Metaprogramování

Multiparadigmatický přístup v softwarovém inženýrství

Kyjev, 2015 — 2025

Anotace

Všechny programy jsou data. Některá data jsou interpretována jako hodnoty, jiná jako typy těchto hodnot a další jako instrukce pro zpracování prvních dvou. Jakákoli paradigmata a programovací techniky jsou jen způsobem, jak vytvořit metadata, která dávají pravidla a posloupnost toku zpracování jiných dat. Multiparadigmatické programování bere to nejlepší ze všech paradigmat a sestavuje z něj syntaktické konstrukce, které umožňují popsat předmětnou oblast jasněji a pohodlněji. Vysokoúrovňové DSL (doménové jazyky) promítáme do nízkoúrovňových strojových instrukcí prostřednictvím mnoha vrstev abstrakcí. Zde je důležité reprezentovat úkol co nejúčinnějším způsobem pro zpracování na úrovni stroje, nikoli fanaticky následovat jedno paradigma. Nejúčinnější je ten, který s menším počtem vrstev a závislostí, nejlépe čitelný, udržovatelný a upravitelný pro člověka, který zajišťuje spolehlivost kódu a testovatelnost, rozšiřitelnost, opětovnou použitelnost, jasnost a flexibilitu konstrukcí metadat na každé úrovni. Věříme, že takový přístup nám umožní získat jak rychlé první výsledky ve vývoji, tak i neztrácet výkon při velkém toku změn ve fázích, kdy projekt již dosáhl vysoké zralosti a složitosti. Pokusíme se zvážit techniky a principy různých programovacích paradigmat prizmatem metaprogramování a změnit jestli ne samotné softwarové inženýrství tím, tak aspoň rozšířit pochopení disciplíny novými generacemi profesionálů.

Obsah

1. Úvod
   1.1. Přístup k výuce programování
   1.2. Příklady v jazycích JavaScript, Python a C
   1.3. Modelování: abstrakce a opětovné použití
   1.4. Algoritmus, program, syntaxe, jazyk
   1.5. Dekompozice a separace odpovědnosti
   1.6. Přehled specializace softwarového inženýra
   1.7. Přehled programovacích paradigmat
2. Základní pojmy
   2.1. Hodnota, identifikátor, proměnná a konstanta, literál, přiřazení
   2.2. Datové typy, skalární, referenční a strukturované typy
   2.3. Operátor a výraz, blok kódu, funkce, smyčka, podmínka
   2.4. Kontexty a lexikální rozsah
   2.5. Procedurální paradigma, volání, zásobník a halda
   2.6. Funkce vyššího řádu, čistá funkce, vedlejší účinky
   2.7. Uzávěry, funkce zpětného volání, zabalení a události
   2.8. Výjimky a řešení chyb
   2.9. Úkoly
3. Stav aplikace, datové struktury a kolekce
   3.1. Stavové a bezstavové přístupy (stateful and stateless)
   3.2. Struktury a záznamy
   3.3. Pole, seznam, sada, n-tice
   3.4. Slovník, hashovací tabulka a asociativní pole
   3.5. Zásobník, fronta, deque
   3.6. Stromy a grafy
   3.7. Projekce a zobrazení datových
   3.8. Odhad výpočetní složitosti
4. Rozšířené koncepty
   4.1. Co je technologický stack
   4.2. Vývojové prostředí a ladění

4.3. Iterace: rekurze, iterátory a generátory
4.4. Stavební bloky aplikací: soubory, moduly, komponenty
4.5. Objekt, prototyp a třída
4.6. Částečná aplikace a curryfikace, skládání funkcí
4.7. Řetězení pro metody a funkce (chaining)
4.8. Mixiny (mixins)
4.9. Závislosti a knihovny
5. Běžná programovací paradigmata
5.1. Imperativní a deklarativní přístup
5.2. Strukturované a nestrukturované programování
5.3. Procedurální programování
5.4. Funkcionální programování
5.5. Objektově orientované programování
5.6. Programování založené na prototypech
6. Návrhové antivzory
6.1. Společné anti-vzory pro všechna paradigmata
6.2. Procedurální antivzory
6.3. Objektově orientované antivzory
6.4. Funkční antivzory
7. Vývojový proces
7.1. Životní cyklus softwaru, analýza předmětné oblasti
7.2. Programovací konvence a normy
7.3. Testování: jednotkové testy, systémové a integrační testování
7.4. Kontrola a refaktoring kódu
7.5. Odhad zdrojů, plán rozvoje a harmonogram
7.6. Analýza rizik, slabá místa, nefunkční požadavky
7.7. Koordinace a úprava procesů
7.8. Průběžná integrace a nasazení
7.9. Optimalizace mnoha aspektů
8. Pokročilé koncepty
8.1. Události, časovače a EventEmitter
8.2. Introspekce a reflexe
8.3. Serializace a deserializace

