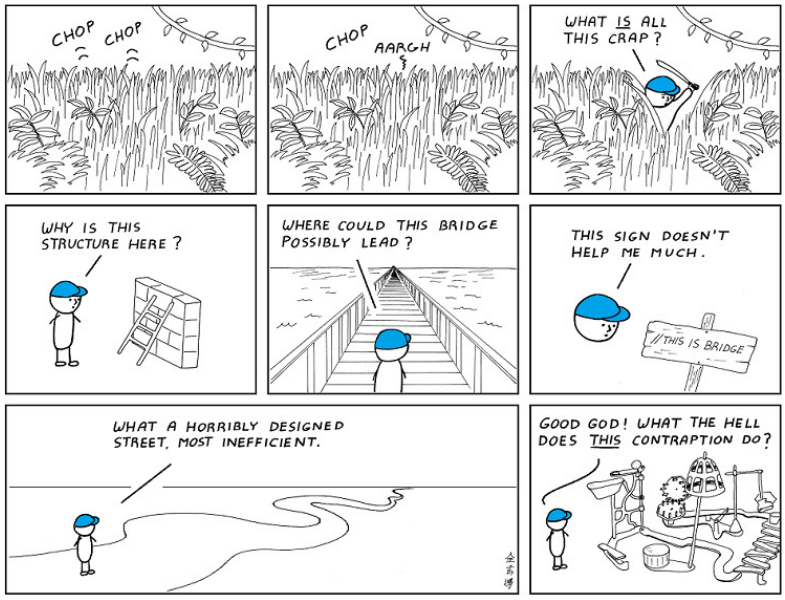


I hate reading
other people's code.

OBJECT-ORIENTED PROGRAMMING

Creational Design Patterns

Lecture #10

TECHNICAL UNIVERSITY OF KOŠICE

Vytváracie návrhové vzory

- Zaoberajú sa inicializovaním a konfiguráciou tried a objektov
- Dnes si ukážeme
  - Factory Method
  - Abstract Factory
  - Singleton
  - Builder
  - Prototype

Vytváracie návrhové vzory

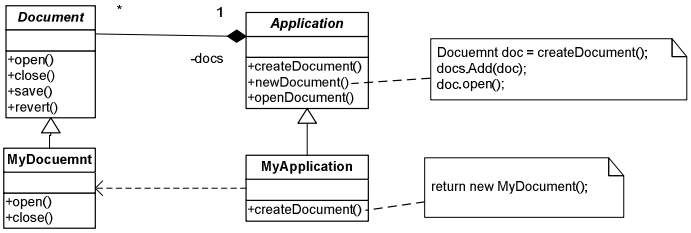
- **Vytváracie vzory** predstavujú abstrakciu procesu vytvárania inštancií objektov. Skrývajú proces vytvárania objektov a pomáhajú budovať systémy nezávislé od spôsobu vytvárania a skladania objektov
- **Vytváracie vzory tried** sa zameriavajú na použitie dedenia pri rozhodovaní o vytváraní objektov
  - **Factory Method, Prototype**
- **Vytváracie vzory objektov** sa zameriavajú na delegovanie procesu vytvárania objektov na iné objekty
  - **Abstract Factory, Builder**

Vytváracie návrhové vzory

- Všetky OO jazyky majú jazykovú konštrukciu pre vytváranie objektov, napr. **`new`**
- Vytváracie vzory umožňujú napísať metódy pre vytváranie objektov bez priameho použitia **`new`**
- To umožňuje písať metódy, ktoré vytvárajú rôzne objekty a môžu byť rozširované pre vytváranie ďalších nových objektov bez nutnosti modifikácie existujúceho kódu

