1 Vysvetlite pojem DSL, porovnajte GPL a DSL

- **DSL (Domain-Specific Language):**
  - počítačový jazyk navrhnutý na riešenie problémov v konkrétnej problémovej doméne (napr. SQL pre databázy, HTML pre dokumenty)
- **GPL (General Purpose Language):**
  - univerzálny programovací jazyk (napr. Java, C++)
  - je Turing-úplný
  - umožňuje zapísať ľubovoľný algoritmus
- **Porovnanie GPL a DSL:**
  - **Rozsah použitia:**
    - **DSL** - je špecializovaný na jednu doménu, riešenie konkrétneho problému
    - **GPL** - je univerzálny (použiteľný na čokoľvek)
  - **Miera abstrakcie:**
    - **DSL** - poskytuje vyššiu úroveň abstrakcie, programátor nerieši nízkoúrovňové problémy, ako napr. správa pamäte, zvolenie vhodnej dátovej štruktúry, definovanie tried
    - **GPL** - má nižšiu úroveň abstrakcie
  - **Cieľ:**
    - **DSL** - sa zameriava na produktivitu a zrozumiteľnosť, programátor nemusí ovládať zložité paradigmy na to aby vedel efektívne vykonať požadovanú akciu
    - **GPL** - poskytuje flexibilitu pre vývoj akéhokoľvek softvéru
  - **Metafora:**
    - **DSL** - špecializovaný nástroj (motorová píla)
    - **GPL** - švajčiarsky nožík

2 Vysvetlite čo je paradigma programovacieho jazyka, uveďte príklady

- **Paradigma programovacieho jazyka:**
  - predstavuje základný spôsob myslenia v programovaní, ktorý je sprostredkovaný počítačovým jazykom
  - paradigmy sa navzájom odlišujú najmä pojmami, abstrakciami a spôsobom, akým sa program vykonáva a štruktúruje
  - veľa moderných jazykov je multiparadigmatických, podporujú viacero paradigiem súčasne, napr. Java (primárne objektovo orientovaná, ale obsahuje už aj funkcionálne prvky ako lambdy a streamy)
- **Príklady paradigmy:**

- **Objektovo-orientovaná paradigma** - základný pojem je objekt so stavom a správaním
- **Funkcionálna paradigma** - založená na matematických funkciách
- **Logické programovanie** - založené na faktoch a pravidlách
- **Aspektovo-orientovaná paradigma** - zameriava sa na prierezové témy (cross-cutting concerns), ako je logovanie alebo bezpečnosť
- **Komponentovo-orientovaná paradigma** - budovanie systémov zo znovupoužiteľných čiernych skriniek (komponentov)

3 Opíšte spôsob použitia DSL v softvérovom vývoji - architektúra spracovania DSL

- **Anatómia jazyka: Každé DSL pozostáva z troch základných zložiek:**
  1. **Konkrétna syntax** - Určuje formu zápisu (textovú, grafickú alebo ich kombináciu)
  2. **Abstraktná syntax** - Definuje vnútornú štruktúru a elementy (pojmy) jazyka
  3. **Sémantika** - Definuje význam viet (programov) prostredníctvom interpretácie alebo generovania kódu
- **Prístupy k implementácii:**
  - **Externé DSL** - Nezávislé od iných jazykov, vyžadujú vlastný jazykový procesor (generovaný napr. cez ANTLR)
  - **Interné DSL** - Využívajú syntax a infraštruktúru hostiteľského jazyka (napr. Java, Haskell)
  - **Jazykové prostredia** - Nástroje na komplexný návrh a modelovanie jazykov (napr. JetBrains MPS)
- **Architektúra spracovania:**
  - Jadrom architektúry je jazykový procesor, ktorý transformuje vetu zapísanú v konkrétnej syntaxi na abstraktnú reprezentáciu (AST alebo ASG)
  - Na túto abstraktnú štruktúru sa následne aplikuje sémantika, ktorá vykoná požadované akcie alebo vygeneruje výsledný softvérový artefakt

4 Porovnajte modely a programy - rozdiely a podobnosti, vykonateľné modely

- **Podobnosti:**
  1. Programy aj modely sú vnímané ako softvérové artefakty
  2. Oba druhy slúžia na vyjadrenie cieľov a zámerov autorov pri realizácii systémov a uplatňujú sa pri nich rovnaké inžinierske princípy, ako sú modularizácia, kompozícia a evolúcia
- **Rozdiely:**
  - Programy sú primárne orientované na konkrétnu syntax (textový zápis) a ich štruktúra je často definovaná bezkontextovými gramatikami

  - Modely sa sústreďujú na abstraktnú syntax a charakterizujú elementy jazyka ako triedy a relácie (metamodely), čo poskytuje prirodzenejšiu formu reprezentácie referencií oproti textovým identifikátorom v programoch

- **Vykonateľné modely:**
  - modely, ktoré majú definovanú sémantiku, čo umožňuje ich priame spracovanie
  - **Vykonateľnosť sa dosahuje dvoma hlavnými prístupmi:**
    1. **Interpretácia** - Veta (model) sa priamo vyhodnocuje, čo vedie k vykonaniu akcií
    2. **Generovanie** - Z modelu sa automaticky vygeneruje kód v inom (vykonateľnom) jazyku

5 Charakterizujte výhody a nevýhody použitia DSL v softvérovom vývoji

- **Výhody:**
  - Zvýšenie úrovne abstrakcie priblížením špecifikácie riešenia k doméne problému
  - Efektívnejšia statická analýza programov vďaka explicitnému používaniu doménových pojmov
  - Zlepšenie komunikácie medzi doménovými expertmi a vývojármi, keďže DSL môže slúžiť ako spoločný jazyk
  - Vyššia produktivita pri tvorbe viet (najmä pri textových DSL) a zjednodušenie realizácie evolúcie systému
- **Nevýhody:**
  - Vysoká náročnosť na vývoj, ktorá vyžaduje expertov na doménu aj na inžinierstvo počítačových jazykov
  - Častejšia evolúcia jazyka, pretože DSL priamo závisia od konkrétnej domény, ktorá sa v čase mení
  - Vysoké náklady na transformáciu existujúcich artefaktov pri prechode na novú verziu jazyka
  - Syntaktický šum alebo nadbytočnosť (napr. pri XML jazykoch), čo môže znižovať efektivitu pri čítaní a zápise

