**ДВНЗ**

**ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ**

**ЗВІТ ДО ЛАБОРАТОРНОЇ РОБОТИ №2**

з дисципліни «Методи інженерії комп’ютерних систем»

Виконав:

Студент гр. КІ-20

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Глущенко Д.С.

(дата, підпис)

Перевірив(ла):

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Любимов А.С.

(дата, підпис) ( ПІБ)

Луцьк – 2023

**Тема: Породжувальні патерни**

**Мета: ознайомлення із призначенням, сферою застосування, структурою та особливостями реалізації породжувальних патернів**

**Завдання:**

1. Abstract Factory:

Дах будинку

Будівельна компанія-посередник займається будівництвом приватних будинків «під ключ». Справа дійшла до будівництва даху, покриття якого в залежності від бажання замовника може бути різним – металевий профіль, керамічна черепиця, звичайний шифер тощо. Кожен постачальник покриття надає й бригаду фахівців з його укладання. Промоделювати цю ситуацію за допомогою абстрактної фабрики, яка створює й об’єкти матеріалу, й фахівців з його укладання.

1. Builder

Реєстрація користувача

Ви розробляєте систему, яка повинна підтримувати процедуру реєстрації користувача. При цьому список полів з інформацією про користувача є досить довгим (ви бажаєте збирати багато статистики!), але тільки частина з них є обов'язково. За допомогою патерну «Будівельник» реалізуйте процес реєстрації користувачів з різним набором введених параметрів.

**Хід роботи:**

Абстрактна фабрика — це породжувальний патерн проектування, що дає змогу створювати сімейства пов’язаних об’єктів, не прив’язуючись до конкретних класів створюваних об’єктів.

Цей патерн використовується коли бізнес-логіка програми повинна працювати з різними видами пов’язаних один з одним продуктів, незалежно від конкретних класів продуктів.

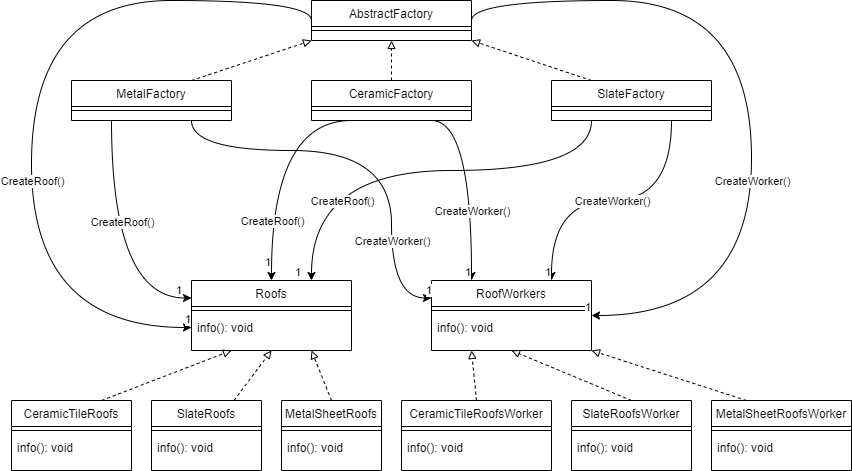


Рисунок 1 – UML-діаграма абстрактної фабрики

**Лістинг програми:**

Зміст AbstractFactory:

**package** main;

**public** **interface** AbstractFactory {

Roofs createRoof();

RoofWorkers createWorker();

}

Зміст CeramicFactory:

**package** main;

**public** **class** CeramicFactory **implements** AbstractFactory {

@Override

**public** Roofs createRoof() {

**return** **new** СeramicTileRoof();

}

@Override

**public** RoofWorkers createWorker() {

**return** **new** CeramicTileRoofWorker();

}

}

Зміст SlateFactory:

**package** main;

**public** **class** SlateFactory **implements** AbstractFactory {

@Override

**public** Roofs createRoof() {

**return** **new** SlateRoof();

}

@Override

**public** RoofWorkers createWorker() {

**return** **new** SlateRoofWorker();

}

}

Зміст MetalFactory:

**package** main;

**public** **class** MetalFactory **implements** AbstractFactory{

@Override

**public** Roofs createRoof() {

**return** **new** MetalSheetRoof();

}

@Override

**public** RoofWorkers createWorker() {

**return** **new** MetalSheetRoofWorker();

}

}

Зміст Roofs:

**package** main;

**public** **interface** Roofs {

**public** **void** info();

}

Зміст RoofWorkers:

**package** main;

**public** **interface** RoofWorkers {

**public** **void** info();

}

Зміст MetalSheetRoof:

**package** main;

**public** **class** MetalSheetRoof **implements** Roofs {

@Override

**public** **void** info() {

System.***out***.println("I'm a metal sheet roof");

}

}

Зміст SlateRoof:

**package** main;

**public** **class** SlateRoof **implements** Roofs {

@Override

**public** **void** info() {

System.***out***.println("I'm a slate roof");

}

}

Зміст СeramicTileRoof:

**package** main;

**public** **class** СeramicTileRoof **implements** Roofs {

@Override

**public** **void** info() {

System.***out***.println("I'm a ceramic tile roof");

}

}

Зміст CeramicTileRoofWorker:

**package** main;

**public** **class** CeramicTileRoofWorker **implements** RoofWorkers {

@Override

**public** **void** info() {

System.***out***.println("I'm a ceramic tile roof worker");

}

}

Зміст MetalSheetRoofWorker:

