NÁVRHOVÉ VZORY

Přednáška 4 – Antipatterns II

II. METODOLOGICKÉ ANTI-PATTERNS

- Metodologické antipatterny jsou běžné úskalí nebo problémy, které mohou nastat při navrhování a implementaci projektu či řešení.
- **Premature Optimization**
  - Předčasná optimalizace je antipattern, kdy optimalizujeme řešení pro výkon ještě před tím, než víme, zda tuto optimalizaci skutečně potřebujeme.
  - Takové optimalizace mají jen náklady a žádné výhody.
  - Mohou vést k několika problémům, včetně zvýšené složitosti, snížené čitelnosti a horší udržovatelnosti.
  - Optimalizovat řešení pro výkon bychom měli jen tehdy, pokud je to skutečně potřeba.
- **Reinventing the Wheel**
  - Antipattern *reinventing the wheel* nastává, když je řešení problému zbytečně znovu vynalézáno místo využití existujícího řešení nebo rozšíření již existující práce.
  - To vede ke zvýšení času vývoje a nákladů.
  - Nové řešení navíc nemusí být tak dobře otestované jako existující řešení.
- **Copy and Paste Programming**
  - Antipattern *copy and paste programming* nastává, když je zdrojový kód kopírován z jednoho místa na jiné místo místo toho, aby byl znovu použit prostřednictvím abstrakce.
  - To vede k vyšším nákladům na údržbu, snížené čitelnosti kódu a menší znovupoužitelnosti.
  - Místo kopírování kódu ho můžeme **refaktorovat a znovu použít**.
  - Například lze kód vložit do pomocné (*utility*) třídy, ke které mají přístup různé části programu.

PREMATURE OPTIMIZATION - PŘÍKLAD

```java
public class StringConcatenator {

    public String concatenate(String[] words) {
        String result = "";
        for (int i = 0; i < words.length; i++) {
            // Předčasná optimalizace: snažíme se ušetřit paměť tím, že nepo
            result += words[i]; // Vytváří se nový objekt při každé iteraci
        }
        return result;
    }
}
```

- Používání operátoru + v cyklu s String vede k neefektivnímu vytváření nových objektů.
- Vývojář se snažil „optimalizovat“ tím, že nepoužil další třídu jako StringBuilder, ale výsledek je horší výkon.
- Navíc, pokud by se ukázalo, že metoda se volá zřídka nebo s malým polem, optimalizace by byla zbytečná.

ŘEŠENÍ

```java
public class StringConcatenator {

    public String concatenate(String[] words) {
        StringBuilder result = new StringBuilder();
        for (String word : words) {
            result.append(word);
        }
        return result.toString();
    }
}
```

- Použití StringBuilder je standardní a efektivní způsob spojování řetězců.
- Kód je čitelnější, jednodušší na údržbu a výkonnější.
- Optimalizace je provedena až poté, co je jasné, že výkon je důležitý (např. na základě profilování)

- **Neoptimalizujme dřív, než víme, že je to potřeba.**
- **Nejprve pišme čistý, srozumitelný kód.**
- **Optimalizujme na základě dat – např. pomocí profilování nebo benchmarků**

REINVENTING THE WHEEL - PŘÍKLAD

- Vývojář vytváří vlastní řešení pro problém, který už má kvalitní, ověřené řešení – často ve formě knihovny nebo frameworku.
- Místo toho, aby použil existující nástroj, „vynalézá kolo znovu“ – což vede ke zbytečné práci, chybám a horší údržbě.

```
public class CustomJsonParser {

    public Map<String, String> parse(String json) {
        Map<String, String> result = new HashMap<>();
        json = json.replace("{", "").replace("}", "");
        String[] pairs = json.split(",");
        for (String pair : pairs) {
            String[] keyValue = pair.split(":");
            result.put(keyValue[0].trim(), keyValue[1].trim());
        }
        return result;
    }
}
```

```java
import com.google.gson.Gson;
import java.util.Map;

public class JsonParserUsingGson {

    public Map<String, String> parse(String json) {
        Gson gson = new Gson();
        return gson.fromJson(json, Map.class);
    }
}
```

ŘEŠENÍ

- Kód je kratší, čitelnější a spolehlivější.
- Gson řeší složité případy, typovou bezpečnost, validaci a je dobře testovaný.
- Vývojář se může soustředit na logiku aplikace, ne na znovuvynalézání základních nástrojů.