8.4. Regulární výrazy
8.5. Memoizace
8.6. Návrhové vzory: Factory, Poll
8.7. Typovaná pole
8.8. Projekce
8.9. I/O (vstup-výstup) a soubory
9. Architektura
9.1. Dekompozice, pojmenování a spojování
9.2. Interakce mezi softwarovými komponentami
9.3. Propojení přes jmenné prostory
9.4. Interakce s voláními a zpětnými voláními
9.5. Interakce s událostmi a zprávami
9.6. Rozhraní, protokoly a smlouvy
9.7. Cibulová (onion) nebo vícevrstvá architektura
10. Základy paralelních výpočtů
10.1. Asynchronní programování
10.2. Paralelní programování, sdílená paměť a synchronizační primitiva
10.3. Asynchronní primitiva: Thenable, Promise, Future, Deferred
10.4. Koprogramy, gorutiny, async/await
10.5. Adaptéry mezi asynchronními kontrakty
10.6. Asynchronní a paralelní kompatibilita
10.7. Přístup předávání zpráv a model aktorů
10.8. Asynchronní fronty i asynchronní kolekce
10.8. Bezzámkové datové struktury (lock-free)
11. Pokročilá programovací paradigmata
11.1. Generické programování
11.2. Událostní a reaktivní programování
11.3. Programování automatů: konečné automaty (stavové stroje)
11.4. Jazykově orientované programování a DSL
11.5. Programování toku dat
11.6. Metaprogramování
11.7. Dynamická interpretace metamodelu

12. Databáze a perzistentní úložiště

12.1. Historie databáze a navigační databáze

12.2. Klíč-hodnota a další abstraktní datové struktury

12.3. Relační datový model a ER-diagramy

12.4. Bezschémové, objektově orientované a dokumentově orientované databáze

12.5. Hierarchické a grafové databáze

12.6. Sloupcové databáze a databáze v paměti

12.7. Distribuované databáze

13. Distribuované systémy

13.1. Meziprocesová komunikace

13.2. Bezkonfliktní replikované datové typy (CRDT)

13.3. Konzistence, dostupnost a distribuce

13.4. Strategie řešení konfliktů

13.5. Konsensuální protokoly

13.6. CQRS, EventSourcing

1. Úvod

Neustálé přehodnocování své činnosti, i té nejjednodušší, by mělo provázet inženýra celý život. Zvyk zapisovat si své myšlenky slovy a zdokonalovat své formulace v tom hodně pomáhá. Tento text se objevil jako moje fragmentární poznámky, psané v různých letech, které jsem shromažďoval a mnohokrát kriticky korigoval. Já jsem často nesouhlasil sám se sebou, když jsem si znovu četl pasáž poté, co chvíli ležela. Proto jsem na textu pracoval tak dlouho, dokud jsem sám nesouhlasil s tím, co bylo napsáno po dlouhé době držení materiálu. Bylo mým úkolem psát co nejstručněji a velké fragmenty jsem opakovaně přepisoval, když jsem zjistil, že by se daly vyjádřit stručněji. Struktura textu a obsah se začaly objevovat po prvním roce výuky, ale v desátém ročníku jsem se rozhodl šířit materiály nejen ve formě otevřených videopřednášek, jak jsem to dělal už asi pět let, ale i ve formě textu. To umožnilo všem ze společenství Metarhia podílet se na tvorbě knihy, díky čtenářům rychle objevovat překlepy a nepřesnosti, a také pro mnohé je formát knihy prostě pohodlnější. Aktuální verzi naleznete na https:// github.com/ HowProgrammingWorks/ Book, bude neustále aktualizována. Žádosti o opravy a doplnění směřujte v angličtině na [issues] (https:// github.com/ HowProgrammingWorks/ Book/ issues), nové nápady zasílejte v libovolném jazyce na [discussions] (https:// github.com/ HowProgrammingWorks/ Book/ discussions), vaše vlastní doplňky a opravy by měly být provedeny formou pull-request do repositáře knihy.

