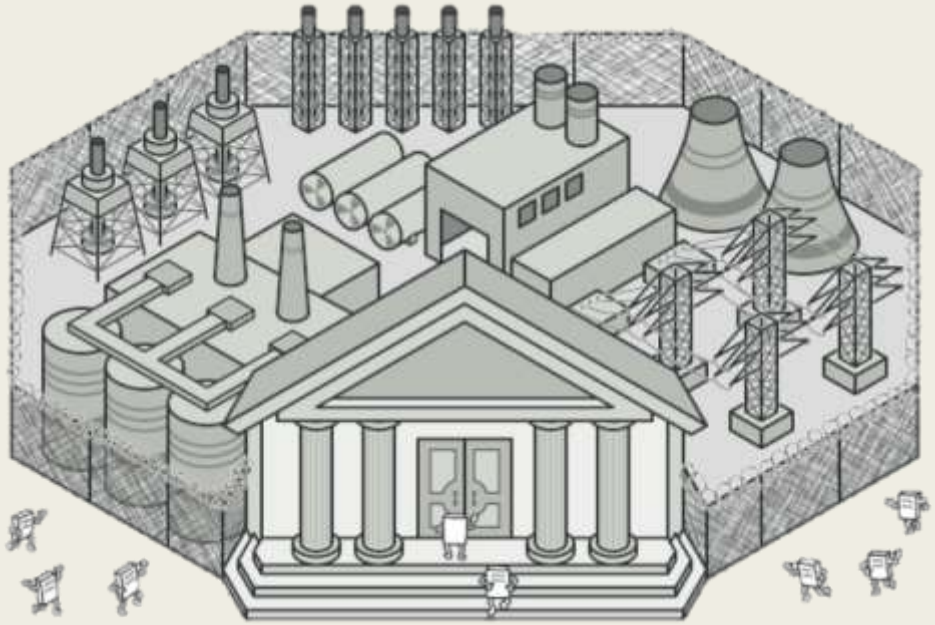


FACADE

Motivace – domácí kino

Při **spuštění** filmu musíme:

- Zatáhnout žaluzie
- Zapnout projektor
- Zapnout DVD přehrávač
- Zapnout ozvučení

Při **ukončení** filmu musíme:

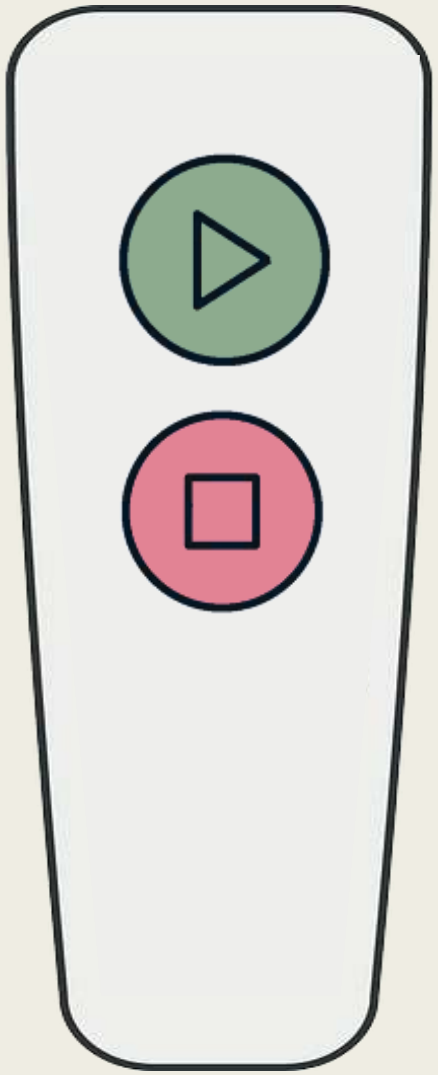
- Vypnout DVD přehrávač
- Vypnout ozvučení
- Vypnout projektor
- Vytáhnout žaluzie

Motivace – domácí kino

Jeden ovladač s funkcemi:

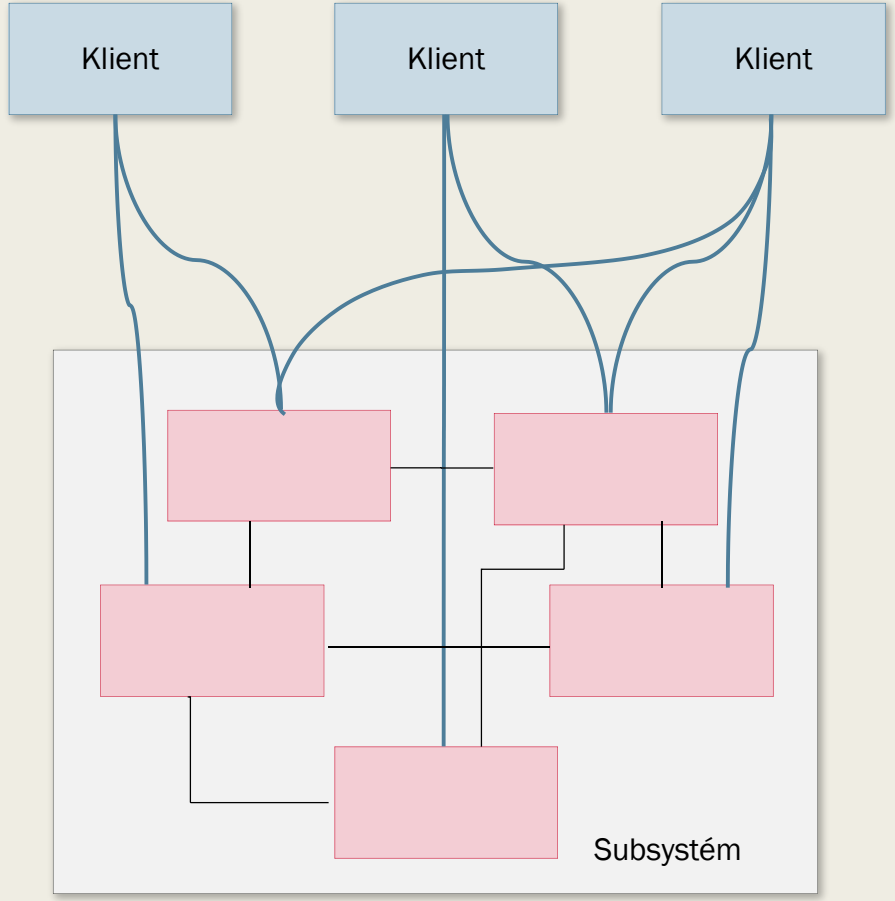
- **Play**
  - Zařídí všechno pro **spuštění** filmu.
- **Stop**
  - Zařídí všechno pro **ukončení** filmu

Zbylé ovladače si ponecháme.



