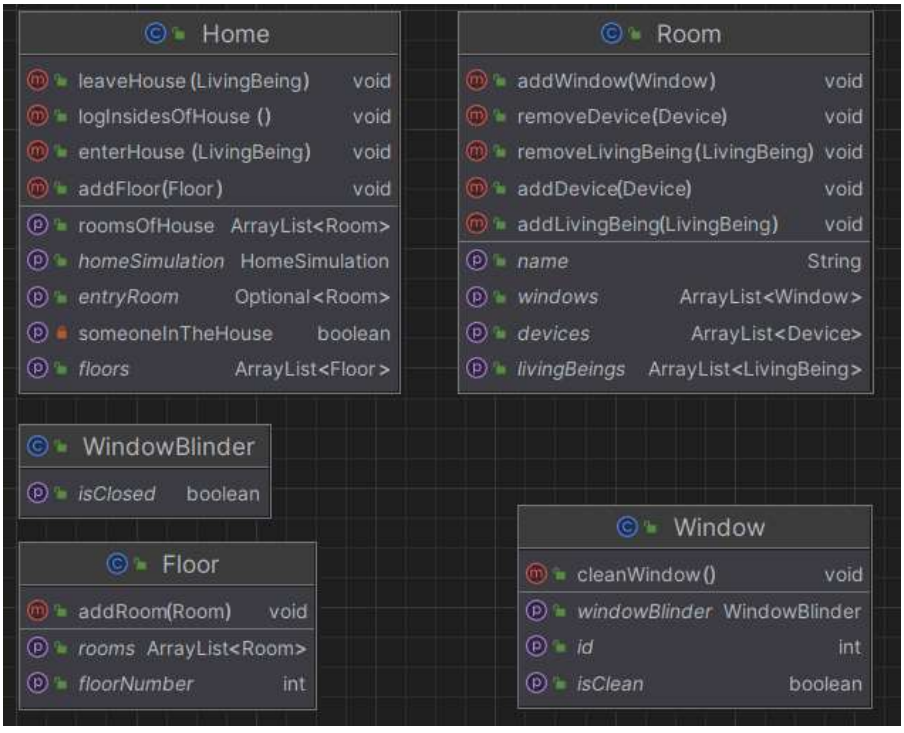
Projekt: Smart Home

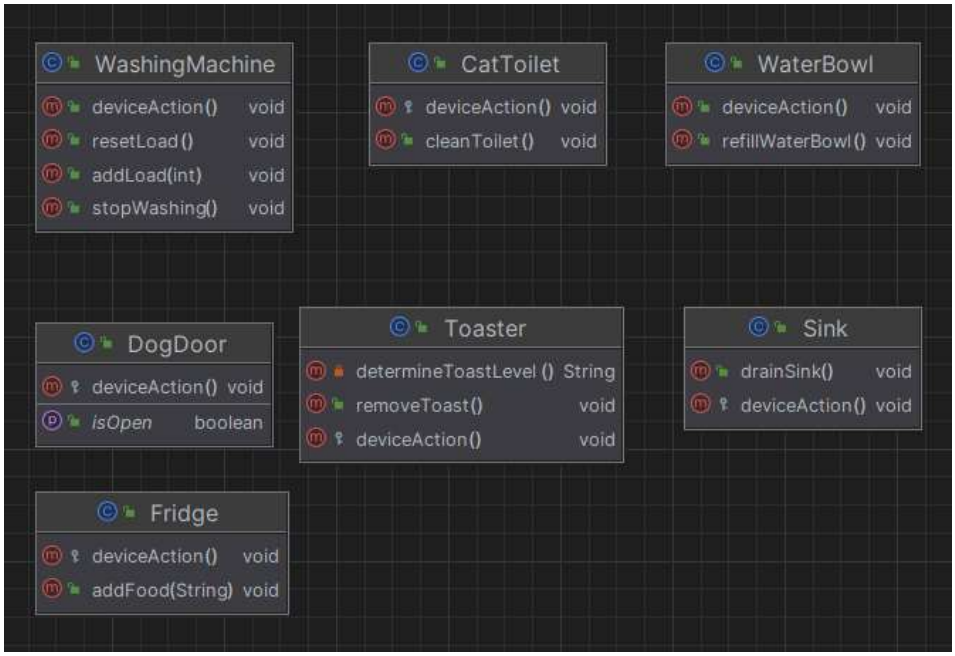
Last edited by **Cerman, Jakub** 2 months ago