1.1. Přístup k výuce programování

Většina si myslí, že základní dovedností programátora je psaní kódu. Ve skutečnosti programátoři nejčastěji kód čtou a opravují ho. A hlavními kritérii kvality kódu jsou srozumitelnost, čitelnost a jednoduchost. Jak řekl Harold Abelson: "Programy musí být napsány pro lidi, kteří je budou číst, a stroje, které budou tyto programy spouštět, jsou vedlejší."

Hlavní dovednosti programátora jsou čtení a úprava kódu

Každé téma obsahuje příklady dobrého a špatného kódu. Tyto příklady jsou převzaty z praxe programování a posuzování projektů. Speciálně připravené příklady špatného kódu budou funkční, ale jsou plné návrhových antivzorců a problémů, které je třeba identifikovat a opravit. Již první praktická práce v kurzu bude souviset s opravou kódu a

zvýšením jeho čitelnosti. Jestli zadávat tradiční úkoly (napsat funkce podle signatury, algoritmu, třídy), začátečník to samozřejmě neimplementuje tím nejlepším způsobem, bude bránit svůj kód, protože je to první věc, kterou napsal. A pokud je úkolem "vzít si příklad cizího špatného kódu, najít problémy a opravit", nikoli nepřepisovat od nuly, ale vylepšovat v několika krocích, fixovat tyto kroky a uvědomovat si, pak se zapíná kritický přístup.

Oprava špatného kódu je jedním z nejúčinnějších způsobů, jak se učit

Začátečník obdrží příklady kontroly kódu a analogicky se snaží opravit i svůj úkol. Takové iterace se mnohokrát opakují bez ztráty kritického postoje. Je velmi dobré mít mentora, který sleduje zlepšení a může korigovat a navrhovat. Mentor by ale v žádném případě neměl dělat práci za začátečníka, ale tlačit ho na to, jak o programování přemýšlet a kde hledat řešení.

Mentor je nepostradatelný v jakékoli fázi profesního růstu

Dále budou úkoly psaní vlastního kódu. Důrazně doporučujeme začátečníkům, aby tato řešení vzájemně sdíleli za účelem křížové revize. Samozřejmě před tím musíte použít lintry a formátovače kódu, které analyzují syntaxi, najdou chyby a identifikují problémové oblasti pro velké množství šablon kódu. Je velmi důležité zajistit, aby kolega porozuměl vaší myšlence a neztrácel čas syntaxí a formátováním.

Používejte přátelské code review, křížové kontroly, lintery a formátovače

Přecházíme ke cvičením zredukování zaháknutí mezi několika abstrakce, poté mezi moduly, tedy tak, abyste o datových strukturách jedné části programu věděli co nejméně od jiné jeho části. Snížení jazykového fanatismu je dosaženo paralelním učením několika programovacích jazyků od začátku a překlady z jednoho jazyka do druhého. Překlad z **JavaScript** do **Python** je velmi snadný. Co se týče **C**, je to o něco těžší, ale tyto tři jazyky, ať už jsou jakékoli, nelze z kurzu vynechat.

Od prvních kroků nepřipouštějte žádný fanatismus: jazykový,

Omezení frameworkového fanatismu – pro začátečníky to znamená zakáz používat knihovny a frameworky a místo toho se zaměřit na nativní kód bez závislostí. Redukce paradigmatického fanatismu je pokusem o spojení procedurálního, funkčního, OOP, reaktivního a automatického programování. Pokusíme se ukázat, jak nám tyto kombinace umožňují zjednodušit návrhové vzory a principy z GoF a SOLID.

Další důležitou součástí kurzu je studium antipatternů a refaktoringu. Nejprve si dáme přehled a poté si procvičíme na reálných příkladech kódu z živých projektů.