- **„Nepišme vlastní knihovnu, pokud ji už někdo napsal lépe.“**

- Vždy se nejprve podívejme, zda už neexistuje kvalitní řešení.
- Používejme knihovny, které jsou aktivně udržované, dokumentované a komunitou ověřené.
- Vlastní implementace má smysl jen tehdy, když máme specifické požadavky, které žádná knihovna nesplňuje.

```java
public class OrderProcessor {
    public void processOrder(Order order) {
        double totalPrice = order.getItemPrice() * order.getQuantity();
        totalPrice += totalPrice * 0.1; // Přidání daně
        System.out.println("Order ID: " + order.getId());
        System.out.println("Total Price: " + totalPrice);
    }
}

public class InvoiceGenerator {
    public void generateInvoice(Order order) {
        double totalPrice = order.getItemPrice() * order.getQuantity();
        totalPrice += totalPrice * 0.1; // Přidání daně
        System.out.println("Invoice ID: " + order.getId());
        System.out.println("Total Price: " + totalPrice);
    }
}
```

COPY AND PASTE PROGRAMMING

- Stejný výpočet ceny a výpis se opakuje.
- Pokud se změní logika výpočtu (např. sazba daně), musí se upravit na více místech.
- Vede to k chybám, nejasnostem a vyšší náročnosti na údržbu

ŘEŠENÍ

```java
public class OrderUtils {

    public static double calculateTotalPrice(Order order) {
        double totalPrice = order.getItemPrice() * order.getQuantity();
        return totalPrice + totalPrice * 0.1;
    }

    public static void printOrderDetails(String label, Order order) {
        double totalPrice = calculateTotalPrice(order);
        System.out.println(label + " ID: " + order.getId());
        System.out.println("Total Price: " + totalPrice);
    }
}

public class OrderProcessor {
    public void processOrder(Order order) {
        OrderUtils.printOrderDetails("Order", order);
    }
}

public class InvoiceGenerator {
    public void generateInvoice(Order order) {
        OrderUtils.printOrderDetails("Invoice", order);
    }
}
```

- Kód je modulární, přehledný a snadno udržovatelný.
- Změna výpočtu se provede na jednom místě.
- Dodržuje princip DRY – Don't Repeat Yourself.
- „Když kopírujeme kód, kopírujeme i chyby.“
- Vždy se zamysleme, zda by se kód neměl extrahovat do metody nebo třídy.
- IDE jako IntelliJ nebo Eclipse mají funkce jako „Extract Method“, které nám s tím pomůžou.
- Používejme utility třídy, design patterns (např. Strategy, Template Method) nebo funkcionální přístup, pokud se hodí.

III. SOFTWARE TESTING ANTI-PATTERNS

- Antipatterny softwarového testování jsou běžné chyby nebo špatné postupy ve fázi testování softwaru. Tyto antipatterny mohou vést ke sníženému pokrytí testy, zvýšeným nákladům na údržbu a nižší spolehlivosti.
- **Wrong Kind of Test**
  - Když používáme nesprávný typ testu, neúmyslně aplikujeme antipattern *wrong kind of test* na naši kódovou bázi.
  - Je zásadní porozumět různým typům testů a vědět, kdy který použít.
  - Použití nesprávného typu testu může vést ke sníženému pokrytí, vyšším nákladům na údržbu a nižší spolehlivosti.
- **Testing Internal Implementation**
  - Antipattern *testing internal implementation* nastává, když jsou testy napsány tak, že jsou vázány na interní implementaci softwarové komponenty místo na její externí chování.
  - To zvyšuje náklady na údržbu, protože testy je nutné aktualizovat při každé změně interní implementace.
- **Happy Path**
  - Když se při testování kódu zaměřujeme pouze na nejběžnější a očekávané případy, aplikujeme antipattern *happy path*.
  - Je důležité testovat i neočekávané případy a scénáře, jinak je vysoké riziko chyb a nepředvídatelného chování.

WRONG KIND OF TEST - PŘÍKLAD

```java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

```java
public class CalculatorTest {

    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();

        // Špatný test: kontroluje vnitřní stav nebo konkrétní implementaci
        int result = calculator.add(2, 3);
        assertEquals(5, result); // OK, ale ...

        // Zbytečný test: kontroluje, že metoda používá operátor +
        // (např. pomocí reflection nebo mockování, což je zbytečné pro takto
    }
}
```

- Test je příliš jednoduchý, ale v praxi se často stává, že testy kontrolují konkrétní implementaci místo chování.
- Pokud by se metoda změnila (např. použila jiný způsob výpočtu), test by selhal, i když výsledek by byl správný.
- Testy by měly být odolné vůči refaktoringu.

