**GoF patterns**

**Creational** роблять процес створення і ініціювання процесів більш адаптивним і динамічним. Builder, Prototype, Factory, AbstractFactory, Singleton.

**Builder** – використовують для створення різного предаставлення складних обєктів використовуючи один і той самий механізм для створення. Наприклад, у нас є User який має 5 полів і ми можемо створити обєкт використовуючи конструктор:

public User (String firstName, String lastName, int age, String phone, String address){

    this.firstName = firstName;

    this.lastName = lastName;

    this.age = age;

    this.phone = phone;

    this.address = address;

А якшо у нас тільки 2 поля у юзера є mandatory , то нам терба робити ще більше конструкторів. І якшо ми хочемо добавити 6-те поле, то нам терба ще більше конструторів:

public User (String firstName, String lastName, int age){ ...   }

public User (String firstName, String lastName){ ...    }

Тут ми можемо застосувати патерн Builder ми можемо створити в класі User внутрішній клас UserBuilder (шоб потім не забути його ментейнити при зміні обєкта User) який матиме набір тих самих полів шо й юзер тільки замість геттерів матиме публічний метод для кожного поля який повертатиме UserBuild і метод сам метод build() який повертатиме User.

public UserBuilder address(String address) {

            this.address = address;

            return this;

        }

//Return the finally consrcuted User object

public User build() {

            User user =  new User(this);

            validateUserObject(user);

            return user;

}

Ну і тепер ми можемо створювати User на основі UserBuilder просто заінлайнивши всі поля які нам потрібні для конкретного репрезентейшена обєкта на приклад як це робить StringBuilder.append():

User user1 = new User.UserBuilder("John", "Doe")

    .age(30)

    .phone("1234567")

    .address("Fake address 1234")

    .build();

**Prototype** – використовують коли треба створити багато інстансів класу які мають одинаковий або дуже схожий стан. Це ніби шаболон обєкта. Прототип обєкта створюється один раз, а потім просто клонується – це дешевше в плані ресурсів ніж постійно робити нові обєкти. Схожий на AbstractFactory але дає більший контроль над процесом створення обєкта. Наприклад у нас є аплікейшн якому дуже часто треба створювати інстанси класів Movie і Show, то ми робимо інтерфейс який буде мати метод clone() що повертатиме тип даного інтерфейсу і тоді наш клас Movie буде екстендити цей інтерфейс:

public interface PrototypeCapable extends Cloneable

{

    public PrototypeCapable clone() throws CloneNotSupportedException;

}

public class Movie implements PrototypeCapable

{

    private String name = null;

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    @Override

    public Movie clone() throws CloneNotSupportedException {

        System.out.println("Cloning Movie object..");

        return (Movie) super.clone();

    }

}

Ну і у нас буде якась PrototypeFactory яка буде в мапі зберігати прототипи наших обєктів і буде просто клонити їх з мапи якшо буде потрібний новий інстанс:

public class PrototypeFactory

{

    public static class ModelType

    {

        public static final String MOVIE = "movie";

        public static final String SHOW = "show";

    }

    private static java.util.Map<String , PrototypeCapable> prototypes = new java.util.HashMap<String , PrototypeCapable>();

    static

    {

        prototypes.put(ModelType.MOVIE, new Movie());

        prototypes.put(ModelType.SHOW, new Show());

    }

    public static PrototypeCapable getInstance(final String s) throws CloneNotSupportedException {

        return ((PrototypeCapable) prototypes.get(s)).clone();

    }

}

І далі просто будемо її юзати:

PrototypeFactory.getInstance(ModelType.MOVIE)

**Factory** – використовується для створення відносно різних комплексних обєктів які схожі по функціоналу і діляться на категорії. Дозволяє централізувати створення обєктів і знизити затрати при якихось змінах щоб не приходилося міняти створення обєктів по всьому коду. Наприклад у нас є CarFactory яка буде повертати Car по його типу – small, sedan, luxury.

public enum CarType {

    SMALL, SEDAN, LUXURY

}

Для всіх машин зробимо парент клас Car в якому буде загальна логіка і конструктор який буде приймати тип моделі і робити якусь дефолтну логіку типу заказу запчастин:

public abstract class Car {

    public Car(CarType model) {

        this.model = model;

        arrangeParts();

    }

    private void arrangeParts() {

        // Do one time processing here

    }

    // Do subclass level processing in this method

    protected abstract void construct();

    private CarType model = null;

    public CarType getModel() {

        return model;

    }

    public void setModel(CarType model) {

        this.model = model;

    }

}

Далі ми чайлдами екстендимо Car де викликаємо конструктор парента з нашим типом і дефолтною логікою, плюс імплементуємо специфічну для нашого CarType логіку зборки:

public class LuxuryCar extends Car {

    LuxuryCar() {

        super(CarType.LUXURY);

        construct();