Class diagram

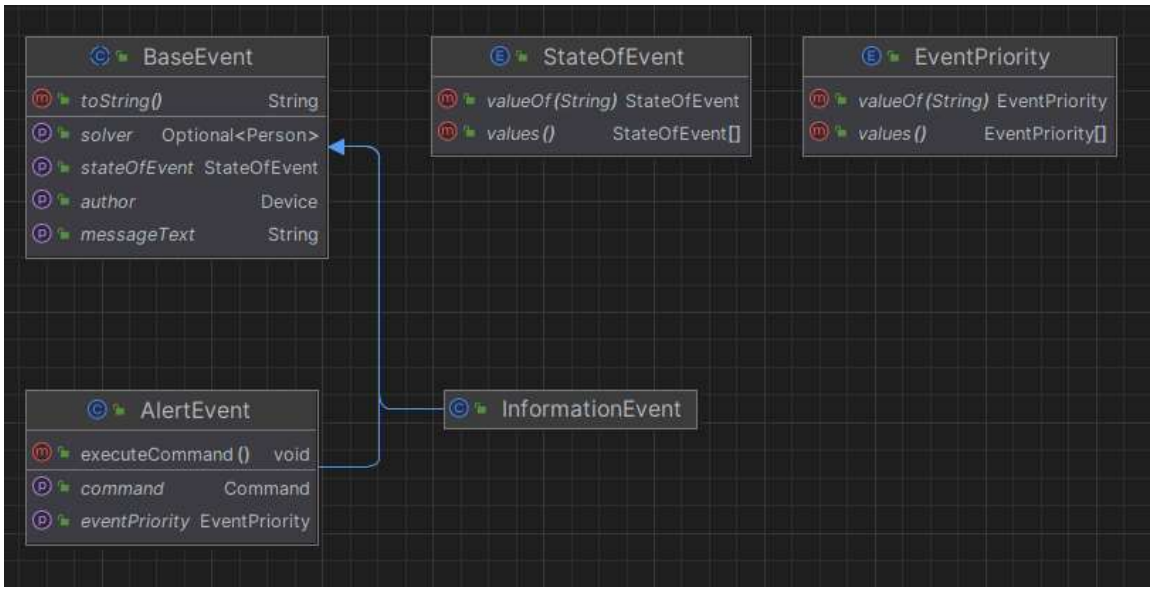
▼ Co obsahuje dům



▼ Devices



▼ **Eventy**



Class diagram podle kterého jsme vyvíjeli v drawio:

OMO_class_diagram_s_design_paterny_ucesany.drawio.pdf

POPÍS APLIKACE

Aplikace Smart Home je simulace chytré domácnosti, která obsahuje různé místnosti, zařízení a obyvatele. Zařízení mají API pro ovládání a sběr dat, které umožňuje sledovat jejich stav (Active, StandBy, Disabled) a spotřebu. Mohou obsahovat předměty (např. lednice jídlo). Obyvatelé (osoby a zvířata) provádějí aktivity, které ovlivňují zařízení nebo jiné osoby, a generují náhodné eventy.

Popis funkcionalit

- [x] **Jednotlivá zařízení v domu mají API na ovládání**. Zařízení mají stav, který lze měnit pomocí API na jeho ovládání. Akce z API jsou použitelné podle stavu zařízení.
- [x] **Zařízení muže mít i obsah**: lednice má jídlo, CD přehrávač má CD.
- [x] **Zařízení má stáv**: Active, StandBy, Disabled.
- [ ] **Zařízení má API na sběr dat**: spotřeba elektřiny, vody a teploty.
- [x] **Person a Animal mohou provádět aktivity(akce),** které mají nějaký efekt na zařízení nebo jinou osobu.
- [x] Jednotlivá Device, Person a Animal se v každém okamžiku vyskytují v jedné místnosti (pokud nesportují) a náhodně generují eventy (eventem může být důležitá informace a nebo alert).
- [x] **Eventy jsou přebírány a odbavovány vhodnou osobou (osobami)** nebo zařízením (zařízeními).
  - [ ] čidlo na vítr (vítr) => vytažení venkovních žaluzií
  - [ ] jistič (výpadek elektřiny) => vypnutí všech nedůležitých spotřebičů (v provozu zůstávají pouze ty nutné)
  - [ ] čidlo na vlhkost (prasklá trubka na vodu) => máma -> zavolání hasičů, táta -> uzavření vody;dcera -> vylovení křečka
  - [ ] Miminko potřebuje přebalit => táta se skrývá, máma -> přebalení
  - [ ] Zařízení přestalo fungovat => ...

- [ ] V lednici došlo jídlo => ...
- [ ] **Vygenerování reportů**:
  - [ ] HouseConfigurationReport: veškerá konfigurační data domu zachovávající hieararchii - dům -> patro -> místnost -> okno -> žaluzie atd. Plus jací jsou obyvatelé domu.
  - [ ] EventReport: report eventů, kde grupujeme eventy podle typu, zdroje eventů a jejich cíle (jaká entita event odbavila)
  - [ ] ActivityAndUsageReport: Report akcí (aktivit) jednotlivých osob a zvířat, kolikrát které osoby použily které zařízení.
  - [ ] ConsumptionReport: Kolik jednotlivé spotřebiče spotřebovaly elektřiny, vody. Včetně finančního vyčíslení.
- [ ] **Při rozbití zařízení musí obyvatel domu prozkoumat dokumentaci k zařízení**: najít záruční list, projít manuál na opravu a provést nápravnou akcí (např. Oprava svépomocí, koupě nového atd.). Manuály zabírají mnoho místa a trvá dlouho než je najdete.
- [ ] Rodina je aktivní a volný čas tráví zhruba v poměru (50% používání spotřebičů v domě a 50% sport kdy používá sportovní náčiní kolo nebo lyže). Když není volné zařízení nebo sportovní náčiní, tak osoba čeká.

