

A. Předmět Počítačové a informační systémy

File (18)

[Analýza informačního systému](#)

[Databázové systémy](#)

[Datové modely](#)

[Dotazovací jazyky](#)

[Návrh a implementace datové vrstvy](#)

[Transakce](#)

[Vykonávání dotazů v databázových systémech](#)

[Datové typy](#)

[Paradigmata programování](#)

[Programovací techniky](#)

[Programování](#)

[Reprezentace čísel v počítači](#)

[Vývoj aplikací](#)

[Informační systém](#)

[Návrhové vzory a praktiky pro vývoj informačních systémů](#)

[Softwarové inženýrství](#)

[UML](#)

[Životní cyklus vývoje software](#)

Programování

Obsah

- [Paradigmata programování](#)
- [Datové typy](#)
- [Programovací techniky](#)
- [Vývoj aplikací](#)
- [Reprezentace čísel v počítači](#)

Programování (FPR, UPR, OOP, SJ, C#/JAVA I a II, UTI, ZDS)

- Paradigmata programování (deklarativní a imperativní, funkcionální, strukturované a objektově orientované paradigma, specifické vlastnosti a principy jednotlivých paradigmát).
- Datové typy (statická a dynamická alokace, dynamické typování, životní cyklus paměti, jednoduché, strukturované, abstraktní a generické typy, kolekce, soubory, streamy).
- Programovací techniky (řídící struktury, funkce, jejich parametry a návratové hodnoty, rekurze, polymorfismus, kolekce, výjimky a jejich strukturovaná obsluha, paralelismus a vlákna).
- Vývoj aplikací (C#/Java, typový systém, virtuální stroj a kompilace, tvorba uživatelského rozhraní, delegáty a události, přístup k databázím, práce s textovými daty, asynchronní programování, serializace, síťová komunikace).
- Reprezentace čísel v počítači (číselné soustavy a převody, celá čísla, čísla s pevnou řádovou čárkou, čísla s pohyblivou řádovou čárkou vyjádřená v binárním, decimálním a hexadecimálním základu, aritmetika s čísly v různých reprezentacích, kódování znaků).

Příklad otázky

Jako programátor máte vyřešit úlohu předávání dat mezi dvěma aplikacemi. Jaké formy předávání dat/komunikace mezi aplikacemi znáte, na základě čeho byste vybrala/vybral nejvhodnější formu a jak byste ji realizovala/realizoval ve vámi zvoleném programovacím jazyce?

Status

File (5)

Done

[Datové typy](#)

true

[Paradigmata programování](#)

true

[Programovací techniky](#)

true

[Reprezentace čísel v počítači](#)

true

[Vývoj aplikací](#)

true


Paradigmata programování

- Paradigmata programování (deklarativní a imperativní, funkcionální, strukturované a objektově orientované paradigma, specifické vlastnosti a principy jednotlivých paradigmát).

Paradigmata programování

jsou vzorce, nebo modely myšlení při programování. Podle programovacích paradigmát třídíme programovací jazyky

Deklarativní paradigma

 [Wiki](#)

***Deklarativní paradigma** je založeno na myšlence programování aplikací pomocí definic **co se má udělat**, a ne **jak se to má udělat**. Opakem tohoto principu je imperativní programování popisující jednotlivé úkony pomocí algoritmů. Zjednodušeně to lze popsat tak, že imperativní programy obsahují algoritmy, kterými se dosáhne chtěný cíl, zatímco deklarativní jazyky specifikují cíl a algoritmizace je ponechána programu (interpretu) daného jazyka.*

Důraz na **co se má udělat**, například:

```
SELECT *  
FROM table  
WHERE name = "Karel"
```

U jazyka **SQL** není důležité jak se věci stanou, který algoritmus se použije na vyhledání, ale popisuje pouze co se má stát. SQL je doménově specifický jazyk, neboli jazyk pro řešení konkrétního problému. Tyto jazyky jsou obvykle turingovsky neúplné (není univerzální, nedokáže vyřešit stejné problémy jako turingův stroj).

- **Specifické vlastnosti:** Programátor popisuje požadovaný výsledek, aniž by podrobně specifikoval, jak má být dosažen.
- **Principy:** Důraz je kladen na popis problému a jeho vztahů, místo na specifikaci postupů. Typickými příklady jsou SQL dotazy nebo deklarativní programování v rámci některých funkcionálních jazyků.

Imperativní paradigma

Pomocí imperativního programovacího paradigmátu vysvětlujeme počítači, **jak** má věci udělat, tedy postup krok po kroku, jak dosáhnout tíženého výsledku. Základními kameny jsou:

- přiřazení hodnot proměnným (a = 20)

- podmínky (if-else)
- cykly (for, while)

```
#Příklad
items = [1, 2, 3, 4];
result = 0
for i in items:
    result += i
print(result)
```

Imperativní paradigma je velmi rozšířené a je používáno v mnoha programovacích jazycích, včetně jazyků jako je **C**, **Java**, **Python** a mnoho dalších. Jeho hlavní výhodou je přímá kontrola nad tím, jak jsou operace vykonávány, což může být výhodné pro některé typy aplikací, jako jsou systémové programy nebo aplikace s vysokým výkonem.

- **Specifické vlastnosti:** V tomto paradigmatu programátor přímo specifikuje kroky, které počítač musí provést k dosažení požadovaného výsledku.
- **Principy:** Důraz je kladen na manipulaci s proměnnými, přiřazování hodnot, řízení toku programu pomocí podmínek a smyček.

Strukturované paradigma

Strukturované programování je paradigmatický přístup k psaní programů, který zdůrazňuje používání strukturovaných metod pro návrh a organizaci kódu. Tento přístup se snaží minimalizovat používání nekontrolovaných skoků (např. goto příkazů) a upřednostňuje použití strukturovaných kontrolních struktur, jako jsou podmínky, smyčky a podprogramy.

Zde je několik klíčových prvků strukturovaného programování:

1. **Seřaditelnost (Sequencing):** Programy jsou psány jako sekvence instrukcí, které jsou vykonávány jeden po druhém, od začátku do konce.
2. **Podmíněné zpracování (Conditional Execution):** Použití podmínek umožňuje programu rozhodnout, které části kódu vykonat na základě aktuálních podmínek.
3. **Cykly (Loops):** Cykly umožňují opakování určitých částí kódu, dokud je splněna určitá podmínka.
4. **Strukturované podprogramy (Structured Subroutines):** Podprogramy nebo funkce jsou používány k rozdělení kódu na menší, snadno spravovatelné části. Tyto podprogramy jsou volány z hlavního programu, což zlepšuje modularitu a znovupoužitelnost kódu.
5. **Návraty (Returns):** Strukturované programování preferuje jednoznačné ukončení podprogramů pomocí návratových instrukcí, což zvyšuje jasnost a přehlednost kódu.
6. **Rekurze (Recursion):** Rekurze, tj. schopnost funkce volat sama sebe, je také součástí strukturovaného programování a může být použita pro řešení určitých problémů.

Tento přístup k programování má za cíl zlepšit čitelnost, údržbu a laditelnost kódu tím, že zavádí jasné definované struktury a omezuje nekontrolované skoky, což usnadňuje pochopení a správu programu.

Příklady mohou být **Python, C, C++, ADA, Fortran**.

- **Specifické vlastnosti:** Důraz je kladen na strukturované programování, které zahrnuje používání podmínek, smyček a podprogramů pro zlepšení čitelnosti a údržby kódu.
- **Principy:** Hlavním cílem je zlepšit přehlednost a snadnou údržbu kódu pomocí strukturovaných konstrukcí.

```
// Příklad: Součet prvků v poli pomocí cyklu for
#include <stdio.h>

int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int sum = 0;
    for (int i = 0; i < 5; i++) {
        sum += numbers[i];
    }
    printf("Součet prvků v poli je: %d\n", sum);
    return 0;
}
```

Objektově orientované paradigma

Objektově orientované programování (OOP) je paradigmatický přístup k psaní programů, který se zaměřuje na modelování reálného světa pomocí objektů a interakcí mezi nimi. Toto paradigma organizuje kód do objektů, které kombinují data a metody (funkce), které s těmito daty pracují. Zde jsou klíčové prvky objektově orientovaného paradigmatu:

1. **Objekty:** Objekty jsou základní stavební jednotkou objektově orientovaného programování. Každý objekt má svůj stav (data) a chování (metody).
2. **Třídy:** Třídy jsou šablony pro vytváření objektů. Definují atributy (data) a metody (funkce), které budou mít objekty dané třídy.
3. **Dědičnost (Inheritance):** Dědičnost umožňuje vytvářet nové třídy (potomky) na základě již existujících tříd (rodičů), přičemž potomci zdědí vlastnosti a chování svých rodičů.
4. **Polymorfismus:** Polymorfismus umožňuje jedné metodě nebo funkci pracovat s různými typy objektů. To znamená, že stejný kód může být použit s různými typy objektů, což zvyšuje flexibilitu a znovupoužitelnost kódu.
5. **Zapouzdření (Encapsulation):** Zapouzdření umožňuje skrýt vnitřní implementaci objektů a umožňuje přístup k nim pouze prostřednictvím definovaných rozhraní. To pomáhá chránit data objektů a zajišťuje konzistenci a bezpečnost programu.

Objektově orientované programování usiluje o vytváření modulárního, snadno udržitelného a znovupoužitelného kódu tím, že organizuje kód do samostatných a dobře definovaných objektů. Tento přístup se často používá pro větší a komplexnější projekty, kde je důležitá rozsáhlost a organizace

kódu. Některé jazyky, jako je Java, C++, Python a C#, mají silnou podporu pro objektově orientované programování.

- **Specifické vlastnosti:** Modeluje programování kolem objektů, které kombinují data a metody pro manipulaci s těmito daty.
- **Principy:** Důraz je kladen na zapouzdření, dědičnost a polymorfismus. Objekty jsou základními stavebními jednotkami programu, které komunikují pomocí zpráv.

```
// Příklad: Třída reprezentující automobil s vlastnostmi a metodami
public class Car {
    private String brand;
    private String model;
    private int year;

    public Car(String brand, String model, int year) {
        this.brand = brand;
        this.model = model;
        this.year = year;
    }

    public void drive() {
        System.out.println("Jedu autem " + brand + " " + model + " " + year);
    }

    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Corolla", 2022);
        myCar.drive();
    }
}
```

Funkcionální paradigma

Funkcionální paradigma programování se zaměřuje na vytváření programů pomocí funkcí a vyhýbá se používání stavových proměnných. Zde jsou specifické vlastnosti a principy funkcionálního paradigmatu:

- **Specifické vlastnosti:**
 - Funkce jsou považovány za základní stavební kameny programu.
 - Stav programu je minimalizován a měněn pouze při volání funkcí a předávání argumentů.
 - Vedlejší efekty jsou minimalizovány nebo eliminovány.
 - Rekurze je běžně používána pro opakování operací.
- **Principy:**
 - **Čistota funkcí:** Funkce nemají vedlejší efekty, což znamená, že volání funkce s určitými vstupy vždy vrátí stejný výstup, a nemění stav programu.

- **Imutabilitu dat:** Data jsou obvykle neměnná (imutabilní), což znamená, že se nemohou po vytvoření změnit. Namísto toho se vytvářejí nová data na základě stávajících dat.
- **Vyhodnocování výrazů:** Výrazy jsou vyhodnocovány na základě jejich hodnot a ne na základě stavu programu.
- **Funkcionální kompozice:** Funkce jsou skládány do větších funkcí pomocí operací jako je kompozice a aplikace.
- **Lazy vyhodnocování:** V některých funkcionálních jazycích se výrazy nevyhodnocují okamžitě, ale až když jsou potřeba.

Funkcionální paradigma se často používá pro paralelní a distribuované programování, zpracování dat a matematické výpočty, ale lze ji využít pro různé typy aplikací. Jazyky, které podporují funkcionální programování, zahrnují Haskell, Lisp, Scala, F#, Clojure a mnoho dalších.

```
-- Příklad: Funkce na výpočet faktoriálu
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Datové typy

- Datové typy (statická a dynamická alokace, dynamické typování, životní cyklus paměti, jednoduché, strukturované, abstraktní a generické typy, kolekce, soubory, streamy).

Datové typy jsou základní stavební bloky programovacích jazyků, které určují, jak jsou data reprezentována a jak s nimi může program pracovat. Zde je stručný přehled různých typů:

Statická a dynamická alokace paměti

- **Statická alokace:** Paměť pro proměnné je přidělena během kompilace a zůstává konstantní během běhu programu. Statické proměnné mají definovanou velikost a rozsah platnosti.
- **Dynamická alokace:** Paměť je přidělena a uvolněna během běhu programu pomocí funkcí jako `malloc` a `free` v jazyce C, nebo pomocí konstrukcí jako `new` a `delete` v jazyce C++.

**Dynamické typování

- V programovacích jazycích s dynamickým typováním je typ proměnné určen až v době běhu programu, nikoli v době kompilace. Proměnné mohou měnit svůj datový typ během provádění programu.

Životní cyklus paměti:

- Životní cyklus paměti určuje, jak dlouho jsou data v paměti platná. Může se lišit v závislosti na tom, zda jsou data uložena na zásobníku (lokální proměnné funkcí) nebo na haldě (dynamicky

alokované proměnné).

Jednoduché, strukturované, abstraktní a generické typy

- **Jednoduché typy:** Také známé jako primitivní typy, jako například celá čísla, desetinná čísla, znaky atd.
- **Strukturované typy:** Tyto zahrnují pole, struktury a třídy, které umožňují seskupení různých datových typů do jedné entity.
- **Abstraktní typy:** Tyto typy poskytují abstrakci nad konkrétními datovými typy a implementují určitou funkcionalitu, aniž by poskytovaly podrobnosti o své vnitřní implementaci.
- **Generické typy:** Generické typy umožňují definovat typy nezávislé na konkrétním datovém typu. Jsou často používány v jazycích jako Java nebo C#.

Kolekce

- Kolekce jsou datové struktury, které umožňují ukládat a manipulovat soubory dat. Mohou zahrnovat seznamy, pole, mapy, fronty atd.
- obvykle je možné nějakým způsobem kolekcemi iterovat v javě třída `Iterable`

Soubory a streamy

- **Soubory:** Umožňují ukládání dat na trvalé úložiště jako pevný disk. Programy mohou číst data ze souborů a zapisovat do nich.
- **Streamy:** Streamy jsou obecně kanály pro přenos dat. Mohou zahrnovat nejen soubory, ale také síťové připojení, standardní vstup a výstup (`stdin`, `stdout`), paměťové proudy atd. Používají se pro asynchronní a sekvenční zpracování dat. Jsou jako proud, takže obvykle co je přečteno se rovnou zpracuje a nedá se k tomu lehce znova přistoupit.

Programovací techniky

- Programovací techniky (řídící struktury, funkce, jejich parametry a návratové hodnoty, rekurze, polymorfismus, kolekce, výjimky a jejich strukturovaná obsluha, paralelismus a vlákna).

Řídící struktury

Řídící struktury jsou základními stavebními kameny algoritmů v programování. Pomáhají programu provádět různé akce podle určitých podmínek. Existují tři hlavní typy řídících struktur:

- **Podmíněné struktury:** Nejčastější formou podmíněných struktur je `if-else`. Tato struktura umožňuje provádět kód v závislosti na pravdivosti určité podmínky. Alternativně můžete použít `switch-case`, který umožňuje vyhodnotit výraz a provést odpovídající kód na základě jeho hodnoty.

- **Cykly:** Cykly umožňují opakovat část kódu několikrát. Nejběžnější cykly jsou `for`, `while` a `do-while`. Cyklus `for` se používá, když je známo, kolikrát se má kód opakovat. `while` cyklus se používá, když není známo, kolikrát se má kód opakovat, ale podmínka musí být splněna. Cyklus `do-while` je podobný cyklu `while`, ale garantuje, že se kód vykoná alespoň jednou, než se podmínka ověří.
- **Skoky:** Skoky umožňují přeskočit určitou část kódu. Příklady skoků zahrnují `break`, který ukončuje cyklus, `continue`, který přeskočí zbytek aktuální iterace cyklu, a `return`, který okamžitě ukončuje funkci a vrací hodnotu.

Funkce

Funkce jsou bloky kódu, které provádějí určitou činnost a mohou být volány z jiných částí programu. Funkce mohou mít parametry, které jsou vstupními hodnotami, a mohou vracet hodnotu jako svůj výstup. Funkce mohou být buď vestavěné (např. funkce pro práci s řetězcí, matematické funkce atd.) nebo uživatelsky definované.

Rekurze

Rekurze je technika, kde funkce volá sama sebe. Tento přístup se často používá k řešení problémů, které mají podobnou strukturu, jako například problémy na základě rozdělení a panování. Každé rekurzivní volání snižuje velikost problému, dokud není dosažena podmínka ukončení, která zabrání nekonečné rekurzi.

Polymorfismus

Polymorfismus umožňuje používat objekty s rozdílnou implementací pomocí stejného rozhraní. Statický polymorfismus je dosažen přetížením funkcí nebo operátorů, zatímco dynamický polymorfismus je dosažen pomocí dědičnosti a virtuálních funkcí, což umožňuje objektům různých tříd používat stejné metody.

Kolekce

Kolekce jsou datové struktury, které umožňují ukládat a manipulovat s více prvky dat. Běžné typy kolekcí zahrnují pole, seznamy, spojové seznamy, fronty, zásobníky, mapy a mnoho dalších. Každý typ kolekce má své vlastnosti a vhodné použití podle konkrétních potřeb.

Výjimky a jejich strukturovaná obsluha

Výjimky jsou neočekávané situace, které mohou nastat během běhu programu. Jsou způsobeny chybami v kódu nebo vnějšími faktory, jako je chybný vstup od uživatele nebo nedostupnost sítě. Strukturovaná obsluha výjimek umožňuje programátorům zachytit a řídit tyto situace pomocí bloků `try-catch`, což umožňuje programu pokračovat ve správném chodu i při výskytu chyb.

Paralelismus a vlákna

Paralelismus je technika, která umožňuje programu provádět několik úloh současně. Vlákna jsou jednotky výpočtu, které mohou být prováděny nezávisle. Paralelní programování může zlepšit výkon a využití systémových prostředků, ale vyžaduje pečlivou správu sdílených zdrojů a synchronizaci mezi vlákny. Používání knihoven a frameworků pro paralelní programování může usnadnit tuto složitou práci a umožnit programátorům využívat výhody paralelismu efektivně.

Vývoj aplikací

- Vývoj aplikací (C#/Java, typový systém, virtuální stroj a kompilace, tvorba uživatelského rozhraní, delegáty a události, přístup k databázím, práce s textovými daty, asynchronní programování, serializace, síťová komunikace).

Java

Java je objektově orientovaný programovací jazyk, který je známý pro svou přenositelnost, bezpečnost a robustnost. Jeho syntaxe je podobná jazykům jako C++ a C#, ale Java je kompilována do bajtkódu, který je spouštěn na virtuálním stroji Java (JVM). To znamená, že Java aplikace jsou nezávislé na platformě a mohou být spuštěny na různých zařízeních, které podporují JVM.

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
}
```