Factory Method

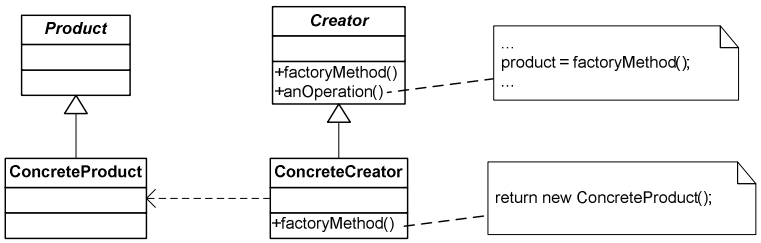
- Zámer použitia
  - Definuje rozhranie pre vytváranie objektov, ale umožňuje podtriedam rozhodovať o tom, objekty ktorých tried budú vytvárané
- Motivácia



- Metóda `createDocument()` predstavuje vzor **Factory Method**

Factory Method

- Použitie
  - V prípade, že trieda nemôže predpokladať akej triedy budú vytvárané objekty
  - Trieda očakáva, že jej podtriedy špecifikujú vytvárané objekty
- Štruktúra



Factory Method

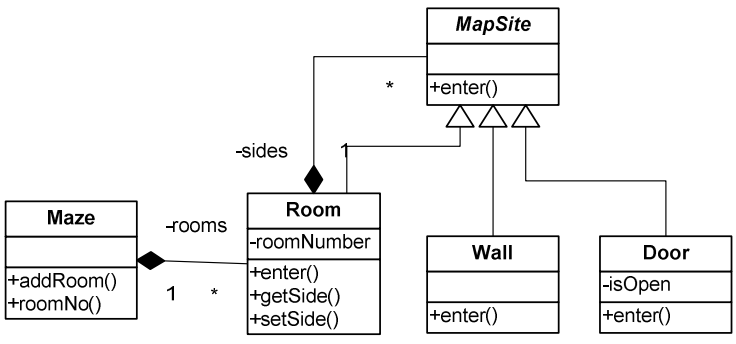
- Zúčastnené prvky
  - `Product` – definuje rozhranie pre typ objektov, ktoré sa budú vytvárať
  - `ConcreteProduct` – implementuje `Product` rozhranie
  - `Creator` – deklaruje vytváraciu metódu, ktorá vracia objekt typu `Product`
  - `ConcreteCreator` – prepisuje vytváraciu metódu a vracia inštanciu `ConcretePorduct`
- Spolupráca prvkov
  - `Creator` sa spolieha na svoje podtriedy, ktoré implementujú vytváraciu metódu, že vrátia príslušnú inštanciu `ConreteProduct`

Factory Method

- Trieda `Creator` je vytváraná bez znalosti aká aktuálna trieda `ConcreteProduct` bude inštanciovaná. To ktorá trieda `ConcreteProduct` bude inštanciovaná je jednoznačne dané príslušnou `ConcreteCreator` podtriedou použitou v aplikácii
- Neznamená to však, že podtrieda sa rozhodne počas vykonávania, ktorý konkrétny typ sa použije!

Factory Method (príklad)

- Majme hru s bludiskom (Maze Game)



Factory Method (príklad)

- Trieda `MazeGame` s metódou `createMaze()`

```
public class MazeGame {
  public Maze createMaze() {
    Maze m = new Maze();
    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door d = new Door(r1, r2);
    m.addRoom(r1);
    m.addRoom(r2);
    r1.setSide(MazeGame.North, new Wall());
    r1.setSide(MazeGame.East, d);
    r1.setSide(MazeGame.South, new Wall());
    r1.setSide(MazeGame.West, new Wall());
    r2.setSide(MazeGame.North, new Wall());
    r2.setSide(MazeGame.East, new Wall());
    r2.setSide(MazeGame.South, new Wall());
    r2.setSide(MazeGame.West, d);
    return m;
  }
}
```


Factory Method (príklad)

- Takáto funkcia `createMaze()` nie je flexibilná
- Čo ak by sme chceli mať iné typy miestností, dverí a stien, napr. `MagicRoom`, `MagicDoor`, `DoorWithLock`, `WallWithHiddenDoor`?
- Takto by sme museli urobiť výrazné zmeny priamo v metóde `createMaze()`, pretože explicitne používame operátor **`new`**
- Riešením je použitie vzoru Factory Method

