Template Method

Aurel Farkašovský

Příklad ze života

- Čaj
  1. uvař vodu

     umyj nádobí
  2. dej do hrnku sáček čaje
  3. zalij hrnek
  4. přisyp cukr a citrón

- Káva
  1. uvař vodu

     popovídej si se zákazníkem
  2. nasyp kávu do hrnku
  3. zalij hrnek
  4. přidej cukr nebo mléko

```
uvař vodu
```

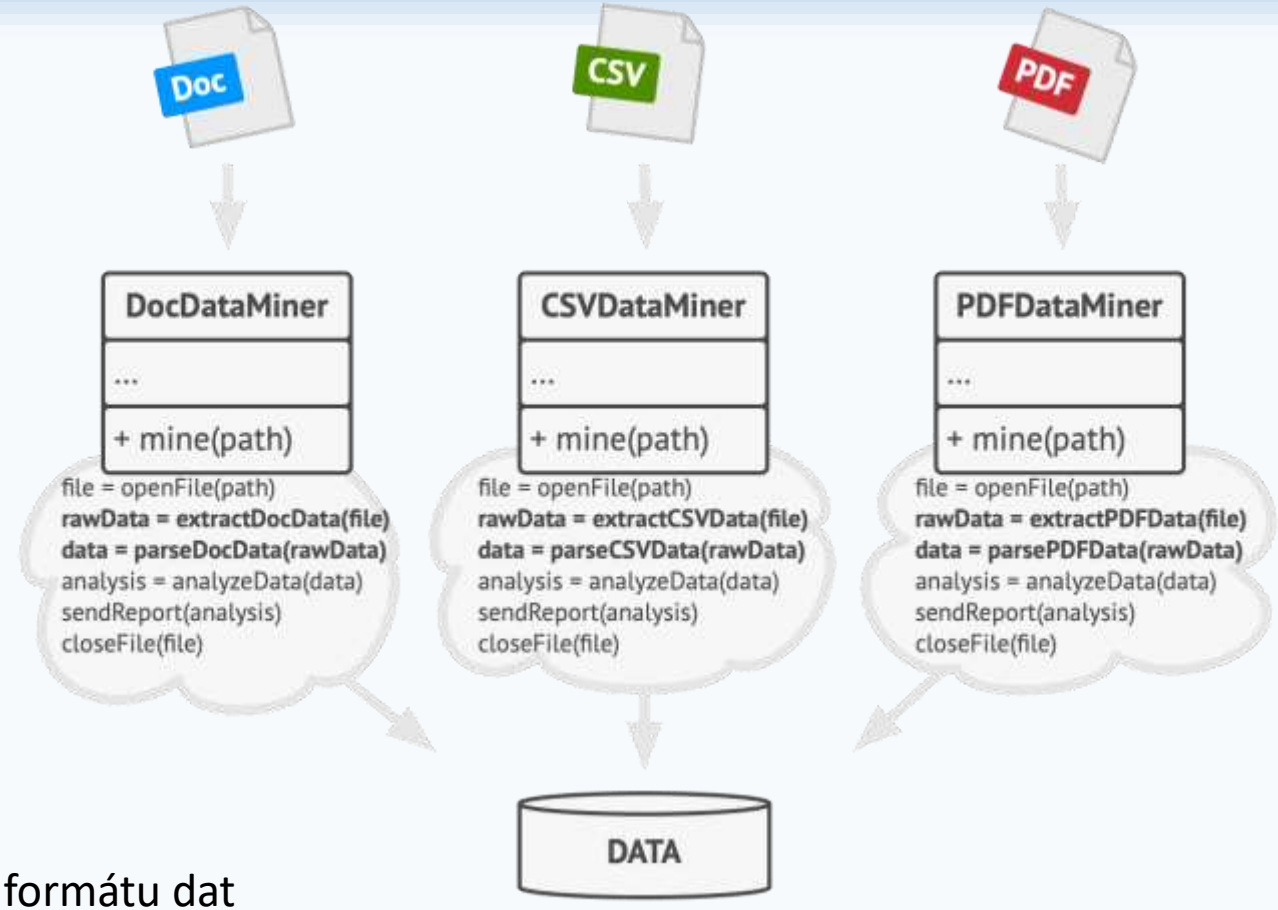


```
dej základ do hrnku
zalij hrnek
přidej doplňky
```

```
základ:  vložit čaj
doplňky: cukr, citrón
```