6 Argumentujte prečo je nejaký konkrétny jazyk DSL, resp. GPL

1. **Prečo je SQL doménovo-špecifický jazyk (DSL)?:**
   - **Doména** - Práca s relačnými dátami
   - **Abstrakcia** - SQL používa pojmy ako SELECT, JOIN, WHERE, programátor špecifikuje čo chce (deklaratívny prístup), nie ako sa majú dáta fyzicky načítať z disku
   - **Obmedzenie** - SQL nie je navrhnuté na tvorbu používateľského rozhrania alebo ovládanie hardvéru, chýba mu univerzálnosť
2. **Prečo je Java univerzálny jazyk (GPL)?:**

- **Turingovská úplnosť** - V Jave môžeme implementovať ľubovoľný algoritmus
- **Technické detaily** - Programátor musí narábať s technickými konštrukciami (triedy, vlákna, správa výnimiek), ktoré nie sú špecifické pre jednu biznis oblasť, ale pre softvérový vývoj ako taký

7 Opíšte a rozlíšte konkrétnu a abstraktnú syntax jazyka a sémantiku jazyka

- **Konkrétna syntax** - Určuje formu zápisu (textovú, grafickú alebo ich kombináciu)
- **Abstraktná syntax** - Definuje vnútornú štruktúru a elementy (pojmy) jazyka
- **Sémantika** - Definuje význam viet (programov) prostredníctvom interpretácie alebo generovania kódu
- **Zhrnutie rozdielov:**
  - Hlavný rozdiel spočíva v tom, že konkrétna syntax sa zaoberá formou zápisu (ako jazyk vyzerá), abstraktná syntax sa sústreďuje na logickú štruktúru a pojmy (z čoho sa skladá) a sémantika určuje samotný zmysel a vykonávanie týchto štruktúr (čo vyjadrujú)
  - Jeden jazyk s jednou abstraktnou syntaxou a sémantikou môže mať viacero rôznych konkrétnych reprezentácií

8 Charakterizujte jazykový (sémantický) model a jeho zápis v OOP, uveďte príklad zápisu jazykového modelu v OOP

- **Jazykový (sémantický) model** charakterizuje v abstraktnej forme elementy (pojmy) jazyka a definuje pravidlá pre ich kompozíciu (abstraktnú syntax), pričom na túto štruktúru je priamo naviazaná sémantika jazyka
- **Zápis v OOP sa realizuje vo forme doménového modelu tvoreného hierarchiou tried a ich vzájomných väzieb:**
  - **Triedy** - definujú jednotlivé jazykové pojmy
  - **Vzťah „je“ (dedičnosť/implementácia)** - vyjadruje špecializáciu pojmov (napr. cyklus je príkaz)
  - **Vzťah „má“ (atribúty/premenné)** - vyjadruje zloženie pojmov (napr. cyklus má podmienku a telo)
  - **Metódy tried** - predstavujú sémantické funkcie, ktoré definujú význam daných konštrukcií

```
// Abstraktny pojem
public abstract class Expression {
    public abstract long eval(); // Semanticka funkcia
}

// Konkretny terminalny pojem
public class Number extends Expression {
    private long value;
    public long eval() { return value; }
}
```

```
// Zlozeny pojem reprezentujuci vztahy
public class Add extends Expression {
    private Expression expression1; // Vztah "ma"
    private Expression expression2;

    public long eval() {
        return expression1.eval() + expression2.eval();
    }
}
```

9 Porovnajte externé a interné DSL, ich vlastnosti, výhody a nevýhody ich aplikácie

- **Externé DSL:**
  - **Podstata** - samostatné jazyky, ktoré sú úplne nezávislé od iných programovacích jazykov
  - **Vlastnosti** - vyžadujú vytvorenie vlastného jazykového procesora (prekladača), ktorý pozná ich konkrétnu a abstraktnú syntax
  - **Výhody** - ponúkajú úplnú voľnosť pri návrhu syntaxe a umožňujú vykonávať špecifickú syntaktickú aj sémantickú kontrolu
  - **Nevýhody** - implementácia je náročná, pretože autor musí mať expertné znalosti z oblasti gramatík a generátorov jazykových procesorov
- **Interné (vložené) DSL:**
  - **Podstata** - implementované v rámci existujúceho hostiteľského jazyka všeobecného použitia (napr. Java, Haskell)
  - **Vlastnosti** - nevyžadujú špeciálny procesor, pretože využívajú prostriedky a infraštruktúru hostiteľa
  - **Výhody** - jednoduchšia implementácia, pri ktorej autor nemusí ovládať technológie spracovania textu ani gramatiky
  - **Nevýhody** - syntax je priamo obmedzená a ovplyvnená pravidlami hostiteľského jazyka, čo môže viesť k prítomnosti nežiadúcich elementov (syntaktický šum)

10 Aký je rozdiel medzi použitím generovania a interpretácie pri implementácii DSL

- **Generovanie** - Zo vstupnej vety v DSL sa ako výstup vygeneruje veta v inom jazyku
- **Interpretácia** - Veta sa zo vstupu priamo vyhodnocuje, čoho dôsledkom je okamžité vykonanie akcií
- Výber medzi týmito prístupmi závisí od rozhodnutia autora nástrojovej podpory a nie je vopred určený syntaxou jazyka

Vlastnosť	**Generovanie**	**Interpretácia**
Výkon	Vyšší (optimalizovaný kód)	Nižší (runtime overhead)
Rýchlosť vývoja	Pomalšia (edit-gen-compile)	Rýchlejšia (edit-run)
Platforma	Závislá na cieľovom kompilátore	Vyžaduje runtime engine (interpret)

11 Rozlíšte vzory pre implementácie interných DSL

- **Function Sequence (Postupnosť funkcií)** - Veta je zapísaná ako postupnosť volaní samostatne definovaných funkcií (často s využitím statických importov), ktoré sa vykonávajú rad za radom
- **Nested Function (Vnorená funkcia)** - Veta je vyjadrená pomocou volaní funkcií, ktoré sú do seba vnorené, pričom jedna funkcia slúži ako argument (parameter) pre druhú funkciu
- **Method Chaining (Zreťazenie metód)** - Veta vzniká reťazením volaní metód na objektoch (tzv. fluid interface), kde každé volanie typicky vracia referenciu umožňujúcu okamžité nadväzujúce volanie ďalšej metódy
- **Literal List/Map** - Veta je vyjadrená pomocou natívnych dátových štruktúr hostiteľského jazyka (napr. zoznamy), ktoré priamo reprezentujú hierarchické dáta alebo konfiguráciu bez nutnosti volania metód