Factory Method (príklad)

- Pridajme vytváracie metódy do triedy `MazeGame` a použijeme ich v `createMaze()` namiesto operácie **`new`**

```
public class MazeGame {

  public Maze makeMaze() {
    return new Maze();
  }

  public Room makeRoom(int n) {
    return new Room(n);
  }

  public Wall makeWall() {
    return new Wall();
  }

  public Door makeDoor(Room r1, Room r2) {
    return new Door(r1, r2);
  }
```


Factory Method (príklad)

```
  public Maze createMaze() {
    Maze m = makeMaze();
    Room r1 = makeRoom(1);
    Room r2 = makeRoom(2);
    Door d = makeDoor(r1, r2);
    m.addRoom(r1);
    m.addRoom(r2);
    r1.setSide(MazeGame.North, makeWall());
    r1.setSide(MazeGame.East, d);
    r1.setSide(MazeGame.South, makeWall());
    r1.setSide(MazeGame.West, makeWall());
    r2.setSide(MazeGame.North, makeWall());
    r2.setSide(MazeGame.East, makeWall());
    r2.setSide(MazeGame.South, makeWall());
    r2.setSide(MazeGame.West, d);
    return m;
  }
}
```

Factory Method (príklad)

- Teraz je metóda `createMaze()` trošku zložitejšia, ale určite flexibilnejšia
- Príklad:

```
public class MagicMazeGame extends MazeGame {
  @Override
  public Maze makeMaze() {
    return new MagicMaze();
  }
  @Override
  public Room makeRoom(int n) {
    return new MagicRoom(n);
  }
  @Override
  public Wall makeWall() {
    return new MagicWall();
  }
  @Override
  public Door makeDoor(Room r1, Room r2) {
    return new MagicDoor(r1, r2);
  }
}
```

- Metóda `createMaze()` z triedy `MazeGame` je zdedená v triede `MagicMazeGame` a môže byť použitá pre vytváranie správnych bludísk bez akejkoľvek modifikácie

Factory Method (príklad)

- Korelácia príkladu so vzorom

`Creator` = `MazeGame`
`ConcreteCreator` = `MagicMazeGame`
(tiež aj `MazeGame`)
`Product` = `MapSite`
`ConcreteProduct` = `Room`, `Wall`, `Door`, `MagicRoom`, `MagicWall`, `MagicDoor`

Factory Method

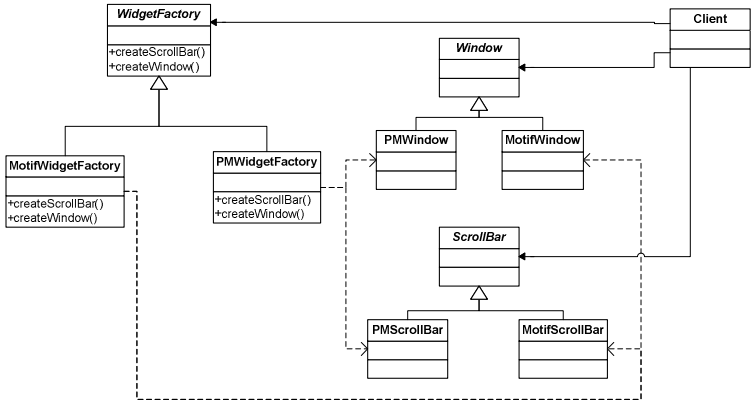
- Výhody
  - Kód je omnoho flexibilnejší a znovu použiteľný vďaka odstráneniu inštanciovania aplikačne špecifických objektov
  - Kód pracuje iba s rozhraním `Product` a tým môže používať akékoľvek triedy `ConcreteProduct`, ktoré implementujú toto rozhranie
- Implementácia
  - `Creator` môže byť abstraktná aj konkrétna trieda
  - Ak má vytváracia metóda vytvárať viac typov objektov, môže mať napr. vstupný parameter a rozhodovať pomocou `if-else` konštrukcie