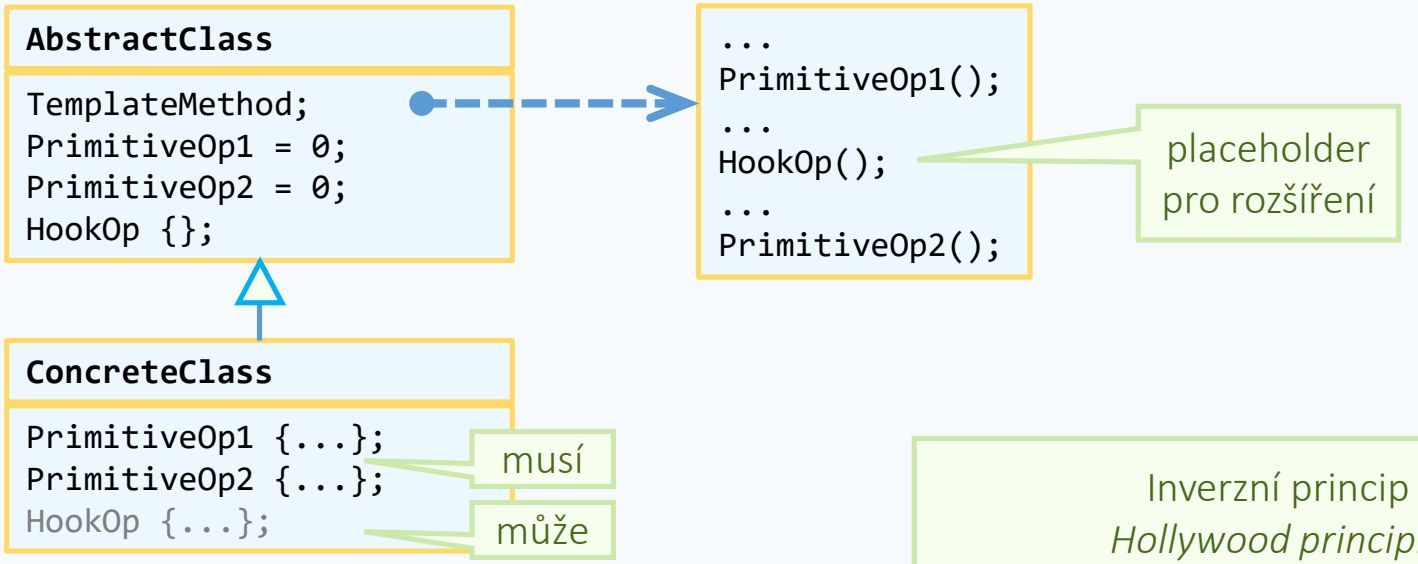
```
základ:  nasypat kávu
dopňky:  cukr, mléko
```

Příklad ze života informatika



- stejná struktura
  - konkrétní kroky záleží na formátu dat
- možné řešení
  - abstraktní metoda ProcessData
    - v každém potomkovi se znovu píše celá implementace
  - nemusí se dodržovat stejná osnova
- Template method