Typový systém

Java používá statický typový systém, což znamená, že typy proměnných jsou ověřovány při kompilaci programu. Typový systém Java obsahuje základní datové typy jako `int`, `double`, `boolean`, atd., a umožňuje také vytvářet uživatelsky definované třídy a rozhraní.

```
int number = 10;  
double pi = 3.14;  
boolean isTrue = true;  
String text = "Hello";
```

Virtuální stroj a kompilace

Java aplikace jsou kompilovány do bajtkódu, který je spouštěn na virtuálním stroji Java (JVM). JVM interpretuje bajtkód a provádí ho na cílovém operačním systému. To umožňuje Java aplikacím běžet na různých platformách bez nutnosti přepisování kódu.

```
javac HelloWorld.java
java HelloWorld
```

Tvorba uživatelského rozhraní

Pro tvorbu uživatelského rozhraní v Javě se obvykle používá knihovna Swing nebo JavaFX. Tyto knihovny poskytují sadu komponent pro vytváření interaktivních uživatelských rozhraní, včetně tlačítek, textových polí, seznamů a dalších prvků.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class Main extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(event -> System.out.println("Hello World!"));
        StackPane root = new StackPane();
        root.getChildren().add(btn);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

Delegáty a události

V Javě se události a jejich obsluha obvykle řeší pomocí rozhraní a anonymních tříd. Například pro zachycení události tlačítka se vytvoří anonymní třída implementující rozhraní `ActionListener`, které obsahuje metodu pro zpracování události.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

public class Main {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Event Example");
```

```

        JButton button = new JButton("Click me");

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button clicked");
            }
        });

        frame.add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

```

Přístup k databázím

Pro přístup k databázím v Javě se obvykle používá standardní API pro práci s databázemi (JDBC). JDBC umožňuje Java aplikacím vytvářet spojení s databází, provádět SQL dotazy a zpracovávat výsledky.

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class Main {
    public static void main(String[] args) {
        try {
            Connection connection =
                DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase", "username",
                    "password");
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery("SELECT * FROM mytable");

            while (resultSet.next()) {
                System.out.println(resultSet.getString("column1") + " " +
                    resultSet.getString("column2"));
            }

            connection.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Práce s textovými daty

Pro práci s textovými daty v Javě se obvykle používá třída `java.io.BufferedReader`, která umožňuje číst textová data ze vstupního proudu. Existují také různé třídy pro práci s textem, jako například `java.lang.String` pro práci s textovými řetězci.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("input.txt")))
        {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Asynchronní programování

Asynchronní programování v Javě je obvykle řešeno pomocí vláken a rozhraní

`java.util.concurrent.Future`. Vlákná umožňují provádět úlohy paralelně a asynchronně, což zlepšuje výkon aplikace a reaktivitu uživatelského rozhraní.

```
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            System.out.println("Hello from thread!");
        });
        thread.start();

        System.out.println("Hello from main thread!");
    }
}
```

Serializace

Serializace v Javě je proces převádění objektů na bajtový tok, který může být uložen nebo přenesen přes síť. K serializaci objektu v Javě se obvykle používá rozhraní `java.io.Serializable`.

```
import java.io.*;

class MyClass implements Serializable {
```

```

    int number;
    String text;

    public MyClass(int number, String text) {
        this.number = number;
        this.text = text;
    }
}

public class Main {
    public static void main(String[] args) {
        try {
            // Serializace
            ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("data.txt"));
            MyClass obj = new MyClass(42, "Hello");
            out.writeObject(obj);
            out.close();

            // Deserializace
            ObjectInputStream in = new ObjectInputStream(new
FileInputStream("data.txt"));
            MyClass newObj = (MyClass) in.readObject();
            System.out.println(newObj.number);
            System.out.println(newObj.text);
            in.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Síťová komunikace

Pro síťovou komunikaci v Javě se obvykle používá balíček `java.net`, který obsahuje třídy pro vytváření a správu síťových spojení. Například třída `java.net.Socket` umožňuje vytvořit klienta pro komunikaci se serverem pomocí TCP/IP.

```

import java.io.*;
import java.net.*;

public class Server {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(12345);
            System.out.println("Server started.");

            while (true) {