ANALÝZA - výběr paternů

0. Command patern

Rozhraní `Command` definuje metodu `execute()`, kterou implementují všechny konkrétní příkazy.

Každá třída představuje konkrétní akci:

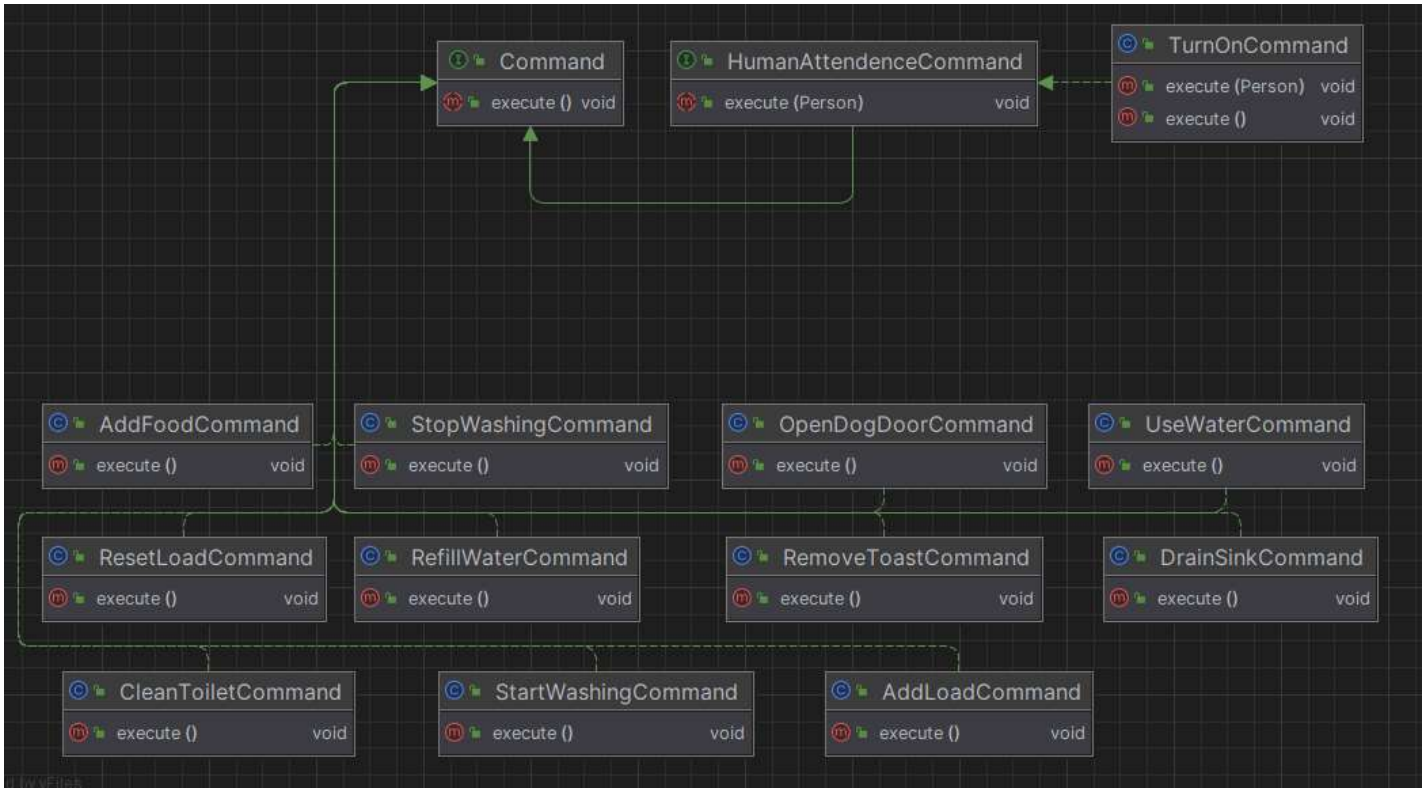
1. `AddFoodCommand` — přidává jídlo.
2. `StartWashingCommand` — spouští praní.
3. `TurnOnCommand` — zapíná zařízení.
4. `StopWashingCommand`, `CleanToiletCommand`, `RemoveToastCommand` a další implementují různé specifické akce.

HumanAttendanceCommand `je specializace, která umožňuje vykonávání příkazů s kontextem osoby (`Person`), například obsluhující uživatel.

1. Odděluje požadavek (příkaz) od jeho vykonání.
2. Umožňuje snadné zpracování příkazů (například jejich plánování, ukládání a případné vracení zpět — undo).
3. Škálovatelný a flexibilní design pro přidávání nových příkazů bez změny existujícího kódu.

Systémy domácí automatizace, kde každé zařízení nebo akce má vlastní příkaz, například zapnutí světel, doplnění vody, spuštění praní.

▼ **Click to expand**



Rozhraní `Consumable` **a** `DeviceUsage` rozdělují funkcionality zařízení:

1. Consumable `obsahuje metody pro měření spotřeby (`measureAccumulatedConsumption`,` measureCurrentConsumption`).
2. `DeviceUsage` obsahuje metody pro ovládání zařízení (`turnOnDevice`, `turnOffDevice`, `useDevice`).