    }

    @Override

    protected void construct() {

        System.out.println("Building luxury car");

        // add accessories

    }

}

Потім робимо CarFactory яка матиме статичний метод buildCar(CarType type) який нам в якомусь світчі повертатиме інстанс обєкта по його типу:

public class CarFactory {

    public static Car buildCar(CarType model) {

        Car car = null;

        switch (model) {

        case LUXURY:

            car = new LuxuryCar();

            break;

        default:

            // throw some exception

            break;

        }

        return car;

    }

}

Ну і викликаєм:

CarFactory.buildCar(CarType.LUXURY)

**Abstract Factory** – використовують коли нам потрібен більш вищий рівень абстракції над групою фабрик. Наприклад ми маємо фабрику яка робить машини по типу машини іле наша аплікуха розширилась і нам треба робити машини з специфікою до певного регіону, наприклад праворульні чи ліворульні машини. Тому ми робимо наприклад, USACarFactory і AsiaCarFactory для різних регіонів. Регіони:

public enum Location {

  DEFAULT, USA, ASIA

}

Описуємо парент обєкт Car в конструкторі якого крім моделі буде ще й інформація про локацію машини.

public abstract class Car {

  public Car(CarType model, Location location){

    this.model = model;

    this.location = location;

  }

  protected abstract void construct();

  private CarType model = null;

  private Location location = null;

  //getters and setter

}

Реалізуємо чайлда Car з певним типом:

public class LuxuryCar extends Car

{

  public LuxuryCar(Location location)

  {

    super(CarType.LUXURY, location);

    construct();

  }

  @Override

  protected void construct() {

    System.out.println("Building luxury car");

    //add accessories

  }

}

Описуєм CarFactory для певного регіону яка буде повертати машину по типу і для певного регіону:

public class USACarFactory

{

  public static Car buildCar(CarType model)

  {

    Car car = null;

    switch (model)

    {

      case LUXURY:

      car = new LuxuryCar(Location.USA);

      break;

      default:

      //throw some exception

      break;

    }

  return car;

  }

}

І робимо нашу CarFactory яка в метод buildCar прийматиме тип машини CarType яку ми хочемо і з сетінгів буде брати значення регіону і відповідно до нього буде повертати машину через якусь фабрику для певного регіону:

public class CarFactory

{

  private CarFactory() {

    //Prevent instantiation

  }

  public static Car buildCar(CarType type)

  {

    Car car = null;

    Location location = Location.ASIA; //Read location property somewhere from configuration

    //Use location specific car factory

    switch(location)

    {

      case USA:

        car = USACarFactory.buildCar(type);

        break;

      case ASIA:

        car = AsiaCarFactory.buildCar(type);

        break;

      default:

        car = DefaultCarFactory.buildCar(type);

    }

  return car;

  }

}

Ну і викликаємо: CarFactory.buildCar(CarType.LUXURY)

**Singleton –** використовують коли потрібно мати тільки один інстанс обєекта на всю JVM (private static volatile Singleton singleton). Може бути сінглтон з **eager ініціалізацією**, тобто створюватися на старті апілікейшена не залежно від того чи він потрібен в даний момент чи ні. Якщо обєкт не важкий, то це непоганий підхід:

public class EagerSingleton {

    private **static volatile** EagerSingleton instance = new EagerSingleton();

    // private constructor

    private EagerSingleton() {

    }

    public static EagerSingleton getInstance() {

        return instance;

    }

}

Також може бути singleton з **lazy ініціалізацією**. В такому випадку інстанс обєкта створюється тільки тоді, коли він потрібен, а не при старті аплікейшена. Коли потрібно ми викличем getInstance() метод і він перевірить чи обєкт null, якшо так, то метод бере ставить монітор на клас і тоді віддає new LazySingleton. Якшо обєкт не null то просто віддається вже існуючий інстанс. Причина робити double check в синхронізованій секції на те чи інстанс не null полягає в тому, шо в середину можуть по зайти два потока і перевірити шо інстанс null і тоді кожен з потоків створить по одному інстансу обєета.

public class LazySingleton {

    private **static volatile** LazySingleton instance = null;

    // private constructor

    private LazySingleton() {

    }

    public static LazySingleton getInstance() {

        if (instance == null) {

            synchronized (LazySingleton.class) {

                // Double check

                if (instance == null) {

                    instance = new LazySingleton();

                }

            }

        }

        return instance;

    }

}

Також є **Singleton з static ініціалізацією**. Тобто ми можем описати створення інстанса в статичному блоці ініціалізації і сінглтон створиться ще до виконання конструктора класу:

public class StaticBlockSingleton {

    private static final StaticBlockSingleton INSTANCE;

    static {

        try {

            INSTANCE = new StaticBlockSingleton();

        } catch (Exception e) {

            throw new RuntimeException("Uffff, i was not expecting this!", e);

        }

    }

    public static StaticBlockSingleton getInstance() {

        return INSTANCE;

    }

    private StaticBlockSingleton() {

        // ...