12 Porovnajte regulárne a bezkontextové jazyky - štruktúra gramatiky a automatu

- **Regulárne jazyky:**
  - **Štruktúra gramatiky** - Sú špecifikované regulárnymi výrazmi, ktoré v procese spracovania jazyka slúžia na definíciu lexikálnych jednotiek (tokenov)
  - **Štruktúra automatu** - Sú rozpoznávané konečným (stavovým) automatom
- **Bezkontextové jazyky:**
  - **Štruktúra gramatiky** - Sú definované bezkontextovou gramatikou (často v Backus-Naurovej forme – BNF); pozostávajú z množiny neterminálov, terminálov a produkčných pravidiel, pričom definujú stromovú štruktúru vety (abstraktný syntaktický strom)
  - **Štruktúra automatu** - Sú rozpoznávané zásobníkovým automatom (pushdown automat), ktorý môže byť v určitých prípadoch deterministický

13 Opíšte lexikálnu, syntaktickú a sémantickú analýzu - fázy spracovania vety jazyka

- **Lexikálna analýza** - Identifikuje a extrahuje zo vstupného textu základné lexikálne jednotky (tokeny), ako sú identifikátory, čísla (literály) alebo kľúčové slová
- **Syntaktická analýza** - Overuje, či je veta v súlade s pravidlami konkrétnej syntaxe (gramatiky), a transformuje ju na abstraktný syntaktický strom (AST), ktorý reprezentuje jej vnútornú štruktúru
- **Sémantická analýza** - Určuje význam vety a vykonáva vyhľadávanie referencií (riešenie odkazov), čím transformuje pôvodný strom na komplexnejší abstraktný syntaktický graf (ASG)

14 Porovnajte prístupy pri spracovaní vety z jazyka - zhora nadol (top-down) a zdola nahor (bottom-up)

- **Zhora nadol (top-down)** - Proces spracovania vychádza zo začiatočného symbolu (koreňa) gramatiky a postupne ho rozkladá na menšie časti až po terminálne symboly (listy), ktoré zodpovedajú vstupnému textu
- **Zdola nahor (bottom-up)** - Proces začína od konkrétnych symbolov vo vete (listov) a postupne ich spája (redukuje) do čoraz širších celkov až po dosiahnutie koreňového uzla. Pri tomto prístupe platí, že v čase konštruovania objektu uzla sú už skonštruované všetky objekty reprezentujúce jeho podstromy.

15 Opíšte ako funguje spracovanie viet na základe oddeľovačov - delimiter-directed translation (parsing)

- Podstata spracovania na základe oddeľovačov (delimiter-directed translation) spočíva v použití lexikálnych symbolov (oddeľovačov) na jednoznačné určenie štruktúry vety v konkrétnej syntaxi bez priameho vplyvu na jej sémantiku
- **Tieto oddeľovače fungujú ako značky, ktoré:**
  - predchádzajú konštrukcii (napr. begin)
  - nasledujú za konštrukciou (napr. end)
  - oddeľujú jednotlivé prvky v postupnostiach (napr. čiarka medzi parametrami)

16 Napíšte gramatiku v BNF pre určený jazyk

- **Základný tvar pravidla** - Každé pravidlo má formu Neterminál ::= výraz, kde symbol ::= znamená „je definovaný ako“
- **Symboly:**
  - **Neterminály** - (vystupujú na ľavej strane a v zložených výrazoch)
  - **Terminály** - (konkrétne symboly abecedy, napr. 'start', '+', čísla)
- **Operátory pre kompozíciu:**
  - | - vyjadruje alternatívu (výber z viacerých možností)
  - + - znamená jeden alebo viac výskytov
  - * - znamená nula alebo viac výskytov
  - ? - označuje voliteľnosť (nula alebo jeden výskyt)

17 Opíšte stromové reprezentácie viet - strom odvodenia (parse tree), abstraktný syntaktický strom - ich vytvorenie a prechádzanie

- Stromové reprezentácie slúžia na zachytenie štruktúry vety v počítačovom jazyku pre jej ďalšie automatizované spracovanie
- **Strom odvodenia (Parse Tree):**
  - **Podstata** - Reprezentuje konkrétnu syntaktickú štruktúru vety presne podľa pravidiel gramatiky
  - **Charakteristika** - Obsahuje všetky detaily konkrétnej syntaxe (napr. terminálne symboly ako zátvorky alebo čiarky), ktoré sú potrebné na správne

rozpoznanie vety

- **Abstraktný syntaktický strom (AST):**
  - **Podstata** - Reprezentuje vnútornú (abstraktnú) štruktúru vety, kde každý uzol určuje výskyt konkrétneho jazykového pojmu
  - **Charakteristika** - Na rozdiel od stromu odvodenia odstraňuje nepodstatné syntaktické detaily (šum) a sústreďuje sa na význam a pravidlá kompozície elementov, slúži ako hlavný podklad pre definovanie sémantiky (významu) programu
- **Vytvorenie:**
  - Stromy konštruuje jazykový procesor (parser) počas analýzy vstupnej vety
  - V moderných prístupoch (napr. nástroj YAJCo) sa AST vytvára automaticky ako strom objektov, ktoré sú inštanciami tried doménového modelu
- **Prechádzanie:**
  - **Na spracovanie vytvorených stromov sa využívajú mechanizmy prechádzania uzlov, najčastejšie prostredníctvom dvoch vzorov:**
    - **Listener** - Prechádza strom a vyvoláva udalosti pri vstupe do uzla a výstupe z neho
    - **Visitor (Návštevník)** - Priamo navštevuje uzly stromu za účelom vykonania sémantickej akcie alebo výpočtu (napr. validácia alebo generovanie kódu)

18 Charakterizujte prácu so symbolmi v programoch - tabuľka symbolov, oblasti viditeľnosti

- Práca so symbolmi v programoch spočíva v používaní identifikátorov (mien), ktoré označujú pojmy a slúžia predovšetkým na odkazovanie (referencovanie) na konkrétne výskyty týchto pojmov v texte programu
- **Tabuľka symbolov** - (v širšom zmysle proces vyhľadávania referencií) umožňuje priradiť meno k jeho významu v danom kontexte, čím sa textové odkazy transformujú na priame väzby v abstraktnom syntaktickom grafe
- **Oblasti viditeľnosti (kontexty)** - definujú rozsah, v ktorom musí byť identifikátor jedinečný, identifikátor je určený nielen svojím menom, ale aj kontextom, v ktorom sa vyskytuje; to umožňuje, aby rovnaké meno označovalo rôzne lokálne pojmy v odlišných oblastiach (napr. v rôznych procedúrach), zatiaľ čo globálne pojmy musia byť jedinečné v celom programe