```

```

        Socket clientSocket = serverSocket.accept();
        System.out.println("Client connected: " +
clientSocket.getInetAddress());

        PrintWriter out = new PrintWriter(clientSocket.getOutputStream(),
true);

        out.println("Hello from server!");
        out.close();

        clientSocket.close();
    }
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

Reprezentace čísel v počítači

- Reprezentace čísel v počítači (číselné soustavy a převody, celá čísla, čísla s pevnou řádovou čárkou, čísla s pohyblivou řádovou čárkou vyjádřená v binárním, decimálním a hexadecimálním základu, aritmetika s čísly v různých reprezentacích, kódování znaků).

Číselné soustavy

Binární soustava

- Binární soustava je základem pro reprezentaci čísel v počítačích.
- Každé číslo se reprezentuje pomocí kombinace bitů, kde každý bit může být buď 0 nebo 1.
- Hodnota každého bitu je mocninou čísla 2, přičemž nejnižší bit má nejnižší hodnotu a každý další bit má hodnotu dvojnásobku předchozího bitu.
- Například: Binární číslo 1011 znamená $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11$ v desítkové soustavě.

Decimální soustava

- Standardní číselná soustava, kterou používáme v každodenním životě.
- Má základ 10, což znamená, že používáme deset symbolů (0 až 9).
- Každá pozice v čísle má hodnotu mocniny čísla 10.
- Například: Číslo 123 znamená $1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 = 123$.

Hexadecimální soustava:

- Používá 16 symbolů, včetně číslic 0 až 9 a písmen A-F.
- Je často používána pro zjednodušení a zkrácení dlouhých binárních řetězců.
- Každý symbol má hodnotu mocniny čísla 16.
- Například: Číslo A1 v hexadecimální soustavě znamená $10 \cdot 16^1 + 1 \cdot 16^0 = 161$ v desítkové soustavě.

Převody mezi soustavami

Binární na desítkovou:

- Každému bitu přiřadíte hodnotu na základě jeho pozice (mocniny čísla 2) a následně sečtete všechny hodnoty.

Desítkovou na binární:

- Číslo postupně dělíte 2 a zaznamenáváte zbytek až do doby, kdy dělení nedá nulu.

Binární na hexadecimální a zpět:

- Binární číslo se rozdělí do skupin po čtyřech bitech, každá skupina se převede na odpovídající hexadecimální symbol.

Hexadecimální na binární a zpět:

- Každý hexadecimální symbol se převede na ekvivalentní binární číslo a naopak.

Aritmetika s čísly

Sčítání:

- Provádí se podobně jako sčítání v desítkové soustavě, počínaje nejnižšími řády a přenesením případného přetečení na vyšší řády.

Odčítání:

- Používá se doplňkový kód a postupuje se obdobně jako u sčítání, přičemž lze využít přenosu při odečítání.

Násobení a dělení:

- Provádí se postupným posouváním jednoho z operandů vlevo (násobení) nebo vpravo (dělení) a přičítáním (násobení) nebo odečítáním (dělení).
- ☐ Naučit se tyto operace

BCD (Binary Coded Decimal)

- Je to způsob kódování desítkových čísel pomocí binárních číslic.
- Každá desítková číslice (0 až 9) je kódována čtyřbitovým binárním číslem (0000 až 1001).
- Například: Číslo 25 je kódováno jako 0010 (pro 2) a 0101 (pro 5).

Dvojkový doplněk (Two's Complement)

- Dvojkový doplněk je způsob reprezentace záporných čísel v binární soustavě.
- Pro kladná čísla se používá klasická binární reprezentace.
- Pro záporná čísla se používá doplněk k jedničce.
- To znamená, že k zápornému číslu se přičte jeho absolutní hodnota představovaná v binární podobě.
- Například: Pro číslo -5, binární reprezentace čísla 5 je 0101, takže dvojkový doplněk čísla -5 je 1011.

Čísla s plovoucí desetinnou čárkou

- Čísla s plovoucí desetinnou čárkou jsou reprezentována v počítači pomocí tzv. formátu s plovoucí řádovou čárkou (floating-point format).
 - Nejčastěji používaným standardem pro reprezentaci čísel s plovoucí desetinnou čárkou je standard IEEE 754.
 - Tento standard definuje formát, který zahrnuje znaménko, exponent a mantisu.
 - Mantisa představuje samotné číslo, exponent určuje, kam se čárka posune, a znaménko označuje, zda je číslo kladné nebo záporné.
 - Při provádění aritmetických operací s čísly v tomto formátu se používají speciální algoritmy, které zohledňují rozdíly v řádové velikosti a přesnosti reprezentace čísel s různými počty bitů pro mantisu a exponent.
- ☐ Naučit se to počítat

Kódování znaků

- Znaky, jako jsou písmena, číslice a speciální symboly, jsou v počítači reprezentovány pomocí kódů.
- ASCII (American Standard Code for Information Interchange) je jedním z nejčastějších kódování znaků, které přiřazuje každému znaku sedmibitový kód.
- Unicode je rozšíření ASCII, které podporuje mnohem širší rozsah znaků a používá 16-bitové nebo 32-bitové kódy.

Unicode

- Unicode je standard, který přiřazuje jedinečný číselný identifikátor každému znaku a symbolu používanému v počítačových systémech.
- Pro reprezentaci znaků používá 16-bitové (UTF-16) nebo 32-bitové (UTF-32) kódování.
- Při použití UTF-16 jsou běžné znaky kódovány jedním 16-bitovým kódem, ale pro méně časté znaky jsou potřeba dva 16-bitové kódy.

☐ Naučit se převádět

Softwarové inženýrství

Obsah

- [Životní cyklus vývoje software](#)
- [UML](#)
- [Informační systém](#)
- [Návrhové vzory a praktiky pro vývoj informačních systémů](#)

Zadání

Softwarové inženýrství (SWI, VIS)

- Životní cyklus vývoje software (typické fáze, jejich náplň, základní modely vývoje).
- UML (typy diagramů, statický náhled na systém, dynamický náhled na systém, mapování na zdrojový kód, využití v rámci vývoje).
- Informační systém (životní cyklus, návrh a architektura informačního systému).
- Návrhové vzory a praktiky pro vývoj informačních systémů (vzory GoF, doménová logika, datové zdroje, objektově-relační chování a struktury, doménově specifické jazyky).

 **Příklad otázky**

Vysvětlete, v jaké situaci je vhodné použít vzor Composite, použijte UML diagram pro jeho schématické vyjádření a promítněte tento diagram do zdrojového kódu ve vámi zvoleném programovacím jazyce.

Informační systém

 [Podívat se na prezentace od kudělký](#)

- Informační systém (životní cyklus, návrh a architektura informačního systému).

Informační systém je soubor vzájemně propojených komponent, které shromažďují, ukládají, zpracovávají a poskytují informace pro podporu rozhodování, koordinaci, kontrolu, analýzu a vizualizaci v organizaci. Životní cyklus informačního systému zahrnuje několik fází od jeho plánování a vývoje až po údržbu a zlepšování. Návrh a architektura informačního systému jsou klíčové pro zajištění jeho efektivnosti a spolehlivosti.

Životní cyklus informačního systému

Životní cyklus informačního systému lze rozdělit do několika hlavních fází:

1. Plánování:

- **Náplň:** Identifikace potřeb a cílů organizace, analýza stávajícího systému, stanovení požadavků na nový systém, zvažování alternativ a provádění studie proveditelnosti.
- **Výstupy:** Plán projektu, specifikace požadavků, studie proveditelnosti, rozpočet a časový plán.

2. Analýza požadavků:

- **Náplň:** Shromažďování a analýza požadavků od uživatelů a dalších zainteresovaných stran, vytváření detailních specifikací a modelů procesů.
- **Výstupy:** Dokumentace požadavků (např. SRS - Software Requirements Specification), modely procesů, případně prototypy.

3. Návrh systému:

- **Náplň:** Vytváření architektonického návrhu systému, návrh databáze, uživatelského rozhraní a definování technických specifikací. Návrh zahrnuje jak logickou, tak fyzickou architekturu systému.
- **Výstupy:** Architektonické diagramy, ER diagramy (Entity-Relationship diagramy), návrh databáze, návrhy uživatelského rozhraní.

4. Implementace:

- **Náplň:** Kódování, integrace a testování jednotlivých modulů systému. Programátoři píší zdrojový kód a provádějí jednotkové a integrační testy.

- **Výstupy:** Zdrojový kód, binární soubory, průběžné verze systému.

5. Testování:

- **Náplň:** Systematické testování systému pro zajištění, že splňuje požadavky a funguje podle specifikací. Zahrnuje různé typy testů (jednotkové, integrační, systémové, akceptační).
- **Výstupy:** Testovací plány, testovací skripty, zprávy o chybách, výsledky testů.

6. Nasazení:

- **Náplň:** Instalace a konfigurace systému v produkčním prostředí, migrace dat, školení uživatelů a příprava podpůrné dokumentace.
- **Výstupy:** Nasazený systém, uživatelské příručky, školení materiály.

7. Údržba a podpora:

- **Náplň:** Pravidelná údržba systému, opravy chyb, aktualizace softwaru, zlepšování funkcionalit a poskytování technické podpory uživatelům.
- **Výstupy:** Opravy chyb, aktualizace, servisní balíčky, záznamy o údržbě.

Návrh a architektura informačního systému

Návrh a architektura informačního systému zahrnují několik klíčových aspektů:

1. Logická architektura:

- Zahrnuje návrh logické struktury systému, jako jsou třídy, objekty, entity a jejich vztahy. Tato část zahrnuje tvorbu ER diagramů, třídových diagramů a dalších UML diagramů pro modelování struktury systému.

2. Fyzická architektura:

- Zaměřuje se na fyzické komponenty systému, jako jsou servery, databázové systémy, síťová infrastruktura a další hardware. Tato část zahrnuje nasazovací diagramy a diagramy komponent, které ukazují, jak budou fyzické části systému propojeny a jak budou komunikovat.

3. Architektura aplikace:

- Určuje strukturu softwarových komponent a jejich interakce. To zahrnuje definování vrstev architektury (např. prezentační vrstva, aplikační vrstva, databázová vrstva), návrh rozhraní a protokolů pro komunikaci mezi komponentami.

4. Datová architektura:

- Zahrnuje návrh databázových struktur, schémat a způsobů ukládání a přístupu k datům. To zahrnuje normalizaci databází, definování datových modelů a plánování zálohování a obnovy dat.

5. Bezpečnostní architektura:

- Zahrnuje návrh bezpečnostních opatření pro ochranu dat a systémů. To zahrnuje autentizaci, autorizaci, šifrování, zálohování a další bezpečnostní mechanismy.

Využití návrhu a architektury v rámci vývoje

- **Plánování a analýza:** Logická a fyzická architektura pomáhají v raných fázích projektu při plánování a analýze požadavků. Poskytují přehled o struktuře a komponentách systému, což

usnadňuje rozhodování.

- **Návrh a implementace:** Detailní návrhy a modely usnadňují programátorům implementaci systému podle specifikací. Zajišťují, že všechny části systému budou správně integrovány.
- **Testování:** Architektura systému je základem pro vytváření testovacích plánů a scénářů. Pomáhá identifikovat kritické komponenty a závislosti, které je třeba otestovat.
- **Nasazení a údržba:** Fyzická a bezpečnostní architektura jsou klíčové při nasazení systému a jeho údržbě. Zajišťují, že systém bude spolehlivý, bezpečný a snadno udržovatelný.

Informační systém a jeho životní cyklus tedy zahrnují pečlivé plánování, návrh, implementaci, testování, nasazení a údržbu. Kvalitní návrh a architektura jsou klíčové pro úspěšný vývoj a provoz informačního systému, který splňuje potřeby a požadavky organizace.

Informační systém (životní cyklus, návrh a architektura informačního systému)

Životní cyklus informačního systému

Životní cyklus informačního systému (IS) se skládá z několika fází, které zajišťují systematický přístup k vývoji a nasazení IS. Tyto fáze jsou:

1. Vize:

- **Náplň:** Identifikace potřeb organizace a definování hlavních cílů IS. Tento krok zahrnuje strategické rozhodnutí, co by měl systém řešit a proč je potřeba.
- **Výstupy:** Dokument popisující systém z pohledu zákazníka.

2. Analýza:

- **Náplň:** Shromažďování a analýza požadavků uživatelů a dalších zainteresovaných stran. Vytváření use-case modelů a specifikace funkčních požadavků.
- **Výstupy:** Funkční specifikace, use-case diagramy, diagramy aktivit.

3. Logický návrh:

- **Náplň:** Vytváření modelu domény, který zahrnuje třídní diagramy, vztahy a interakce mezi třídami.
- **Výstupy:** Návrh doménového modelu (statický diagram tříd, sekvenční diagramy, použité vzory).

4. Technologický návrh:

- **Náplň:** Definování technologických platform, databázových struktur a rozhraní. Rozhodování o architektuře systému a volbě technologií.