1. Builder (Stavitel).

Třída `HomeBuilder` poskytuje metody pro krokové sestavení domu, například:

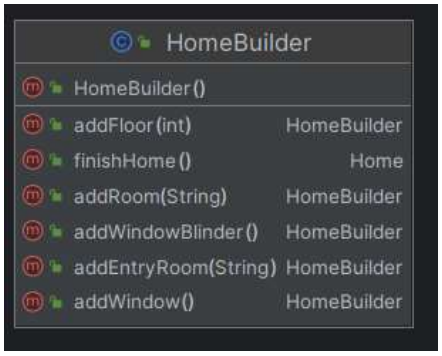
1. addFloor(int)` pro přidání podlaží.

2. addRoom(String)` pro přidání místností.
3. addWindow() `nebo` addWindowBlinder()` pro přidání oken a žaluzií.
4. Metoda `finishHome()` vrací hotový objekt `Home`.

Když potřebujete vytvořit objekt s mnoha volitelnými vlastnostmi nebo složitou strukturou krok za krokem.

Umožňuje flexibilní a čitelný proces vytváření složitých objektů (například domů), aniž by se použily přetížené konstruktory.

▼ **Click to expand**



2. Template method

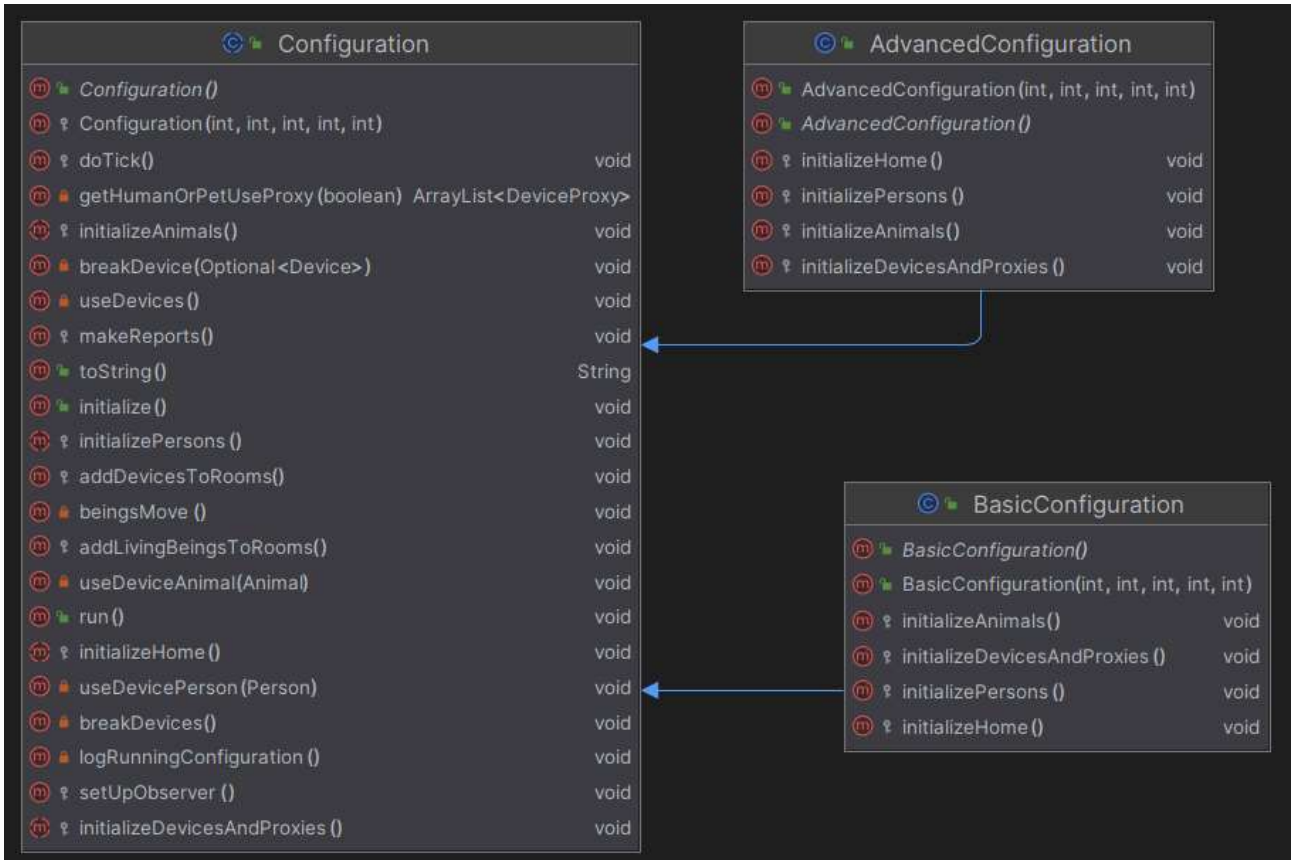
Třída `Configuration` definuje obecnou strukturu algoritmu s metodami jako:

1. initializePersons() `,` initializeAnimals() `,` initializeDevicesAndProxies()` pro inicializaci různých částí systému.
2. run() `a` doTick()` kombinují různé kroky pro provádění procesů.
3. Podtřídy `AdvancedConfiguration` a `BasicConfiguration` přizpůsobují specifické kroky inicializace, například způsob práce s osobami nebo zařízeními.