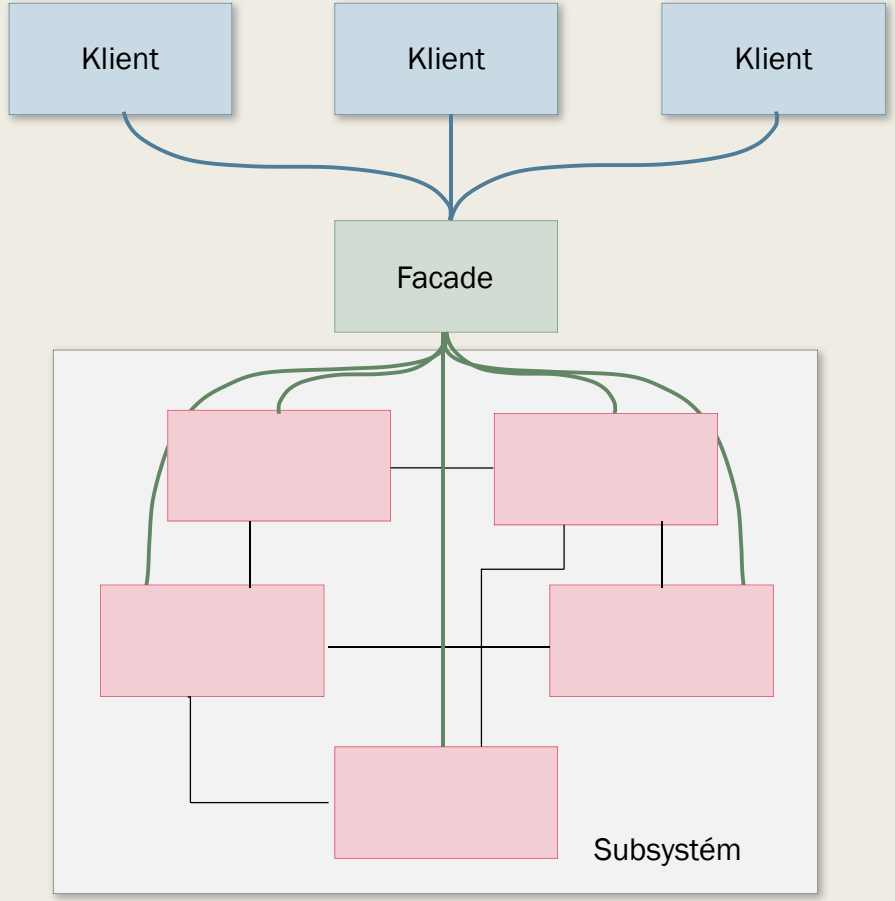
Problém

- Máme složitý subsystém.
  - *Velké množství objektů (tříd a jejich metod).*
- Klienti přistupují přímo k subsystému.
  - *Musí inicializovat objekty.*
  - *Držet si přehled o závislostech.*
  - *Volat metody ve správném pořadí.*
- Logika klientských tříd je svázaná s implementací subsystému.