```java
public class CalculatorTest {

    @Test
    public void shouldReturnSumOfTwoNumbers() {
        Calculator calculator = new Calculator();
        assertEquals(5, calculator.add(2, 3));
        assertEquals(0, calculator.add(-2, 2));
        assertEquals(-5, calculator.add(-2, -3));
    }
}
```

ŘEŠENÍ

- Testuje se chování metody – různé vstupy a očekávané výstupy.
- Test je jednoduchý, čitelný a nezávislý na konkrétní implementaci.
- Pokud se změní způsob výpočtu, ale výsledek zůstane stejný, test stále projde

- „Dobrý test je jako bezpečnostní pás – chrání tě, ale neomezuje v pohybu.“

- Piš testy, které ověřují chování, ne konkrétní implementaci.
- Testy by měly být jednoduché, izolované a smysluplné.
- Refaktoring by neměl nutit k přepisování testů, pokud se nezměnilo chování.

WRONG KIND OF TEST – PŘÍKLAD 2

```java
// Třída, kterou testujeme
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

- Tento test je křehký — testuje implementační detaily, ne funkční chování.
- Pokud by se metoda interně změnila (např. používala long a přetypovala výsledek), test by zbytečně selhal, i když výsledek by byl správný.

```java
// Test – špatný typ testu
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalculatorTest {
    @Test
    public void testInternalImplementation() {
        Calculator calc = new Calculator();

        // Předpokládáme, že metoda add používá konkrétní interní proměnnou "sum"
        // (což vůbec není garantováno rozhraním)
        int a = 3;
        int b = 5;
        int expected = 8;

        int result = calc.add(a, b);
        assertEquals(expected, result);

        // ❌ Navíc testujeme implementační detail (např. že metoda vrací přesně int)
        assertTrue(((Object)result) instanceof Integer); // zbytečné
    }
}
```

ŘEŠENÍ 2

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalculatorTest {
    @Test
    public void testAddReturnsCorrectSum() {
        Calculator calc = new Calculator();

        assertEquals(8, calc.add(3, 5));
        assertEquals(0, calc.add(-3, 3));
        assertEquals(-10, calc.add(-7, -3));
    }
}
```

- Tento test kontroluje chování, nikoli způsob, jakým je metoda implementována.
- Pokud se v budoucnu implementace změní (např. použije BigDecimal, nebo jinou interní logiku), test stále zůstane platný.

WRONG KIND OF TEST – PŘÍKLAD 3

```java
// REST controller
@RestController
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/{id}")
    public ResponseEntity<User> getUser(@PathVariable Long id) {
        return ResponseEntity.ok(userService.findUserById(id));
    }
}
```

```java
// Špatný test
@SpringBootTest
@AutoConfigureMockMvc
class UserControllerWrongTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void testInternalServiceIsCalled() throws Exception {
        UserController controller = new UserController();
        UserService mockService = Mockito.mock(UserService.class);
        controller.userService = mockService;

        controller.getUser(1L);

        // ❌ Testujeme interní implementaci – že metoda "findUserById" byla zavolána
        // místo aby nás zajímal výsledek API
        Mockito.verify(mockService).findUserById(1L);
    }
}
```

ŘEŠENÍ 3

- testuje chování API z pohledu klienta,nezajímá se o to, jak je služba implementována
- bude stále platný i při vnitřní změně (např. jiná služba, jiná databáze).

```
@SpringBootTest
@AutoConfigureMockMvc
class UserControllerGoodTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void testGetUserReturnsUserData() throws Exception {
        mockMvc.perform(get("/users/1"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.id").value(1))
                .andExpect(jsonPath("$.name").value("Alice"));
    }
}
```

TESTING INTERNAL IMPLEMENTATION - PŘÍKLAD

```java
public class UserService {
    private int retryCount = 3;

    public boolean login(String username, String password) {
        // nějaká logika
        return true;
    }
}
```

```java
public class UserServiceTest {

    @Test
    public void testRetryCountInternalState() throws Exception {
        UserService service = new UserService();

        // Špatný test: přístup k privátnímu poli pomocí reflexe
        Field field = UserService.class.getDeclaredField("retryCount");
        field.setAccessible(true);
        int value = (int) field.get(service);

        assertEquals(3, value); // Testuje vnitřní stav, ne chování
    }
}
```

- Test porušuje zapouzdření (encapsulation).
- Změna názvu pole nebo jeho typu rozbije test, i když login stále funguje.
- Test neověřuje, zda login funguje správně – jen že existuje konkrétní implementace.