Když chcete vytvořit obecný algoritmus s možností přizpůsobení určitých kroků ve specializovaných podtřídách.

Sdílená logika je implementována v základní třídě (`Configuration`), zatímco detaily implementace jsou ponechány na podtřídách.

▼ **Click to expand**



3. Decorator

Třída `Sensor` definuje společné rozhraní a základní vlastnosti pro všechny senzory, například:

1. measureCurrentConsumption(Person)`.
2. measureAccumulatedConsumption(Person)`.

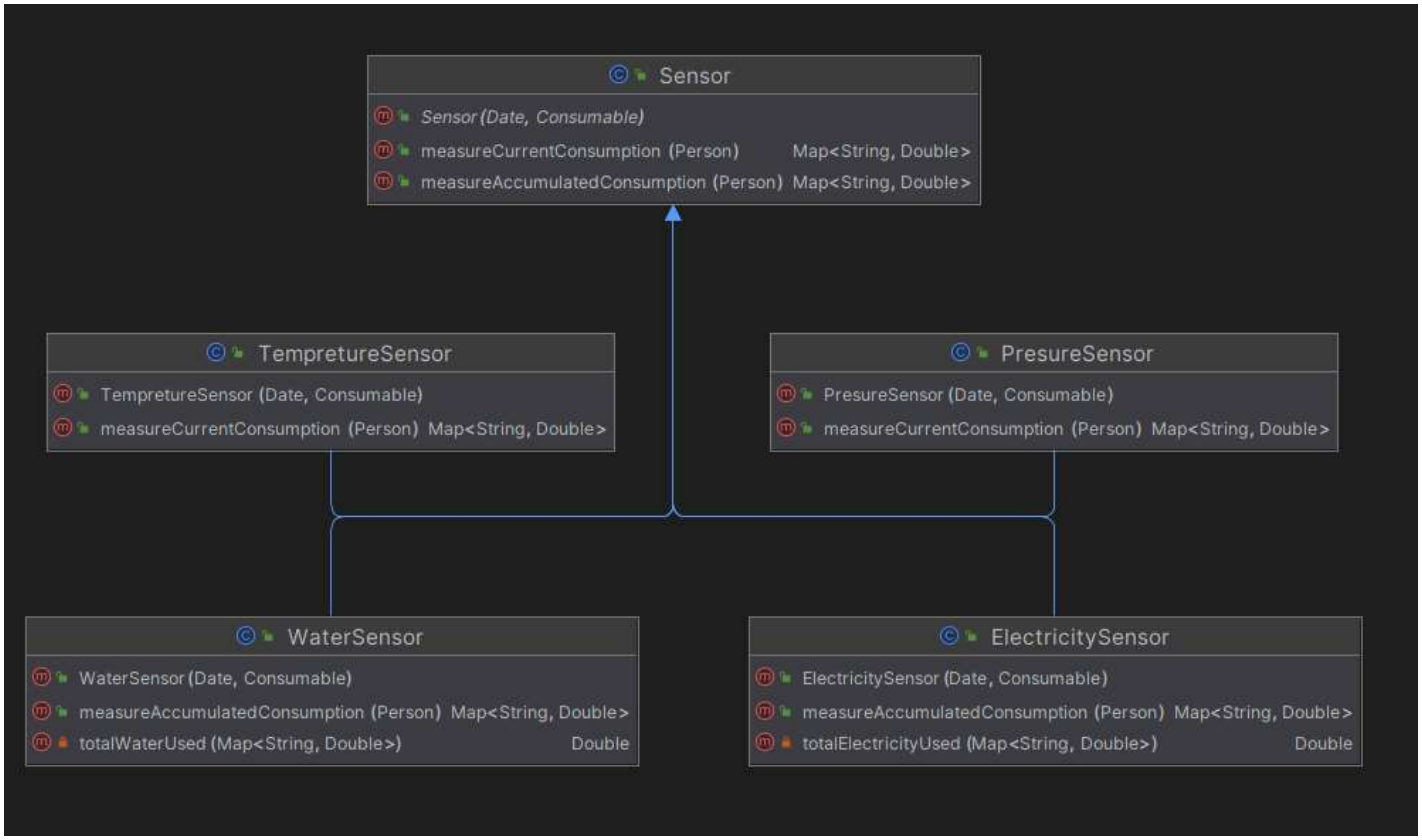
3. Podtřídy jako `TemperatureSensor`, `PressureSensor`, `WaterSensor`, a `ElectricitySensor` přidávají specifické chování, například: totalWaterUsed() `v` WaterSensor `a` totalElectricityUsed() `v` ElectricitySensor`.

Factory Method umožňuje vytvářet konkrétní typy senzorů podle potřeby bez přímé závislosti na jejich implementaci.

Základní třída (`Sensor`) poskytuje jednotné rozhraní, zatímco podtřídy implementují konkrétní logiku.