1.2. Příklady v jazycích JavaScript, Python a C

Příklady kódu budeme psát v různých jazycích, ale přednost nebudou mít ty nejlepší z ních, krásné a rychlé, ale ty, bez kterých se nelze obejít. **JavaScript** budeme brát jako nejrozšířenější, **Python**, protože jsou oblasti, kde to bez něj nejde, a jazyk **C**, ten je docela blízký assembleru a je stále aktuální a má největší vliv na moderní jazyky z pohledu syntaxe a vložených do něj myšlenek. Všechny tři mají k mému vysněnému jazyku hodně daleko, ale tohle je to, co máme. **Python** se na první pohled velmi liší od **JavaScript** a dalších **C-podobných** jazyků, i když je to jen na první pohled, ukážeme si, že je **JavaScriptu** velmi podobný právě díky tomu, že typový systém, datové struktury a zejména vestavěné v nich kolekce jsou skoro stejné. Syntakticky rozdíl v organizaci bloků kódu pomocí odsazení a složených závorek **{}** je markantní, ale ve skutečnosti takový rozdíl není příliš významný a **JavaScript** a **Python** mají mnohem více společného, než oba mají s jazykem **C**.

Nezačneme učením syntaxe, ale rovnou čtením špatného kódu a hledáním tam chyb. Podívejme se na následující úryvky, první bude v jazyku **JavaScript**:

```
let first_num = 2;
let secord_num = 3;
let sum = firstNum + secondNum;
console.log({ sum });
```

Pokuste se porozumět tomu, co je zde napsáno a kde mohou být chyby. A pak porovnejte tento kód s jeho překladem do **C-ečka**:

```
#include <stdio.h>

int main() {
  int first_num = 2;
  int secord_num = 3;
  int sum = firstNum + secondNum;
  printf("%d\n", sum);
}
```

Chyby jsou zde stejné, snadno je pozná i člověk, který nezná ani základy programování, pokud si ovšem kód prozkoumá. Další část kódu bude v **Pythonu**, dělá to samé a se stejnými chybami.

```
first_num = 2;
secord_num = 3;
sum = firstNum + secondNum;
print({ 'sum': sum  });
```

Dále budeme často porovnávat příklady kódu v různých jazycích, hledat a opravovat chyby, optimalizovat kód a zlepšovat především jeho čitelnost a srozumitelnost.

1.3. Modelování: abstrakce a opětovné použití

Srdcem programování je modelování, tedy vytvoření modelu řešení problému neboli modelu objektů a procesů v paměti počítačů. Programovací jazyky poskytují syntaxi pro navrhování omezení při vytváření modelů. Jakákoli konstrukce a struktura navržená pro rozšíření funkčnosti a zavedená do modelu vede k dalším omezením. Zvýšení úrovně abstrakce může naopak některá omezení odstranit a snížit složitost modelu a programového kódu, který tento model vyjadřuje. Neustále balancujeme mezi rozšiřováním funkcí a jejich sbalováním do zobecněného modelu. Tento proces může a měl by být iterativní.

Člověk překvapivě dokáže úspěšně řešit problémy, jejichž složitost přesahuje možnosti jeho paměti a myšlení, pomocí sestavení modelů a abstrakcí. Přesnost těchto modelů určuje jejich užitečnost pro rozhodování a rozvoj kontrolních akcí. Model je vždy nepřesný a odráží pouze malou část reality: jednu nebo více jejích stran či aspektů. V omezených podmínkách použití však může být model shodný s reálným

objektem předmětné oblasti. Existují fyzikální, matematické, simulační a další modely, ale nás budou zajímat především informační a algoritmické modely.

Abstrakce je technika zobecnění, která redukuje mnoho různých, ale podobných případů na jediný model. Zajímají nás datové abstrakce a abstraktní algoritmy. Nejjednoduššími příklady abstrakce v algoritmech jsou cykly (iterativní zobecnění) a funkce (procedury a podprogramy). Pomocí cyklu můžeme popsat mnoho iterací jedním blokem příkazů za předpokladu jeho vícenásobného opakování, s různými hodnotami proměnných. Funkce se taky mnohokrát opakují s různými argumenty. Příkladem abstrakce dat jsou pole, asociativní pole, seznamy, množiny atd. V aplikacích je potřeba abstrakce kombinovat ve vrstvách, nazýváme je abstraktní vrstvy. Nízkoúrovňové abstrakce jsou zabudovány do programovacího jazyka (proměnné, funkce, pole, události). Abstrakce vyšší úrovně jsou obsaženy ve frameworkech, runtimech, standardních knihovnách a externích knihovnách, nebo si je můžete vytvořit sami z jednoduchých abstrakcí. Abstrakce se tak nazývají, protože řeší zobecněné abstraktní úlohy nesouvisející s předmětnou oblastí.