```java
public class UserService {

    private int retryCount = 3;

    public boolean login(String username, String password) {
        for (int i = 0; i < retryCount; i++) {
            if (authenticate(username, password)) {
                return true;
            }
        }
        return false;
    }

    private boolean authenticate(String username, String password) {
        return "admin".equals(username) && "secret".equals(password);
    }
}
```

```java
public class UserServiceTest {

    @Test
    public void shouldLoginWithCorrectCredentials() {
        UserService service = new UserService();
        assertTrue(service.login("admin", "secret"));
    }

    @Test
    public void shouldFailLoginWithWrongCredentials() {
        UserService service = new UserService();
        assertFalse(service.login("user", "wrong"));
    }
}
```

ŘEŠENÍ

- Testy ověřují funkční chování, ne detaily implementace.
- Pokud se změní počet pokusů nebo způsob autentizace, testy stále fungují, pokud chování zůstane stejné.
- Kód je odolný vůči refaktoringu.

- „Test by měl být jako zákazník – zajímá ho výsledek, ne jak jsi ho dosáhl.“

- Testujme co třída dělá, ne jak to dělá.
- Vyhýbejme se reflexi, přístupu k privátním polím nebo metodám.
- Pokud potřebujeme testovat vnitřní logiku, zvažme refaktoring do samostatné třídy nebo metody s veřejným rozhraním.

HAPPY PATH - PŘÍKLAD

```java
public class PaymentService {
    public String processPayment(String cardNumber, double amount) {
        // Předpokládáme, že karta je vždy platná a částka je vždy kladná
        return "Payment of $" + amount + " processed for card " + cardNumber;
    }
}
```

```java
public class PaymentServiceTest {

    @Test
    public void testSuccessfulPayment() {
        PaymentService service = new PaymentService();
        String result = service.processPayment("4111111111111111", 100.0);
        assertEquals("Payment of $100.0 processed for card 4111111111111111",
    }
}
```

- Test pokrývá pouze „šťastnou cestu“ – validní vstup, žádné chyby.
- Co když je částka záporná? Co když je číslo karty neplatné nebo null?
- Kód neobsahuje žádné ošetření výjimek nebo validaci vstupů.

```java
public class PaymentService {
    public String processPayment(String cardNumber, double amount) {
        if (cardNumber == null || cardNumber.isEmpty()) {
            throw new IllegalArgumentException("Card number is invalid");
        }
        if (amount ≤ 0) {
            throw new IllegalArgumentException("Amount must be positive");
        }
        return "Payment of $" + amount + " processed for card " + cardNumber;
    }
}
```

```java
public class PaymentServiceTest {

    @Test
    public void testSuccessfulPayment() {
        PaymentService service = new PaymentService();
        String result = service.processPayment("4111111111111111", 100.0);
        assertEquals("Payment of $100.0 processed for card 4111111111111111",
    }

    @Test
    public void testInvalidCardNumber() {
        PaymentService service = new PaymentService();
        assertThrows(IllegalArgumentException.class, () → {
            service.processPayment("", 100.0);
        });
    }

    @Test
    public void testNegativeAmount() {
        PaymentService service = new PaymentService();
        assertThrows(IllegalArgumentException.class, () → {
            service.processPayment("4111111111111111", -50.0);
        });
    }
}
```

ŘEŠENÍ

- Kód je robustní, ošetřuje chybové stavy.
- Testy pokrývají jak „happy path“, tak „sad path“.
- Snižuje se riziko výpadků v produkci kvůli neošetřeným vstupům.

- „Testovat jen šťastnou cestu je jako jezdit autem bez rezervy – funguje, dokud nepřijde problém.“
- Piš testy, které pokrývají validní vstupy, nevalidní vstupy, okrajové případy a výjimky.
- V produkčním kódu vždy ošetřuj vstupy, které mohou být neplatné nebo nečekané.
- Používej princip defenzivního programování – předpokládej, že vstupy mohou být chybné.

HAPPY PATH - PŘÍKLAD 2

```java
// Jednoduchá služba pro převod měny
public class CurrencyConverter {
    public double convert(double amount, String from, String to) {
        if (from.equals("USD") && to.equals("EUR")) {
            return amount * 0.9;
        } else if (from.equals("EUR") && to.equals("USD")) {
            return amount * 1.1;
        }
        throw new IllegalArgumentException("Unsupported currency pair");
    }
}
```

```java
// Špatný test – testuje jen happy path
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CurrencyConverterTest {
    @Test
    public void testUsdToEurConversion() {
        CurrencyConverter converter = new CurrencyConverter();
        double result = converter.convert(100, "USD", "EUR");
        assertEquals(90.0, result);
    }
}
```

- Testuje jen jeden ideální případ, kdy vše funguje správně.
- Netestuje:
  - Neplatné měny ("XXX"),
  - Převod mezi stejnými měnami,
  - Záporné hodnoty,
  - Výjimky.