    }

}

Негативна сторона такого підходу, шо якшо у нас декілька статичних полів, то вони будуть ініціалізуватися всеодно потрібні вони на даний момент в аплікейшені чи ні.

Також є **Enum singleton**. Так як Енами в джаві є тред сейф і гарантують тільки один інстанс, то можна їх використовувати для реалізації сінглтона. Робимо енам і описуємо потрібні нам методи:

public enum EnumSingleton {

    INSTANCE;

    public void someMethod(String param) {

        // some class member

    }

}

**Structural patterns** допомагають нам поєднувати різні частини системи таким чином щоб вони були гнучкими і могли розширюватися. І коли ми робимо зміни в одній частині системи, то десь в іншій нічого не зламалося. Тобто патерни які зменшують код комплекміті, покращують луз каплінг і орієнтовані на open\close принципи.

Adapter, Decorator, Bridge, Composit, Façade, FlyWeigth, Proxy.

**Adapter** – перетворює інтерфейс класу в інший який очікує клієнт. Таким чином класи можуть працювати разом не зважаючи на різні інтерфейси.Наприклад ми берем карту з фотоапарата і хочемо її вставити в комп, але так як той не має такого розєму – ми берем карт рідер який дозволяє карті працювати з пк. Приклад з джави, коли ми хочем зчитати символи консолі, то ми використовуємо System.in який повертає масив байтів і передаєм його в BufferedReader але той очікує шо йому прийде масив символів. Тому ми використовуємо між ними адаптер InputStreamReader який хендлеть цю різницю в інтерфейсах.

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

**Bridge** – дозволяє нам розділити клас на дві частини – абстракцію і імплементацію. Завдяки цьому в майбутньому вони можуть розвиватися не афектаючи один одного. Наприклад, ми імплементуємо збереження файлів в різних операційних системах. Ми робимо інтерфейс абстракцію яка матиме метод store():

public interface FileDownloaderAbstraction

{

    public boolean store(Object object);

}

Далі ми робимо ще один інтерфейс з методом storeFile(), який буде ніби провайдером для імплементації збереження на різних системах.

public interface FileDownloadImplementor

{

    public boolean storeFile(Object object);

}

І тепер наша імплементація буде імплементувати інтерфейс абстракції і імплементувати з неї метод save(), але в самій імплементації викликати метод storeFile() класу, який описаний в імплементації другого інтерфейсу провайдера. Таким чином нашим провайдером може бути провайдер любої операційної системи:

public class FileDownloaderAbstractionImpl implements FileDownloaderAbstraction {

    private FileDownloadImplementor provider = null;

    public FileDownloaderAbstractionImpl(FileDownloadImplementor provider) {

        super();

        this.provider = provider;

    }

    @Override

    public boolean store(Object object)

    {

        return provider.storeFile(object);

    }

}

**Composit** паттерн дозволяє нам модифікувати структуру обєкта. Найбільше він підходить якшо наша аплікуха має їєрархічну структуру і ми розглядаємо структуру обєкта як деревовидну ієрархію. Тобто в обєкті кожна нода є композитною нодою, або листком. Таким чином ми можемо трітати і компонувати обєкт під наші потреби через простий інтерфейс компонента. А коли нам треба добавити додаткові компоненти то це відбувається навіть без змін на клієнті. Наприклад у клієнта банку є декілька рахунків і нам треба зібрати загальний баланс клієнта по всім рахункам і операціям.

**Decorator** – використовують, коли потрібно добавити якогось функціоналу для конкретного інстансу класу не змінюючи при цьому інші інстанси цього класу. Тобто декоратор міняє поведінку обєкта але не його інтерфейс таким чином у нас є обєкти з додатковою поведінкою без зміни коду класу який стоїть в основі. На практиці ми берем інстанс і в нього добавляємо ще один інстанс, вони обоє підтримують один або схожий інтерфейс. В такому випадку зовнішній називають декоратором який і використовують поки він маскує зміни інстанса в середині себе чи використовує методин існтанса в середині. Наприклад, ми маємо репорт і він має два пдітипа, якийсь ClientReport i EmployeeReport які є чайлдами репорта. І ці репорти по різному відмальовуються, наприклад мають різний колір бекграунда. Тобто ми в цих двох репортах описуємо метод який буде просетувати нам колір бекграунда. А якщо нам через пару днів терба буде щоб репорти малювалися різним шрифтом, то нам знову в двох місцях терба буде додавати методи makeFont(). А якщо типів репорту стане ще більше, то це ще більше місць де треба добавляти поведінку і це суперечить принципу Open/Close. Тому можна добавити якийсь ReportFontDecorator який буде наслідувати якийсь ReportDataDecorator, а він в свою чергу матиме наслідуватиметься і врапатиме в собі Report і потім можна збирати цей телеском з ClientReporta який кладеться в конструктор ClientFontDecorator і який кладеться в конструктор SupportBackgroundDecorator. Наприклад в джаві коли ми хочемо читати з файлу, ми файл передаємо в FileInputStream який передаєм в BufferedInputStream.