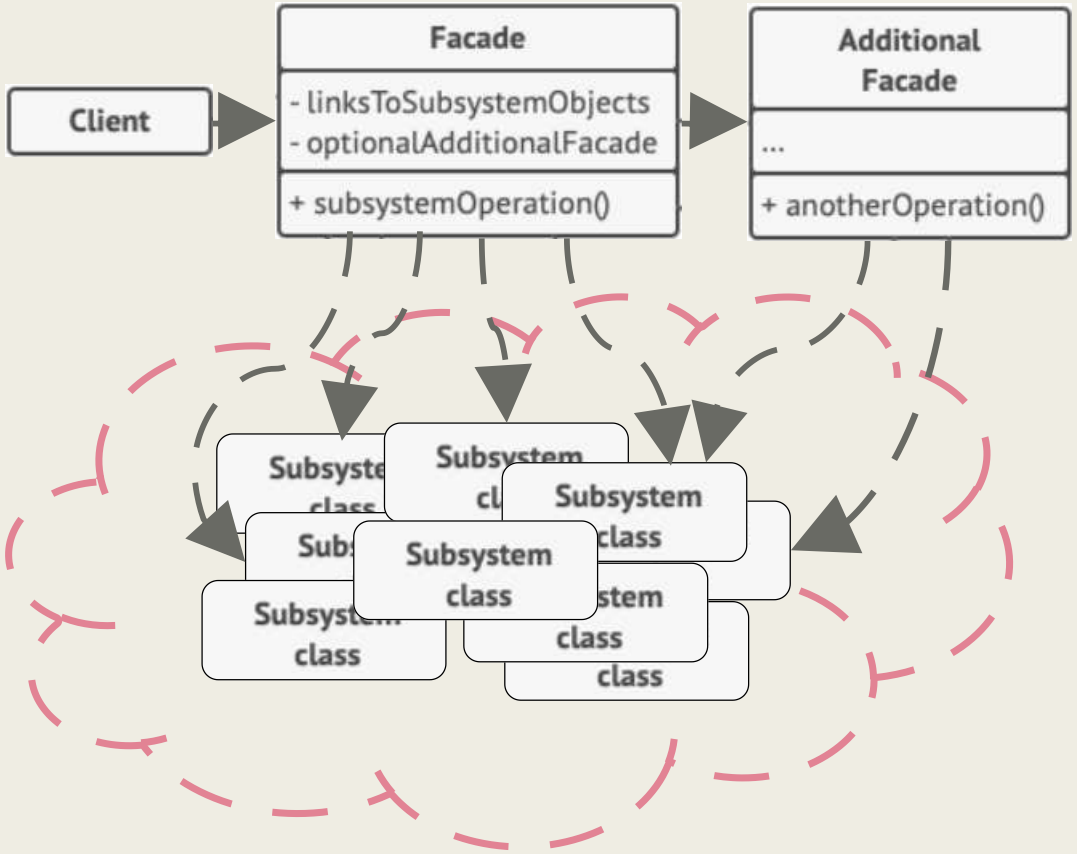
Řešení

- Minimalizujeme komunikaci a závislosti mezi subsystémy použitím Fasády.
- Fasáda = objekt poskytující zjednodušené rozhraní pro komunikaci se subsystémem.
- Klienti nevědí s jakými metodami subsystému pracují.



Struktura

- Komplexní **subsystém** s velkým množstvím objektů.
- **Fasáda** zprostředkovává přístup k podmnožině funkcí subsystému.
  - *Může přidávat funkcionalitu.*
  - *Neskrývá přístup k metodám subsystému.*
- **Klient** používá subsystém skrze fasádu.
- **Další fasáda** zprostředkovává přístup k jiné části subsystému.
  - *Další fasády mohou volat jak klienti, tak jiné fasády.*



Struktura

- Třídy subsystému implementují funkcionalitu subsystému.
- Dostávají pokyny od Fasády.
- O Fasádě nic nevědí, nemají na ní referenci.

```
class Subsystem1:

    def operation1(self) -> str:
        return "Subsystem1: Ready!"

    def operation_n(self) -> str:
        return "Subsystem1: Go!„

    # ...

class Subsystem2:

    def operation1(self) -> str:
        return "Subsystem2: Get ready!"

    def operation_z(self) -> str:
        return "Subsystem2: Fire!„

    # ...
```

Struktura

```
class Facade:

    def __init__(self) -> None:
        self._subsystem1 = Subsystem1()
        self._subsystem2 = Subsystem2()

    def operation(self) -> str:
        results = []
        results.append("Facade initializes subsystems:")
        results.append(self._subsystem1.operation1())
        results.append(self._subsystem2.operation1())

        results.append("Facade orders subsystems to perform the action:")
        results.append(self._subsystem1.operation_n())
        results.append(self._subsystem2.operation_z())

        return "\n".join(results)
```

- Fasáda ví, které třídy jsou zodpovědné za požadavky klientů.
- Deleguje klientovy požadavky odpovídajícímu objektu subsystému.

```
class Subsystem1:

    def operation1(self) -> str:
        return "Subsystem1: Ready!"

    def operation_n(self) -> str:
        return "Subsystem1: Go!„

    # ...

class Subsystem2:

    def operation1(self) -> str:
        return "Subsystem2: Get ready!"

    def operation_z(self) -> str:
        return "Subsystem2: Fire!„

    # ...
```

Struktura

```
class Facade:

    def __init__(self) -> None:
        self._subsystem1 = Subsystem1()
        self._subsystem2 = Subsystem2()

    def operation(self) -> str:
        results = []
        results.append("Facade initializes subsystems:")
        results.append(self._subsystem1.operation1())
        results.append(self._subsystem2.operation1())

        results.append("Facade orders subsystems to perform the action:")
        results.append(self._subsystem1.operation_n())
        results.append(self._subsystem2.operation_z())

        return "\n".join(results)
```

```
def client_code(facade: Facade) -> None:
    facade = Facade()
    print(facade.operation(), end="")
```

```
class Subsystem1:

    def operation1(self) -> str:
        return "Subsystem1: Ready!"

    def operation_n(self) -> str:
        return "Subsystem1: Go!„

    # ...

class Subsystem2:

    def operation1(self) -> str:
        return "Subsystem2: Get ready!"

    def operation_z(self) -> str:
        return "Subsystem2: Fire!„

    # ...
```

Výhody

Zjednodušuje práci se subsystémem.