Abstract Factory

- Zámer použitia
  - Poskytuje rozhranie pre vytváranie skupín príbuzných alebo závislých objektov bez špecifikácie ich konkrétnych tried
  - Vytváranie objektov je delegované na iné objekty pomocou kompozície a používa vzor Factory Method pre vytváranie konkrétnych typov objektov

Abstract Factory

- Motivácia

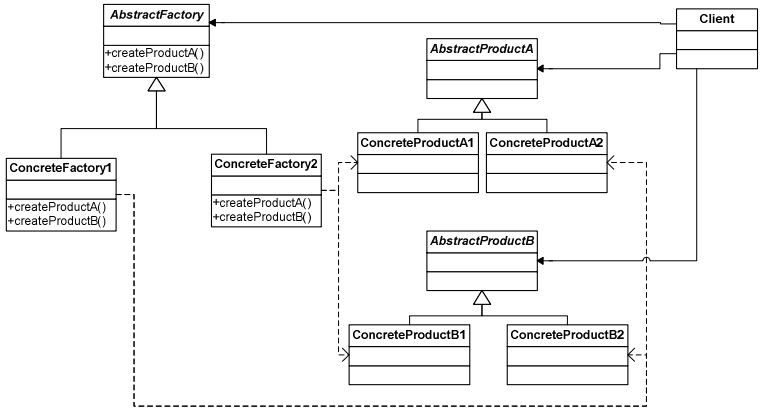


Abstract Factory

- Použitie
  - Systém má byť nezávislý na tom ako sú vytvárané jeho prvky
  - Trieda nemôže predpokladať akej triedy budú vytvárané objekty
  - Systém musí používať iba jednu skupinu príbuzných prvkov
  - Prvky z jednej skupiny príbuzných prvkov musia byť používané súčasne

Abstract Factory

- Štruktúra



Abstract Factory

- Zúčastnené prvky
  - `AbstractFactory` – deklaruje rozhranie pre operácie, ktoré vytvárajú objekty abstraktných produktov
  - `ConcreteFactory` – implementujú operácie pre vytváranie konkrétnych objektov
  - `AbstractProduct` – deklaruje rozhranie pre typ objektov produktov
  - `ConreteProduct` – definuje objekt produktu, ktorý má byť vytváraný konkrétnou vytváracou triedou, implementuje rozhranie `AbstractProduct`
  - `Client` – používa iba rozhrania deklarované ako abstraktné (`AbstractFactory`, `AbstractProduct`)

Abstract Factory

- Spolupráca prvkov
  - Je vytvorená iba jediná inštancia `ConcreteFactory` (viď vzor Singleton), táto vytvára objekty produktov podľa konkrétnej implementácie. Pre vytváranie iných typov produktov je potrebná inštancia inej `ConcreteFactory`
  - Trieda `AbstractFactory` necháva implementáciu vytvárania objektov na svojich podtriedach

Abstract Factory (príklad)

- Implementujme vzor Abstract Factory do našej hry

```
public class MazeFactory {
  public Maze makeMaze() {
    return new Maze();
  }
  public Room makeRoom(int n) {
    return new Room(n);
  }
  public Wall makeWall() {
    return new Wall();
  }
  public Door makeDoor(Room r1, Room r2) {
    return new Door(r1, r2);
  }
}
```

- Trieda `MazeFactory` je len množina vytváracích metód!
- Trieda `MazeFactory` predstavuje `AbstractFactory` a zároveň aj `ConcreteFactory`

Abstract Factory (príklad)

- Použijeme objekt triedy `MazeFactory` ako parameter pre metódu `createMaze()` v triede `MazeGame`

```
public class MazeGame {
  public Maze createMaze(MazeFactory factory) {
    Maze m = factory.makeMaze();
    Room r1 = factory.makeRoom(1);
    Room r2 = factory.makeRoom(2);
    Door d = factory.makeDoor(r1, r2);
    m.addRoom(r1);
    m.addRoom(r2);
    r1.setSide(MazeGame.North, factory.makeWall());
    r1.setSide(MazeGame.East, d);
    r1.setSide(MazeGame.South, factory.makeWall());
    r1.setSide(MazeGame.West, factory.makeWall());
    r2.setSide(MazeGame.North, factory.makeWall());
    r2.setSide(MazeGame.East, factory.makeWall());
    r2.setSide(MazeGame.South, factory.makeWall());
    r2.setSide(MazeGame.West, d);
    return m;
  }
}
```