BufferedInputStream bs = new BufferedInputStream(new FileInputStream(new File(“File1.txt”)));

**Facade** – представляє собою високорівневий інтерфейс над інтерфейсами. Він надає уніфікований інтерфейс до сету інтерфейсісв в системі таким чином якшо у нас складна система то ми клієнту можемо показувати простий інтерфейс в вигляді фасаду. Наприклад, нам треба взяти дані з бази і для цього треба встановити конекшн з базою. Ми просто робимо якийсь dataSource.getConnection(), а він собі під капотом вже завантажує драйвер, створює конекшн чи бере його з пула і повертає його. Чи, наприклад, у нас є складні репорти, то ми можемо зробити якийсь ReportGeneratorFacade який матиме метод generateReport(), а робоиа з всякими хедерами, футерами і даними репортів буде збиратися в середині нього – таким чином ми матимем простий інтерфейс для складної логіки що спростить клієнту життя. Різниця з Адаптером в тому, що адаптер міняє інтерфейс для кдлієнта, а Фасад спрощує його. Різниця з Медіатором в тому, що в мкдіаторі відбувається зв'язок в дві сторони, тобто система знає про посередника, а в фасаді тільки фасад спілкується з системою, тобто тільки в одну сторону.

**FlyWeight** – представляє собою шарений обєкт який може використовуватися в різних контекстах і бути незалежним обєктом в кожному з контекстів. Використовують, коли в системі потрібно мати велику кількість унікальних лише декількома параметрами обєктів, а вся інша поведінка для них є загальною. Також паттерн корисний коли нам треба контролювати споживання памяті при великій кількості обєктів – тоді робляться обєкти і шаряться між контекстами. Паттерн має два атрибута, один зовнішній для клієнта, а інший внутрішній, який буде загальний для обєктів даного типу. Наприклад у нас є ручка, яку можна заправляти чорнилом різного кольору. В такому випадку заправка кольору буде зовнішнім атрибутом обєкта, а все решта таке як еорпус ручки, її перо і тпе дплі – будуть внутрішнім атрибутом ручки. Далі, якщо нам потрібна ручка з червоним кольором, ми створюєм за допомогою FlyweightFactory нову ручку в яку передаєм червоний колір. Ця створена фабрикою ручка кладеться в якусь її мапку і далі, якщо комусь треба буде червону ручку, то фабрика віддасть вже створений шарений інстанс цієї червоної ручки з мапи. Також прикладом є пул строк.

**Proxy** – використовують для того щоб зробити плейсхолдер для обєкта тим самим ховаючи його і контролювати доступ до нього, наприклад для лейзі лоадінгу, коли ми не хочемо одразу тянути цілий обєкт поки він нам не знадобиться або для якихось додаткових секюрних причин. Наприклад в Hibernate коли ми тягнемо ентіті з бази, то хібернейт повертає обєкт з проксі таким чином що ми можемо працювати з обєктом, а коли на буде дійсно потрібен той чи інший підобєкт, то він його підтяне. З точки зору коду у нас є інтерфейс яким користується клієнт, далі є обєкт який імплементує цей інтерфейс і є проксі, який екстендить цей обєкт і саме з цим проксі замість реального обєкта і працює клієнт.

public interface RealObject

{

    public void doSomething();

}

public class RealObjectImpl implements RealObject {

    @Override

    public void doSomething() {

        System.out.println("Performing work in real object");

    }

}

public class RealObjectProxy extends RealObjectImpl

{

    @Override

    public void doSomething()

    {

        //Perform additional logic and security

        //Even we can block the operation execution

        System.out.println("Delegating work on real object");

        super.doSomething();

    }

}

Різниця з Декоратором в тому, що декоратор додає функціоналу, а Проксі більше зосереджений на контролі доступу до обєкта.

Типи проксі: **Protection** **proxy** (займається перевіркою доступу до обєкту. Наприклад перевіряти якісь пермішени перед виконанням методу), **Virtual** **proxy** (займажться лейзі ініціалізацією обєктів), **Smart** **proxy** (виконує якусь додаткову роботу перед викликом обєкту, наприклад, чекає чи не залоканий обєкт перед його зміною), **Remote** **proxy** (бере на себе всю роботу по комунікації з ремоут обєктами щоб клієнт мав до цього відношення).