Budování abstraktních vrstev je téměř nejdůležitější programovací úkol, jehož úspěšné řešení určuje takové vlastnosti softwarového řešení, jako je flexibilita přizpůsobení, snadná modifikace, schopnost integrace s jinými systémy a životnost řešení. Všechny vrstvy, které nejsou vázány na předmět a konkrétní aplikované úkoly, se budou nazývat systémové vrstvy. Nad systémovými vrstvami programátor staví aplikační vrstvy, jejichž abstrakce se naopak snižuje, univerzálnost klesá a aplikace se stává specifičtější, vázána na konkrétní úkoly.

Abstrakce různých úrovní mohou být umístěny jak v jednom adresním prostoru (jeden proces nebo jedna aplikace), tak i v různých. Je možné je od sebe oddělit a zavést mezi nimi interakci pomocí programovacích rozhraní, modularity, komponentního přístupu a jednoduše silou vůle, vyhnout se přímým voláním ze středu jedné softwarové komponenty doprostřed jiné, pokud programovací jazyk nebo platforma se nestará o to, aby takové schopnosti zabránily. To by mělo být provedeno i v rámci jednoho procesu, kde by jakékoli funkce, komponenty a moduly mohly být přístupné z jakéhokoli jiného, i když logicky patří do různých vrstev. Důvodem je potřeba snížit vzájemnou závislost mezi vrstvami a softwarovými komponentami, což umožňuje, aby byly zaměnitelné, znovu použitelné a mohly být vyvíjeny samostatně. Zároveň je nutné zvýšit konektivitu v rámci vrstev, komponent a modulů, což zajišťuje růst výkonu kódu, jeho snadné čtení, pochopení a úpravy. Pokud se dokážeme

vyhnout propojenosti mezi různými úrovněmi abstrakce a použít dekompozici, abychom zajistili, že jeden modul může být vždy zcela pokryt pozorností jednoho inženýra, pak se vývojový proces stane škálovatelným, řiditelným a předvídatelnějším. Podobná myšlenka je základem architektury mikroslužeb, ale obecnější princip platí pro jakýkoli systém, a nezáleží na tom, zda se jedná o samostatně běžící mikroslužby nebo moduly běžící ve stejném procesu.

Je třeba poznamenat, že čím lépe je systém distribuován, tím lépe je centralizován. Protože řešení problémů v takových systémech probíhá na adekvátních úrovních, kde je již dostatek informací pro rozhodování, zpracování a získávání výsledků, nedochází k rigidnímu propojení modelů různých úrovní abstrakce. Díky tomuto přístupu se vyhneme zbytečným eskalacím úkolu na vyšších úrovních, je zabráněno "přehřívání" rozhodovacích uzlů, minimalizován přenos dat a zvýšena provozní rychlost.

1.4. Algoritmus, program, syntaxe, jazyk

Existuje mnoho termínů souvisejících s programováním. Pro jistotu bychom měli zjistit rozdíl mezi nimi. Je lepší zaměřit se na neformální porozumění než na formální definici. Nejstarším pojmem je zde algoritmus, který si všichni pamatujeme ze školního kurzu matematiky. Například Euklidův algoritmus pro nalezení největšího společného dělitele dvou celých čísel.

Algoritmus

Algoritmus ještě není program, je to nápad na řešení problému, popsaný formálně. Tak, aby to bylo pro ostatní srozumitelné, ověřitelné a realizovatelné. Algoritmus nelze spustit, lze jej převést do kódu v některém programovacím jazyce. Algoritmus obsahuje popis operací a může být zapsán různými způsoby: vzorcem, vývojovým diagramem, seznamem akcí v lidské řeči. Vždy se omezuje na určitou třídu problémů, které řeší v konečném čase. Zúžením třídy problémů často můžeme zjednodušit a optimalizovat algoritmus. Například přechodem ze sčítání součtu celých a zlomkových čísel ke sčítání součtu pouze celých čísel můžeme provést efektivnější implementaci. Lze také rozšířit i třídu problémů pro tento příklad, což umožní rovněž zadávaní do vstupu řetězcové reprezentace čísel. Díky tomu bude algoritmus všestrannější, ale méně účinný. Musíme si vybrat, co přesně optimalizujeme. V tomto

případě je lepší rozdělit algoritmus na dva, jeden převede všechna čísla na požadovaný datový typ a druhý bude sčítat.