```java
public class CurrencyConverterTest {

    private final CurrencyConverter converter = new CurrencyConverter();

    @Test
    public void testUsdToEurConversion() {
        assertEquals(90.0, converter.convert(100, "USD", "EUR"));
    }

    @Test
    public void testEurToUsdConversion() {
        assertEquals(110.0, converter.convert(100, "EUR", "USD"));
    }

    @Test
    public void testUnsupportedCurrencyThrowsException() {
        assertThrows(IllegalArgumentException.class, () -> converter.convert(100, "USD", "GBP"));
    }

    @Test
    public void testNegativeAmountHandled() {
        double result = converter.convert(-100, "USD", "EUR");
        assertEquals(-90.0, result);
    }

    @Test
    public void testSameCurrencyHandledGracefully() {
        // třeba později metoda bude rozšířena, tady můžeme testovat současné chování
        assertThrows(IllegalArgumentException.class, () -> converter.convert(100, "USD", "USD"));
    }
}
```

ŘEŠENÍ 2

- Pokrývá i chybové a hraniční případy,
- Testuje výjimky a chování systému při neočekávaných vstupech,
- Poskytuje reálnou jistotu, že kód obstojí i mimo „šťastnou cestu“.

IV. JAK ROZPOZNAT A VYHNOUT SE ANTIPATERNS

- Je zásadní antipatterny rozpoznat a vyhnout se jim, protože mohou vést k několika negativním důsledkům:
  - Neefektivní nebo neúčinná řešení problémů
  - Zvýšená složitost a náklady na údržbu
  - Snížená čitelnost a srozumitelnost kódu
  - Snížená produktivita a morálka týmu
- Rozpoznáním a vyhýbáním se antipatternům můžeme zlepšit naše řešení a zajistit, že budou **efektivní, účinná a udržitelná**.
- To může vést k lepším výsledkům projektů a příznivějšímu pracovnímu prostředí.

IDENTIFIKACE ANTIPATTERNŮ

- Při identifikaci antipatternů v našem kódu nebo návrhu je důležité **mít otevřenou mysl a zpochybňovat své předpoklady**.
- Někdy se můžeme přichytit k řešení, které si vyžádalo hodně času a úsilí, přesto může existovat lepší řešení.
- Abychom tomu předešli, je užitečné **požádat o zpětnou vazbu od ostatních**.
- Ostatní lidé často vidí náš problém z jiné perspektivy a mohou odhalit problémy nebo neefektivnosti, které jsme přehlédli.
- Další způsob, jak antipatterny identifikovat, je **hledat varovné signály**.
- Příliš složitá nebo těžko pochopitelná řešení mohou být známkou antipatternu.
- Také **studium osvědčených postupů a běžných úskalí** nám může pomoci vyhnout se chybám.
- Je rovněž důležité **nebát se zrušit kód a začít znovu**.
- Pokud zjistíme, že jsme uvízli v antipatternu, nové začátky mohou být tím nejlepším řešením.
- Dodržováním těchto tipů můžeme zvýšit šance na **rozpoznání antipatternů** v naší práci a zlepšit naše celkové dovednosti v programování a návrhu.

VYHÝBÁNÍ SE POUŽITÍ ANTIPATERNŮ

- Při práci na softwarových projektech je zásadní **být si vědom běžných úskalí**, která mohou vést k antipatternům.
- Jednou ze strategií, jak se těmto úskalím vyhnout, je **udělat krok zpět a zvážit širší kontext problému**.
- Porozumění problému jako celku pomůže při hledání dobrého řešení.
- Další strategií je **používat osvědčené návrhové vzory a nejlepší praktiky**.
- Řešení, která se osvědčila u ostatních s podobnými problémy, jsou dobrou volbou a mohou ušetřit spoustu času.
- Užitek může přinést i **seznámení se s běžnými návrhovými vzory**.
- Další strategií je **rozložit velké problémy na menší části**.
- To pomáhá vyhnout se pocitu zahlcení a usnadňuje odhalování problémů a neefektivností.
- Také se **nemáme bát požádat o pomoc**, pokud si s něčím nejsme jisti.
- Vždy existuje někdo zkušenější, kdo je ochotný pomoci a sdílet své znalosti.
- Je rovněž důležité **průběžně kontrolovat náš kód**, aby bylo zajištěno, že používáme nejlepší možné řešení.
- Jakmile se o problému a jeho řešení dozvíme více, můžeme změnit svůj přístup a vyvinout lepší řešení.

ČISTÝ KÓD

- KISS
- DRY
- SOLID
- YAGNI

- „Clean Code“ označuje psaní kódu, který je snadno pochopitelný, udržovatelný a upravitelný.
- Zaměřuje se na čitelnost, jednoduchost a přehlednost, což usnadňuje spolupráci vývojářů nad společnou kódovou základnou.
- Clean Code se řídí principy a postupy, jako jsou smysluplné názvy proměnných, konzistentní formátování, správné odsazování a modulární návrh.
- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ: Prentice Hall.

SOLID KONEC

- Každý vývojář by měl znát těchto pět návrhových principů, které mají za cíl usměrňovat návrh a vývoj softwaru tak, aby vznikal udržitelnější, flexibilnější a lépe škálovatelný kód.
- Každý princip se zaměřuje na jiný aspekt návrhu softwaru, ale dohromady směřují k tvorbě softwarových systémů, které jsou časem snáze pochopitelné, upravitelné a rozšiřitelné.

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

- „Každá třída by měla mít pouze jeden důvod ke změně.“
- To znamená, že třída by neměla nést více odpovědností najednou a jedna odpovědnost by neměla být rozptýlena mezi více tříd nebo smíchaná s jinými odpovědnostmi.
- Je to podobné jako ve firmě – je efektivnější, když má každý svou vlastní roli, než aby šéf, zaměstnanec, řidič a kuchař byla jedna a tatáž osoba.

SRP PŘÍKLAD