- **Výstupy:** Technická specifikace, modely komponent a jejich rozhraní.

5. Vývoj:

- **Náplň:** Kódování a integrace jednotlivých komponent podle návrhu. Implementace funkcionalit a propojení systému.

- **Výstupy:** Zdrojový kód, binární soubory, průběžné verze systému.

6. Nasazení a provoz:

- **Náplň:** Instalace systému v produkčním prostředí, migrace dat, školení uživatelů a spuštění systému. Průběžná údržba a aktualizace systému.
- **Výstupy:** Nasazený systém, uživatelské příručky, záznamy o údržbě.

Návrh a architektura informačního systému

Návrh a architektura IS zahrnují několik klíčových aspektů:

1. Architektura:

- **Definice:** Architektura IS zahrnuje organizaci komponent systému, jejich vzájemné vztahy a vztahy k okolí systému. Zahrnuje také principy návrhu a vývoje systému.
- **Pohledy na architekturu:**
 - **Doménový pohled:** Z pohledu zákazníka, zahrnuje skupinu souvisejících "věcí" a procesů.
 - **Vývojářský pohled:** Globální struktura systému, chování částí a jejich propojení.
 - **Datový pohled:** Přístup k datům a toky dat v systému.
 - **Fyzický pohled:** Fyzické rozmístění komponent.

2. Statická vs. dynamická architektura:

- **Statická architektura:** Pevná struktura systému, která se nemění za běhu.
- **Dynamická architektura:** Podporuje vznik a zánik komponent za běhu systému podle předem definovaných pravidel.
- **Mobilní architektura:** Rozšiřuje dynamickou architekturu o mobilní prvky, které se mohou přesouvat.

3. Komponenty:

- **Definice:** Komponenta je softwarový balík, služba nebo modul, který zajišťuje určitou funkčnost a má definované rozhraní.
- **Příklady:** Modul pro věrnostní program, databázová služba.

4. Návrh vs. nasazení:

- **Návrh:** Popisuje logickou strukturu systému, jak funguje a s čím pracuje.
- **Nasazení:** Popisuje, kde systém běží (HW, SW platforma).

5. Proces návrhu architektury:

- **Kroky:**
 1. Identifikace požadavků.
 2. Dekompozice systému do komponent.
 3. Přidělení požadavků k jednotlivým komponentám.
 4. Ověření, že všechny požadavky byly přiděleny.

Využití návrhu a architektury v rámci vývoje

- **Plánování a analýza:** Architektura a návrh poskytují základní přehled, který usnadňuje rozhodování a plánování.
- **Návrh a implementace:** Umožňují strukturovaný přístup k implementaci a zajišťují správnou integraci všech částí systému.
- **Testování:** Architektonické modely jsou základem pro vytváření testovacích plánů a scénářů.
- **Nasazení a údržba:** Fyzická a bezpečnostní architektura zajišťují spolehlivost a bezpečnost systému, což je klíčové pro jeho údržbu a provoz.

Na základě prezentací z přednášek a vypracovaných otázek můžeme shrnout, že kvalitní návrh a architektura jsou nezbytné pro úspěšný vývoj a provoz informačního systému, který splňuje požadavky a potřeby organizace.

Návrhové vzory a praktiky pro vývoj informačních systémů

- Návrhové vzory a praktiky pro vývoj informačních systémů (vzory GoF, doménová logika, datové zdroje, objektově-relační chování a struktury, doménově specifické jazyky).

Návrhové vzory a praktiky jsou důležitými nástroji pro efektivní a kvalitní vývoj informačních systémů. Tyto vzory poskytují osvědčená řešení běžných problémů a zlepšují strukturu, udržovatelnost a flexibilitu kódu.

Vzory GoF (Gang of Four)

Vzory GoF jsou základní návrhové vzory, které byly popsány ve známé knize "Design Patterns: Elements of Reusable Object-Oriented Software" od Ericha Gamma, Richarda Helma, Ralpha Johnsona a Johna Vlissidese. Tyto vzory se dělí do tří kategorií:

1. Creational (tvořivé) vzory:

- **Abstract Factory:** Poskytuje rozhraní pro vytváření rodin souvisejících nebo závislých objektů bez specifikace jejich konkrétních tříd.
- **Builder:** Odděluje konstrukci složitého objektu od jeho reprezentace, což umožňuje stejný konstrukční proces vytvářet různé reprezentace.
- **Factory Method:** Definuje rozhraní pro vytváření objektu, ale nechává podtřídám rozhodnout, jakou třídu instancí vytvořit.
- **Prototype:** Umožňuje klonování objektů pomocí prototypu namísto jejich vytváření od nuly.
- **Singleton:** Zajišťuje, že třída má pouze jednu instanci, a poskytuje globální přístupový bod k této instanci.

2. Structural (strukturální) vzory:

- **Adapter:** Umožňuje spolupráci mezi neslučitelnými rozhraními tak, že jedno rozhraní přizpůsobí jinému.
- **Bridge:** Odděluje abstrakci od její implementace, což umožňuje, aby mohly být obě nezávisle změněny.

- **Composite:** Skládá objekty do stromové struktury pro reprezentaci hierarchie část-celku, což umožňuje klientům zacházet s jednotlivými objekty a jejich složením jednotně.
- **Decorator:** Dynamicky přidává odpovědnosti objektům tím, že je obaluje do jiných objektů.
- **Facade:** Poskytuje jednotné rozhraní k sadě rozhraní v podkladových subsystémech.
- **Flyweight:** Efektivně podporuje velké množství malých objektů tím, že sdílí co nejvíce dat.
- **Proxy:** Poskytuje zástupný objekt, který řídí přístup k jinému objektu.

3. Behavioral (behaviorální) vzory:

- **Chain of Responsibility:** Deleguje požadavek na objekty v řetězci, dokud není požadavek zpracován.
- **Command:** Přeměňuje požadavky nebo jednoduché operace do objektů.
- **Interpreter:** Definuje gramatiku pro jazyk a interpretuje věty v tomto jazyce.
- **Iterator:** Poskytuje sekvenční přístup k prvkům agregovaného objektu, aniž by odhaloval jeho podkladovou reprezentaci.
- **Mediator:** Definuje objekt, který zapouzdřuje způsob, jakým soubor objektů spolu interaguje.
- **Memento:** Umožňuje zachytit a externalizovat vnitřní stav objektu, aniž by došlo k porušení zapouzdření, takže objekt může být vrácen do tohoto stavu později.
- **Observer:** Definuje závislost 1: mnoho mezi objekty, takže když se změní stav jednoho objektu, všechny jeho závislé objekty jsou upozorněny a automaticky aktualizovány.
- **State:** Umožňuje objektu měnit své chování, když se změní jeho vnitřní stav.
- **Strategy:** Definuje rodinu algoritmů, zapouzdří každý z nich a činí je zaměnitelnými.
- **Template Method:** Definuje kostru algoritmu v metodě, přičemž některé kroky deleguje na podtřídy.
- **Visitor:** Představuje operaci, která se má provádět na prvcích struktury objektu.

Vzory doménové logiky

Vzory doménové logiky se zaměřují na organizaci byznys logiky v aplikaci. Mezi hlavní vzory patří:

1. Transaction Script:

- Organizes business logic by procedures where each procedure handles a single request from the presentation.
- Vhodný pro jednoduché aplikace s malým množstvím byznys logiky.

2. Domain Model:

- An object model of the domain that incorporates both behavior and data.
- Vhodný pro složitější aplikace s bohatou doménovou logikou.

3. Table Module:

- A single instance that handles the business logic for all rows in a database table or view.
- Vhodný pro aplikace, kde byznys logika je úzce spjata s daty v databázi.

4. Service Layer:

- Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.

- Vhodný pro aplikace s komplexními operacemi a potřebou jasně definovaných služeb.

Vzory pro práci s datovými zdroji

Vzory pro práci s datovými zdroji se zaměřují na způsoby, jakým aplikace komunikuje s databází a jinými úložišti dat:

1. Table Data Gateway:

- An object that acts as a gateway to a database table. One instance handles all the rows in the table.
- Vhodný pro jednoduchou doménovou logiku a práci s transakčními skripty.

2. Row Data Gateway:

- An object that acts as a gateway to a single record in a data source. There is one instance per row.
- Vhodný pro práci s jednoduchou doménovou logikou.

3. Active Record:

- An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.
- Vhodný pro složitější doménovou logiku s jednoduchými operacemi přímo mapovanými na tabulky.

4. Data Mapper:

- A layer of mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.
- Vhodný pro složitou doménovou logiku a potřebu nezávislosti mezi objekty a databází.

Vzory objektově-relačního chování a struktury

Tyto vzory se zaměřují na způsoby mapování objektů na relační databáze:

1. Identity Field:

- Saves a database ID field in an object to maintain identity between an in-memory object and a database row.

2. Foreign Key Mapping:

- Maps an association between objects to a foreign key reference between tables.

3. Association Table Mapping:

- Saves an association as a table with foreign keys to the tables that are linked by the association.

4. Dependent Mapping:

- Has one class perform the database mapping for a child class.

5. Embedded Value:

- Maps an object into several fields of another object's table.

6. Serialized LOB:

- Saves a graph of objects by serializing them into a single large object (LOB), which it stores in a database field.

Vzory objektově-relačního chování

1. Unit of Work:

- Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.

2. Identity Map:

- Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks up objects using the map when referring to them.

3. Lazy Load:

- An object that doesn't contain all of the data you need but knows how to get it.

Doménově specifické jazyky (DSL)

Doménově specifické jazyky jsou programovací jazyky určené pro specifické domény, což usnadňuje vyjadřování a práci v těchto doménách:

1. External DSL:

- Language separate from the main language of the application it works with (e.g. XML, SQL, regular expressions).

2. Internal DSL:

- A particular way of using a general-purpose language to make it more expressive for a specific domain.

3. Language Workbench:

- A specialized IDE for defining and building DSLs.

Doménově specifické jazyky zlepšují produktivitu vývojářů a usnadňují komunikaci s doménovými experty, ale mohou mít také problémy jako "language cacophony" nebo náklady na vývoj a údržbu.

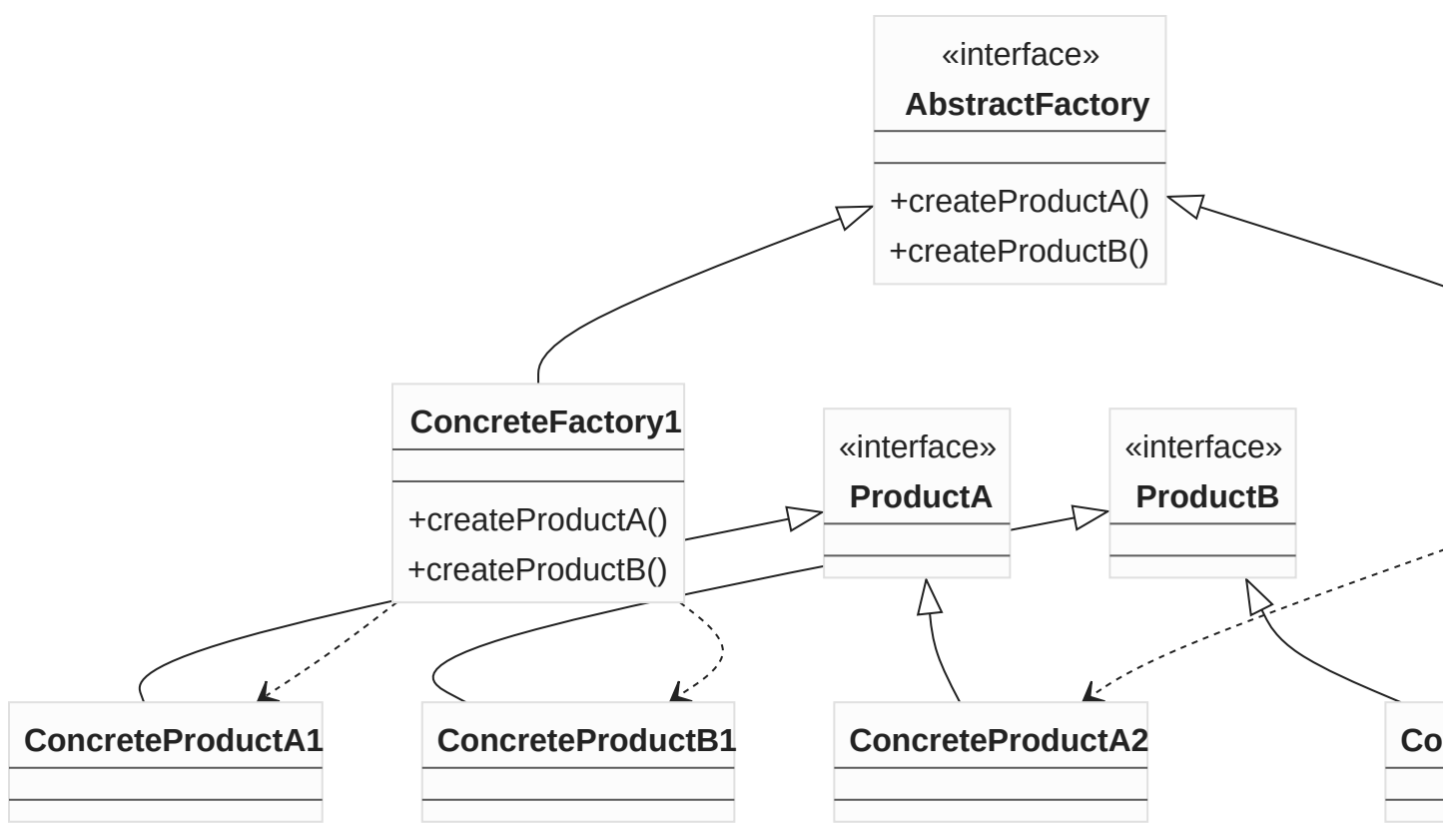
Závěr

Návrhové vzory a praktiky jsou klíčové pro efektivní vývoj informačních systémů. Poskytují osvědčené řešení pro různé problémy, zlepšují strukturu kódu a usnadňují jeho údržbu. Je důležité vybrat správné vzory podle specifických potřeb a kontextu projektu.

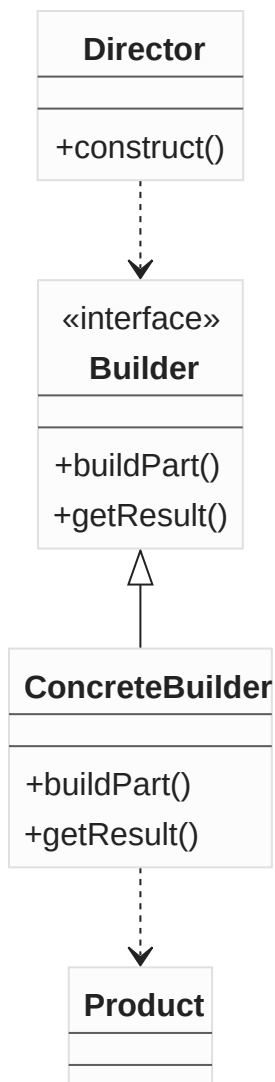
Návrhové vzory a praktiky pro vývoj informačních systémů s diagramy v Mermaid

Vzory GoF (Gang of Four)

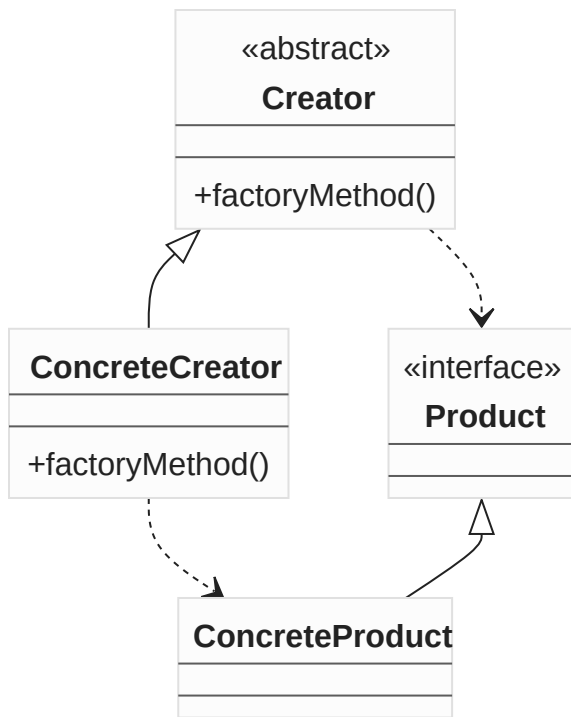
1. Abstract Factory:



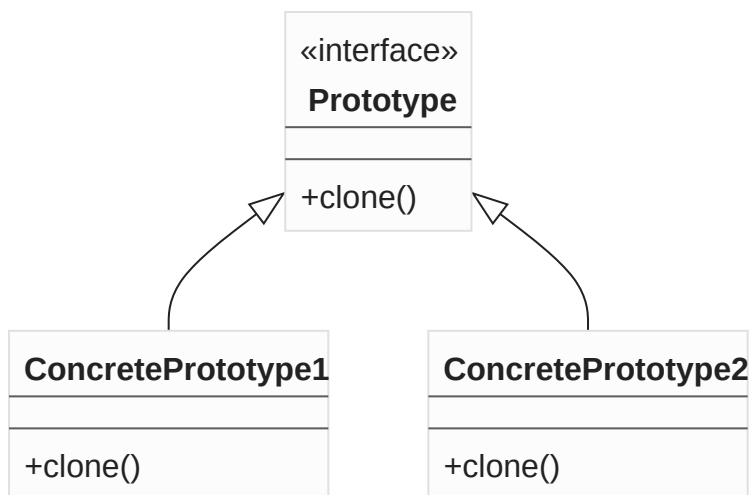
2. Builder:



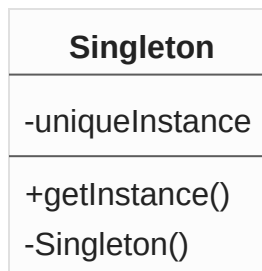
3. Factory Method:



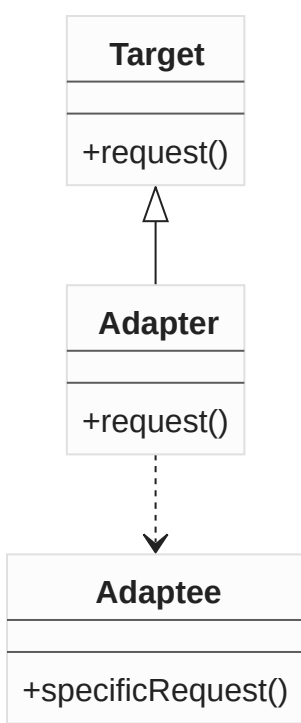
4. Prototype:



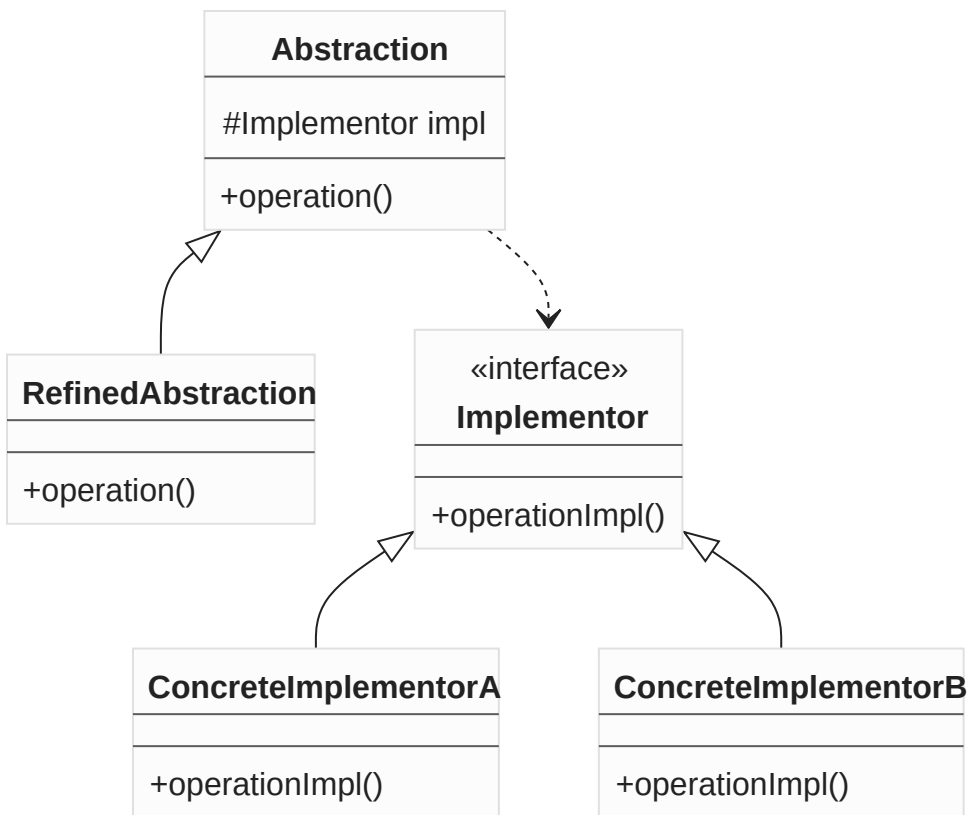
5. Singleton:



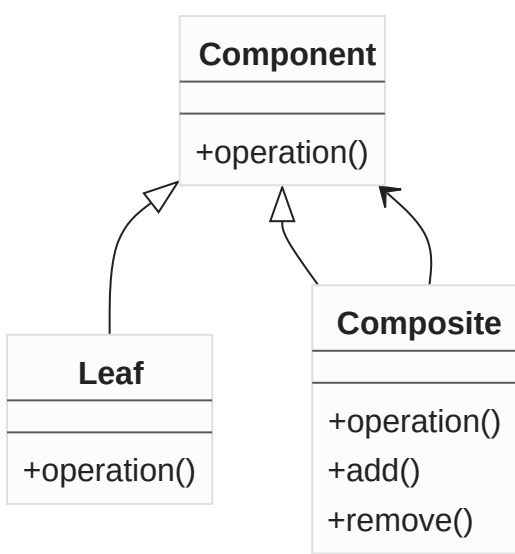
6. Adapter:



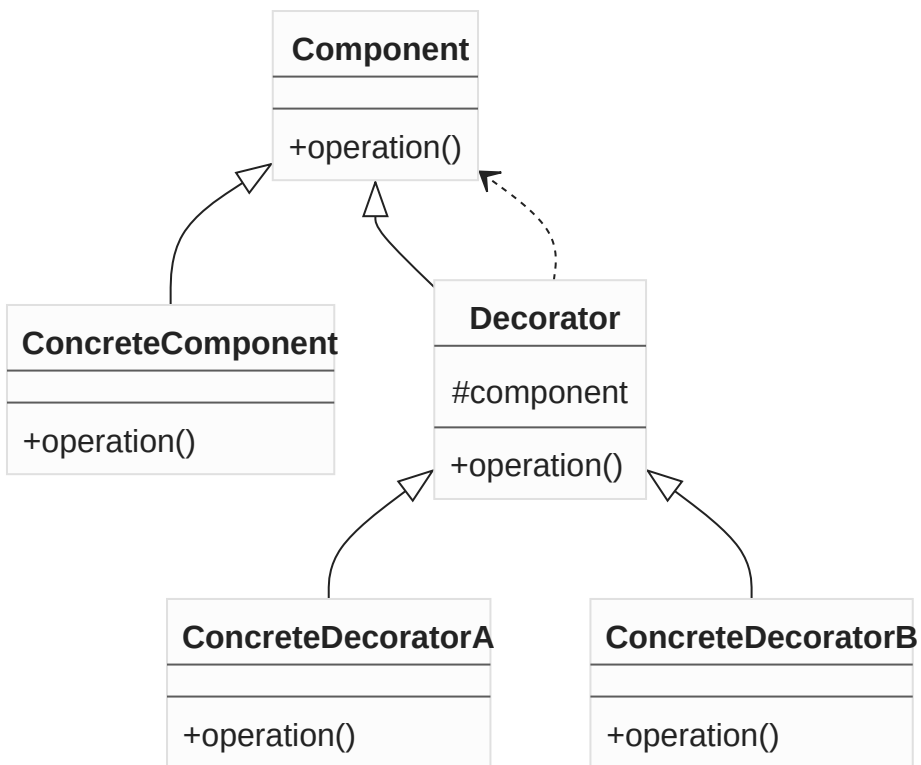
7. Bridge:



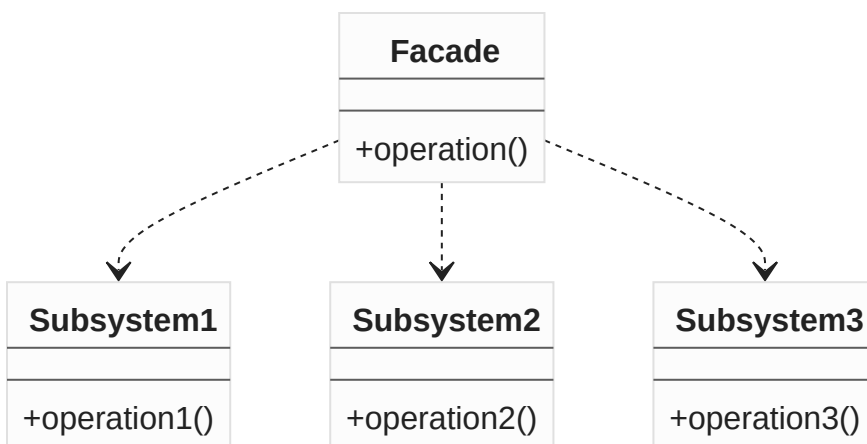
8. Composite:



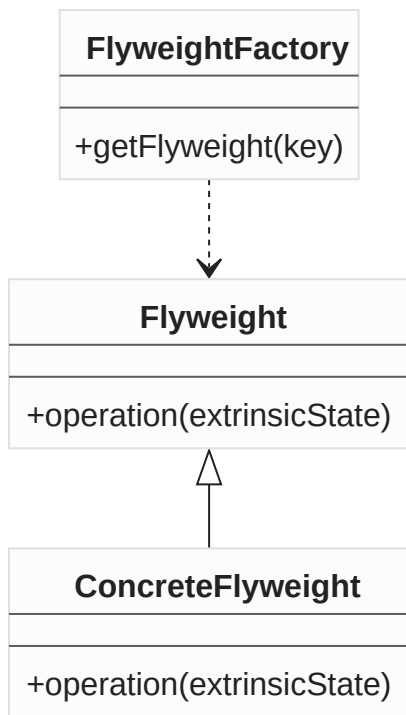
9. Decorator:



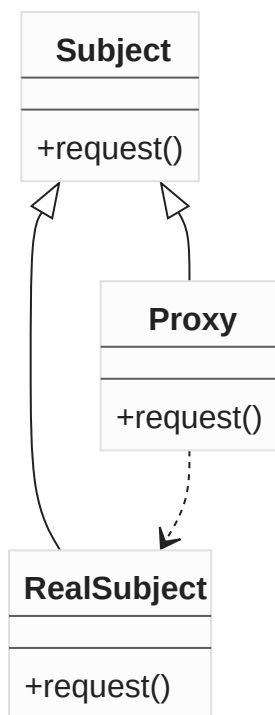
10. Facade:



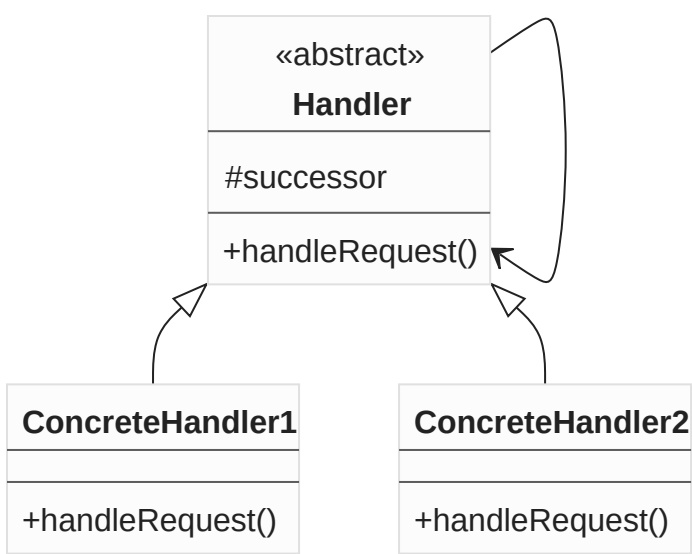
11. Flyweight:



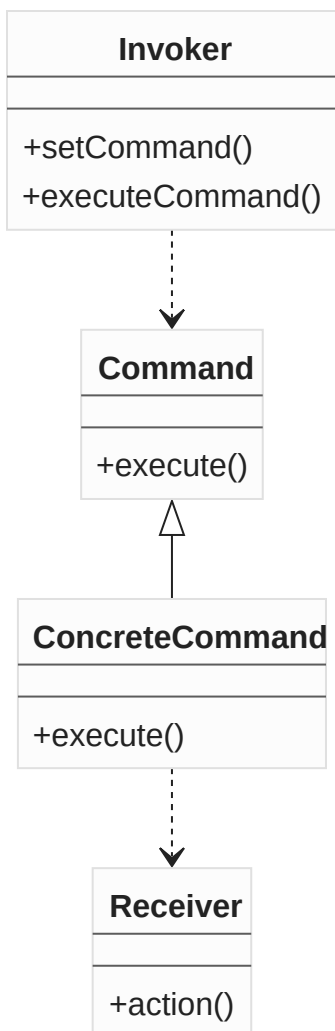
12. Proxy:



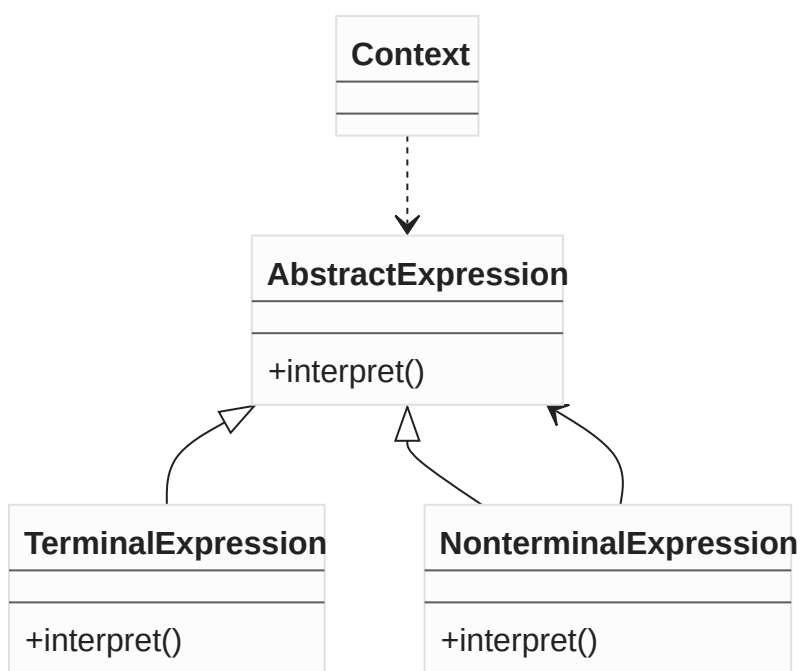
13. Chain of Responsibility:



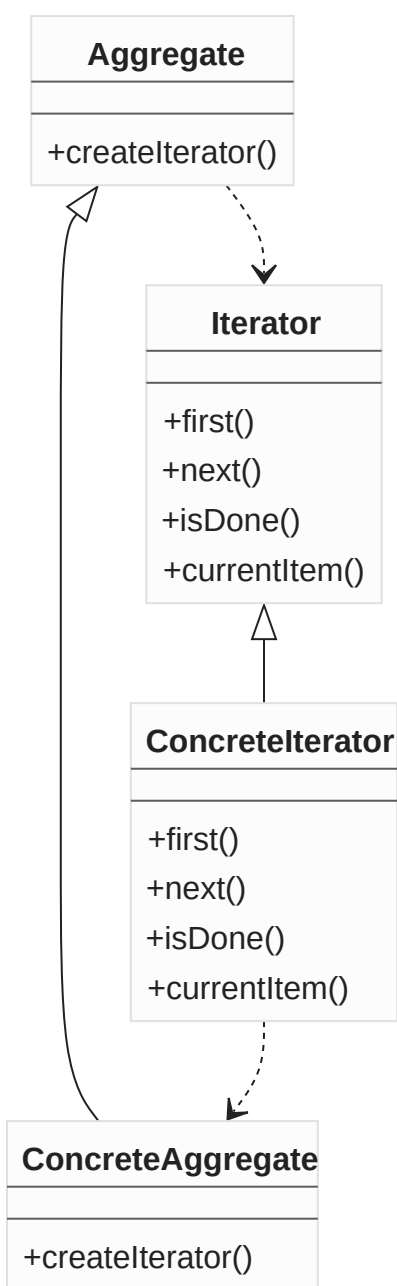
14. Command:



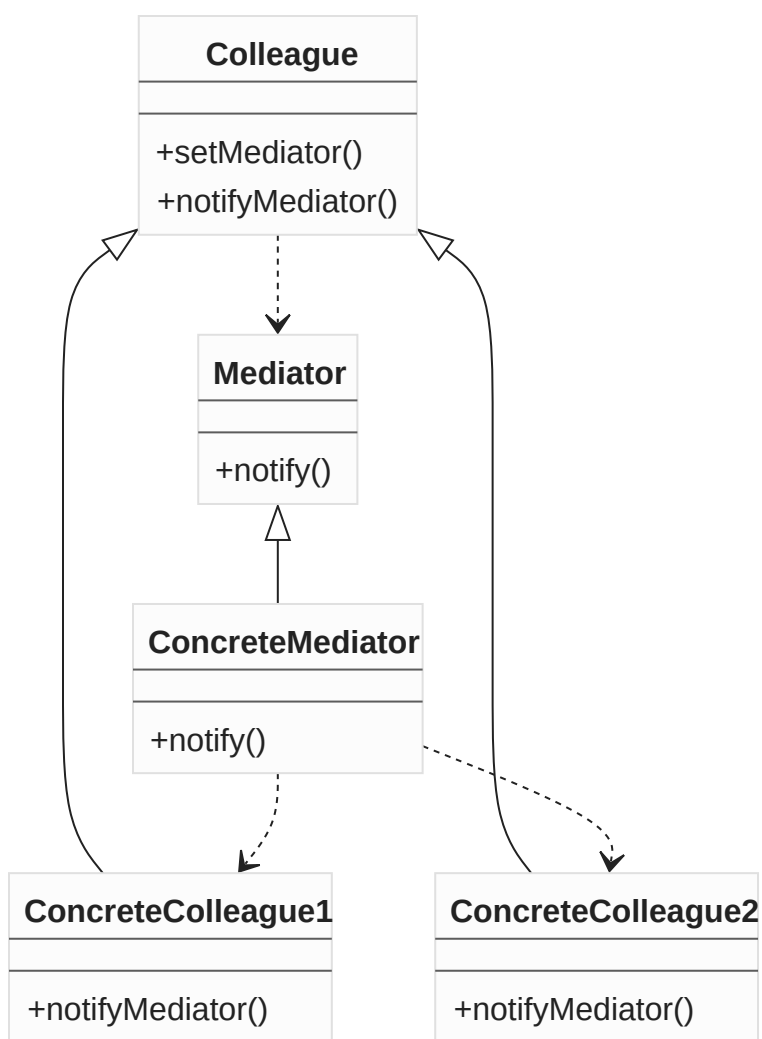
15. Interpreter:



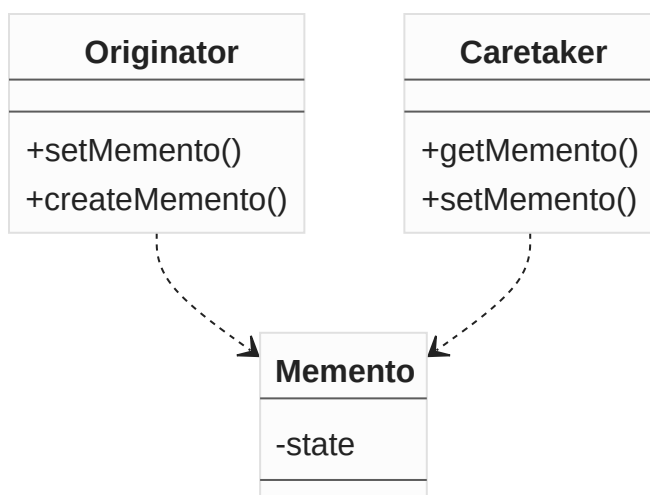
16. Iterator:



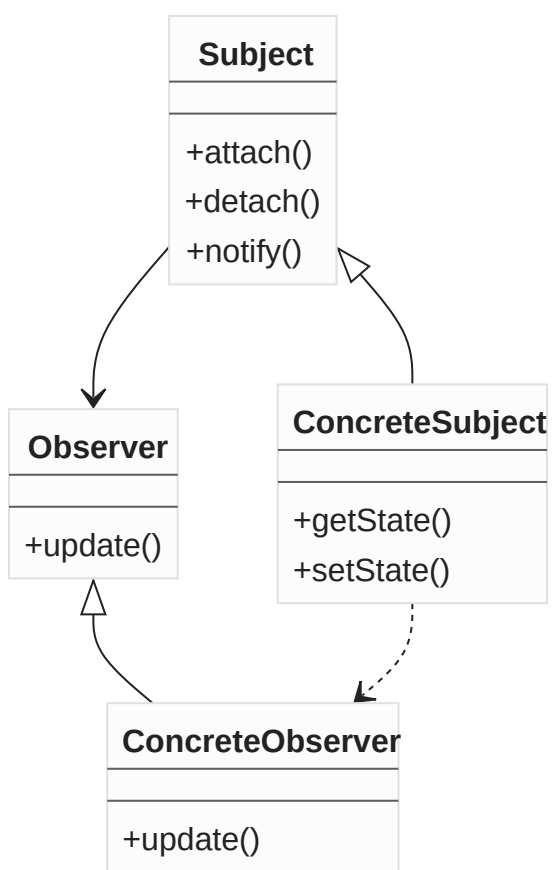
17. Mediator:



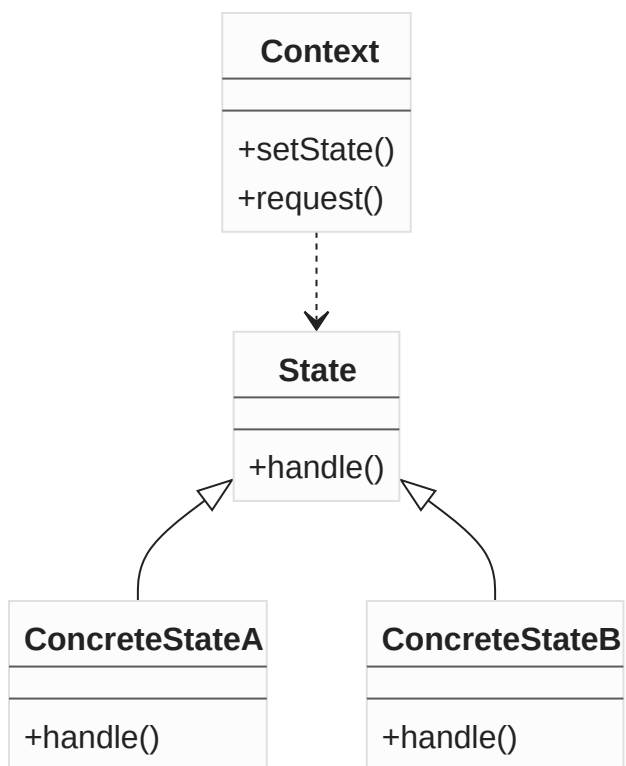
18. Memento:



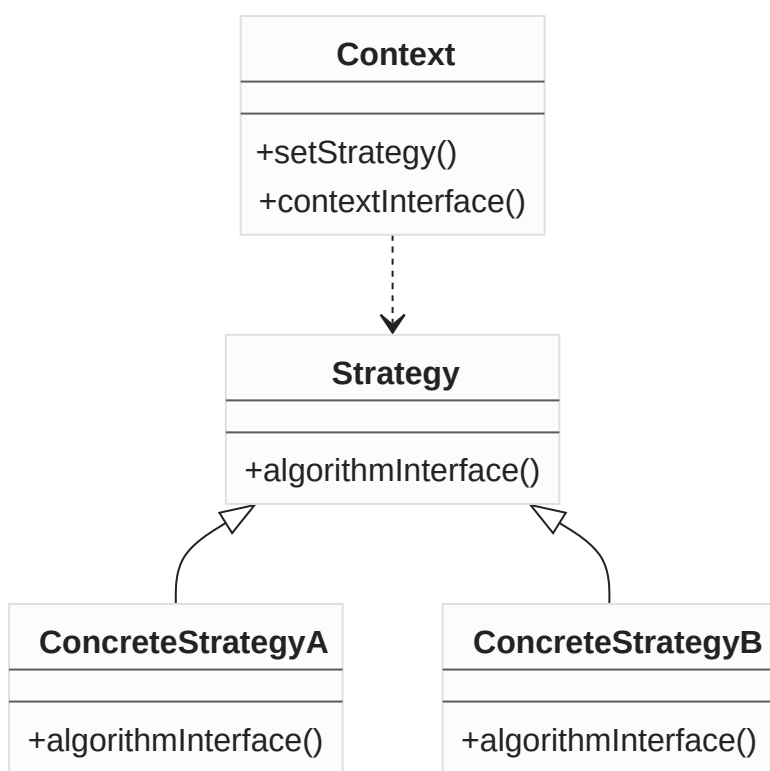
19. Observer:



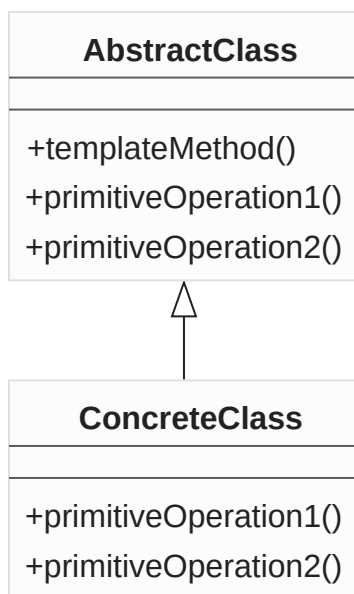
20. State:



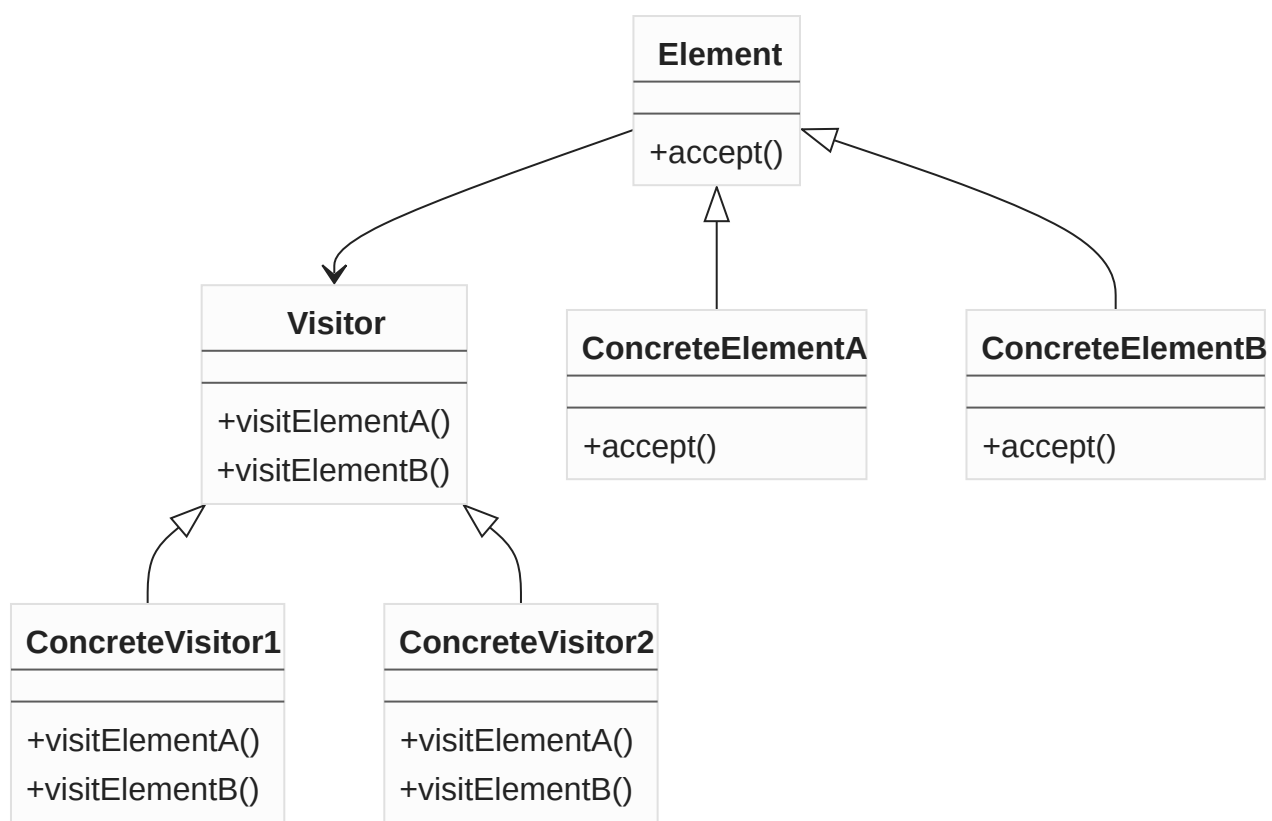
21. Strategy:



22. Template Method:

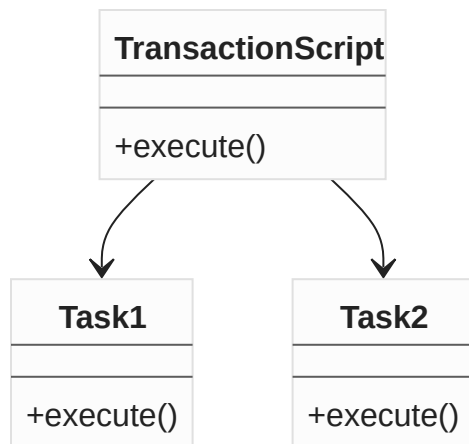


23. Visitor:

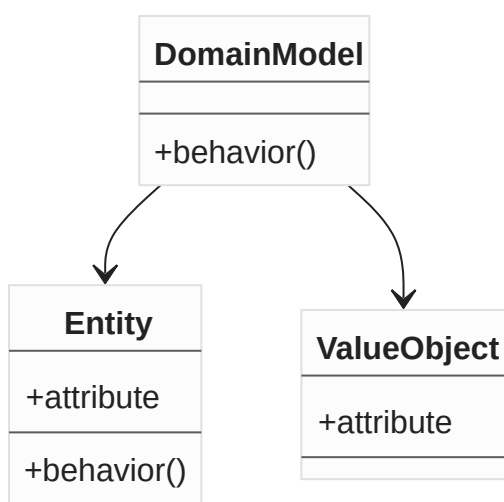


Vzory doménové logiky

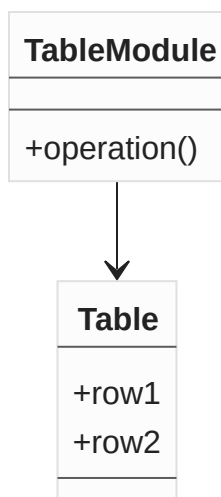
1. Transaction Script:



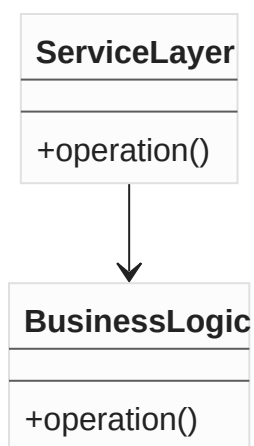
2. Domain Model:



3. Table Module:

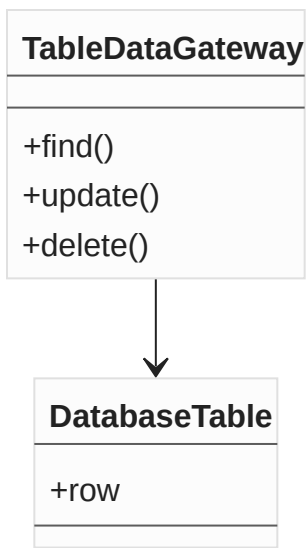


4. Service Layer:

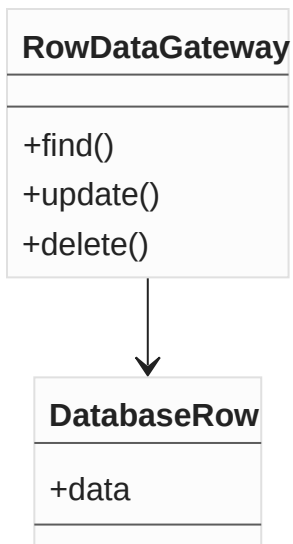


Vzory pro práci s datovými zdroji