**Behavioral patterns** використовують коли мова йдепро взаємодію між обєктами. Вони представляють собою абстракцію дії над обєктом, який виконує якусь дію. Тобто ми можемо міняти обєкт чи його поведінку не міняючи для клієнта базовий інтерфейс.

Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Template, Visitor.

**Chain of Responsibility** – патерн який дозволяє відправнику реквеста не бути строго завязаним на отримувача реквеста таким чином щоб декілька обєктів могли опрацьовувати реквест незалежно. Тобто якшо у нас є декілька обєктів які можуть обробити реквест який не очікує якогось специфічного хендлера, то можна використати чейн оф респонсібіліті який в рантаймі буде визначати який хендлер буде працювати над реквестом. **Хендлер** в нашому випадку буде якийсь інтерфейс який передасть реквест наступному хендлеру і він не має знати нічого, що відбувається далі по чейну. **Конкретний** **хендлер** – це вже якийсь конкретний хендлер з чейну який замається своєю задачею по порядку. Ну і тут ще є клієнт, який безпосередньо посилає реквест на хендлер. Хорошим прикладом є колл центр якоїсь установи, наприклад банку. Клієн звонить колл менеджерам з якоюсь проблемою, якщо оператор не може її вирішити, то передає запит по чейну далі до когось з більшими повноваженнями чи з потрібною відповідальністю і так далі.

**Command** дозволяє розбити логіку на абстрактні дії і енкапсулює в собі всю інформацію яка дозволяє зробити якусь бізнес логіку чи трігнути івент. Підхід схожий як в чейн оф респосібіліті, але там декілька хендлерів задіяні, а тут один. З точки зору коду ми маєм якийсь **Command** **Interface** який описує набір операцій, маємо **ConreteCommandClass** який імплементує методи щоб задіяти бізнес логіку, є **Invoker** який робить запит на якусь дію і **Receiver** який вже трігає якусь дію. При чому Invoker може трігати дії навіть нічого не знаючи про Receiver.

**Iterator –** дозволяє проводити якісь дії по черзі в різних напрямках над агреговваними обєктами з якоїсь колекції не показуючи на зовні їх внутрішню будову. Наприклад в джаві є ListIterator який імплементує інтерфейс Iterator шоб могти бігати по колекції.

**Mediator** – допомагає зменшити код комплексіті шляхом того, що обєкти яким треба якось взаємодіюти не комунікують на пряму а через медіатор. Це дозволяє винести всі звязки в окремий клас і обєкти замість того щоб взаємодіяти між собою делегують свою взаємодію медіатору. Тобто Many-to-many звязки перетворюються в one-to-many. Також він дозволяє ізолювати зміни компонента не афектаючи при цьому інші компоненти. Наприклад взяти якийсь чат де є багато учасників. Якщо робити конекшн кожного з кожним то це буде дуже накладно і складно контрольовано. Замість цього робиться якась рума до якої всі підключені – ця рума і виступає в ролі медіатора. Приклад з Джави це метод execute() з Executors. З точки зору коду є інтерфейс **Mediator** який описує взаємодію між **Colegues** обєктами, далі є **ConcreteMediator** який імплементує інтерфейс Mediator щоб кординувати взаємодію між Colegues обєктами бо знає про них і як вони мають взаємодіяти, ну і відповідно є інтерфейс **Colegue** і **ConcreteColegues** який описує як Colegue взіємодіє з іншими Colegues через медіатор. Різниця з фасадом полягає в тому, що медіатор не працює з інтерфейсом одного обєкта, а робить мультиінтерфейс для багатьох обєктів. Якщо переборщити з логікою в медіаторі то можна прийти в кінці до антипатерна God class.

**Memento** (snapshot) – використовують коли треба відновлювати попередній стан обєкта. Мається на увазі, що обєкт на протязі свого лайфсайкла має ресторпойнти які ми можемо використати. Наприклад коли ми пишемо код, не закомітали і рестартанули IDEшку, то вона завантажиться з змінами які ми зробили. Або нажимати CTRL+Z коли хочемо відмінити зміни. Так само можуть бути чекпойнти на транакціях бази даних шоб ролбекнути все до попереднього стану якшо шось зафейлиться. З точки зору коду є **Originator** який має методи **createMemento**() i **restore**(memnto) відповідно вміє робити снепшоти обєкта і ресторити їх стан; **Caretaker** який слідкує за багатьма memento i каже **Originatory** коли робити createMemento() a коли restore(memento); ну і сам **Memento** який реперезентує собою стан обєкта і має бути імьютабл щоб ніхто його не поміняв.