Když potřebujete flexibilně vytvářet různé typy objektů (např. senzory) na základě specifického kontextu.

▼ **Click to expand**



4. Facade (Fasáda).

Třída `HomeSimulation` poskytuje jednoduché rozhraní pro ovládání funkcí domácnosti,například:

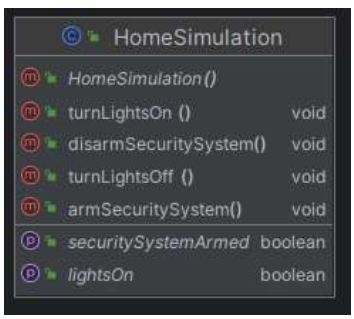
1. `turnLightsOn()` a `turnLightsOff()` pro správu osvětlení.
2. `armSecuritySystem()` a `disarmSecuritySystem()` pro zabezpečovací systém.
3. Pole jako `securitySystemArmed` a `lightsOn` sledují stav jednotlivých systémů.

Když potřebujete zjednodušit přístup k více funkcím a podpořit čitelnost kódu při práci s komplexním systémem.

Fasáda zjednodušuje interakci s komplexním systémem domácnosti tím, že skrývá detaily implementace a poskytuje jednoduché a přehledné API.

Umožňuje oddělení klientského kódu od složité logiky systému.

▼ **Click to expand**



5. Factory Method (Tovární metoda).

Třída `EventFactory` poskytuje metody pro vytvoření různých typů událostí:

1. `createInformationEvent(String, Device)` pro vytvoření informační události (`InformationEvent`).
2. `createAlertEvent(String, Device, Command, EventPriority)` pro vytvoření výstražné události (`AlertEvent`).

Když potřebujete oddělit proces vytváření objektů od jejich konkrétní implementace a chcete flexibilně vytvářet různé typy událostí.

Centralizuje logiku vytváření objektů `InformationEvent` a `AlertEvent`, což usnadňuje správu a rozšíření.

Klientský kód nemusí znát detaily konstrukce objektů, pouze zavolá odpovídající metodu továrny.

▼ **Click to expand**

6. Observer (Pozorovatel).

Rozhraní `Observer` definuje metody, které reagují na události, například:

1. `handleEvent(AlertEvent)` pro výstražné události.
2. `handleEvent(InformationEvent)` pro informační události. Každý pozorovatel implementuje vlastní logiku zpracování těchto událostí.

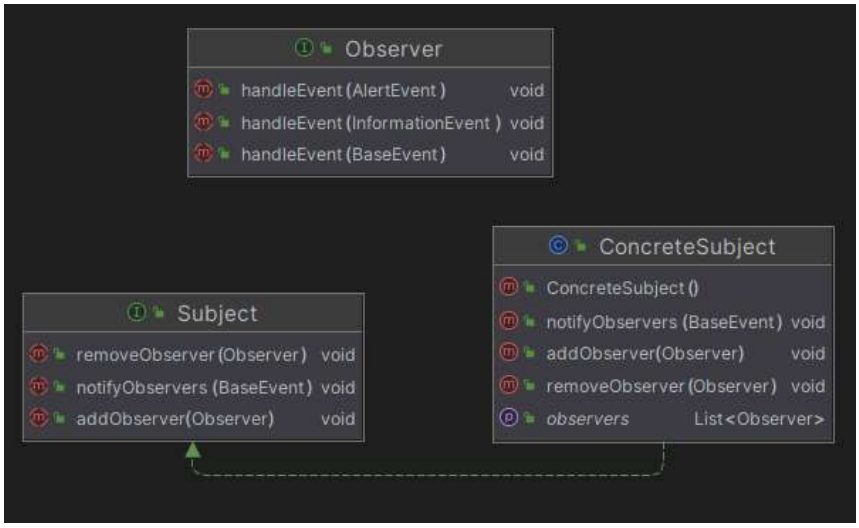
Rozhraní `Subject` umožňuje spravovat seznam pozorovatelů:

1. `addObserver(Observer)` pro přidání pozorovatele.
2. `removeObserver(Observer)` pro odstranění pozorovatele.
3. `notifyObservers(BaseEvent)` pro notifikaci všech registrovaných pozorovatelů. `ConcreteSubject` implementuje tuto logiku a uchovává seznam pozorovatelů (`observers`).

Když chcete mít flexibilní systém, kde více objektů (pozorovatelů) reaguje na změny stavu jiného objektu (subjektu).

Umožňuje implementaci vzoru Publish/Subscribe, kdy subjekt oznamuje změny více pozorovatelům, aniž by znal jejich konkrétní implementaci.

▼ **Click to expand**



7. Proxy (Zástupce).

Třída `DeviceProxy` funguje jako prostředník pro přístup k objektu `Device` a poskytuje kontrolovaný přístup k jeho funkcím.