19 Opíšte generátory jazykových procesorov - načo slúžia a ako sa používajú (napr. Antlr)

- Generátor jazykových procesorov (angl. parser generator) je nástroj, ktorý automaticky generuje zdrojový kód jazykového procesora na základe jeho formálnej špecifikácie
- **Načo slúžia:**

  - Slúžia na zjednodušenie implementácie jazykových procesorov určených na spracovanie viet konkrétneho jazyka (napr. doménovo-špecifického jazyka) so znalosťou jeho syntaxe a sémantiky
  - Šetria čas tým, že autor nemusí ručne programovať algoritmy na rozpoznávanie textu
- **Ako sa používajú:**
  - **Vstup** - Autor vytvorí špecifikáciu, ktorou je zvyčajne bezkontextová gramatika zapísaná v určenom formáte (často BNF – Backus-Naurova forma)
  - **Proces** - Generátor spracuje túto gramatiku a pomocou zvolenej technológie (napr. LL, LR, LALR) vytvorí algoritmus na rozpoznávanie viet
  - **Výstup** - Výsledkom je zdrojový kód v cieľovom programovacom jazyku (napr. Java, C#), ktorý po preložení dokáže analyzovať vstupný text a často z neho automaticky budovať abstraktný syntaktický strom (AST)

20 Opíšte vzťah doménovo-špecifických jazykov a modelom riadeného vývoja softvéru - DSL (Domain-specific language) a MDSD (Model driven software development)

- DSL predstavuje kľúčový prístup a nástroj k budovaniu softvéru v kontexte modelom riadeného vývoja (MDSD)
- V rámci MDSD majú DSL významné postavenie pri definovaní abstraktnej syntaxe jazyka prostredníctvom metamodelov, čo umožňuje formálne opísať štruktúru softvérového systému
- Používanie DSL v modelmi riadenom vývoji zjednodušuje realizáciu evolúcie (zmien a údržby) softvérových systémov vďaka využívaniu vyššej úrovne abstrakcie

21 Charakterizujte jazykové prostredie - funkcie, význam, príklady (language workbenches)

- **Charakteristika:**
  - Jazykové prostredia sú komplexné nástroje určené na efektívnu tvorbu doménovo-špecifických jazykov (DSL) a doménové modelovanie. Ich význam rastie najmä vďaka zjednodušovaniu procesu vývoja špecializovaných softvérových nástrojov
- **Funkcie:**
  - Tvorba grafických dizajnérov pre vizuálne jazyky a editorov s podporou (dopĺňanie kódu, refaktoring)
  - Definovanie transformácií z modelov do výstupného (cieľového) jazyka
  - Generovanie výsledných viet (kódu) v cieľovom jazyku
  - Konfigurácia generických nástrojov prostredníctvom metamodelov, ktoré špecifikujú syntax a sémantiku domény
- **Príklady:**
  - **Microsoft Visual Studio DSL** - nástroj na návrh grafických DSL v prostredí Visual Studio

  - **JetBrains Meta Programming System (MPS)** - prostredie podporujúce jazykovo-orientované programovanie
  - **MetaEdit+** - prostredie využívajúce metamodelovací jazyk GOPPRR
  - **GME (Generic Modeling Environment)** - konfigurovateľné prostredie pre syntézu programov a doménové modelovanie

22 Opíšte princíp projekčných editorov, porovnajte ich s klasickými textovými editormi

- **Princíp projekčných editorov**
  - Projekčné editory (ako napr. Jetbrains MPS) sú založené na priamej manipulácii s abstraktnou syntaxou (doménovým modelom/AST). Používateľ neupravuje text, ale priamo prvky modelu, pričom editor mu tento model len „premieta“ (projektuje) do zvolenej konkrétnej formy (textovej, grafickej alebo ich kombinácie)
- **Porovnanie s klasickými textovými editormi:**
  - **Smer spracovania** - Klasický prístup postupuje od konkrétnej syntaxe k abstraktnej (používateľ píše text, ktorý procesor transformuje na strom), zatiaľ čo projekčný prístup postupuje od abstraktnej syntaxe ku konkrétnej (model je primárny, zobrazenie je sekundárne)
  - **Orientácia** - Textové editory sú orientované na gramatiku a zápis (konkrétnu syntax), zatiaľ čo projekčné editory sa sústreďujú na pojmy a ich sémantiku (abstraktnú syntax)
  - **Technológia** - Textové editory vyžadujú na rozpoznanie štruktúry parser (napr. LL, LR), zatiaľ čo projekčné editory konštruujú model priebežne počas editácie, čím eliminujú potrebu klasického parsovania textu

23 Opíšte generovanie kódu z modelu a porovnajte: generovanie pomocou šablón vs. generovanie transformáciou (Templated Generation vs. Transformer Generation); generovanie s použitím modelu a bez (Model-Aware Generation vs. Model Ignorant Generation)

- **Templated vs. Transformer Generation:**
  - **Generovanie pomocou šablón (Templated Generation)** - Využíva šablónovacie systémy (napr. Velocity), kde je výstupná forma definovaná ako textový vzor. Tento prístup je vhodný, ak je veľká časť výsledného kódu nemenná, pričom generátor do šablóny len vkladá konkrétne hodnoty z modelu
  - **Generovanie transformáciou (Transformer Generation)** - Výstup sa generuje programovo (napr. pomocou metód v Jave), ktoré prechádzajú štruktúru modelu (abstraktný syntaktický graf) a postupne transformujú jednotlivé doménové pojmy na výsledný text
- **Model-Aware vs. Model-Ignorant Generation:**
  - **Model-Aware Generation** - Generátor je orientovaný na abstraktnú syntax a priamo pracuje s doménovým modelom (pojmami a ich vzťahmi)
  - **Model-Ignorant Generation** - Ide o klasický prístup orientovaný primárne na

konkrétnu syntax (gramatiku), generátor sa sústreďuje na rozpoznanie textových reťazcov a ich syntaktickú správnosť, pričom model (abstraktná syntax) v ňom často nie je explicitne definovaný a je ponechaný na autorovi procesora