```java
// Třída pro generování reportu
class ReportGenerator {
    public String generateReport() {
        return "Report content";
    }
}

// Třída pro ukládání reportu
class ReportSaver {
    public void saveToFile(String content) {
        System.out.println("Saving report: " + content);
    }
}

// Třída pro tisk reportu
class ReportPrinter {
    public void printReport(String content) {
        System.out.println("Printing report: " + content);
    }
}

// Použití
public class Main {
    public static void main(String[] args) {
        ReportGenerator generator = new ReportGenerator();
        ReportSaver saver = new ReportSaver();
        ReportPrinter printer = new ReportPrinter();

        String report = generator.generateReport();
        saver.saveToFile(report);
        printer.printReport(report);
    }
}
```

```java
class Report {
    public String generateReport() {
        return "Report content";
    }

    public void saveToFile(String content) {
        // logika pro uložení do souboru
        System.out.println("Saving report: " + content);
    }

    public void printReport(String content) {
        // logika pro tisk
        System.out.println("Printing report: " + content);
    }
}
```

OPEN/CLOSED PRINCIPLE, OCP

- Princip otevřenosti/uzavřenosti (Open/Closed Principle, OCP): „Softwarové entity by měly být otevřené pro rozšíření, ale uzavřené pro změny.“
- Tento princip říká, že třída by měla být snadno rozšiřitelná, aniž by bylo nutné měnit její základní implementaci.
- Musím snad kopat a měnit základy celého domu, abych si k němu přidal balkon?

```java
// Abstrakce
interface Shape {
    double calculateArea();
}

// Konkrétní implementace
class Circle implements Shape {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

class Rectangle implements Shape {
    private double width, height;
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public double calculateArea() {
        return width * height;
    }
}

// Kalkulátor, který neporušuje OCP
class AreaCalculator {
    public double calculateArea(Shape shape) {
        return shape.calculateArea();
    }
}

// Použití
public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle(5);
        Shape rectangle = new Rectangle(4, 6);

        AreaCalculator calculator = new AreaCalculator();
        System.out.println("Circle area: " + calculator.calculateArea(circle)
        System.out.println("Rectangle area: " + calculator.calculateArea(rect
    }
}
```

OCP - PŘÍKLAD

```java
class AreaCalculator {
    public double calculateArea(Object shape) {
        if (shape instanceof Circle) {
            Circle c = (Circle) shape;
            return Math.PI * c.getRadius() * c.getRadius();
        } else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle) shape;
            return r.getWidth() * r.getHeight();
        }
        return 0;
    }
}
```

- Problém: Pokud přidáme nový tvar (např. trojúhelník), musíme upravit metodu calculateArea, čímž porušujeme OCP.
- Výhody:
- Snadné rozšíření – přidání nového tvaru nevyžaduje úpravu existujícího kódu
- Lepší udržovatelnost – méně rizika při změnách
- Čistší architektura – každá třída má jasnou odpovědnost

LISKOV SUBSTITUTION PRINCIPLE, LSP

- Princip substituce podle Liskovové (Liskov Substitution Principle, LSP): „Podtypy by měly být zaměnitelné za své základní typy, aniž by se narušila správnost programu.“
- Pokud program nebo modul používá základní třídu, pak by odvozená třída měla být schopna tuto třídu rozšířit, aniž by měnila její původní implementaci.
- Pokud to vypadá jako kachna, kváká jako kachna, ale potřebuje baterky, pravděpodobně máte špatnou abstrakci.

LSP - PŘÍKLAD

```java
class Rectangle {
    protected int width;
    protected int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}

class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        this.width = width;
        this.height = width; // nutí výšku být stejná jako šířka
    }