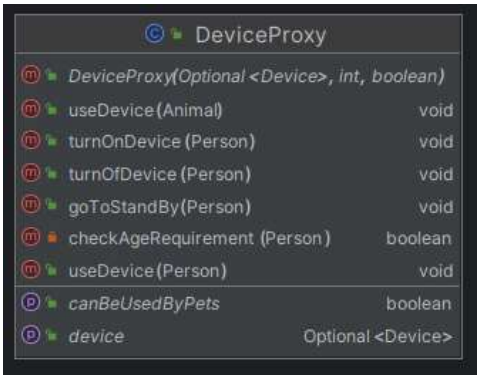
Implementuje metody jako:

1. `useDevice(Person)` a `useDevice(Animal)` pro interakci s zařízením.
2. `turnOnDevice(Person)` a `turnOffDevice(Person)` pro zapnutí a vypnutí zařízení.
3. `checkAgeRequirement(Person)` pro ověření věku uživatele zařízení. Obsahuje atribut `device` (odkaz na skutečné zařízení) a další logiku, jako je pole `canBeUsedByPets` .

Když chcete kontrolovat přístup nebo přidat další logiku (např. validaci) při práci s objektem, aniž byste měnili jeho implementaci.

Proxy přidává další vrstvy zabezpečení nebo logiky (např. ověřování uživatele nebo režim stand-by) před přístupem k reálnému objektu `Device` .

▼ **Click to expand**



8. State (Stav).

Třída `DeviceState` definuje rozhraní pro různé stavy zařízení a implementuje základní logiku, například:

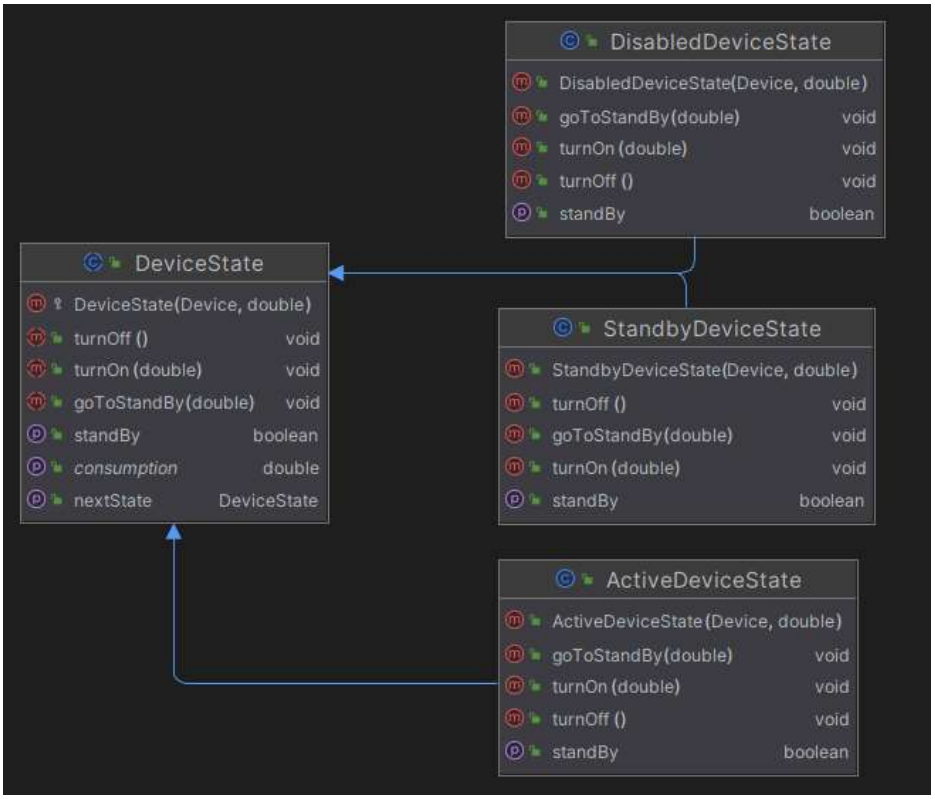
1. `turnOn(double)`, `turnOff()`, a `goToStandBy(double)`.
2. Pole `nextState` umožňuje přepínání mezi stavy. Konkrétní stavy (`DisabledDeviceState`, `ActiveDeviceState`, `StandbyDeviceState`) implementují specifické chování zařízení v daném stavu.

Když zařízení nebo objekt má několik stavů (např. aktivní, nečinný, zakázaný) a jeho chování se liší v závislosti na stavu.

Umožňuje měnit chování objektu dynamicky na základě jeho aktuálního stavu, aniž by se měnila jeho třída.

Odděluje logiku stavů do samostatných tříd, což zvyšuje čitelnost a usnadňuje rozšiřování.

▼ **Click to expand**



9. Strategy (Strategie).

Rozhraní `BehaviourStrategy` definuje metodu `executeBehavior(Person)` pro provádění konkrétního chování.

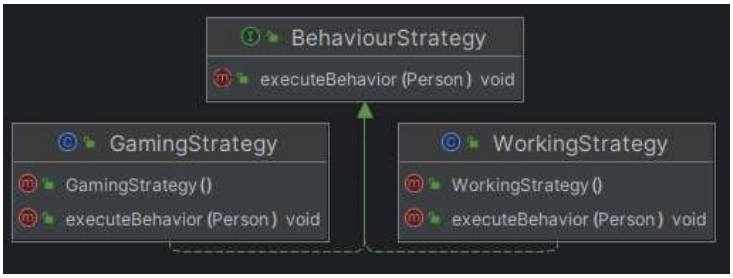
Konkrétní implementace strategií:

1. GamingStrategy` implementuje herní chování.
2. `WorkingStrategy` implementuje pracovní chování. Objekty mohou dynamicky měnit strategii chování podle kontextu.