**Observer** – описує one-to-many взаємодію так що якщо один з обєктів змінив свій стан, то всі залежні від нього класи інформуються про це і відповідно до логіки міняють свій стан автоматично. Наприклад, ми підписалися на когось в файсбуці і тепер коли він шось буде постити, то ми будемо нотіфіковані про його пост і якщо нам шось не підійде, то ми можемо відписатися від людини коли завгодно. З точки зору коду є **Subject** і його імплементація **ConreteSubject** які описують **attach** і **detach** обсерверів до них навіть в рантаймі і безпосередню нотифікацію заатачених обсерверів якщо відбулася зміна обєкта; Також є **Observer** i його імплементація **ConcreteObserver** які знають що робити, коли вони будуть нотифіковані. При чому різні типи обсерверів можуть підписуватися на один і той же Subject оскільки вони імплементують один Observer. Різниця з чейн оф респонсібіліті полягає в тому, що в обсервері одразу всі нотифікуються і змінюють свій стан одночасно, а в чейн оф респонсібіліті обєекти проходить через всю цепочку один за одним.

**State** - вирішує проблему, коли треба щоб обєкт міняв свою поведінку, якщо його стан зміниться. Для цього кожен стан обєкта має мати інший відповідний клас для обєкта який прийме або відхилить конкретний стан обєкта при його зміні. Це дозволяє зменшити кількість if-else і дає можливість додавати нові state обєкта не афектаючи попередні. Наприклад у нас є пульт до телевізора який при натисканні кнопки міняє його стан – виключає звук, перемикає канал чи вимикає його – це можливі стани обєкта, а от якщо телевізор вимкнений, то він вже може мати тільки стан включення, коли ми його включемо. З ДЖави приклад це стани потоків які можуть мінятися відповідно до лайфсайклу потока (NEW (ще не запущено), RUNNABLE (виконується), BLOCKED (призупинено бо чекає на блокування), TERMINATED (закінчив виконання), TIMED\_WAITING (призупинив виконання на певний час. Наприклад після sleep(millis), join(), wait() ), WAITING (зупинив виконання бо чекає на виконання якоїсь дії.). З точки зору коду є **State** i **ConcreteState** які описують поведінку при аевному стані; і є **Context** який мейнтейнить референси на конкретні стани і делегує до обєктів для різного стану.

Рекомендують State обєкти завжди робити сінглтонами.

**Strategy** – патерн який дозволяє вибирати алгоритм для якоїсь задачі в рантаймі з декількох алгоритмів для даної задачі. З точки зору коду ми створюємо інтерфейс який представляє абстракцію алгоритму. Потім ми робимо імплементації цієї абстракції які називаються стратегіями. Клієнт буде використовувати абстракцію і передавати в неї контекст обєкт який і вирішуватиме яку стратегію вибрати. Наприклад CD Project RED випустили свій лаунчер для ігр з різних лаунчерів по типу Steam, Origin і так далі. Він дозволяє зібрати в одному місці всі твої ігри і друзів з усіх лаунчерів. І, наприклад, коли я хочу шось написати другу, то я в контексті передаю його нікнейм і лаунчер в якому він у мене є і мені відкриється чат з ним при чому я навіть не буду знати що відбулося під капотом. Це все приблизно виглядає як якийсь абстрактний інтерфейс **LauncherStrategy** який містить в собі абстактну дію connectToLAuncher(launcherName), **LauncherContext** який буде вибирати конкретну імплементацію стратегії і сама **Implementation** стратегії.

Приклад з Джави це Collections.sort(list, comparator) в якому клієнт передає потрібний компаратор в рантаймі і базуючись на різних компараторах одна колекція може бути постортована по різному.

**Template** – використовують коли ми маємо логіку в якій чітко видно що є декілька кроків які повинні повторятися одне за одним без зміни. Тоді є сенс цю незмінну цепочку винести в абстрактний клас який матиме метод що трігатиме цю послідовність. Таким чином ми уникнем дублікації коду і гарантуємо собі послідовне виконання степів потрібного алгоритму. Наприклад, ми хочемо побудувати дім. Для цього ми знаємо шо є послідовність методів типу зробити фундамент, побудувати стіни, зробити дах, вставити вікна і двері. Якщо ми дозволимо виконати ці методи в довільном порядку, то дім побудовано не буде. Для цього ми робимо абстрактний клас **House** і в нім метод **buildHouse**() в якому буде описана правильна послідовність в вигляді абстрактних методів шоб їх імплементнули в нащадках. А якщо ми, наприклад будуємо фундамент і дах який має бути загальним для всіх, то ці методи темплейта можемо описати в абстрактному класі **House** як **private final**.

public abstract class House {

    public final void buildhouse() {

        constructBase();

constructWalls();

        constructRoof();

        constructWindows();

    }

    public abstract void constructDoors();

    public abstract void constructWindows();

    public abstract void constructWalls();

    private final void constructRoof() {

        System.out.println("Roof has been constructed.");

    }

    private final void constructBase() {

        System.out.println("Base has been constructed.");