    @Override
    public void setHeight(int height) {
        this.height = height;
        this.width = height; // nutí šířku být stejná jako výška
    }
}
```

- Problém: Pokud použijeme Square tam, kde očekáváme Rectangle, může dojít k nečekanému chování. Například:

```java
Rectangle r = new Square();
r.setWidth(5);
r.setHeight(10);
System.out.println(r.getArea()); // Očekáváme 50, ale dostaneme 100
```

LSP - PŘÍKLAD

```
interface Shape {
    int getArea();
}

class Rectangle implements Shape {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}

class Square implements Shape {
    private int side;

    public Square(int side) {
        this.side = side;
    }

    public int getArea() {
        return side * side;
    }
}
```

INTERFACE SEGREGATION PRINCIPLE, ISP

- Princip segregace rozhraní (Interface Segregation Principle, ISP): „Klienti by neměli být nuceni záviset na rozhraních, která nepoužívají.“

- Žádný klient by neměl být nucen implementovat metody, které nepotřebuje, a smlouvy (rozhraní) by měly být rozděleny na menší a užší.
- Pták umí létat, pták je zvíře, takže zvířata umí létat.
- Znamená to tedy, že všechna zvířata umí létat?

ISP - PŘÍKLAD

```java
interface Worker {
    void work();
    void eat();
}

class Developer implements Worker {
    public void work() {
        System.out.println("Developer is coding.");
    }

    public void eat() {
        System.out.println("Developer is eating.");
    }
}

class Robot implements Worker {
    public void work() {
        System.out.println("Robot is working.");
    }

    public void eat() {
        throw new UnsupportedOperationException("Robot does not eat.");
    }
}
```

- Problém: Robot je nucen implementovat metodu eat(), kterou nepotřebuje. To porušuje ISP.

ISP - PŘÍKLAD

```java
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

class Developer implements Workable, Eatable {
    public void work() {
        System.out.println("Developer is coding.");
    }

    public void eat() {
        System.out.println("Developer is eating.");
    }
}

class Robot implements Workable {
    public void work() {
        System.out.println("Robot is working.");
    }
}
```

- Výhoda:
- Každá třída implementuje jen to, co potřebuje
- Rozhraní jsou specifická a přehledná
- Kód je čistší, flexibilnější a lépe testovateln

DEPENDENCY INVERSION PRINCIPLE (DIP):

- Princip inverze závislostí (Dependency Inversion Principle – DIP):
  „Moduly na vysoké úrovni by neměly záviset na modulech na nízké úrovni; oba typy by měly záviset na abstrakcích.“
- Tento princip říká, že mezi komponentami softwaru by neměla být těsná vazba.
- Aby se tomu předešlo, měly by komponenty záviset na abstrakci (např. rozhraní nebo obecné definici chování), nikoli na konkrétní implementaci.
- Příklad s autem:
  Při návrhu auta se nepočítá s konkrétními příslušenstvími (např. typem rádia nebo konkrétní značkou pneumatik).
- Místo toho se auto navrhuje tak, aby bylo kompatibilní s různými typy těchto doplňků – tedy závisí na abstrakci, ne na konkrétním provedení.
- To umožňuje flexibilitu a snadnou výměnu komponent bez zásahu do samotného návrhu auta.

DIP - PŘÍKLAD

```
class LightBulb {
    public void turnOn() {
        System.out.println("LightBulb is turned on");
    }

    public void turnOff() {
        System.out.println("LightBulb is turned off");
    }
}

class Switch {
    private LightBulb bulb;

    public Switch(LightBulb bulb) {
        this.bulb = bulb;
    }

    public void operate(boolean on) {
        if (on) bulb.turnOn();
        else bulb.turnOff();
    }
}
```

- Problém: Třída Switch přímo závisí na třídě LightBulb.
- Pokud byste chtěli přepnout na jiné zařízení (například ventilátor), museli byste upravit třídu Switch, čímž by došlo k porušení principu otevřenosti/uzavřenosti (Open/Closed Principle).

```java
// Abstraction
interface Switchable {
    void turnOn();
    void turnOff();
}

// Low-level module
class LightBulb implements Switchable {
    public void turnOn() {
        System.out.println("LightBulb is turned on");
    }

    public void turnOff() {
        System.out.println("LightBulb is turned off");
    }
}

// High-level module
class Switch {
    private Switchable device;

    public Switch(Switchable device) {
        this.device = device;
    }

    public void operate(boolean on) {
        if (on) device.turnOn();
        else device.turnOff();
    }
}

// Main
public class Main {
    public static void main(String[] args) {
        Switchable bulb = new LightBulb();
        Switch lightSwitch = new Switch(bulb);
        lightSwitch.operate(true);  // Output: LightBulb is turned on
        lightSwitch.operate(false); // Output: LightBulb is turned off
    }
}
```

DIP - ŘEŠENÍ

- Výhody principu závislosti na abstrakcích (DIP)
- Volné propojení: Třída Switch nezávisí na konkrétním zařízení, které ovládá.
- Flexibilita: LightBulb lze snadno nahradit jiným zařízením, které implementuje rozhraní Switchable.
- Testovatelnost: Rozhraní Switchable lze snadno nahradit falešnou implementací (mockem) při jednotkovém testování.
- Škálovatelnost: Nová zařízení lze přidávat bez nutnosti měnit logiku ve vyšších vrstvách aplikace.

