва з екрана.URL: https://ela.kpi.ua/server/api/core/bitstreams/854443cb-3375-4e76-961a-1f9a74a15b07/content