Abstract Factory (príklad)

- Teraz môžeme jednoducho rozšíriť ďalšie vytváracie triedy pre špecifické bludiská

```
public class MagicMazeFactory extends MazeFactory {
  @Override
  public Maze makeMaze() {
    return new MagicMaze();
  }
  @Override
  public Room makeRoom(int n) {
    return new MagicRoom(n);
  }
  @Override
  public Wall makeWall() {
    return new MagicWall();
  }
  @Override
  public Door makeDoor(Room r1, Room r2) {
    return new MagicDoor(r1, r2);
  }
}
```

Abstract Factory (príklad)

- Korelácia príkladu so vzorom

`AbstractFactory` = `MazeFactory`
`ConcreteFactory` = `MagicMazeFactory`
(tiež aj `MazeFactory`)
`AbstractProduct` = `MapSite`
`ConcreteProduct` = `Room`, `Wall`, `Door`, `MagicRoom`, `MagicWall`, `MagicDoor`
`Client` = `MazeGame`

Abstract Factory

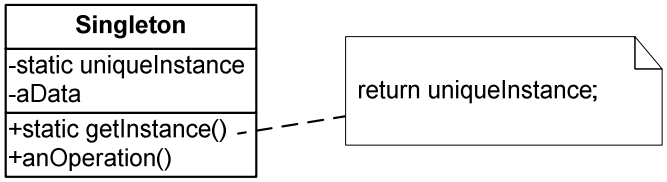
- Výhody
  - Izoluje klienta od konkrétnej implementácie tried
  - Umožňuje jednoduchú výmenu skupiny produktov, keďže konkrétna vytváracia trieda implementuje vytváranie celej skupiny produktov
  - Vynucuje používanie len jednej skupiny objektov produktov
- Implementácia
  - Typická implementácia vyžaduje iba jedinú inštanciu vytváracej triedy
  - V tomto prípade sa používa vzor Singleton
  - V prípade, že konkrétna vytváracia trieda musí vytvárať nové druhy produktov (nezahrnuté v abstraktnej deklarácii), často sa implementuje iba jediná vytváracia metóda s parametrom druh produktu

Singleton

- Zámer použitia
  - Umožňuje vytvorenie iba jedinej inštancie z danej triedy a poskytuje takto globálny prístup k tomuto objektu
- Motivácia
  - Niekedy chceme iba jedinú inštanciu danej triedy
  - Napr. chcem iba jeden objekt spravujúci objekty okien v systéme, alebo iba jeden vytvárací objekt
  - Chceme aby sme sa k tomuto objektu ľahko dostali
  - Chceme mať zaručené, že nemôžeme vytvoriť viac inštancií danej triedy

Singleton

- Štruktúra



- Výhody
  - Kontrolovateľný prístup k jednoznačnému objektu
  - Nedovoľuje vytvorenie rôznych ďalších inštancií

Implementácia vzoru Singleton

- Použijeme statickú metódu pre získanie referencie na jedinečný objekt a budeme mať privátny konštruktor

```
public class Singleton {
  private static Singleton uniqueInstance = null;
  private int data;

  public static Singleton getInstance() {
    if (uniqueInstance == null)
      uniqueInstance = new Singleton();
    return uniqueInstance;
  }

  private Singleton() { this.data = 0; }

  public int getData() { return this.data; }
  public void setData(int data) { this.data = data; }
}
```


Implementácia vzoru Singleton

- Testovací program

```
public class TestSingleton {
  public static void main(String[] args) {
    Singleton s1 = Singleton.getInstance();
    s1.setData(100);
    System.out.println("Prvá referencia: " + s1.getData());
    Singleton s2 = Singleton.getInstance();
    System.out.println("Druhá referencia: " + s2.getData());
  }
}
```