Redukuje závislost mezi klienty a subsystémem.

Umožňuje vrstvení subsystémů.

Nebrání klientům používat třídy subsystému.

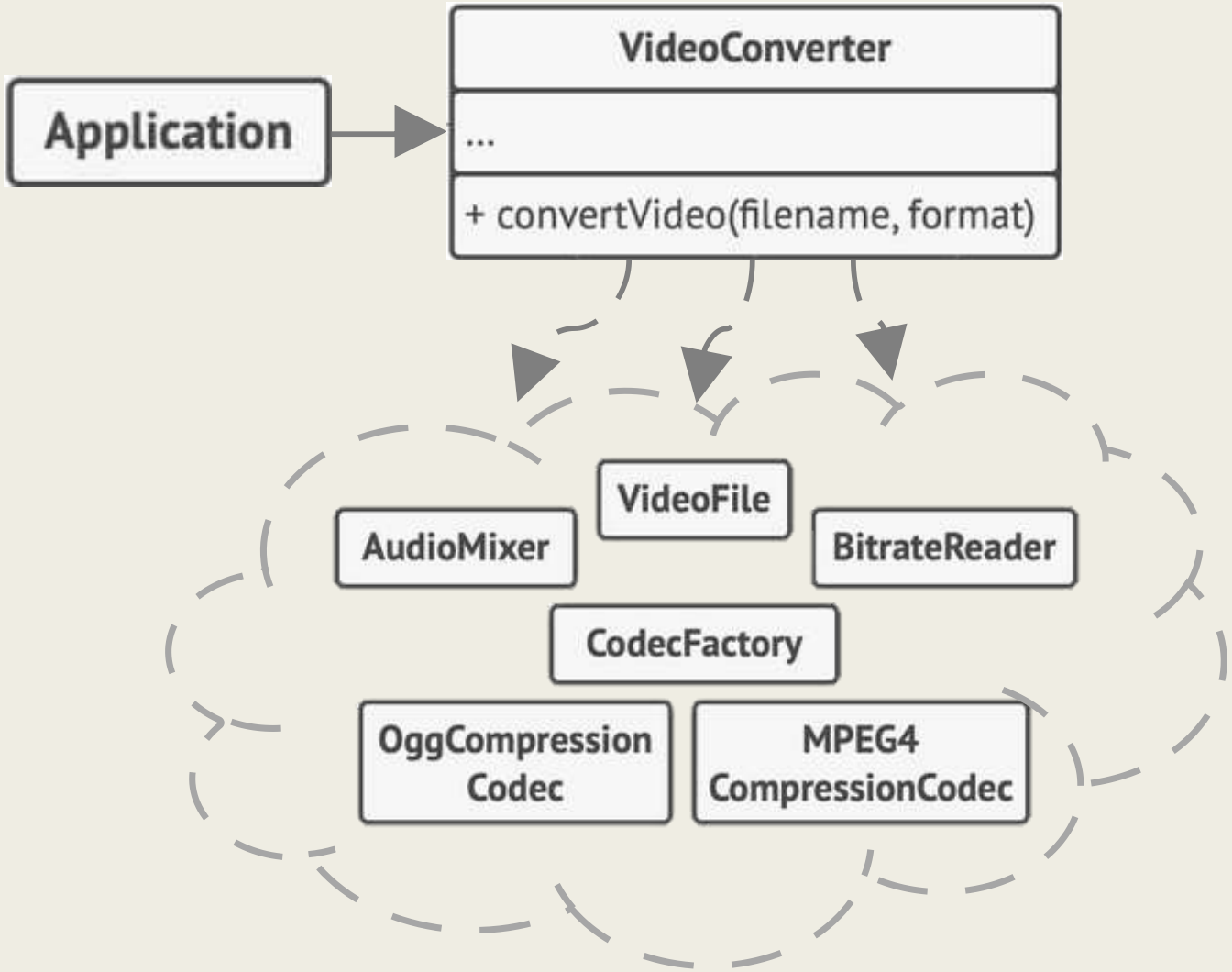
Kdy použít?