```java
public interface Engine {
    void start();
}
```

```java
public class DieselEngine implements Engine {
    @Override
    public void start() {
        System.out.println("Diesel engine starting ... ");
    }
}
```

```java
public class Car {
    private Engine engine;

    public Car(Engine engine) {
        this.engine = engine;
    }

    public void startCar() {
        engine.start();
        System.out.println("Car is running.");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Engine diesel = new DieselEngine();
        Car myCar = new Car(diesel);
        myCar.startCar();
    }
}
```

DIP - PŘÍKLAD

- Car neví nic o konkrétním typu motoru – závisí jen na rozhraní Engine.
- Můžeš snadno přidat další typ motoru (např. ElectricEngine) bez změny třídy Car.
- To je přesně to, co DIP doporučuje: záviset na abstrakcích, ne na konkrétních třídách.

YAGNI (YOU AIN'T GONNA NEED IT – NEBUDEŠ TO POTŘEBOVAT)

- YAGNI doporučuje neimplementovat funkce, dokud nejsou skutečně potřeba, aby se předešlo zbytečné složitosti a zachoval se důraz na aktuální požadavky.
- Příklad:
- „Chystám se do Velké Británie, ale balím si kufr pro Španělsko – pro případ, že bych tam někdy jel. Potřebuji ho teď?“
- → Ne. A právě to je pointa YAGNI.
- Je to jako připravovat se na scénář, který možná nikdy nenastane, místo toho, abys řešil to, co je před tebou. Chceš si to přeložit i do programátorského kontextu, nebo naopak do každodenního života?

KISS (KEEP IT SIMPLE, STUPID) – „DRŽ TO JEDNODUŠE, HLUPÁKU

- Princip KISS podporuje jednoduchost v návrhu a vývoji.
- Upřednostňuje přímočará řešení před složitými, aby se zlepšila srozumitelnost a udržovatelnost kódu.
- Přirovnání:
- „Je jednodušší letět do Velké Británie letadlem než jet autem, že?“
- → Stejně tak je lepší napsat jednoduchý kód, než se ztratit ve zbytečně složitém řešení.

KISS PŘÍKLAD

```
class Calculator {
    public int calculate(String operation, int a, int b) {
        if ("add".equals(operation)) {
            return a + b;
        } else if ("subtract".equals(operation)) {
            return a - b;
        } else {
            throw new IllegalArgumentException("Unsupported operation");
        }
    }
}
```

- Snadno pochopitelné
- Snadno testovatelné
- Dělá přesně to, co má – bez zbytečných podmínek

```
class SimpleAdder {
    public int add(int a, int b) {
        return a + b;
    }
}
```

DRY (DON'T REPEAT YOURSELF – „NEOPAKUJ SE“

- Princip DRY podporuje znovupoužitelnost kódu tím, že se vyhýbá duplicitě kódu nebo logiky. Výsledkem je čistší a lépe udržovatelný kód.
- Přirovnání:
- „Máme si kupovat nové kufry na každou cestu?“
- → Ne, samozřejmě ne.
- Stejně tak bychom neměli psát stejný kód znovu a znovu – stačí ho napsat jednou a používat opakovaně.

DRY - PŘÍKLAD

```java
class InvoicePrinter {
    public void printHeader() {
        System.out.println("=== Invoice ===");
    }

    public void printInvoiceA() {
        System.out.println("=== Invoice ===");
        System.out.println("Customer: Alice");
        System.out.println("Amount: $100");
    }

    public void printInvoiceB() {
        System.out.println("=== Invoice ===");
        System.out.println("Customer: Bob");
        System.out.println("Amount: $200");
    }
}
```

```java
class InvoicePrinter {
    private void printHeader() {
        System.out.println("=== Invoice ===");
    }

    public void printInvoice(String customer, int amount) {
        printHeader();
        System.out.println("Customer: " + customer);
        System.out.println("Amount: $" + amount);
    }
}
```

- Kód je čistší a přehlednější
- Snadno se udržuje – změna hlavičky se provede jen na jednom místě
- Znovupoužitelnost bez zbytečné duplicity