24 Ako je možné prepojiť generovaný kód a ručne napísaný kód - význam jeho oddelenia

- **Spôsob prepojenia** - Prepojenie sa realizuje prostredníctvom doménového modelu, kde generované štruktúry (napr. triedy reprezentujúce uzly abstraktného syntaktického stromu) slúžia ako základ, do ktorého vývojár ručne implementuje sémantické funkcie vo forme metód (napr. eval() alebo code()). Ďalšou možnosťou je využitie aspektovo-orientovaného programovania (AOP) na úplné oddelenie realizačnej logiky od dátovej štruktúry
- **Význam oddelenia** - Hlavným významom je nezávislosť sémantiky od konkrétnej syntaxe, to umožňuje meniť formu zápisu jazyka (textovú alebo grafickú) alebo technológiu rozpoznávania viet bez toho, aby bolo nutné upravovať ručne napísanú realizačnú logiku (sémantiku). Oddelenie zároveň zjednodušuje evolúciu a modularizáciu jazyka, pretože zmeny v jednej časti (napr. pridanie operátora) majú minimálny dopad na ostatné artefakty

25 Vysvetlíte čo je to modularita a prečo je podstatná vo vývoji softvéru

- Modularita je miera, v ktorej môžu byť komponenty systému od seba oddelené a rôzne kombinované. Jej podstata spočíva v rozdelení zložitého celku na menšie, samostatne spravovateľné časti
- **Vo vývoji softvéru je podstatná z týchto hlavných dôvodov:**
  - **Pochopenie kódu** - Rozdelením na moduly dokáže človek porozumieť jednotlivým častiam komplexného systému
  - **Tímová práca** - Umožňuje členom tímu pracovať paralelne a nezávisle na rôznych častiach kódu
  - **Flexibilita** - Uľahčuje vykonávanie zmien tak, aby úprava v jednej časti nežiaduco neovplyvnila iné časti systému
  - **Testovateľnosť** - Moduly sa dajú testovať samostatne v izolácii, čo zjednodušuje hľadanie chýb vďaka menšiemu množstvu premenných
  - **Nasadenie (deployment)** - Umožňuje nezávisle nasadzovať a škálovať jednotlivé služby bez nutnosti koordinácie celého systému
- Pre dosiahnutie dobrej modularity je kľúčové usilovať sa o vysokú kohéziu (súvisiace veci patria k sebe) a nízku prepojenosť (moduly sú od seba čo najmenej závislé)

26 Ako súvisí modularita s testovaním softvéru

- **Izolácia a zjednodušenie:** - Modularita umožňuje testovať časti systému samostatne a nezávisle. Pri testovaní jedného modulu sa pracuje s výrazne menším počtom premenných a ich kombinácií než pri celom systéme, čo robí testovanie jednoduchším a dôkladnejším
- **Použitie mockov:** - Dobre navrhnuté rozhrania a princíp obrátenia závislostí (Dependency Inversion) umožňujú pri testovaní nahradiť zložité externé technológie (napr. databázy) testovacími dvojníkmi (mockmi), čím sa tvorba testov ďalej zjednodušuje

- **Indikátor kvality návrhu:** - Testovanie slúži ako pomôcka pri návrhu – ak sa k nejakej časti kódu píšu testy ťažko, je to zvyčajne príznakom zlého rozdelenia modulov a nevhodnej štruktúry
- **Dôvera v systém:** - Dôsledné otestovanie malých modulov zvyšuje istotu, že aj systém ako celok bude po integrácii fungovať správne

27 Vysvetlíte význam pojmu kohézia (cohesion) v softvéru

- Kohézia (súdržnosť) vyjadruje mieru, do akej prvky vo vnútri jedného modulu patria k sebe
- **Jej podstata spočíva v dvoch kľúčových zásadách:**
  1. **Súvislosť** - Veci, ktoré spolu logicky súvisia, majú byť umiestnené spolu
  2. **Spoločná zmena** - Modul majú tvoriť časti, ktoré sa menia z rovnakých dôvodov a v rovnakom čase (tzv. Common Closure Principle), v softvérovom inžinierstve sa snažíme o vysokú kohéziu, čo znamená, že jeden modul by mal riešiť práve jednu ucelenú vec a nemiešať v sebe nesúvisiace záujmy

28 Vymenujte a stručne vysvetlite hlavné princípy pre dekompozíciu softvérových systémov

- **Vysoká kohézia (High Cohesion)** - Snaha o to, aby prvky vo vnútri jedného modulu úzko súviseli a patrili k sebe. Ideálne by mal mať modul len jednu zodpovednosť a jeden dôvod na zmenu
- **Nízka prepojenosť (Low Coupling)** - Minimalizácia vzájomných závislostí medzi modulmi, cieľom je, aby moduly boli čo najviac nezávislé, čo uľahčuje ich zmeny a testovanie
- **Skrývanie informácií (Information Hiding)** - Moduly by mali skrývať svoje konkrétne návrhové rozhodnutia (napr. spôsob ukladania dát) za abstraktnými rozhraniami, zmena vnútri modulu potom neovplyvní ostatné časti systému
- **Oddelenie záujmov (Separation of Concerns)** - Princíp, pri ktorom sa kód riešiaci jednu konkrétnu vec (aspekt problému) sústreďuje na jednom mieste a nemieša sa s nesúvisiacimi záujmami
- **Abstrakcia** - Potlačenie implementačných detailov a náhodnej zložitosti v prospech podstatných vlastností systému a doménových pojmov
- **Dekompozícia podľa domény** - Moduly by mali vychádzať z podstaty riešeného problému (domény). To, čo je oddelené v realite, by malo byť oddelené aj v softvéri

29 V čom spočíva princíp skrývania informácii (information hiding) v návrhu softvéru a ako súvisí s dekompozíciou

- **Princíp skrývania informácií (information hiding)** - Spočíva v tom, že každý modul v systéme skrýva konkrétne návrhové rozhodnutie (napr. spôsob ukladania dát alebo konkrétny algoritmus) za svoje rozhranie. Toto „tajomstvo“ modulu nie je zvonku prístupné, čím sa minimalizuje dopad budúcich zmien na zvyšok systému
- **Súvis s dekompozíciou** - Skrývanie informácií slúži ako kritérium dekompozície, na rozdiel od prirodzenej, ale menej vhodnej dekompozície podľa krokov spracovania (postupnosti deja), sa pri tomto prístupe systém delí na moduly podľa ich zodpoved-

nosti za jednotlivé návrhové rozhodnutia. Takáto dekompozícia zvyšuje nezávislosť modulov, uľahčuje paralelné vyvíjanie a zlepšuje zmeniteľnosť softvéru