Příklad implementace algoritmu NSD

Euklidův algoritmus pro nalezení NSD (největšího společného dělitele) v **JavaScript** lse zapsat následovně:

```
const gcd = (a, b) => {
  if (b === 0) return a;
  return gcd(b, a % b);
};
```

Nebo ještě kratší formou, ale nebude to o nic méně jasné, pokud porovnáte tyto dvě možnosti a zkontrolujete, které konstrukce jsou nahrazeny:

```
const gcd = (a, b) => (b === 0 ? a : gcd(b, a % b));
```

Tento jednoduchý algoritmus je rekurzivní, tzn. zavolá sám sebe, aby vypočítal další krok, a skončí, když **b** dosáhne **0**. U algoritmů můžeme určit výpočetní náročnost, klasifikovat je podle zdrojů procesorového času a paměti potřebných k řešení problému.

Program

V předchozím příkladu jsme se zabývali funkcí, ta je součástí programu, ale aby fungovala, je třeba funkci zavolat a předat jí data. Programový kód a data, spojené do jednoho celku, tvoří program. Niklaus Wirth, autor mnoha jazyků včetně Pascalu, má knihu s názvem "Algoritmy + Datové struktury = Programy". Její název obsahuje velmi důležitou pravdu, hluboce vtisknutou nejen do světonázoru čtenářů, ale i do názvů kurzů předních univerzit a dokonce i do pohovorů, kde se od uchazeče vyžaduje, aby se na tyto dvě věci soustředil. V prvních 50 letech softwarového průmyslu se ukázalo, že datové struktury jsou stejně důležité jako algoritmy. Mnoho profesionálních programátorů věnuje strukturám více času. Linus Torvalds k tomu má trefnou poznámku: "Špatní programátoři si dělají starosti s kódem. Dobří programátoři se starají o datové struktury a vztahy mezi nimi." Faktem je, že výběr datových struktur do značné míry

určuje, jaký bude algoritmus, omezuje jej v rámci výpočetní náročnosti a sémantiky úlohy, kterou programátor rozumí prostřednictvím dat rozložených v paměti mnohem lépe než pomocí sledu operací.

Eric Raymond to vyjádřil takto: "Inteligentní datové struktury a hloupý kód fungují mnohem lépe než naopak."

Kód umožňuje najít společný jazyk

Ten samý Linus Torvalds nám však také řekl: "Klábosení nestojí za nic. Ukaž mi kód." To není v rozporu s tím, co bylo řečeno výše. Myslím, že zde měl na mysli to, že programový kód nepřipouští dvojznačnost. Programový kód je univerzální metoda, která pomáhá programátorům najít společnou řeč právě tehdy, když tomu přirozené jazyky svou nejednoznačností zabraňují, pak se stačí na kód podívat.

Inženýrství

Využití dostupných zdrojů prostřednictvím vědy, technologie, různých metodik, organizační struktury, technik a znalostí je další úroveň neboli inženýrství.

Pamatuji si, jak pro mě v prvních letech učení programování bylo důležité, aby můj kód používali lidé, zlepšoval jejich život a žil dlouho. Olympiáda se mi zdála nezajímavá, studijní úkoly byly příliš vymyšlené, chtěl jsem se soustředit na to, co budou lidé každý den provozovat na svých počítačích: databázové aplikace, formuláře a tabulky, síťové a komunikační aplikace, programy pro ovládání zařízení, práce se senzory a mnoho nástrojů pro samotné programátory.

Stejně jako v jiných inženýrských oborech je i v programování velmi důležitý přínos pro člověka a ne správnost nebo harmonie konceptu. Inženýrství je povoláno k využívání vědeckých úspěchů, a tam, kde dnes dostupné vědecké poznatky jsou nedostačující, inženýrství uplatňuje intuici, inženýrskou kulturu, pokusy a omyly, využívání nevědomých zkušeností a zkušeností, které nemají dostatečné vědecké porozumění.

To je výhoda i nevýhoda inženýrství. Máme mnoho různých a kontroverzních řešení stejného problému, ne vždy víme, proč něco nefunguje, ale to je v pořádku, občas se divíme, proč něco funguje. Tento přístup vede k hromadění špatných postupů v projektech a takovému prolínání dobrých a špatných praktik, že je velmi obtížné je oddělit a často

naše úsilí vynakládáno opakovaně na již vyřešené úkoly. Niklaus Wirth řekl: "Programy se stávají pomalejšími rychlejí, než hardware se stává rychlejším" a často považujeme, že je snazší program přepsat, než v něm opravovat chyby.