**package** main;

**public** **class** MetalSheetRoofWorker **implements** RoofWorkers {

@Override

**public** **void** info() {

System.***out***.println("I'm a metal sheet roof worker");

}

}

Зміст SlateRoofWorker:

**package** main;

**public** **class** SlateRoofWorker **implements** RoofWorkers {

@Override

**public** **void** info() {

System.***out***.println("I'm a slate roof worker");

}

}

Зміст Main:

**package** main;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

AbstractFactory factory;

factory = **new** SlateFactory();

*CreateRealFactory*(factory);

factory = **new** MetalFactory();

*CreateRealFactory*(factory);

}

**public** **static** **void** CreateRealFactory(AbstractFactory f) {

Roofs roof;

RoofWorkers worker;

roof = f.createRoof();

worker = f.createWorker();

roof.info();

worker.info();

}

}

**Приклад виконання коду з Main:**

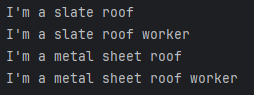


Рисунок 2 – Результат роботи абстрактної фабрики

Будівельник — це породжувальний патерн проектування, що дає змогу створювати складні об’єкти крок за кроком. Будівельник дає можливість використовувати один і той самий код будівництва для отримання різних відображень об’єктів.

Цей патерн потрібно використовувати якщо потрібно позбутися від «телескопічного конструктора», код повинен створювати різні уявлення якогось об’єкта, потрібно збирати складні об’єкти.

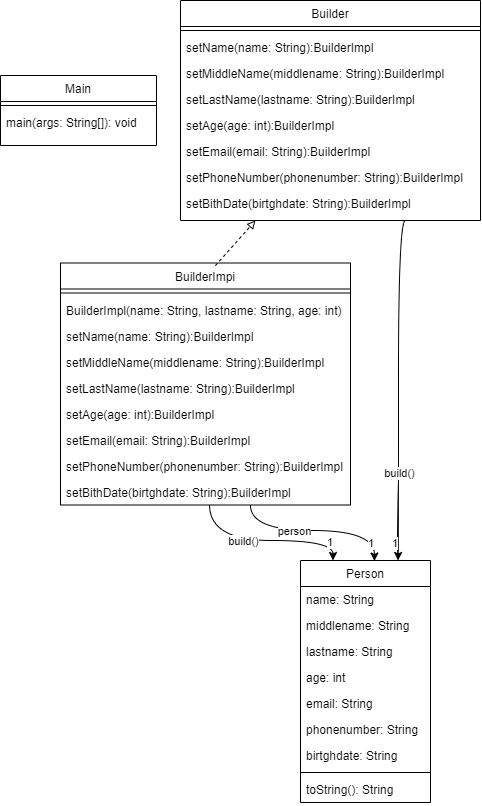


Рисунок 3 – UML-діаграма будівельника за варіантом

**Лістинг програми:**

Зміст Builder:

**package** main;

**public** **interface** Builder {

BuilderImpl setName(String name);

BuilderImpl setMiddleName(String middlename);

BuilderImpl setLastName(String lastname);

BuilderImpl setAge(**int** age);

BuilderImpl setEmail(String email);

BuilderImpl setPhoneNumber(String phonenumber);

BuilderImpl setBirthDate(String birthdate);

Person build();

}

Зміст BuilderImpl:

**package** main;

**public** **class** BuilderImpl **implements** Builder{

Person person = **new** Person();

BuilderImpl(String name,String lastname,**int** age){

person.name=name;

person.lastname=lastname;

person.age=age;

}

@Override

**public** BuilderImpl setName(String name) {

person.name=name;

**return** **this**;

}

@Override

**public** BuilderImpl setMiddleName(String middlename) {

person.middlename=middlename;

**return** **this**;

}

@Override

**public** BuilderImpl setLastName(String lastname) {

person.lastname=lastname;

**return** **this**;

}

@Override

**public** BuilderImpl setAge(**int** age) {

person.age=age;

**return** **this**;

}

@Override

**public** BuilderImpl setEmail(String email) {

person.email=email;

**return** **this**;

}

@Override

**public** BuilderImpl setPhoneNumber(String phonenumber) {

person.phonenumber=phonenumber;

**return** **this**;

}

@Override

**public** BuilderImpl setBirthDate(String birthdate) {

person.birthdate=birthdate;

**return** **this**;

}

**public** Person build() {

**return** person;

}

}

Зміст Person:

**package** main;

**public** **class** Person {

String name;

String middlename;

String lastname;

**int** age;

String email;

String phonenumber;

String birthdate;

**public** String toString() {

**return** "Name: "+ name+

" Middle name: "+middlename+

" Last name: "+lastname+

" Age: "+age+

" E-mail: "+email+

" Phone number: "+phonenumber+

" Birth date: "+birthdate+"\n";

}

}

Зміст Main:

**package** main;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Person mike = **new** BuilderImpl("Bob","Builder",27).setEmail("example@email.com.ua").setPhoneNumber("9847629876").build();

System.***out***.print(mike.toString());

}

}

**Приклад виконання коду з Main:**



Рисунок 4 – Результат роботи будівельника

**Висновок:** я **ознайомився з призначенням, сферою застосування, структурою та особливостями реалізації породжувальний патернів. На практиці використав патерни абстрактної фабрики ти будівельник.**

**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ**

1. Абстрактна фабрика [Електорний ресурс]–Режим доступу: <https://refactoring.guru/uk/design-patterns/abstract-factory>
2. Будівельник [Електорний ресурс]–Режим доступу: <https://refactoring.guru/uk/design-patterns/builder>