Implementácia vzoru Singleton

- Táto implementácia vytvára objekt, len v prípade ak je to nutné – pri prvom volaní `getInstance()` (**late initialization**)
- Čo ak dve paralelné vlákna zavolajú metódu `getInstance()`?
  - Môžu vzniknúť dve inštancie!
- Riešenie
  - Definícia kritickej oblasti v časti vytvárania objektu
  - Vytvorenie inštancie statickým inicializovaním v triede (**early initialization**)

```
public class Singleton {
  private static final Singleton uniqueInstance = new Singleton();
  private int data;
  public static Singleton getInstance() { return uniqueInstance; }
  private Singleton() { this.data = 0; }
  public int getData() { return this.data; }
  public void setData(int data) { this.data = data; }
}
```


Builder

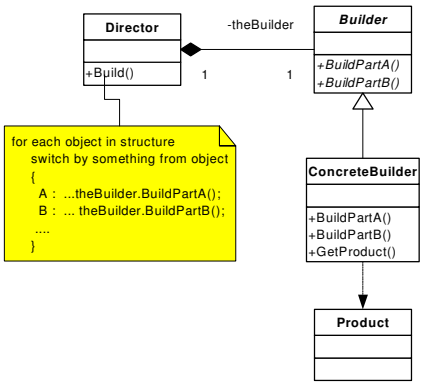
- Zámer použitia
  - Oddeľuje konštrukciu (postup vytvárania) zložitého objektu od jeho reprezentácie (jeho štruktúry)
- Motivácia
  - Prípady, kde potrebujeme vyskladať objekt zložený z iných objektov
  - Chceme pred klientom skryť samotnú implementáciu vyskladania objektu
  - Chceme klientovi poskytnúť variabilitu skladania štruktúrovaných objektov

Builder

- Použitie
  - Oddeliť problém výstavby objektu (z čoho sa má skladať) od konštrukcie (ako sa má skladať) a reprezentácie skladaného objektu (čo sa má skladať)
  - Klient má prístup ku konštrukcii objektu na základe rozhrania, ktoré môže byť implementované rôznymi objektmi

Builder

- Štruktúra

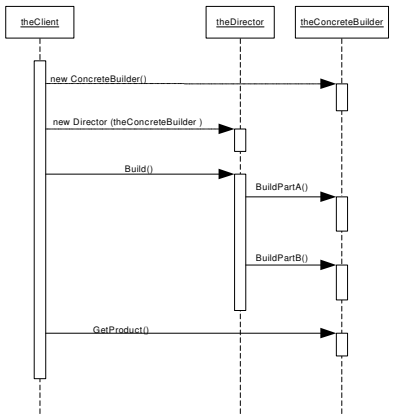


Builder

- Zúčastnené prvky
  - `Builder` – abstraktné rozhranie pre tvorbu častí výsledného objektu
  - `ConcreteBuilder` – implementuje `Builder` a konštruuje konkrétny objekt typu `Product` (vykonáva celú konštrukciu)
  - `Director` – konštruuje štruktúrovaný objekt použitím objektu s rozhraním `Builder` (t.j. `ConcreteBuilder`), pričom nepozná konštruovaný objekt (riadi celú konštrukciu)
  - `Product` – reprezentuje konštruovaný zložený objekt

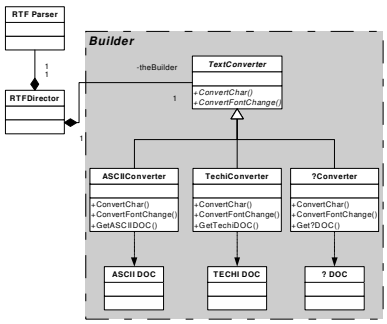
Builder

- Spolupráca prvkov



Builder (príklad)