Softwarové inženýrství

Aplikace inženýrství v softwarovém průmyslu zahrnuje architekturu, výzkum, vývoj, testování, nasazení a údržbu softwaru.

Softwarový průmysl se vyvinul do silného průmyslu, přerostlého podpůrnými technologickými postupy, které snižují dopad jeho již výše uvedených nedostatků a činí konečný produkt dostatečně spolehlivým, aby přinášel zisk, ale ne dostatečně kvalitní, aby mohly být vydávány další a další verze.

"Většina softwaru je dnes jako egyptské pyramidy z milionu cihel na sobě a bez strukturální integrity – jsou postaveny jen hrubou silou a tisíci otroků" // Alan Kay

Programování

Programování je tedy umění a inženýrství řešení problémů pomocí počítačů. Zde je důležité poznamenat, že výpočetní technika výrazně ovlivňuje to, jak programujeme, určuje, která paradigmata a přístupy budou fungovat efektivněji, a poskytne výsledek, který máme k dispozici, a to jak z hlediska zdrojů vynaložených na programování, tak z hlediska výpočetních zdrojů potřebných ke spuštění vytvořených programů.

Kódování

Pokud od programování oddělíme pouze psaní zdrojového kódu programu pomocí určité syntaxe (jazyka), stylu a paradigmatu podle hotového technického úkolu (TÚ), pak toto nazýváme kódování, i když slovo lze považovat za zastaralé.

Vývoj lze rozdělit na návrh a kódování, což z dlouhodobého hlediska poskytuje efektivnější uplatnění sil, ale často musíte začít programovat bez TÚ a bez předběžného návrhu. Takto vyvinuté systémy se nazývají prototypy, MVP (minimum viable product), pilotní systémy nebo stojany. Jejich přínos spočívá v testování hypotéz o užitečnosti pro spotřebitele či

ekonomické efektivitě jejich použití.

Programátor si ne vždy uvědomuje, co dělá, prototyp nebo produkt, a nakonec dostaneme prototyp, který je skoro stejně dobrý jako hotový produkt, nebo hotový produkt vyrobený jako dočasné řešení. Jsou však nadšenci, kteří svou práci milují, a právě na nich se toto odvětví drží, kontroverzní a plné problémů.

"Většina dobrých programátorů nedělá svou práci proto, že očekávají, že dostanou zaplaceno nebo uznání, ale protože je baví programování." // Linus Torvalds

Vývojář vs programátor

Každé z jmen profesí má své příznivce, často sebejmenování vývojář nebo programátor je spojeno se zvláštní profesní hrdostí až arogantním přístupem k příznivcům jiného jména profese. Bylo by dobré tyto profese oddělit zhruba stejně, jako se rozdělovaly profese řidiče a automechanika. Automechanik samozřejmě může říct, že řidiči nerozumí autům, ale právě řidiči masově vozí lidi. Podobně v IT se programátor musí soustředit na abstrakce a tvorbu softwarových komponent, zatímco vývojář by se měl zaměřit na použití hotových komponent k řešení úkolu, který vyžaduje jiné znalosti a dovednosti než programování.

Rozdíl mezi programátorem a vývojářem lze ukázat na příkladu tvorby informačních systémů (IS). Ve světě je potřeba, aby masová výroba IS uspokojila potřeby průmyslu, dopravy, služeb, logistiky, obchodu, medicíny atd. Informační systémy (IS) jsou však nyní drahým potěšením a pro jejich vývoj je zapotřebí vysoce kvalifikovaný personál. IS jako třída systémů zahrnuje databáze, uživatelská rozhraní a obchodní logiku. Téměř vždy je potřeba IS integrovat s ostatními IS. Vývoj IS tedy vyžaduje znalost SŘBD (SQL nebo noSQL), front-endu (formuláře, UX/UI), back-endu (aplikační server) a API. Proto mají IS vysoké náklady na vlastnictví a provoz je spojen s vysokými riziky. Vždyť IS jsou vytvářeny univerzálními softwarovými inženýry, kteří píší spoustu systémového kódu pro každý IS z nuly. Pokud oddělíme aplikované programování a systémové do dvou různých specializací a využijeme výhody vysokoúrovňové platformy, kterou vytvořili systémoví programátoři, pak můžeme znovu použít až 80% kódu v různých systémech. Aplikační programátoři (tedy vývojáři) se pak budou moci soustředit pouze na úkoly související se specifikou předmětné oblasti. To výrazně snižuje nároky na vývojáře aplikací a využití principů svobodného softwaru umožňuje spojit úsilí o vytvoření platformy