30 Čo je to prepojenosť (coupling) v návrhu softvéru a ako ju môžeme znížiť

- Prepojenosť (coupling) vyjadruje mieru vzájomnej závislosti a prepojenia jednotlivých častí (modulov) systému. V kvalitnom návrhu sa snažíme o nízku prepojenosť, aby boli moduly čo najviac nezávislé a systém flexibilný
- **Ako ju môžeme znížiť:**
  - **Definovanie rozhraní** - Medzi modulmi vytvárať čo najmenšie a jasne definované rozhrania, ktoré skrývajú interné detaily
  - **Skrývanie informácií (information hiding)** - Moduly by mali skrývať svoje návrhové rozhodnutia a implementačné detaily
  - **Princíp inverzie závislostí (Dependency Inversion)** - Zabezpečiť, aby moduly záviseli od abstrakcií (rozhraní), a nie od konkrétnych nízkoúrovňových implementácií
  - **Dependency Injection** - Modul by si nemal závislosti vytvárať sám, ale dostávať ich zvonku ako rozhranie
  - **Vzory ako Ports and Adapters** - Oddeliť doménovú logiku od technických detailov (databázy, API) pomocou abstraktných portov a ich konkrétnych adaptérov

31 Vymenujte a stručne vysvetlíte 4 oblastí, ktoré sú súčasťou architektúry softvéru

1. **Štruktúra systému** - Predstavuje základné rozdelenie systému na komponenty a definuje vzťahy medzi nimi
2. **Architektonické charakteristiky** - Ide o kvalitatívne vlastnosti systému (tzv. „-ilities“), ako sú napríklad testovateľnosť, udržiavateľnosť alebo spoľahlivosť
3. **Architektonické rozhodnutia** - Sú to kľúčové voľby a pravidlá o tom, ako bude systém fungovať počas jeho vývoja a prevádzky
4. **Návrhové princípy** - Súbor pravidiel a smerníc, ktoré sa aplikujú pri návrhu jednotlivých častí systému a ich vzájomnej interakcii

32 Aké vlastnosti by mal mať softvérový architekt

- Schopnosť robiť a priebežne analyzovať kľúčové architektonické rozhodnutia o najdôležitejších vlastnostiach systému
- Technický prehľad a kontakt s praxou, čo zahŕňa sledovanie aktuálnych trendov a to, že architekt neprestáva programovať
- Dohľad nad dodržiavaním pravidiel, teda zabezpečenie, aby sa prijaté architektonické rozhodnutia v tíme skutočne realizovali
- Hlboká znalosť domény, aby rozumel biznis oblasti, pre ktorú je softvér navrhovaný (napr. bankovníctvo či medicína)
- Rozvinuté „soft skills“ a komunikácia, ktoré sú nevyhnutné pre vyjednávanie s tímom, zákazníkmi a pohyb v organizačnej štruktúre firmy

33 Vymenujte a stručne vysvetlite SOLID princípy

- **S** - Single Responsibility Principle (Princíp jednej zodpovednosti): Modul by mal mať len jeden dôvod na zmenu, čo znamená, že by mal zodpovedať za požiadavky práve jedného konkrétneho aktéra (používateľa alebo skupiny)
- **O** - Open–Closed Principle (Princíp otvorenosti/uzavretosti): Softvérový artefakt má byť otvorený pre rozširovanie (pridávanie novej funkcionality), ale uzavretý pre modifikáciu (úpravu existujúceho a otestovaného kódu)
- **L** - Liskov Substitution Principle (Liskovovej princíp zastúpenia): Objekty nadtypu musia byť nahraditeľné objektmi svojich podtypov bez toho, aby sa narušilo správanie alebo vlastnosti systému
- **I** - Interface Segregation Principle (Princíp oddelenia rozhraní): Klienti by nemali byť nútení závisieť od metód, ktoré nepoužívajú; rozhrania by mali byť špecifické pre konkrétnych klientov, nie univerzálne
- **D** - Dependency Inversion Principle (Princíp obrátenia závislostí): Vysokoúrovňové moduly (doménová logika) nesmú závisieť od nízkoúrovňových (technické detaily), ale oba majú závisieť od abstrakcií; abstrakcie nesmú závisieť od detailov

34 Vysvetlite princíp „čistej architektúry“

- Podstata čistej architektúry (Clean Architecture) spočíva v striktnom oddelení doménovej logiky od technických detailov, ako sú databázy, používateľské rozhrania či frameworky
- **Jej fungovanie definujú tieto kľúčové body:**
  - **Pravidlo závislostí** - Všetky závislosti v kóde smerujú výhradne dovnútra smerom k jadru (k entitám a scenárom použitia)
  - **Nezávislosť od technológií** - Vnútorné vrstvy (vysokoúrovňová logika) nevedia nič o vonkajších vrstvách (nízkoúrovňové detaily), čo umožňuje jednoduchú výmenu technológií bez dopadu na biznis logiku
  - **Vysoká testovateľnosť** - Vďaka izolácii od externých vplyvov (DB, API) je možné doménové jadro testovať rýchlo a efektívne aj bez týchto systémov

35 Vysvetlíte princíp otočenia závislostí a aký problém je možné pomocou neho riešiť

- Princíp otočenia závislostí (Dependency Inversion Principle) hovorí, že vysokoúrovňové moduly (doménová logika) nesmú závisieť od nízkoúrovňových modulov (technické detaily, napr. databázy, API), ale obe musia závisieť od spoločných abstrakcií (rozhraní). Platí, že abstrakcie nesmú závisieť od detailov, ale detaily od abstrakcií
- **Riešený problém:**
  - **Vysoká prepojenosť (coupling)** - Zabraňuje tomu, aby zmena v technických detailoch (napr. prechod z jednej databázy na inú) vyžadovala zmenu v biznis logike
  - **Slabá testovateľnosť** - Umožňuje izolovať doménovú logiku a testovať ju nezávisle od externých systémov pomocou náhradných objektov (mockov)

  - **Nepružnosť architektúry** - Rieši problém, kedy sú dôležité architektonické rozhodnutia zviazané s konkrétnymi nástrojmi, čo zvyšuje cenu budúcich zmien