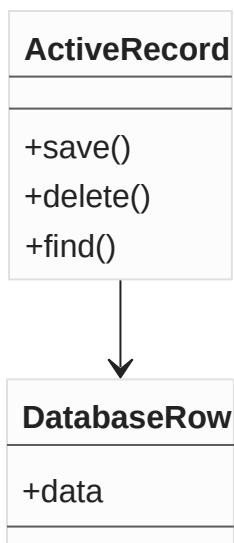
1. Table Data Gateway:



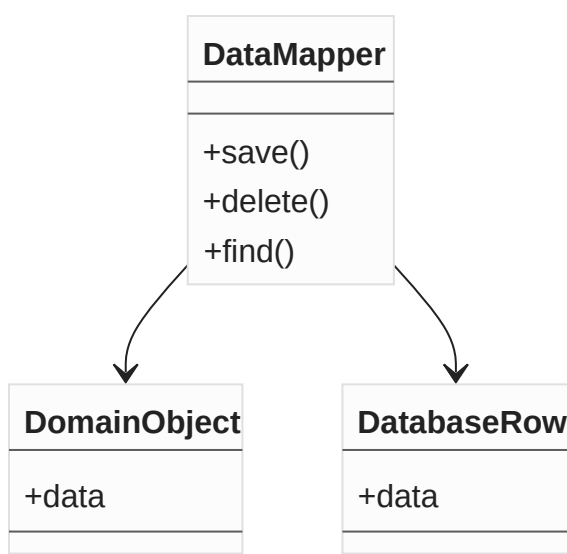
2. Row Data Gateway:



3. Active Record:

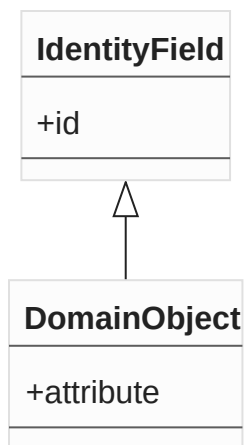


4. Data Mapper:

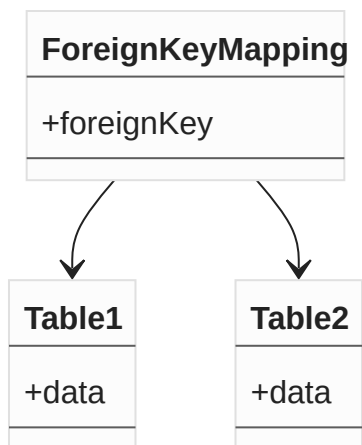


Vzory objektově-relačního chování a struktury

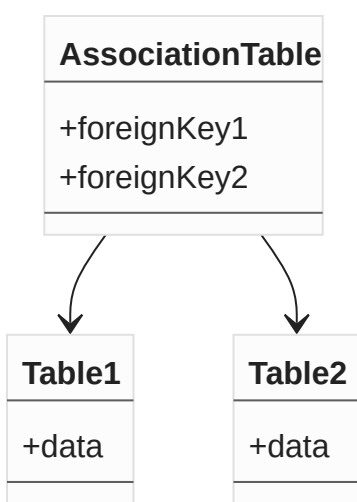
1. Identity Field:



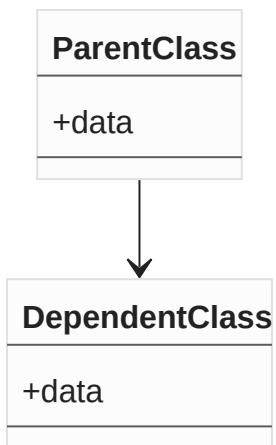
2. Foreign Key Mapping:



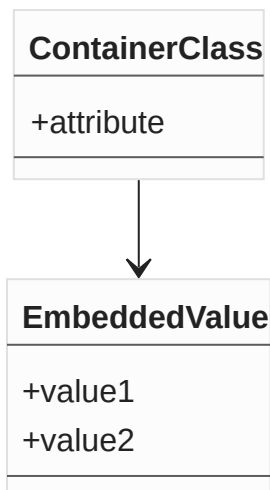
3. Association Table Mapping:



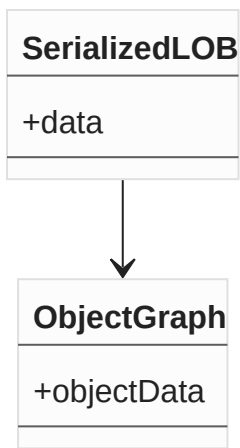
4. Dependent Mapping:



5. Embedded Value:

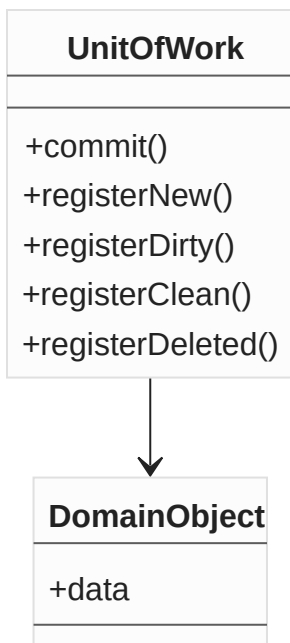


6. Serialized LOB:

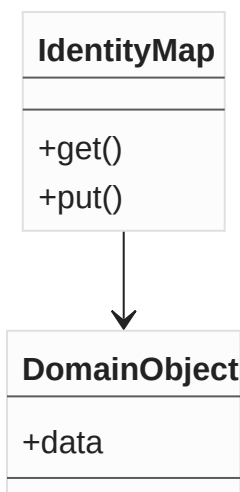


Vzory objektově-relačního chování

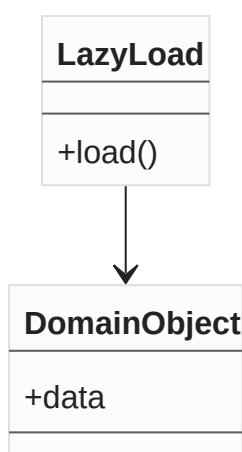
1. Unit of Work:



2. Identity Map:

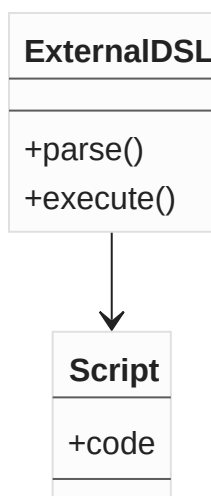


3. Lazy Load:

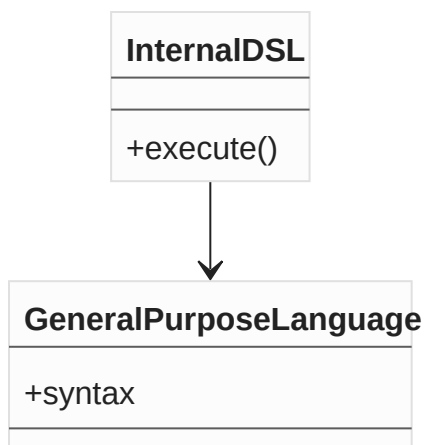


Doménově specifické jazyky (DSL)

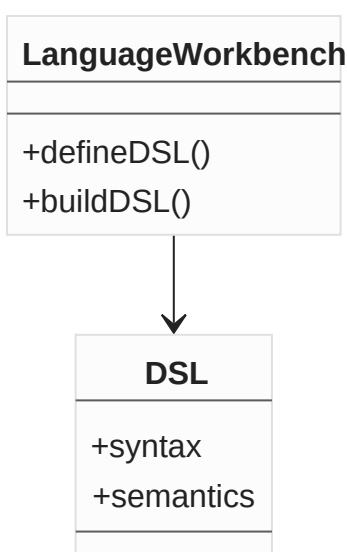
1. External DSL:



2. Internal DSL:



3. Language Workbench:



Tyto diagramy v mermaid formátu vizualizují strukturu a vztahy jednotlivých návrhových vzorů a praktik používaných při vývoji informačních systémů.

UML

Podívat se jak se kreslí

- UML (typy diagramů, statický náhled na systém, dynamický náhled na systém, mapování na zdrojový kód, využití v rámci vývoje).

Unified Modeling Language (UML) je standardizovaný jazyk pro specifikaci, vizualizaci, konstrukci a dokumentaci softwarových systémů. UML zahrnuje různé typy diagramů, které poskytují různé pohledy na systém, a je využíván v různých fázích vývoje softwaru. UML lze rozdělit na statické a dynamické diagramy.

Typy diagramů UML

1. Statické diagramy:

- **Třídní diagram (Class Diagram):** Zobrazuje statickou strukturu systému pomocí tříd, jejich vlastností, metod a vztahů mezi třídami (dědičnost, asociace, agregace, kompozice).
- **Objektový diagram (Object Diagram):** Ukazuje instance tříd v konkrétním okamžiku a jejich vztahy, což pomáhá pochopit strukturu objektů v systému.
- **Komponentový diagram (Component Diagram):** Zobrazuje fyzickou architekturu systému a vztahy mezi komponentami (moduly, knihovny, služby).
- **Nasazovací diagram (Deployment Diagram):** Ukazuje fyzické nasazení softwarových komponent na hardware (servery, síťové uzly).
- **Balíčkový diagram (Package Diagram):** Zobrazuje rozdělení systému do balíčků a závislosti mezi nimi.

- **Profilový diagram (Profile Diagram):** Rozšiřuje UML pro konkrétní domény nebo platformy, umožňuje definovat stereotypy a značky.

2. Dynamické diagramy:

- **Sekvenční diagram (Sequence Diagram):** Zobrazuje interakce mezi objekty v časové posloupnosti, ukazuje zprávy posílané mezi objekty.
- **Komunikační diagram (Communication Diagram):** Ukazuje interakce mezi objekty zaměřením na strukturální organizaci, místo časové posloupnosti.
- **Diagram aktivit (Activity Diagram):** Modeluje tok činností nebo pracovní postupy v systému, podobný vývojovým diagramům.
- **Diagram stavů (State Diagram):** Ukazuje stavy objektu a přechody mezi těmito stavy, obvykle pro modelování chování objektů s komplexním životním cyklem.
- **Diagram případů užití (Use Case Diagram):** Zobrazuje funkční požadavky systému pomocí případů užití, aktérů a jejich interakcí.
- **Časový diagram (Timing Diagram):** Ukazuje změny stavů nebo podmínek objektů v závislosti na čase, často používaný v reálném čase a embedded systémech.
- **Interakční přehledový diagram (Interaction Overview Diagram):** Kombinuje prvky z aktivitních a interakčních diagramů, poskytuje vysokou úroveň přehledu nad interakcemi v systému.

Statický náhled na systém

Statické diagramy, jako jsou třídivé, objektové, komponentové a nasazovací diagramy, poskytují pohled na strukturu systému, jeho architekturu a rozložení komponent. Tyto diagramy jsou užitečné pro:

- **Návrh systému:** Pomáhají při definování tříd a jejich vztahů, při rozdělování systému do modulů a při plánování nasazení.
- **Dokumentace:** Poskytují jasnou a strukturovanou dokumentaci systému, kterou lze použít pro údržbu a další vývoj.
- **Komunikace:** Usnadňují komunikaci mezi členy týmu, zákazníky a dalšími zainteresovanými stranami.

Dynamický náhled na systém

Dynamické diagramy, jako jsou sekvenční, komunikační, aktivitní, stavové a časové diagramy, modelují chování systému, interakce mezi objekty a toky činností. Tyto diagramy jsou užitečné pro:

- **Analýzu požadavků:** Pomáhají pochopit, jak bude systém reagovat na různé vstupy a jak budou probíhat interakce mezi uživateli a systémem.
- **Návrh chování:** Pomáhají při definování, jak budou jednotlivé komponenty spolupracovat a jak budou zpracovávat různé scénáře.
- **Testování:** Umožňují modelovat testovací scénáře a ověřovat chování systému vůči požadavkům.

Mapování na zdrojový kód

UML diagramy mohou být mapovány na zdrojový kód následujícími způsoby:

- **Třídivé diagramy:** Přímé mapování na definice tříd, atributů a metod ve zdrojovém kódu.
- **Sekvenční a komunikační diagramy:** Můžou být použity k vytvoření skeletonu metod a jejich volání.
- **Diagramy aktivit a stavů:** Mohou být mapovány na implementaci pracovních postupů a stavových automatů v kódu.
- **Komponentové a nasazovací diagramy:** Pomáhají při organizaci kódu do modulů a při přípravě nasazovacích skriptů a konfigurace.

Využití UML v rámci vývoje

UML je široce využíváno v různých fázích vývoje softwaru:

- **Požadavky:** Diagramy případů užití pomáhají specifikovat a analyzovat požadavky.
- **Návrh:** Statické a dynamické diagramy jsou využívány pro návrh architektury a chování systému.
- **Implementace:** UML diagramy slouží jako základ pro psaní zdrojového kódu a organizaci kódové báze.
- **Testování:** Diagramy aktivit a sekvenční diagramy pomáhají při tvorbě testovacích scénářů.
- **Údržba a dokumentace:** UML poskytuje srozumitelnou a aktualizovanou dokumentaci, která usnadňuje údržbu a další vývoj softwaru.

UML tedy poskytuje všestranný nástroj pro modelování softwarových systémů, který podporuje celý životní cyklus vývoje softwaru od analýzy požadavků po údržbu a dokumentaci.

Životní cyklus vývoje software

- Životní cyklus vývoje software (typické fáze, jejich náplň, základní modely vývoje).

Životní cyklus vývoje softwaru zahrnuje několik fází, které představují strukturovaný přístup k vytváření softwarového produktu. Tyto fáze jsou navrženy tak, aby zajistily systematický a efektivní vývoj softwaru, minimalizovaly rizika a zajistily, že konečný produkt splňuje požadavky zákazníka. Typické fáze životního cyklu vývoje softwaru zahrnují:

1. Požadavky a analýza:

- **Náplň:** Shromažďování a analýza požadavků od zákazníků a dalších zainteresovaných stran. Tento proces zahrnuje pochopení potřeb a očekávání, definování funkcionalit a stanovení rozsahu projektu.
- **Výstupy:** Dokumenty specifikující požadavky (např. SRS - Software Requirements Specification), modely procesů a případně prototypy.

2. Návrh:

- **Náplň:** Vytvoření architektonického návrhu systému, který zahrnuje rozdělení systému na moduly, definování databázových struktur, návrh uživatelského rozhraní a určení technologií.
- **Výstupy:** Architektonické a detailní návrhy, UML diagramy, specifikace rozhraní a datové modely.

3. Implementace:

- **Náplň:** Kódování podle návrhových specifikací. Programátoři píší zdrojový kód, integrují různé moduly a testují jednotlivé komponenty.
- **Výstupy:** Zdrojový kód, binární soubory a průběžné verze softwaru.

4. Testování:

- **Náplň:** Verifikace a validace softwaru. Testování zahrnuje jednotkové testy, integrační testy, systémové testy a akceptační testy. Cílem je najít a opravit chyby, zajistit kvalitu a ověřit, že software splňuje požadavky.
- **Výstupy:** Testovací plány, testovací skripty, zprávy o chybách a výsledky testů.

5. Nasazení:

- **Náplň:** Instalace a konfigurace softwaru v produkčním prostředí. Zahrnuje také migraci dat a případné školení uživatelů.
- **Výstupy:** Instalovaný software, dokumentace k nasazení, uživatelské příručky.

6. Údržba:

- **Náplň:** Oprava chyb, aktualizace softwaru, přidávání nových funkcionalit a zajištění podpory pro uživatele. Údržba také zahrnuje monitorování výkonu systému a zálohování dat.
- **Výstupy:** Opravy chyb, aktualizace, servisní balíčky.

Základní modely vývoje softwaru

Existuje několik základních modelů vývoje softwaru, které se liší svým přístupem k organizaci a řízení jednotlivých fází životního cyklu:

1. Vodopádový model:

- Sekvenční model, kde každá fáze musí být dokončena před přechodem na další. Je vhodný pro projekty s dobře definovanými požadavky, které se během vývoje nemění.
- Výhody: Jednoduchost, snadná kontrola a sledování pokroku.
- Nevýhody: Nízká flexibilita, obtížné řešení změn v požadavcích.

2. Iterativní a inkrementální model:

- Vývoj probíhá v opakovaných cyklech (iteracích), kde každá iterace zahrnuje analýzu, návrh, implementaci a testování části systému. Každá iterace přidává inkrement nové funkčnosti.
- Výhody: Flexibilita, možnost průběžného zlepšování a rychlejší dodání funkčního softwaru.
- Nevýhody: Možnost vzniku technického dluhu, potřeba dobrého řízení iterací.

3. Agilní modely (např. Scrum, Kanban):

- Zaměřují se na flexibilitu, průběžnou zpětnou vazbu a časté dodávání funkčního softwaru. Týmy pracují v krátkých cyklech (sprintech) a pravidelně vyhodnocují a přizpůsobují svůj

postup.

- Výhody: Vysoká adaptabilita na změny, důraz na spolupráci a kvalitu.
- Nevýhody: Potřeba zkušeného týmu, možné problémy se škálováním na velké projekty.

4. Spirálový model:

- Kombinuje prvky vodopádového a iterativního modelu, klade důraz na analýzu rizik. Vývoj probíhá ve spirálách, kde každá spirála zahrnuje plánování, analýzu rizik, vývoj a hodnocení.
- Výhody: Řízení rizik, flexibilita a možnost postupného zlepšování.
- Nevýhody: Složitost, vyšší náklady na řízení.

5. V-model:

- Rozšíření vodopádového modelu, kde testovací fáze jsou naplánovány paralelně s odpovídajícími vývojovými fázemi. Každá fáze má své testovací aktivity.
- Výhody: Jasně propojení mezi vývojovými a testovacími fázemi, důraz na kvalitu.
- Nevýhody: Podobné jako u vodopádového modelu, nízká flexibilita.

Každý model má své výhody a nevýhody a je vhodný pro různé typy projektů v závislosti na jejich specifikách, požadavcích a rizicích.

Databázové systémy

Obsah

- [Datové modely](#)
- [Dotazovací jazyky](#)
- [Analýza informačního systému](#)
- [Transakce](#)
- [Vykonávání dotazů v databázových systémech](#)
- [Návrh a implementace datové vrstvy](#)

Databázové systémy (DS I, DS II)

- Datové modely (relační datový model: definice, normální formy, funkční závislosti; objektově-relační datový model: základní rysy).
- Dotazovací jazyky (SQL: jazyk pro definici dat, jazyk pro manipulaci s daty, SELECT – spojení, poddotazy, agregační funkce; procedurální rozšíření SQL: PL/SQL, T/SQL, obecné rysy: uložené procedury, kurzory, trigger).
- Analýza informačního systému (konceptuální a datový model, stavová analýza, funkční analýza – minispecifikace, návrh formulářů).

- Transakce (definice, ACID, serializovatelnost transakcí, zotavení, řízení souběhu, úroveň izolace transakcí).
- Vykonávání dotazů v databázových systémech (fyzický návrh databáze, vykonávání dotazů, logické a fyzické operace).
- Návrh a implementace datové vrstvy (objektově-relační mapování, DTO, DAO, efektivní implementace datové vrstvy).

🔗 Příklad otázky

Jaký SQL dotaz může v transakci vracet neočekávané výsledky, pokud použijeme úroveň izolace Read Committed? Popište i relaci se kterou pracujete a napište konkrétní dotaz nad touto relací.

Analýza informačního systému

- Analýza informačního systému (konceptuální a datový model, stavová analýza, funkční analýza – minispecifikace, návrh formulářů).

Konceptuální a datový model

Entity-Relationship Diagram (ERD):

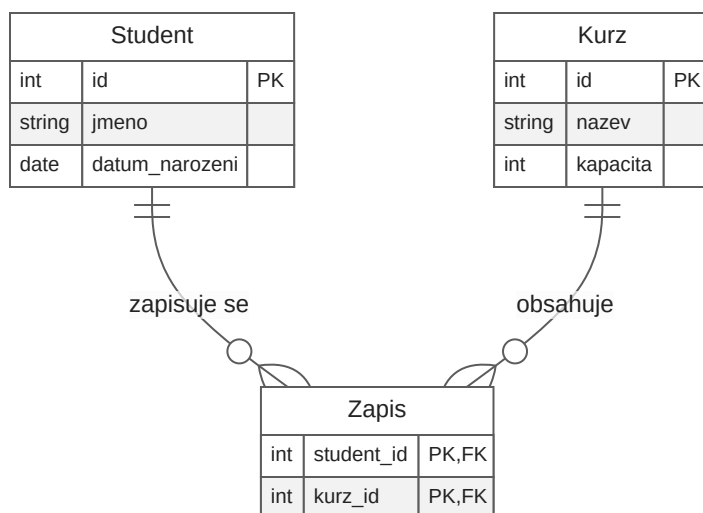


Schéma databáze:

```

CREATE TABLE Student (
  id INT PRIMARY KEY,
  jméno VARCHAR(50),
  datum_narození DATE

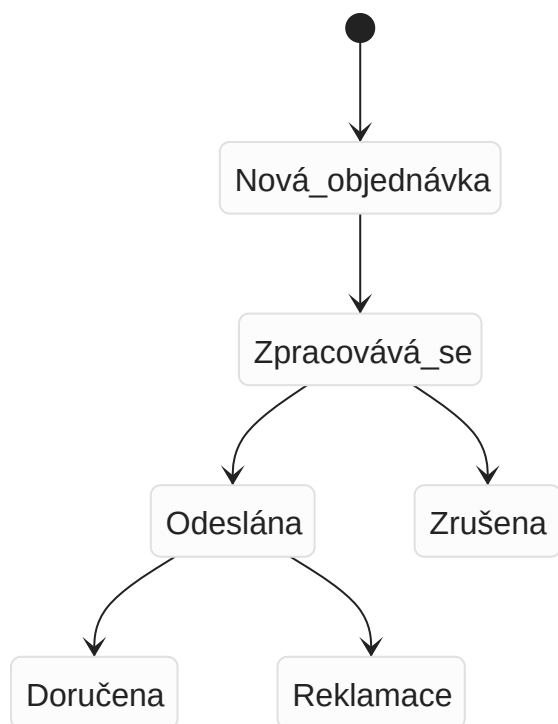
```

```
);
```

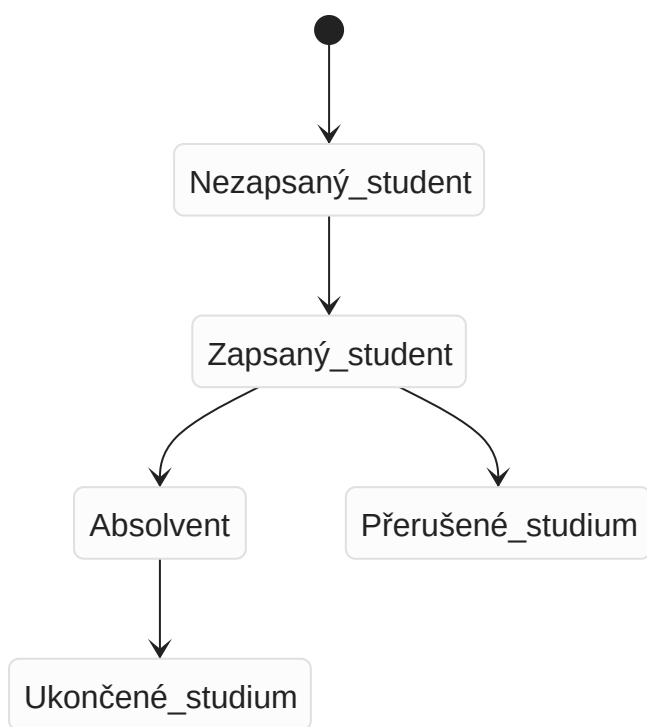
```
CREATE TABLE Kurz (  
    id INT PRIMARY KEY,  
    název VARCHAR(100),  
    kapacita INT  
);  
  