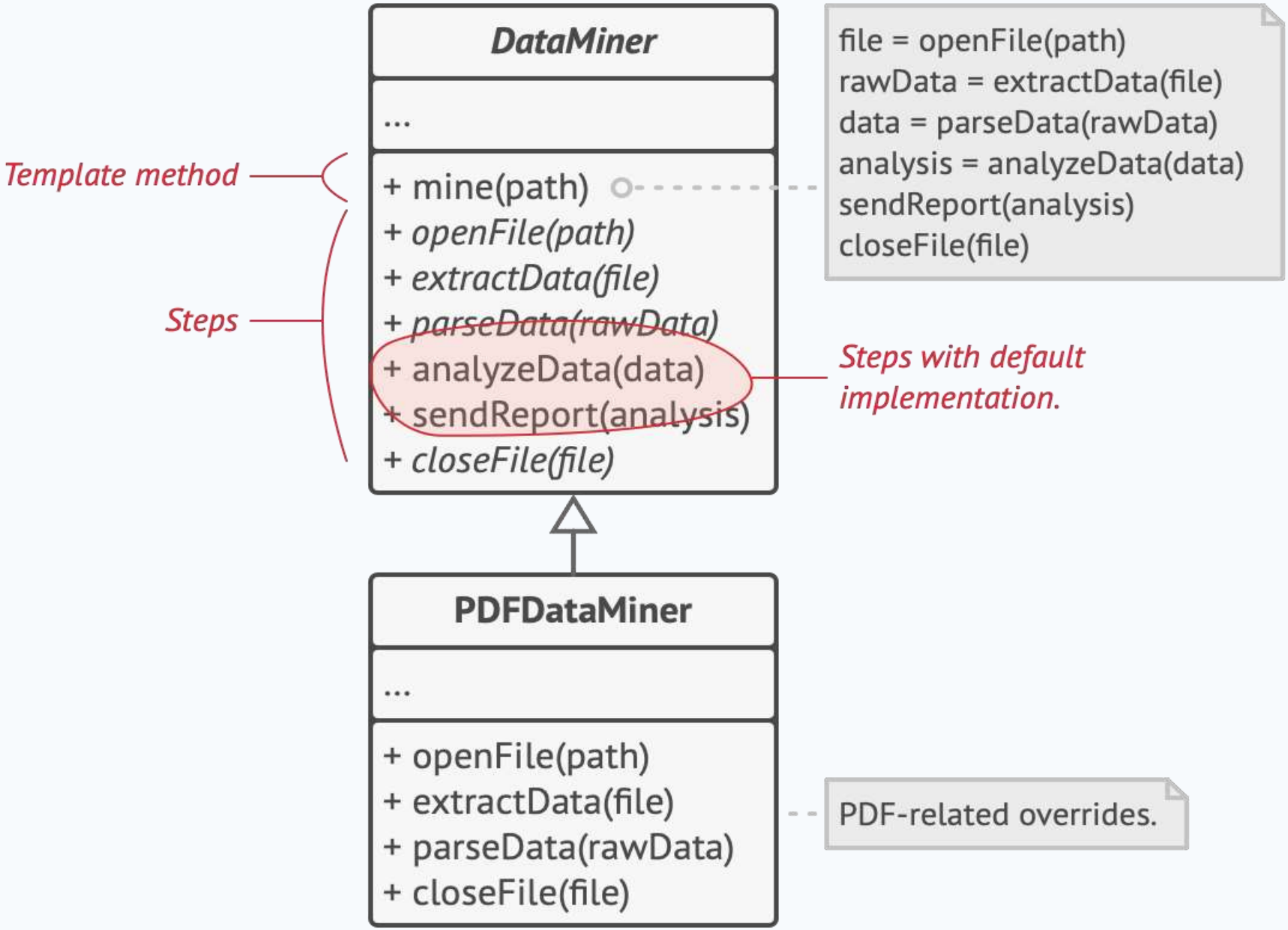
Obecná struktura



Inverzní princip
Hollywood principle
don't call us, we'll call you
rodičovská třída volá metody potomků

- AbstractClass
  - definuje abstraktní primitivní a hook operace
  - implementuje kostru algoritmu
- ConcreteClass
  - implementuje primitivní operace - konkrétní části algoritmu
  - může implementovat hook operace