36 Vymenujte a stručne opíšte aspoň 4 architektonické štýly

1. **Vrstvená architektúra (Layered)** - Systém je rozdelený na logické a technologické vrstvy (napr. používateľské rozhranie, biznis logika a databáza), ktoré majú jasne definované zodpovednosti
2. **Mikrokernel (Microkernel)** - Architektúra pozostávajúca z minimálneho jadra so základnou funkčnosťou, ku ktorému sa dodatočné vlastnosti pridávajú vo forme pluginov (rozšírení)
3. **Mikroslužby (Microservices)** - Rozdelenie systému na veľké množstvo malých, nezávislých služieb na základe domény, ktoré komunikujú cez sieťové protokoly
4. **Založená na udalostiach (Event-Driven)** - Komunikácia medzi časťami systému prebieha prostredníctvom zasielania a spracovania udalostí cez fronty správ (message queues)

37 Aké sú výhody a nevýhody distribuovaných architektúr

- **Výhody:**
  - **Škálovateľnosť** - Jednotlivé moduly môžu bežať na rôznych serveroch alebo vo viacerých inštanciách podľa aktuálnej záťaže
  - **Flexibilita technológií** - Služby môžu byť implementované v rôznych programovacích jazykoch podľa toho, čo je pre danú úlohu najvhodnejšie
  - **Nezávislý vývoj a nasadenie** - Samostatné tímy môžu vyvíjať a automatizovane nasadzovať svoje časti systému bez potreby zložitej koordinácie s ostatnými
  - **Vynútenie hraníc** - Komunikácia cez sieťový protokol núti vývojárov dodržiavať rozhrania, čím bráni príliš silnému previazaniu komponentov, ktoré je bežné v monolitických systémoch
- **Nevýhody:**
  - **Vysoká zložitosť** - Vývojár musí namiesto jednoduchých volaní metód riešiť komplexné problémy sieťovej komunikácie a súbežného spracovania
  - **Nespoľahlivosť siete** - Distribuované systémy trpia neduhmi, ako sú latencia, výpadky spojenia, bezpečnostné riziká a obmedzená prenosová rýchlosť
  - **Výkonnostná réžia** - Sieťová komunikácia (napr. HTTP) je výrazne pomalšia a náročnejšia na zdroje než volanie metódy v rámci jedného procesu
  - **Náročnosť zmien pri zlom návrhu** - Ak sú hranice medzi službami navrhnuté nevhodne a služby musia komunikovať príliš často, dodatočná oprava takéhoto rozdelenia je veľmi drahá a komplikovaná

38 Aký je význam záznamov o architektonických rozhodnutiach (Architecture Decision Records) a aké položky by mali obsahovať

- Význam Architecture Decision Records (ADR) spočíva v zaznamenávaní dôležitých architektonických rozhodnutí a ich kontextu pre potreby tímu a budúcnosť. Umožňujú

spätne pochopiť, prečo bolo dané rozhodnutie prijaté, a v prípade zmeny kontextu ho efektívne prehodnotiť

- **Záznam ADR by mal obsahovať tieto položky:**
  - **Title (Názov)** - jasné pomenovanie rozhodnutia
  - **Context (Kontext)** - opis problému, situácie, požiadaviek a zvažovaných alternatív (výhody/nevýhody)
  - **Decision (Rozhodnutie)** - špecifikácia zvoleného riešenia a dôvod jeho výberu
  - **Status (Stav)** - aktuálny stav rozhodnutia (napr. navrhované, prijaté, zamietnuté alebo nahradené)
  - **Consequences (Následky)** - stručné zhrnutie pozitívnych aj negatívnych dopadov prijatého rozhodnutia

39 Čo je formálna metóda a z čoho pozostáva? Aký je vzťah medzi formálnymi metódami a programovacími jazykmi? Aké sú výhody a nevýhody formálnych metód?

- **Čo je formálna metóda a z čoho pozostáva:**
  - Formálne metódy sú techniky založené na matematickom základe, ktoré slúžia na špecifikáciu, vývoj, analýzu a verifikáciu počítačových systémov
  - **Každá formálna metóda pozostáva z dvoch kľúčových zložiek:**
    1. **Formálny jazyk** - má jednoznačne definovanú syntax a sémantiku (pomocou matematického aparátu, ako je logika alebo teória množín)
    2. **Súbor procedúr** - nástroje a postupy, ktoré umožňujú so zápisom v danom jazyku pracovať (napr. redukcia, syntéza alebo vyhľadávanie)
- **Vzťah medzi formálnymi metódami a programovacími jazykmi:**
  - Hlavný rozdiel spočíva v tom, že sémantika jazykov formálnych metód je definovaná matematicky, zatiaľ čo bežné programovacie jazyky majú sémantiku opísanú v prirodzenom jazyku, čo môže viesť k nejednoznačnej interpretácii
  - Hoci programovacie jazyky majú presnú sémantiku v podobe strojového kódu, ten je pre používateľa nečitateľný. Niektoré formálne metódy (napr. B-metóda) sú priamo určené na to, aby pomocou procesu zjemňovania umožnili prechod od abstraktnej špecifikácie až k spustiteľnému kódu v jazykoch ako C alebo Java
- **Výhody a nevýhody formálnych metód:**
  - **Výhody:**
    * **Jednoznačnosť** - Umožňujú vytvoriť presný opis systému už v skorých fázach vývoja
    * **Efektivita pri hľadaní chýb** - Odstránenie chýb na úrovni špecifikácie je výrazne lacnejšie ako počas implementácie
    * **Vysoká kvalita** - Poskytujú rigorózne techniky na overenie, či systém spĺňa požadované vlastnosti

- **Nevýhody:**
  - **Náročnosť na vzdelanie** - Vývojári si musia osvojiť nový jazyk a prácu s ním, čo si často vyžaduje prítomnosť experta
  - **Výpočtové limity** - Automatické nástroje (napr. overovače modelov) majú vysoké nároky na pamäť a čas
  - **Nedokonalá automatizácia** - Pri niektorých prístupoch (napr. dokazovače teorém) je nevyhnutný zásah používateľa a dôkaz nie je úplne automatický

40 Čo rozumieme pod pojmom vyjadrovacia sila formálnej metódy? Na aké úrovne vieme rozdeliť formálne metódy?