Když chcete flexibilně měnit chování objektu, například přepínat mezi prací a hrou pro různé osoby nebo situace.

Umožňuje definovat různé algoritmy (strategie) a přepínat mezi nimi za běhu programu. Snižuje závislost na podmínkách `if-else` nebo `switch`.

▼ **Click to expand**



10. Visitor (Návštěvník).

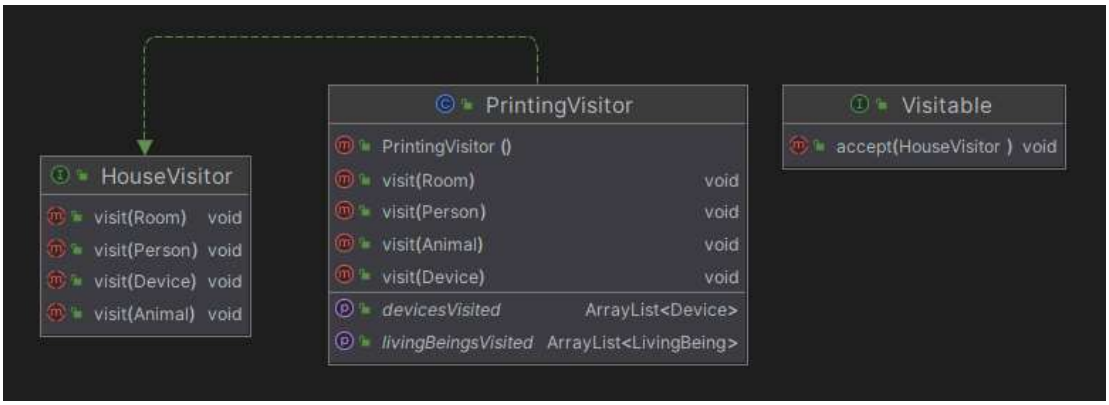
Rozhraní `HouseVisitor` definuje metody `visit` pro různé typy objektů (např. `Room`, `Person`, `Device`, `Animal`). Konkrétní implementace `PrintingVisitor` přidává specifické chování, například ukládání navštívených zařízení (`devicesVisited`) a bytostí (`livingBeingsVisited`).

Rozhraní `Visitable` definuje metodu `accept(HouseVisitor)`, která umožňuje přijetí návštěvníka a následnou aplikaci příslušné metody `visit`.

Když chcete umožnit různým operacím (např. tisk, kontrola) pracovat s heterogenními objekty bez změny jejich implementace.

Odděluje algoritmy (logiku návštěv) od datových struktur (`Room`, `Person`, atd.), což umožňuje snadné přidávání nových operací bez změny stávajících tříd.

▼ **Click to expand**



11. Singleton (Jednoduchý objekt).

Třída `CodeTester` implementuje návrhový vzor **Singleton** a zajišťuje, že existuje pouze jedna instance prostřednictvím pole `instance`.

Metody: 1. `addClassTest(ClassTester)` přidává testovací konfiguraci. 2. `testCode(boolean)` a `testEnabledAssertions()` provádějí testování kódu a kontrolu zapnutých podmínek (assertions). 3. Zajišťuje centralizovanou správu všech testů v systému.

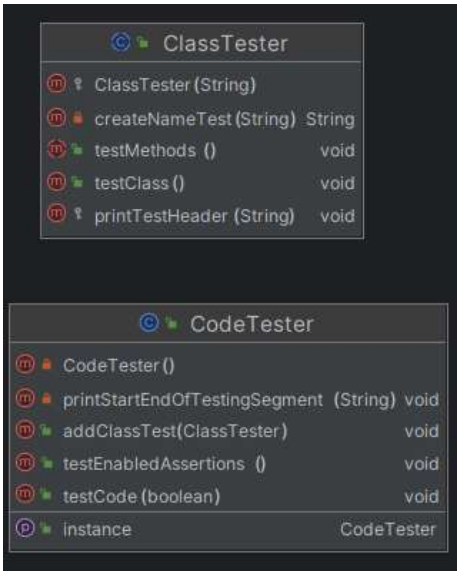
ClassTester:

1. Používá se pro testování na úrovni jednotlivých tříd.
2. Metody jako `testMethods()` a `testClass()` provádějí specifické testy.

Zaručuje, že existuje pouze jedna instance `CodeTester`, což umožňuje centralizované řízení testování.

Když potřebujete jednotný přístup ke správě testovacího systému prostřednictvím jedné instance.

▼ **Click to expand**


ClassTester
ClassTester (String)
createNameTest (String) String
testMethods () void
testClass () void
printTestHeader (String) void
CodeTester
CodeTester ()
printStartEndOfTestingSegment (String) void
addClassTest(ClassTester) void
testEnabledAssertions () void
testCode (boolean) void
instance CodeTester

Comments