CREATE TABLE Zápis (  
    student_id INT,  
    kurz_id INT,  
    PRIMARY KEY (student_id, kurz_id),  
    FOREIGN KEY (student_id) REFERENCES Student(id),  
    FOREIGN KEY (kurz_id) REFERENCES Kurz(id)  
);
```

Stavová analýza

Stavový diagram pro proces objednávky:



Stavový diagram pro studentskou registraci:



Funkční analýza – minispecifikace

Minispecifikace: Vložení nového studenta

- **Vstupy:**
 - `jméno` : Řetězec (např. 'Jan Novák')
 - `datum_narození` : Datum (např. '2000-01-01')
- **Výstupy:**
 - Nový záznam v tabulce `Student`
- **Popis:**
 - Funkce vloží nového studenta do databáze. Nejprve ověří, že jméno a datum narození nejsou prázdné. Poté vygeneruje nové ID pro studenta a vloží záznam do tabulky `Student`.
- **SQL kód:**

```
CREATE OR REPLACE PROCEDURE VlozitStudenta (  
    p_jméno IN VARCHAR2,  
    p_datum_narození IN DATE  
) AS  
    v_id INT;  
BEGIN  
    -- Generování nového ID  
    SELECT NVL(MAX(id), 0) + 1 INTO v_id FROM Student;  
  
    -- Vložení nového studenta  
    INSERT INTO Student (id, jméno, datum_narození) VALUES (v_id, p_jméno,  
p_datum_narození);  
END;
```

Návrh formulářů

Formulář pro vložení nového studenta:

Mockup formuláře:

```
-----  
| Formulář pro vložení nového studenta |  
-----  
| Jméno: _____ |  
| Datum narození: _____ |  
| _____ |  
| [ Uložit ] |  
-----
```

HTML kód:

```
<form action="/vlozit_studenta" method="post">  
  <label for="jmeno">Jméno:</label>  
  <input type="text" id="jmeno" name="jmeno" required>  
  <br>  
  <label for="datum_narozeni">Datum narození:</label>  
  <input type="date" id="datum_narozeni" name="datum_narozeni" required>  
  <br>  
  <button type="submit">Uložit</button>  
</form>
```

Shrnutí

Analýza informačního systému zahrnuje:

- **Konceptuální a datový model:** Identifikace entit, atributů a vztahů, následně detailní definice tabulek a jejich struktury.
- **Stavová analýza:** Identifikace stavů systému a přechodů mezi nimi.
- **Funkční analýza – minispecifikace:** Definování funkcionality systému a detailní popisy jednotlivých funkcí.
- **Návrh formulářů:** Vytváření uživatelských rozhraní pro interakci se systémem.

Tyto kroky zajišťují, že informační systém bude splňovat požadavky uživatelů a bude dobře strukturovaný a snadno udržitelný.

Datové modely

- Datové modely (relační datový model: definice, normální formy, funkční závislosti; objektově-relační datový model: základní rysy).•

Relační datový model

Definice:

Relační datový model představuje data v podobě tabulek (relací), kde každý řádek tabulky (entita) reprezentuje jeden záznam a každý sloupec (atribut) reprezentuje typ dat. Relační model je založen na teorii množin a predikátové logice.

Relace:

Relace je tabulka, která obsahuje data ve formě dvourozměrné matice. Je tvořena řádky a sloupci, kde každý řádek reprezentuje jeden záznam a každý sloupec reprezentuje atribut záznamu.

- **Relační schéma:** Definuje strukturu relace, tedy jména sloupců a jejich datové typy. Např. pro relaci `Student` můžeme mít schéma `Student(id: INTEGER, jméno: VARCHAR, datum_narození: DATE)`.
- **Relační instance:** Konkrétní data uložená v relaci v daném okamžiku.

Vlastnosti relací:

1. **Jedinečnost řádků:** Každý řádek v relaci musí být jedinečný. To znamená, že nemůže být více řádků se stejnými hodnotami ve všech sloupcích.
2. **Neuspořádanost řádků a sloupců:** Pořadí řádků a sloupců v relaci nemá význam, což znamená, že přeskupení řádků nebo sloupců nemění logický význam relace.
3. **Atomické hodnoty:** Každá hodnota v relaci musí být atomická, což znamená, že nemůže být dále dělitelná.

Normální formy:

Normální formy jsou pravidla pro strukturování databázových tabulek tak, aby se minimalizovala redundance a eliminovaly anomálie při aktualizaci.

1. První normální forma (1NF):

- Každá tabulka má jedinečný název.
- Každý atribut má jedinečný název.
- Všechny hodnoty atributů jsou atomické (nedělitelné).

2. Druhá normální forma (2NF):

- Splňuje požadavky 1NF.
- Každý neklíčový atribut je plně závislý na primárním klíči (žádné parciální závislosti).

3. Třetí normální forma (3NF):

- Splňuje požadavky 2NF.
- Žádný neklíčový atribut není tranzitivně závislý na primárním klíči (všechny neklíčové atributy závisí přímo na primárním klíči).

4. Boyce-Coddova normální forma (BCNF):

- Splňuje požadavky 3NF.
- Každý determinant je kandidátní klíč.

Funkční závislosti:

Funkční závislost mezi dvěma množinami atributů (A) a (B) v tabulce znamená, že pro každý řádek, pokud dva řádky mají stejné hodnoty atributů v (A), musí mít stejné hodnoty atributů v (B). Formálně: $(A \rightarrow B)$.

- **Triviální funkční závislost:** Pokud (B) je podmnožinou (A), pak $(A \rightarrow B)$ je triviální funkční závislost.
- **Netriviální funkční závislost:** Pokud (B) není podmnožinou (A), pak $(A \rightarrow B)$ je netriviální funkční závislost.

Příklady funkčních závislostí:

- V tabulce `Student` s atributy `id`, `jméno`, `datum_narození`, funkční závislost `id -> jméno`, `datum_narození` znamená, že pro každý student s daným `id` existuje jedno `jméno` a jedno `datum_narození`.

Objektově-relační datový model

Základní rysy:

Objektově-relační datový model (ORDM) rozšiřuje relační model o prvky objektově orientovaného programování, což umožňuje lepší modelování složitých datových struktur a jejich vztahů.

1. Strukturované datové typy:

- Umožňují definovat složené typy s atributy a metodami.
- Např.: `CREATE TYPE student_type UNDER person_type (...);`
- **Příklad použití:** Vytvoření datového typu `Address` s atributy `street`, `city`, `zipcode` a metodami pro formátování adresy.

2. Dědičnost:

- Datové typy mohou být organizovány do hierarchií s podporou dědičnosti.
- Např.: `CREATE TYPE student_type UNDER person_type (...);`
- **Příklad použití:** Definování typu `UndergraduateStudent` a `GraduateStudent` jako podtypy typu `Student`.

3. Ukazatele a reference:

- Umožňují vytvářet vazby mezi tabulkami pomocí ukazatelů.
- Např.: `Manager REF person_type;`
- **Příklad použití:** Modelování vztahu mezi zaměstnanci a jejich nadřízenými pomocí `reference manager REF Employee`.

4. Kolekce dat:

- Umožňují ukládání kolekcí dat (asociativní pole, zahnížděné tabulky, pole).
- Např.: `TYPE jmeno IS TABLE OF element_type INDEX BY PLS_INTEGER | VARCHAR2(sizevarchar);`
- **Příklad použití:** Ukládání seznamu telefonních čísel pro každého zákazníka v databázi.

5. Procedury a trigger:

- Kód jako funkce, procedury a trigger je uložený v SŘBD a může být vykonáván na serveru.
- Např.: `CREATE TRIGGER ...`
- **Příklad použití:** Definování triggeru, který automaticky aktualizuje stav skladu při vložení nové objednávky.

Výhody objektově-relačního modelu:

- **Lepší modelování reálného světa:** Schopnost modelovat složité struktury a vztahy pomocí dědičnosti a složených typů.
- **Zvýšená flexibilita:** Možnost definovat vlastní datové typy a metody.
- **Výkonnostní výhody:** Eliminace potřeby přenášet velké objemy dat mezi klientem a serverem díky funkcím a triggerům běžícím na serveru.

Nevýhody objektově-relačního modelu:

- **Složitost:** Vyšší složitost návrhu a údržby databáze.
- **Kompatibilita:** Možné problémy s kompatibilitou mezi různými SŘBD implementacemi objektově-relačních rozšíření.

Tímto způsobem jsou objektově-relační databáze schopny poskytovat vyšší flexibilitu a lepší modelování reálných světů než tradiční relační databáze.

Dotazovací jazyky

- Dotazovací jazyky (SQL: jazyk pro definici dat, jazyk pro manipulaci s daty, SELECT – spojení, poddotazy, agregační funkce; procedurální rozšíření SQL: PL/SQL, T/SQL, obecné rysy: uložené procedury, kurzory, triggery).

SQL (Structured Query Language)

SQL je standardizovaný jazyk pro práci s relačními databázemi. SQL zahrnuje jazyk pro definici dat (DDL), jazyk pro manipulaci s daty (DML), jazyk pro řízení dat (DCL) a jazyk pro dotazování dat (SELECT).

Jazyk pro definici dat (DDL):

DDL se používá k definování a modifikaci struktury databází a jejich objektů.

- **CREATE:** Vytváří nové objekty v databázi (např. tabulky, indexy).

```
CREATE TABLE Student (  
    id INT PRIMARY KEY,  
    jmeno VARCHAR(50),
```



```
datum_narozeni DATE  
);
```

- **ALTER:** Modifikuje existující objekty v databázi.

```
ALTER TABLE Student ADD email VARCHAR(100);
```

- **DROP:** Odstraňuje objekty z databáze.

```
DROP TABLE Student;
```

Jazyk pro manipulaci s daty (DML):

DML se používá k manipulaci s daty v rámci existujících objektů databáze.

- **INSERT:** Vkládá nová data do tabulek.

```
INSERT INTO Student (id, jmeno, datum_narozeni) VALUES (1, 'Jan Novak', '1990-01-01');
```

- **UPDATE:** Aktualizuje existující data v tabulkách.

```
UPDATE Student SET email = 'jan.novak@example.com' WHERE id = 1;
```

- **DELETE:** Odstraňuje data z tabulek.

```
DELETE FROM Student WHERE id = 1;
```

SELECT – dotazovací jazyk:

SELECT se používá k dotazování dat z jedné nebo více tabulek.

- **Spojení (JOIN):**
Spojení tabulek na základě společného atributu.

```
SELECT s.jmeno, k.nazev_kurzu  
FROM Student s  
JOIN Kurz k ON s.id = k.student_id;
```

- **Poddotazy (Subqueries):**
Dotazy vnořené uvnitř jiných dotazů.

```
SELECT jmeno
FROM Student
WHERE id IN (SELECT student_id FROM Kurz WHERE nazev_kurzu = 'Database');
```

- **Agregační funkce:**

Funkce pro sumarizaci dat.

```
SELECT COUNT(*), AVG(vek), MAX(plat)
FROM Zamestnanec;
```

Procedurální rozšíření SQL

PL/SQL (Procedural Language/SQL):

PL/SQL je procedurální rozšíření SQL používané v Oracle databázích.

- **Uložené procedury:** Bloky kódu, které lze opakovaně volat.

```
CREATE OR REPLACE PROCEDURE ZvysPlat (
    p_id IN Zamestnanec.id%TYPE,
    p_castka IN NUMBER
) AS
BEGIN
    UPDATE Zamestnanec SET plat = plat + p_castka WHERE id = p_id;
END;
```

- **Kurzory:** Umožňují postupně procházet výsledky dotazu.

```
DECLARE
    CURSOR c1 IS SELECT jmeno FROM Student;
    v_jmeno Student.jmeno%TYPE;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO v_jmeno;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_jmeno);
    END LOOP;
    CLOSE c1;
END;
```

- **Triggery:** Automaticky spouštěné bloky kódu při určité akci.

```
CREATE OR REPLACE TRIGGER trg_zamestnanec
BEFORE INSERT ON Zamestnanec
```

```

FOR EACH ROW
BEGIN
    IF :NEW.plat < 0 THEN
        RAISE_APPLICATION_ERROR(-20001, 'Plat nemůže být záporný.');
```

T-SQL (Transact-SQL):

T-SQL je procedurální rozšíření SQL používané v Microsoft SQL Serveru.

- **Uložené procedury:**

```

CREATE PROCEDURE ZvysPlat
    @id INT,
    @castka DECIMAL
AS
BEGIN
    UPDATE Zamestnanec SET plat = plat + @castka WHERE id = @id;
END;
```

- **Kurzory:**

```

DECLARE @jmeno NVARCHAR(50);
DECLARE c1 CURSOR FOR SELECT jmeno FROM Student;
OPEN c1;
FETCH NEXT FROM c1 INTO @jmeno;
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT @jmeno;
    FETCH NEXT FROM c1 INTO @jmeno;
END;
CLOSE c1;
DEALLOCATE c1;
```

- **Triggery:**

```

CREATE TRIGGER trg_zamestnanec
ON Zamestnanec
AFTER INSERT
AS
BEGIN
    IF EXISTS (SELECT * FROM inserted WHERE plat < 0)
    BEGIN
        RAISERROR ('Plat nemůže být záporný.', 16, 1);
        ROLLBACK TRANSACTION;
```

```
END  
END;
```

Obecné rysy procedurálních rozšíření:

- **Uložené procedury:** Seskupení jednoho nebo více SQL příkazů do bloku, který může být uložen a volán podle potřeby.
- **Kurzory:** Mechanismy pro postupné procházení řádků výsledku dotazu.
- **Triggery:** Automatické spouštění akcí na základě změn v databázi, jako je INSERT, UPDATE nebo DELETE.

Rozdíl mezi PL/SQL a T-SQL

1. Syntaxe a funkcionality:

- **PL/SQL (Oracle)**
 - PL/SQL umožňuje použití operátorů %TYPE a %ROWTYPE pro deklaraci proměnných, které mají stejné datové typy jako sloupce tabulek.
 - PL/SQL podporuje konstrukci CREATE OR REPLACE , která umožňuje jednoduše aktualizovat existující procedury a triggery.
 - PL/SQL umožňuje zpracování výjimek s EXCEPTION blokem, který zachytává a zpracovává chyby.

```
DECLARE  
    v_jmeno Student.jmeno%TYPE;  
BEGIN  
    BEGIN  
        SELECT jmeno INTO v_jmeno FROM Student WHERE id = 1;  
    EXCEPTION  
        WHEN NO_DATA_FOUND THEN  
            DBMS_OUTPUT.PUT_LINE('Student nebyl nalezen.');
```

- **T-SQL (Microsoft SQL Server)**
 - T-SQL nemá operátory %TYPE a %ROWTYPE, místo toho používá explicitní deklaraci proměnných.
 - T-SQL podporuje konstrukci CREATE OR ALTER , která je podobná CREATE OR REPLACE v PL/SQL (dostupné od verze SQL Server 2016).
 - T-SQL má jednodušší mechanismus pro zpracování chyb pomocí TRY...CATCH bloků.

```
DECLARE @jmeno NVARCHAR(50);  
BEGIN  
    BEGIN TRY
```

```

        SELECT @jmeno = jmeno FROM Student WHERE id = 1;
    END TRY
    BEGIN CATCH
        PRINT 'Student nebyl nalezen.';
    END CATCH;
END;

```

2. Zpracování kurzorů:

• PL/SQL

- PL/SQL kurzory jsou definovány pomocí `DECLARE CURSOR` a mohou být otevřeny, načteny a zavřeny.
- PL/SQL umožňuje použití implicitních kurzorů pomocí `FOR` smyček.

```

DECLARE
    CURSOR c1 IS SELECT jmeno FROM Student;
    v_jmeno Student.jmeno%TYPE;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO v_jmeno;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_jmeno);
    END LOOP;
    CLOSE c1;
END;

```

• T-SQL

- T-SQL kurzory jsou definovány pomocí `DECLARE CURSOR FOR` a musí být explicitně otevřeny, načteny, uzavřeny a deallokovány.
- T-SQL kurzory vyžadují použití `FETCH NEXT` a `WHILE` smyček pro iteraci přes výsledky.

```

DECLARE @jmeno NVARCHAR(50);
DECLARE c1 CURSOR FOR SELECT jmeno FROM Student;
OPEN c1;
FETCH NEXT FROM c1 INTO @jmeno;
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT @jmeno;
    FETCH NEXT FROM c1 INTO @jmeno;
END;
CLOSE c1;
DEALLOCATE c1;

```

3. Triggery:

- **PL/SQL**

- PL/SQL triggery mohou být definovány pro různé události (např. `BEFORE INSERT`, `AFTER UPDATE`) a mohou používat speciální proměnné `:NEW` a `:OLD` pro přístup k novým a starým hodnotám.

```
CREATE OR REPLACE TRIGGER trg_zamestnanec
BEFORE INSERT ON Zamestnanec
FOR EACH ROW
BEGIN
    IF :NEW.plat < 0 THEN
        RAISE_APPLICATION_ERROR(-20001, 'Plat nemůže být záporný.');
```

- **T-SQL**

- T-SQL triggery používají vložené (`INSERTED`) a smazané (`DELETED`) tabulky pro přístup k novým a starým hodnotám.
- T-SQL triggery jsou definovány pomocí `CREATE TRIGGER` a mohou být spouštěny po (`AFTER`) nebo místo (`INSTEAD OF`) určité události.