- Konverzia dokumentu z typu RTF do iných typov (pre reprezentáciu na zobrazovacom zariadení)
- Dokument sa skladá z množstva objektov, pre klienta nie je podstatné ako vyzerá výsledná štruktúra ani ako sa skladá



Prototype

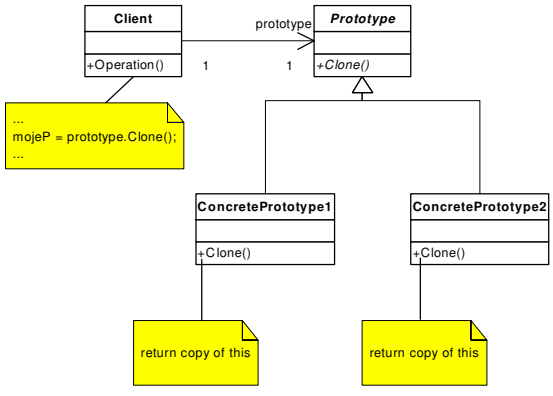
- Zámer použitia
  - Zavedenie skupiny objektov vznikajúcich klonovaním prototypových inštancií pomocou jednotného rozhrania
- Motivácia
  - Pri vytváraní objektov máme často problém výberu triedy výsledného produktu
  - Riešenie pomocou Factory vzorov vyžaduje často prepis celej vytváracej metódy (hoci po zdedení)
  - Vytváranie nových objektov ako kópií iných objektov, bez toho aby sme vedeli o aký skutočný typ ide

Prototype

- Použitie
  - Štruktúra produktov má spoločné rozhranie, ktoré podporuje klonovanie objektov
  - Klient obsahuje zoznam prototypových objektov, z ktorých sú klonované nové objekty
  - Klient pristupuje pri vytváraní objektu cez toto spoločné rozhranie, pričom nezáleží o aký konkrétny typ vytváraného objektu ide

Prototype

- Štruktúra



Prototype

- Zúčastnené prvky
  - `Prototype` – abstraktné rozhranie pre vytváraný objekt, implementuje abstraktnú metódu klonovania objektu
  - `ConcretePrototype` – implementuje `Prototype` a vytvára kópiu inštancie tejto konkrétnej triedy
  - `Client` – používa `Prototype` na vytváranie objektov pomocou metódy `Clone()`, nemusí poznať aké `ConcretePrototype` je použité (nazývaný tiež Prototype Manager a často implementovaný pomocou Factory vzorov)

Prototype (príklad)

- Factory pre vytváranie záznamov o autách a osobách s použitím klonovania prototypov

```
public enum RecordType { Car, Person }        // Typ záznamu

public abstract class Record {                // Prototyp záznamu
  public abstract Record duplicate();
}

public class PersonRecord extends Record {    // Záznam osoby
  String name; int age;                       // Údaje o osobe
  public Record duplicate() {                 // Klonovanie osoby
    return (Record) clone();
  }
}

public class CarRecord extends Record {       // Záznam o aute
  String name; CarID id;                      // Údaje o aute
  public Record duplicate() {                 // Klonovanie auta, nastav nové id
    CarRecord newCar = (CarRecord) clone();
    newCar.setID(CarID.newID());
    return newCar;
  }
}
```

Využijeme klonovanie implementované v triede `Object`

Využijeme klonovanie implementované v triede `Object`

Prototype (príklad)

```
// Factory pre vytváranie záznamov
public class RecordFactory {

  // Zoznam prototypov
  private static Map<RecordType, Record> prototypes =
                                          new HashMap<RecordType, Record>();

  // Naplnenie zoznamu prototypov
  public RecordFactory() {
    prototypes.add(RecordType.Car, new CarRecord());
    prototypes.add(RecordType.Person, new PersonRecord());
  }

  // Vytváranie objektov ako klony prototypov
  public Record createRecord(RecordType type) {
    return prototypes.get(type).duplicate();
  }
}
```


Readings

- GAMMA, E. – HELEM, R. – JOHNSON, R. – VLISSIDES, J. M.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994
  - 3 Creational Patterns