    }

}

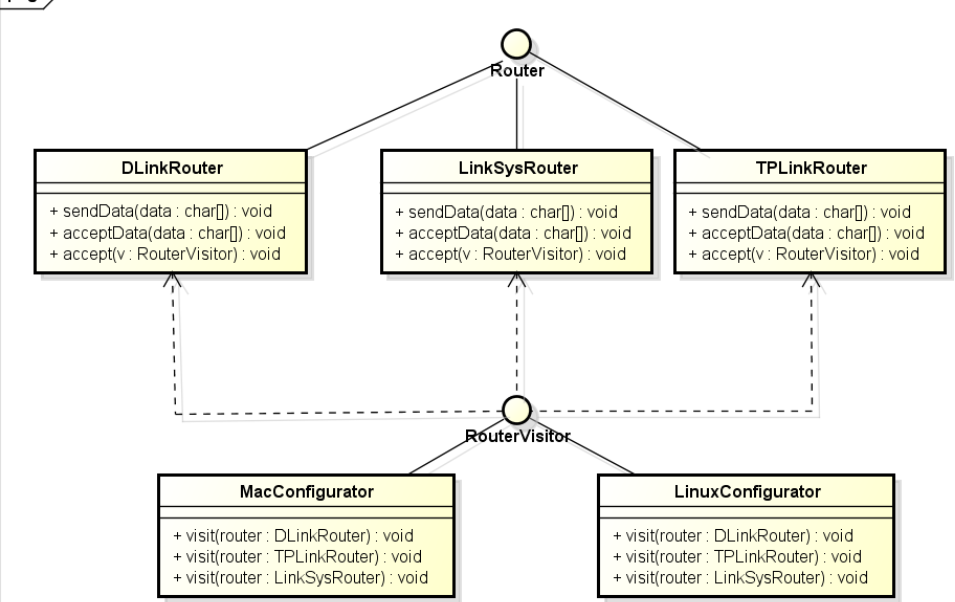
Потім просто використовуємо в імплементації:

House house = new ConcreteWallHouse();

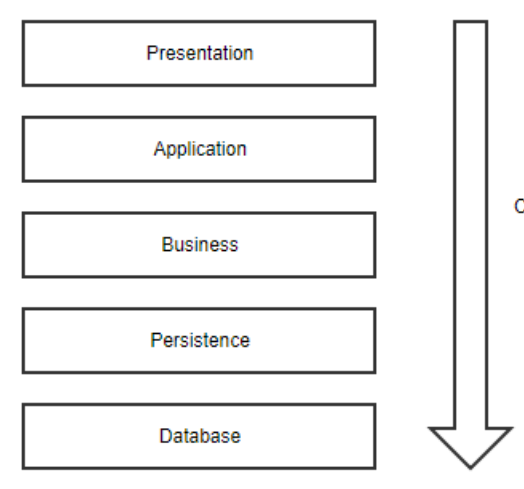
house.buildhouse();

Приклад з джавви – InputStream \ OutputStream.

**Visitor** використовують, коли у нас є ієрахія обєктів і нам треба до неї добавити новий метод але так, щоб не міняти кучу коду в багатьох місцях цієї ієрархії. Наприклад у нас є Router інтерфейсякий імплементують якісь TPLinkRouter, DLinkRouter і так далі. І для кожного з цих роутерів є конфігурації для різних операційних систем. Якщо ми добавимо конфігкрації в Router ігтерфейс, то тоді кожного разу як нам треба буде нову конфігурацію – ми будем повинні додавати цей метод в інтерфейс і відповідно по всій ієрархії. Краще буде не зберігати методи конфігурації в інтерфейсі, а винести їх в окремі класи які будуть передаватися в конкретний метод роутера як RouterVisitor тип. Наприклад маємо WindowsRouterConfiguration i LinuxRouterConfiguration. І тоді, якшо ми захочемо добавити нову конфігуріцію, наприклад MacOsRouterConfiguration, то я ніде не міняю ієрархію, а просто добавляю нову конфігурацію-класс яка також буде наслідуватися від RouterVisitor і імплементую в ній методи visit() для кожного з роутерів.



**Layered architecture** – головно ідея патерна полягає в тому, шо ми розбиваємо аплікуху на леєри, кожен з яких має свою респонсібіліті, колає настіпний леєр під собою і є сервісом для леєра який знаходиться на рівень вище. Кількість і типи леєрів можуть варіюватися але найчастіше ми маємо таку картину:



Presentation layer – зачасту предаставляє собою якийсь UI для юзера.

Application – представляє інтерфейс для Presentation шоб тому нічого не треба було знати про бізнес леєр. Так шоб в теорії можна було поміняти юайний стек і не чіпати нічого більше в аплікусі. Також аплікейшн леєр відіграє роль кординатора в плані що і де колати в бізнес леєрі.