```
CREATE TRIGGER trg_zamestnanec
ON Zamestnanec
AFTER INSERT
AS
BEGIN
    IF EXISTS (SELECT * FROM inserted WHERE plat < 0)
    BEGIN
        RAISERROR ('Plat nemůže být záporný.', 16, 1);
        ROLLBACK TRANSACTION;
    END
END;
```

4. Zpracování chyb:

- **PL/SQL** používá bloky `EXCEPTION` k zachycení a zpracování výjimek.
- **T-SQL** používá bloky `TRY...CATCH` k zachycení a zpracování výjimek.

5. Specifické funkce a rozšíření:

- **PL/SQL** má bohatou sadu vestavěných balíčků, jako je `DBMS_OUTPUT` pro výstup z PL/SQL bloků.
- **T-SQL** poskytuje specifické funkce pro práci s Windows Azure a dalšími Microsoft technologiemi.

Shrnutí

PL/SQL a T-SQL jsou mocná procedurální rozšíření SQL, každé s vlastními specifiky a rozšířeními pro konkrétní databázový systém. Znalost těchto rozdílů je klíčová pro efektivní práci s databázemi Oracle a Microsoft SQL Server.

Návrh a implementace datové vrstvy

- Návrh a implementace datové vrstvy (objektově-relační mapování, DTO, DAO, efektivní implementace datové vrstvy).

Návrh a implementace datové vrstvy

Návrh a implementace datové vrstvy je klíčovým krokem při vývoji softwarového systému, který zajišťuje správu a přístup k datům uloženým v databázi. Tento proces zahrnuje několik konceptů a technik, jako je objektově-relační mapování (ORM), Data Transfer Objects (DTO), Data Access Objects (DAO) a efektivní implementace datové vrstvy.

Objektově-relační mapování (ORM)

Objektově-relační mapování je technika, která spojuje objektově-orientovaný programovací model s relačním databázovým modelem. ORM nástroje umožňují vývojářům pracovat s databázovými daty jako s objekty v programovacím jazyce, což usnadňuje práci s daty a zlepšuje produktivitu.

Hlavní výhody ORM:

- **Abstrakce databázových operací:** Vývojáři pracují s objekty namísto přímého psaní SQL dotazů.
- **Automatické mapování:** ORM automaticky mapuje tabulky a sloupce na třídy a atributy.
- **Podpora různých databází:** ORM nástroje obvykle podporují různé databázové systémy, což umožňuje snadnou změnu databáze bez velkých úprav kódu.

Příklad ORM (Hibernate, JPA):

```
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String jmeno;
    private LocalDate datumNarozeni;

    // Gettery a settery
}
```

Data Transfer Objects (DTO)

DTO je vzor používaný k přenosu dat mezi vrstvami aplikace. DTO jsou jednoduché objekty, které nemají žádnou logiku, pouze obsahují data.

Výhody DTO:

- **Oddělení vrstev:** DTO oddělují prezentační a datovou vrstvu, což zlepšuje modularitu.
- **Optimalizace přenosu dat:** DTO mohou obsahovat pouze potřebná data, čímž se minimalizuje objem přenášených dat.

Příklad DTO:

```
public class StudentDTO {  
    private Long id;  
    private String jmeno;  
    private LocalDate datumNarozeni;  
  
    // Gettery a settery  
}
```

Data Access Objects (DAO)

DAO je vzor používaný k oddělení aplikační logiky od logiky přístupu k datům. DAO poskytují metody pro CRUD operace (Create, Read, Update, Delete) nad entitami.

Výhody DAO:

- **Oddělení logiky:** DAO odděluje přístup k datům od aplikační logiky.
- **Znovupoužitelnost:** Kód pro přístup k datům je centralizovaný a znovupoužitelný.
- **Snadná údržba:** Změny v databázové logice jsou omezeny na DAO třídy, což usnadňuje údržbu.

Příklad DAO:

```
public interface StudentDAO {  
    void save(Student student);  
    Student findById(Long id);  
    List<Student> findAll();  
    void update(Student student);  
    void delete(Student student);  
}  
  
public class StudentDAOImpl implements StudentDAO {  
    // Implementace CRUD operací pomocí ORM frameworku (např. Hibernate)  
}
```

Efektivní implementace datové vrstvy

Efektivní implementace datové vrstvy zahrnuje několik klíčových aspektů, které zajišťují výkon, škálovatelnost a udržitelnost aplikace.

1. Použití ORM frameworku:

- ORM frameworky, jako jsou Hibernate nebo JPA, zjednodušují práci s databází a zajišťují automatické mapování mezi objekty a databázovými tabulkami.

2. Optimalizace dotazů:

- Používání lazy loading pro odložení načítání souvisejících entit, dokud nejsou skutečně potřeba.
- Využití nativních SQL dotazů nebo pojmenovaných dotazů pro složité dotazy, které by byly neefektivní v HQL nebo JPQL.

3. Cache:

- Implementace cache na úrovni druhé úrovně (second-level cache) pro snížení počtu dotazů na databázi.
- Využití cache frameworků, jako je Ehcache nebo Hazelcast.

4. Transakční řízení:

- Použití transakčního řízení k zajištění atomických operací a zachování konzistence dat.
- Využití @Transactional anotace v Spring frameworku nebo ekvivalentních mechanismů v jiných frameworkech.

5. Jednotkové testy a integrační testy:

- Vytváření jednotkových testů pro DAO vrstvy pomocí testovacích rámců jako JUnit nebo TestNG.
- Použití nástrojů jako H2 databáze pro integrační testy bez potřeby přístupu k produkční databázi.

Příklad DAO implementace s použitím Spring a JPA:

```
@Repository
public class StudentDAOImpl implements StudentDAO {

    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    @Override
    public void save(Student student) {
        entityManager.persist(student);
    }

    @Override
    public Student findById(Long id) {
        return entityManager.find(Student.class, id);
    }

    @Override
```

```

public List<Student> findAll() {
    String query = "SELECT s FROM Student s";
    return entityManager.createQuery(query, Student.class).getResultList();
}

@Transactional
@Override
public void update(Student student) {
    entityManager.merge(student);
}

@Transactional
@Override
public void delete(Student student) {
    if (entityManager.contains(student)) {
        entityManager.remove(student);
    } else {
        entityManager.remove(entityManager.merge(student));
    }
}
}

```

Shrnutí

Návrh a implementace datové vrstvy zahrnuje použití několika designových vzorů a technik, jako je ORM pro zjednodušení práce s databází, DTO pro efektivní přenos dat mezi vrstvami, DAO pro oddělení přístupu k datům od aplikační logiky a různé optimalizační techniky pro zajištění výkonu a škálovatelnosti aplikace. Tyto techniky společně zajišťují, že datová vrstva je dobře strukturovaná, udržitelná a výkonná.

Transakce

- Transakce (definice, ACID, serializovatelnost transakcí, zotavení, řízení souběhu, úroveň izolace transakcí).

Definice

Transakce je základní jednotka zpracování, která provádí sekvenci operací na databázi jako jediný logický celek. Transakce začíná příkazem `BEGIN TRANSACTION` a končí příkazem `COMMIT` nebo `ROLLBACK`.

- **BEGIN TRANSACTION:** Zahájení transakce.
- **COMMIT:** Potvrzení a trvalé uložení všech změn provedených transakcí.
- **ROLLBACK:** Zrušení všech změn provedených transakcí a návrat databáze do stavu před začátkem transakce.

ACID

ACID je akronym pro čtyři základní vlastnosti transakcí, které zaručují spolehlivost zpracování dat v databázových systémech.

1. Atomicity (Atomická):

- Zajišťuje, že všechny operace v rámci transakce jsou provedeny buď všechny, nebo žádná. Pokud dojde k chybě, všechny změny jsou vráceny zpět.
- Příklad: Při převodu peněz mezi dvěma účty musí být částka odečtena z jednoho účtu a přičtena k druhému účtu jako jediný atomický celek.

2. Consistency (Konzistence):

- Zaručuje, že transakce převede databázi z jednoho konzistentního stavu do druhého konzistentního stavu. Všechny definované pravidla a omezení musí být dodrženy.
- Příklad: Při vložení záznamu do databáze musí být dodrženy všechny integritní omezení, jako jsou primární a cizí klíče.

3. Isolation (Izolace):

- Zajišťuje, že současně prováděné transakce se vzájemně neovlivňují. Změny provedené jednou transakcí nejsou viditelné pro jiné transakce, dokud nejsou potvrzeny.
- Příklad: Pokud dvě transakce současně aktualizují stejné záznamy, každá transakce by měla vidět pouze své vlastní změny, dokud nejsou obě potvrzeny.

4. Durability (Trvalost):

- Zaručuje, že jakmile je transakce potvrzena, její změny jsou trvalé a přetrvávají i v případě výpadku systému.
- Příklad: Po potvrzení transakce zůstávají všechny provedené změny v databázi uloženy i po restartu databázového systému.

Serializovatelnost transakcí

Serializovatelnost je vlastnost plánů transakcí, která zajišťuje, že výsledek provádění transakcí je ekvivalentní jejich nějakému sériovému provedení, i když jsou prováděny současně.

- **Sériový plán:** Transakce jsou provedeny jedna po druhé, bez překrývání.
 - Příklad: Transakce A a B jsou provedeny v pořadí A, pak B.
- **Serializovatelný plán:** Plán provádění transakcí, který je ekvivalentní nějakému sériovému plánu.
 - Příklad: Transakce A a B jsou provedeny v takovém pořadí, že jejich kombinovaný efekt je stejný jako u sériového provedení.

Zotavení

Zotavení databáze je proces, který zajišťuje návrat databáze do konzistentního stavu po selhání, ať už jde o selhání systému, chyby aplikací nebo jiné problémy.

- **UNDO:** Vracení zpět změn provedených neúspěšnou transakcí.

- **REDO:** Opětovné provedení změn provedených úspěšnou transakcí, které nebyly trvale zapsány do databáze.

Techniky zotavení:

1. Odložená aktualizace (Deferred Update):

- Změny provedené transakcemi jsou nejprve zaznamenány do logu a do databáze se zapíší až po úspěšném potvrzení transakce.
- Pouze operace REDO jsou nutné při zotavení, protože neúspěšné transakce neprováděly žádné změny v databázi.

2. Okamžitá aktualizace (Immediate Update):

- Změny jsou zaznamenány do logu a ihned aplikovány na databázi během provádění transakce.
- Při zotavení jsou nutné operace UNDO pro neúspěšné transakce a operace REDO pro potvrzené transakce, které nebyly trvale zapsány.

Řízení souběhu

Řízení souběhu je proces, který zajišťuje správnou izolaci a správnost transakcí prováděných současně. Existuje několik technik řízení souběhu:

1. Zamykání (Locking):

- Zajišťuje exkluzivní přístup k datům pomocí zámků. Sdílené zámky (S) umožňují čtení, výlučné zámky (X) umožňují zápis.
- **Dvoufázový zamykací protokol (Two-Phase Locking, 2PL):** Transakce získává zámky během fáze růstu a uvolňuje je během fáze zkracování.
 - **Striktní 2PL (Strict 2PL):** Zámky jsou uvolněny až po potvrzení nebo zrušení transakce.

2. Optimistické řízení souběhu (Optimistic Concurrency Control):

- Předpokládá, že konflikty mezi transakcemi jsou vzácné. Transakce pracují bez zámků a konflikty se kontrolují před potvrzením.
- Pokud se zjistí konflikt, transakce je zrušena a musí být opakována.

3. Časová razítka (Timestamp Ordering):

- Každé transakci je přiřazeno jedinečné časové razítko. Operace transakcí jsou prováděny v pořadí podle jejich časových razítek.
- Konflikty jsou řešeny na základě časových razítek, starší transakce mají přednost.

Úroveň izolace transakcí

Úroveň izolace transakcí určuje, jak jsou transakce izolovány jedna od druhé a jaké typy anomálií mohou nastat. SQL standard definuje čtyři úrovně izolace:

1. Read Uncommitted (Nezávazně přečteno):

- Transakce mohou číst data, která byla změněna jinými transakcemi, ale ještě nebyla potvrzena.
- Může nastat: Špinavé čtení (dirty read), Neopakovatelné čtení (non-repeatable read), Fantomové čtení (phantom read).

2. Read Committed (Závazně přečteno):

- Transakce mohou číst pouze data, která byla potvrzena jinými transakcemi.
- Může nastat: Neopakovatelné čtení, Fantomové čtení.
- Nemůže nastat: Špinavé čtení.

3. Repeatable Read (Opakovatelné čtení):

- Zajišťuje, že pokud transakce čte stejná data vícekrát, vždy dostane stejné hodnoty. Žádné jiné transakce nemohou měnit čtená data, dokud není tato transakce potvrzena.
- Může nastat: Fantomové čtení.
- Nemůže nastat: Špinavé čtení, Neopakovatelné čtení.

4. Serializable (Serializovatelné):

- Nejvyšší úroveň izolace, která zajišťuje úplnou izolaci transakcí. Transakce jsou prováděny tak, jako by byly prováděny sériově.
- Nemůže nastat: Špinavé čtení, Neopakovatelné čtení, Fantomové čtení.

Shrnutí

Transakce jsou základními stavebními kameny databázových systémů, které zajišťují integritu a spolehlivost zpracování dat. Dodržování vlastností ACID, správná úroveň izolace, efektivní řízení souběhu a robustní mechanismy zotavení jsou klíčové pro dosažení spolehlivého a konzistentního chování databázových systémů.

Vykonávání dotazů v databázových systémech

- Vykonávání dotazů v databázových systémech (fyzický návrh databáze, vykonávání dotazů, logické a fyzické operace).

Vykonávání dotazů v databázových systémech

Vykonávání dotazů v databázových systémech je komplexní proces, který zahrnuje několik klíčových komponent, jako je fyzický návrh databáze, logické a fyzické operace a optimalizace dotazů. Tyto komponenty společně zajišťují, že dotazy jsou vykonávány efektivně a rychle.

Fyzický návrh databáze

Fyzický návrh databáze se zaměřuje na optimální uspořádání dat na úložném médiu. Tento návrh zahrnuje výběr vhodných datových struktur, indexů a strategií ukládání, které zlepšují výkon dotazů.

1. Tabulky typu Halda (Heap Tables):

- Záznamy jsou ukládány v pořadí, v jakém jsou vkládány, bez specifického uspořádání.
- Výhody: Rychlé vkládání dat ($O(1)$).
- Nevýhody: Pomalejší vyhledávání ($O(n)$).

2. Indexované tabulky:

- Indexy umožňují rychlé vyhledávání záznamů na základě hodnot v jedné nebo více sloupcích.
- Typy indexů:
 - **Jednoduché indexy:** Indexy na jednom sloupci.
 - **Složené indexy:** Indexy na více sloupcích.
- Indexy jsou často implementovány pomocí stromových struktur, jako je B-strom nebo B+-strom.
- Výhody: Rychlejší vyhledávání ($O(\log n)$).
- Nevýhody: Vyšší režie při vkládání, aktualizaci a mazání dat.

3. Clustering:

- Záznamy jsou fyzicky uspořádány na disku podle jednoho nebo více sloupců.
- Výhody: Rychlejší dotazy, které potřebují přistupovat k rozsahu hodnot.
- Nevýhody: Vyšší režie při vkládání a aktualizaci dat.

4. Particionování:

- Tabulky jsou rozděleny na menší části (partice), které mohou být uloženy a spravovány samostatně.
- Výhody: Lepší výkon při práci s velmi velkými tabulkami.
- Nevýhody: Komplexnější správa databáze.

Vykonávání dotazů

Vykonávání dotazů zahrnuje několik kroků, které zahrnují překládání SQL dotazů do interní reprezentace, optimalizaci dotazů a vykonání dotazů pomocí logických a fyzických operací.

1. Překlad dotazu:

- SQL dotaz je přeložen do interní reprezentace, často ve formě dotazovacího stromu nebo grafu, který reprezentuje jednotlivé operace dotazu.

2. Optimalizace dotazu:

- Optimalizátor dotazu přetváří dotaz na ekvivalentní, ale efektivnější verzi.
- Transformace zahrnují například přesun filtrů, použití indexů a eliminaci nepotřebných operací.

3. Generování plánu dotazu:

- Optimalizátor vytváří množinu možných plánů vykonání dotazu a vybírá nejlevnější plán na základě odhadovaných nákladů (např. I/O operace, CPU čas).

Příklad: Plán dotazu

```
SELECT * FROM Student WHERE jméno = 'Jan Novák';
```

1. Překlad dotazu:

- Dotaz je přeložen do interního dotazovacího stromu.

2. Optimalizace dotazu:

- Optimalizátor zjistí, zda existuje index na sloupci `jméno`, který může být použit pro rychlejší vyhledávání.

3. Generování plánu dotazu:

- Možné plány:
 - Full Table Scan: Procházení celé tabulky `Student`.
 - Index Scan: Použití indexu na sloupci `jméno` (pokud existuje).
- Výběr plánu: Index Scan, pokud existuje vhodný index, jinak Full Table Scan.

Logické a fyzické operace

Logické operace:

- Logické operace jsou vysokou úrovní operací, které jsou nezávislé na fyzické implementaci dat. Příklady zahrnují selekci, projekci, joiny, agregace atd.

1. Selekt (Selection):

- Výběr řádků, které splňují danou podmínku.
- $\sigma_{\{podmínka\}}(Tabulka)$

2. Projekce (Projection):

- Výběr specifických sloupců z tabulky.
- $\pi_{\{sloupce\}}(Tabulka)$

3. Spojení (Join):

- Kombinace řádků ze dvou tabulek na základě společného atributu.
- $Tabulka1 \bowtie_{\{podmínka\}} Tabulka2$

4. Agregace (Aggregation):

- Výpočet agregačních funkcí, jako je SUM, COUNT, AVG.
- $\gamma_{\{agregace\}}(Tabulka)$

Fyzické operace:

- Fyzické operace jsou implementace logických operací na úrovni úložiště. Příklady zahrnují sekvenční scan, index scan, nested loop join, sort merge join, hash join atd.

1. Sekvenční scan (Sequential Scan):

- Prochází všechny záznamy v tabulce.

2. Index scan (Index Scan):

- Používá index k nalezení specifických záznamů.

3. **Nested Loop Join:**

- Provádí join pro každou dvojici záznamů z obou tabulek.

4. **Sort Merge Join:**

- Seřadí obě tabulky podle společného atributu a poté je spojí.

5. **Hash Join:**

- Používá hash tabulky k urychlení spojení.

Shrnutí

Vykonávání dotazů v databázových systémech zahrnuje fyzický návrh databáze, překlad a optimalizaci dotazů a provádění logických a fyzických operací. Tento proces zajišťuje efektivní a rychlé zpracování dotazů, minimalizaci nákladů na I/O a CPU a optimální využití dostupných zdrojů.