- Chceme omezené, ale jednoduché rozhraní ke komplexnímu subsystému
- Chceme subsystémy vrstvit.
- Chceme oddělit logiku klientského kódu od závislosti na subsystému.

Na co si dát pozor?

- Fasáda není zapotřebí.
- Z Fasády se stane God objekt.
  - *Anti-pattern, který „dělá moc věcí, a ví toho moc“.*
- Fasáda zakrývá rozhraní subsystému.
- Subsystém o Fasádě ví.

Příklad


Application
VideoConverter
...
+ convertVideo(filename, format)
VideoFile
AudioMixer
BitrateReader
CodecFactory
OggCompression Codec
MPEG4 CompressionCodec

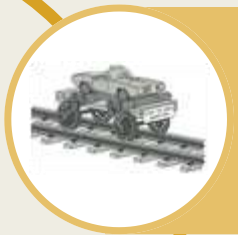
Příklad

```
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)
```

```
class Application is
    method main() is
        convertor = new VideoConverter()
        mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
        mp4.save()
```

```
class VideoFile
// ...

class OggCompressionCodec
// ...

class MPEG4CompressionCodec
// ...

class CodecFactory
// ...

class BitrateReader
// ...

class AudioMixer
// ...
```

Související vzory



Adapter
Oba mění rozhraní objektů. Adapter umožňuje komunikaci objektům s nekompatibilním rozhraním. Fasáda zjednodušuje rozhraní.

Singleton
Fasáda může být implementována jako singleton. Jeden objekt Fasády většinou stačí.

Abstract factory
Může být alternativa Fasády, pokud chceme schovat před klienty tvorbu objektů subsystému.

Mediator
Oba se snaží organizovat komunikaci mezi velkým množstvím propojených tříd. Fasáda pouze zjednodušuje rozhraní subsystému a subsystém o ní neví, zatímco mediátor centralizuje komunikaci a objekty vědí pouze o něm.

Shrnutí

Název	Facade
Problém	Klienti přímo přistupují k složitému rozhraní subsystému.
Řešení	Třída poskytující jednoduché rozhraní pro komunikaci se subsystémem.
Příklady	Video Converter
Související vzory	Abstract factory, Singleton, Adapter, Mediator

Děkuji za pozornost.

Zdroje

- GAMMA, Erich, Richard HELM, Ralph JOHNSON a John VLISSIDES. *Design patterns: elements of reusable object-oriented software* [online]. Boston: Addison-Wesley, 1995 [cit. 2023-03-03]. ISBN 02-016-3361-2. Dostupné z: http://www.javier8a.com/itc/bd1/articulo.pdf
- Facade. *Refactoring Guru* [online]. Kamianets-Podilskyi: Alexander Shvets, c2014-2023 [cit. 2023-03-03]. Dostupné z: https://refactoring.guru/design-patterns/facade
- ZHART, Dmitry. Facade. In: *Refactoring Guru* [online]. Kamianets-Podilskyi: Alexander Shvets, c2014-2023 [cit. 2023-03-03]. Dostupné z: https://refactoring.guru/design-patterns/facade
- ZHART, Dmitry. Structure. In: *Refactoring Guru* [online]. Kamianets-Podilskyi: Alexander Shvets, c2014-2023 [cit. 2023-03-03]. Dostupné z: https://refactoring.guru/design-patterns/facade
- ZHART, Dmitry. Example. In: *Refactoring Guru* [online]. Kamianets-Podilskyi: Alexander Shvets, c2014-2023 [cit. 2023-03-03]. Dostupné z: https://refactoring.guru/design-patterns/facade