Business layer – де ми сторимо модельки і логіку яка відноситься безпосередньо до бізнес задач.

Persistence layer – відповідає за підключення до БД і роботою з нею. (всякі конекшени, квері і так далі)

Database – сама база даних.

Плюси: багато девелоперів знайомі з патерном і аплікухи на його основі легко тестувати.

Мінуси: імплементація веде до моноліта, який з часом все важче сплітати. Також займає більше часу написання коду якшо нам треба шось добавити і прокинути це через всі леєри – якшо це простенький CRUD, то така архітектура є оверкілл для аплікухи.

**MVC** – model / view / controller pattern – використовують щоб розділити задачі аплікухи на різні частини і отримати loose coupling між леєрами. Це архітектурний патерн але він не репрезентує цілу аплікуху. Найбільше він має відношення до UI частини аплікації, так шо нам всеодно можуть бути потрібні бізнес, data access і так далі леєри. Так у нас є

**Model** в якій лежать класи і POJO і логіка яка каже контроллеру що дані оновлено. Model є **пасивною** (коли вью дає інпути контроллеру, той трігає модель і контроллер сповіщає вью про зміну стейту, тоді вью питає модель за новим стейтом) і **активно** (коли вью даю інпути контроллеру, той нотифікує модель, модель робить логіку і віддає якийсь івент для вью).

**View** – візуалізує дані які знаходяться в моделі.

**Controller** – слідкує за діями юзера і контролює флоу як дані потраплаються в модель і апдейтить view якшо дані змінилися. Таким чином він тримає модель і vіew роздільно.

**Inversion of control –** архітектурний підхід який каже, що обєкти які нам потрібні ми не створюємо самі, а делегуємо це комусь іншому. Тобто замість того шоб писати SomeObject obj = new SomeObj() це за нас робить хтось інший, а ми просто описуємо в себе в класі які залежності нам потрібні. Як правило це IoC контейнер. В ітозі ми маємо обєкти які контролюються контейнером і звичайні ті які контролює девелопер. Контейнер створює обєкти і контролює їх лайфсайкл.

**DI –** одна з реалізацій IoC. Є ще Factory method i Service Locator. Основна його ідея полягає в тому, що у нього є зборщик який в поле класу підставляє реалізацію потрібного інтерфейса. Є Condtructor Injection (депенденсі передається через конструктор класу), Setter Injection (контейнер колає дефолтний конструктор, а потім сеттер), Interface Injection (юзає рефлексію, більш затратний).

**Антипатерни**:

**Спагетті** **код** – коли ми маєм тучу іфів в іфах які іфами поганяють, чи постійні якісь переміщення через goto якщо ми не говоримо про джаву. Тобто це код який сплутаний-переалутаний і в якому тяжко розібратися.

**Лазанья** **код** – коли система має аж забагато леєрів і один або декілька з цих леєрів можна сказати, що лишні. Наприклад якшо леєр нічого не робить а просто ганяє дані з леєра в леєр через себе, то можна й без нього обійтися.

**Шотган** **surgery** – переводиться як стрільба дробью і тут навіть з назви можна зрозуміти, що це код який нам треба, наприклад, поміняти але він юзався таким чином, що нам прийдеться залізти в дуже багато місць шоб його змінити. Тобто це не одне місце яке ми міняємо і маємо спокій.

**Golden** **hammer** – це вже більш індивідуальна річ. Індивідуальна по відношенню до девелопера бо скажем людина на якомусь проекті чи підпроекті використовувала якисйсь підхід до вирішення задачі і потім робить то саме навіть якщо задача віддалено нагадує попередні. Тобто проблема полягає в тому, шо інша ситуація може мати інші нюанси і їх треба враховувати.

**God** **object** – коли ми маєм клас який робить всьо і для всіх. Наприклад, грубо говоря, має ендпойнти, лізе в базу, шукає шось в еластіку і шле імейли. Тобто клас не є якоюсь абстрактною одиницею і не відповідає принципу single responsebility.

**MVVM** – model view view model – патерн який розділяє логіку від відображення і забезпечує двухсторнній зв'язок між View i ViewModel що дозволяє швидко і автоматично реагувати на зміни в ViewModel (на відміну від MVC який має тільки односторонній звязок). Як правило цей патерн базований на патерні Observer який дозволяє нотифікувати View одразу як є зміни на ModelView. **ViewModel** – рівень який містить методи і проперті які мейнтейнять стан View і маніпулюють Model після дій на View. Різниця з MVC полягає в тому, що в MVC View читає Model на пряму і йде через Controller якшо треба маніпулювати даними і сама Model живе на сервері, а в MVVM наша ViewModel веде себе як проксі для Model при чому надаючи потрібний функціонал для View і як правило ViewModel живе на клієнті.