Konkrétní struktura



Implementace

- operace v Template method
  - konkrétní operace abstraktní třídy
    - užitečné pro všechny podtřídy
  - primitivní operace
    - abstraktní, **musí** být implementovány v potomkovi
  - hook operace
    - defaultní implementace, **může** být přepsána v potomkovi
    - jasně definované body rozšíření
- přístupová práva a modifikátory
  - template method - public, **nevirtuální**
  - primitivní operace - protected, čistě virtuální
  - hook operace - protected, virtuální s implementací
    - typicky prázdná implementace
- minimalizace počtu primitivních operací
  - všechny je nutné definovat
  - větší množství komplikuje implementaci

```
class Base {
protected:
  virtual void preHook() {};
  virtual void postHook() {};
  virtual void doFirst() = 0;
  virtual void doSecond() = 0;
  void invariantFunc() {...};
public:
  void execute() {
    preHook();
    doFirst();
    invariantFunc();
    doSecond();
    postHook();
  }
};
```

```
class Derived1 : public Base {
  void doFirst() override {...};
  void doSecond() override {...};
  void postHook() override {...};
};
```

```
class Derived2 : public Base {
  void doFirst() override {...};
  void doSecond() override {...};
  void preHook() override {...};
};
```

Policy classes

- dědičnost - runtime flexibilita / režie
- šablony - compile-time

```
class Base {
protected:
  virtual void doFirst() = 0;
  virtual void doSecond() = 0;
public:
  void execute() {
    doFirst();
    doSecond();
  }
};
```

```
class Derived : public Base {
  void doFirst() override {};
  void doSecond() override {};
};
```

```
std::unique_ptr<Base> tmp =
std::make_unique<Derived>();
tmp->execute();
```

```
template<class Policy>
class Base {
public:
  void execute() {
    Policy policy;
    policy.doFirst();
    policy.doSecond();
  }
};
```

```
class Policy {
public:
  void doFirst() {};
  void doSecond() {};
};
```

```
Base<Policy> tmp;
tmp.execute();
```

Porovnání se Strategy

- Template Method
  - jednotlivé *'malé'* kroky společného algoritmu s pevnou kostrou
  - data předka
  - dědičnost (politiky)

```
class AbstractTemplate {
protected:
  virtual void primitiveOp() = 0;
public:
  void execute() {
    primitiveOp();
  }
};

class ConcreteAlgo : AbstractTemplate {
  void primitiveOp() {...};
};
```

```
std::unique_ptr<AbstractTemplate> tmp =
std::make_unique<ConcreteAlgo>();
tmp->execute();
```

- Strategy
  - jeden konkrétní *'velký'* algoritmus
  - vlastní data
  - kompozice (kontext) / delegace

```
class IStrategy;

class ClientCode {
private:
  IStrategy _algo;
public:
  ClientCode(IAlgorithm algo) :
    _algo(algo) {
  }
  void execute() {
    _algo.primitiveOp();
  }
};

class ConcreteAlgo : IStrategy {
public:
  void primitiveOp() {...};
};
```

```
ConcreteAlgo algo;
ClentCode code(algo);
code.execute();
```

Použití

- modifikovatelné postupy (*'algoritmy'*)
- frameworky (GUI, Web...)
- hry (common behavior)

- JUnit testy
- při každém automatickém testu je třeba spustit několik akcí:
  - runBare() – Template method
  - setUp() – připraví zdroje
  - runTest() – vykoná test
  - tearDown() – uvolní zdroje


```
public abstract class TestCase {

  public void runBare() {
    setUp();
    try {
      runTest();
    } finally {
      tearDown();
    }
  }

  protected void setUp() {};
  protected void tearDown() {};
  protected void runTest() {};
};
```

Shrnutí

- ☺ výhody
  - eliminace copy-and-paste
  - vynucení pevné struktury algoritmu
  - definice vhodných míst pro rozšíření
- 😐 trade-off
  - počet primitivních operací
- ☹ nevýhody
  - rozdrobenost
    - fragmenty kódu jsou vytržené z kontextu
  - složitější přidávání nových funkcí
    - změna algoritmu vyžaduje změnu předka
    - zásadnější změny - změna všech potomků
- 😱 problém
  - definice kostry obecného algoritmu
  - pevná struktura, měnitelné detialy
  - 🚫 copy-and-paste
- 😍 řešení
  - základní třída implementuje kostru algoritmu
  - podtřída implementuje jednotlivé kroky
- 👫 související NV
  - Strategy
    - změna celého algoritmu delegací
    - Template method může definovat kroky
  - Factory method
    - může sloužit i jako primitivní operace
    - např. DoCreateFile()

Děkuji za pozornost!

Aurel Farkašovský