- Pod pojmom vyjadrovacia sila rozumieme schopnosť formálneho jazyka opísať určitý typ systému, pričom sa táto sila bežne porovnáva voči Turingovmu stroju
- **Úrovne formálnych metód Podľa J.P. Bowena delíme formálne metódy (resp. ich použitie) na 3 úrovne:**
  1. **Úroveň 0 (Formálna špecifikácia)** - Použitie formálnej notácie (jazyka) na vytvorenie jednoznačného opisu systému, pričom dostupné procedúry slúžia najmä na kontrolu syntaxe alebo simuláciu
  2. **Úroveň 1 (Formálna verifikácia alebo vývoj)** - Využitie rigoróznych techník na overenie vlastností systému alebo na prechod od abstraktnej špecifikácie k implementovateľnej podobe
  3. **Úroveň 2 (Počítačom vykonaný dôkaz)** - Najvyššia úroveň, kde sa využívajú softvérové nástroje (overovače modelov alebo dokazovače teorém) na realizáciu samotných formálnych dôkazov

41 Ako je formálne definovaná syntax a sémantika Petriho sietí?

- **Syntax (Štruktúra) Petriho sieť je formálne definovaná ako usporiadaná štvorica N=(P,T,pre,post):**
  - **P** - je konečná množina miest (graficky krúžky)
  - **T** - je konečná množina prechodov (graficky obdĺžniky)
  - pre a post sú funkcie ($P \times T \to N$), ktoré definujú orientované hrany medzi miestami a prechodmi a ich váhy
- **Sémantika (Správanie) Sémantika opisuje, ako sa sieť vyvíja v čase prostredníctvom zmien stavov:**
  - **Značenie (m)** - Funkcia $m:P \to N$, ktorá definuje distribuovaný stav siete priradením počtu značiek (tokenov) jednotlivým miestam
  - **Vykonateľnosť prechodu** - Prechod je vykonateľný v značení m, ak v každom jeho vstupnom mieste je aspoň toľko značiek, koľko vyžaduje váha hrany definovaná funkciou pre
  - **Vykonanie (odpálenie) prechodu** - Ak sa prechod vykoná, dôjde k zmene značenia — z jeho vstupných miest sa značky odoberú (podľa pre) a do jeho výstupných miest sa značky pridajú (podľa post)

42 Opíšte postup výpočtu invariantov v Petriho sieťach (môžete aj na príklade)

- Postup výpočtu invariantov v Petriho sieťach (konkrétne v P/T sieťach) je založený na riešení sústav lineárnych rovníc odvodených z grafu siete
- **Rozlišujeme dva základné typy:**
  - **S-invarianty (miestové)** - Charakterizujú nemenný pomer medzi značeniami miest v každom dosiahnuteľnom stave
  - **T-invarianty (prechodové)** - Charakterizujú sekvencie prechodov, ktoré sieť vrátia do pôvodného značenia
- **Postup výpočtu:**
  1. **Zostavenie incidenčnej matice C** - Matica C je definovaná ako rozdiel post-matice a pre-matice (C=PostPre)
  2. **Sformulovanie sústavy homogénnych lineárnych rovníc:**
     - Pre S-invarianty hľadáme stĺpcový vektor X, pre ktorý platí: CT * X=0
     - Pre T-invarianty hľadáme stĺpcový vektor Y, pre ktorý platí: C * Y=0
  3. **Riešenie sústavy** - Sústava má zvyčajne nekonečne veľa riešení, preto sa hľadajú tzv. základné invarianty, ktoré predstavujú minimálne nezáporné celočíselné riešenia tvoriace bázu
  4. **Interpretácia (Axiómy)** - Získané S-invarianty sa prepisujú do rovníc (axióm) v tvare m * X=m_0 * X, ktoré platia pre každé dosiahnuteľné značenie m

43 Čo je B-Metóda? Aké špecifikačné komponenty sa v nej používajú? Ako je definovaný vývojový proces v B-Metóde?

- B-metóda je formálna metóda orientovaná na vývoj softvéru a sekvenčných systémov, ktorá je postavená na matematických základoch klasickej logiky a teórie množín. Umožňuje vytvoriť formálnu špecifikáciu systému a pomocou matematických dôkazov ju postupne transformovať až do podoby spustiteľného kódu
- **V B-metóde sa používajú tri základné typy komponentov, ktoré tvoria sekvenciu vývoja:**
  1. **Abstraktný stroj (Machine)** - Základná jednotka špecifikácie, ktorá zapuzdruje stavové premenné a operácie; podobá sa triede v objektovom programovaní
  2. **Zjemnenie (Refinement)** - Komponent, ktorý predstavuje medzikrok v návrhu, kde sa abstraktné údaje a operácie konkretizujú
  3. **Implementácia (Implementation)** - Posledný článok reťazca, z ktorého je už možné priamo generovať zdrojový kód v programovacom jazyku (napr. C, Java)
- **Vývojový proces je definovaný ako rigorózny cyklus pozostávajúci z týchto fáz:**
  1. **Vývoj špecifikácie** - Formalizácia požiadaviek do abstraktných strojov, ich validácia animáciou a overenie vnútornej konzistencie pomocou povinných dôkazov (POb)

2. **Návrh (Zjemňovanie)** - Postupná transformácia abstraktnej špecifikácie cez sériu zjemnení až po implementáciu, pričom každý krok sa matematicky overuje (POb), či zachováva vlastnosti predchádzajúcej úrovne
3. **Generovanie kódu** - Automatický prevod finálnych implementácií do cieľového programovacieho jazyka

44 Čo je zovšeobecnená substitúcia (generalised substitution)? Ako je formálne definovaná sémantika zovšeobecnenej substitúcie?

- **Čo je zovšeobecnená substitúcia:**
  - Zovšeobecnená substitúcia (GS) je základný konštrukt jazyka GSL (Generalized Substitution Language), ktorý sa využíva v B-metóde na zápis operácií. Jej hlavným cieľom je poskytnúť jednotný formálny zápis systému v celom procese vývoja – od vysoko abstraktných strojov až po konkrétny procedurálny kód v implementácii. Medzi GS patria príkazy ako jednoduché priradenie, prázdna substitúcia (skip), viazaná voľba, pre-podmienka či cyklus
- **Ako je formálne definovaná sémantika zovšeobecnenej substitúcie:**
  1. Sémantika GS je definovaná pomocou predikátových transformérov a kalkulu najslabšej pre-podmienky (Weakest precondition calculus)
  2. Formálne sa zapisuje ako [S]P, čo označuje najslabšiu pre-podmienku, pri ktorej výpočet substitúcie S zaručene skončí (terminuje) a výsledný stav bude spĺňať predikát P. Pre základnú substitúciu (priradenie x:=e) je sémantika definovaná ako: $[x := e]P \equiv P[x := e]$ (textuálna substitúcia výrazu e za premennú x v predikáte P).