a eliminovat rizika spojená s vlastnictvím platformy. **Open source** licence brání prodejcům svévolně měnit své politiky vůči spotřebitelům a systémovým integrátorům, protože nemají blokující kontrolu nad platformou a mohou být nahrazeni konkurenty.

Tento přístup byl již opakovaně použit a výrazně zvýšil dostupnost účetního a kancelářského softwaru, tvorby webových stránek, dokonce ani počítačové hry se dnes nepíší od nuly, ale využívají platformy, na kterých mohou i začátečníci rychle ukázat působivé výsledky.

Abychom takový přístup pro IS zavedli, potřebujeme nové profese, nový systém školení, nový přístup ke stanovení cílů, dokonce i zákazník takového IS by o nich měl přemýšlet úplně jinak. Stávající poptávka by se měla výrazně změnit, řádově se zvýšit díky tomu, že nyní budou specializované IS dostupné mnohem širšímu spektru spotřebitelů a již nebudou luxusem.

Složitost a jednoduchost

Snažme se, aby naše programy byly jednoduché jak pro uživatele, tak pro nás samotné, jako lidi, kteří je budou mnohokrát upravovat a neustále narážet na řešení, která jsme do nich vložili při prvotním vývoji. A pokud jsme časově omezeni a nuceni psát neefektivní nebo obskurní kód, pak bychom měli naplánovat jeho zpracování, refaktoring a optimalizaci, než zapomeneme na jeho strukturu a všechny nápady na vylepšení zmizí. Hromadění problémů v kódu se nazývá "technický dluh" a vede nejen k tomu, že se programy stávají méně flexibilními a srozumitelnými, ale také k tomu, že naši mladší kolegové, napojení na projekty, čtou a vstřebávají ne úplně ty nejlepší pracovní postupy a praktiky a asimilují naší přetechnizovanost (over-engineering). Jednoduchost při řešení složitých problémů je cílem dobrého programátora a skrýt složitost za softwarové abstrakce je metodou zkušeného inženýra.

"Vždycky jsem snil o tom, že můj počítač lze používat stejně snadno jako telefon; můj sen se stal skutečností: už nemůžu přijít na to, jak používat svůj telefon." // Björn Stroustrup

1.5. Dekompozice a separace odpovědnosti

No translation

1.6. Přehled specializace softwarového inženýra

Kolem programování, stejně jako kolem jakékoli oblasti své činnosti, člověk si dokázal vybudovat obrovské množství předsudků a bludů. Prvním zdrojem problémů je terminologie, protože různá paradigmata, jazyky a ekosystémy si vnucují vlastní terminologii, která je nejenom uvnitř sebe rozporuplná, ale i nelogická i v rámci samostatné komunity. Kromě toho mnoho programátorů samouků a samotářů vymýšlí originální, unikátní terminologii a koncepty, které se navzájem duplikují. Zběsilí marketéři mají také destruktivní vliv na IT průmysl jako celek, na formování světového názoru a terminologie. Překrucují samozřejmost, zamotají hlavu, komplikují koncepty, což vytváří nevyčerpatelnou lavinu problémů, na kterých spočívá celý softwarový byznys. Průmysloví giganti lákají uživatele a programátory svými technologiemi, často vytvářejí velmi poutavé a věrohodné koncepty, které nakonec vedou k nekompatibilitě, válkám standardů a přímé závislosti na dodavateli softwarové platformy. Trendy upoutávající pozornost lidí parazitují na jejich zájmech a rozpočtech po celá desetiletí. Někteří vývojáři, poháněni pýchou a ješitností, sami šíří pochybné a někdy zjevně slepé myšlenky. Koneckonců, pro výrobce se absolutně nevyplácí vyrábět dobrý software. Situace je mnohem lepší v oblasti svobodného softwaru a open source, ale decentralizovaní nadšenci jsou příliš rozpolcení na to, aby účinně odolávali mocné propagandě průmyslových gigantů.

Jakmile se technologie nebo ekosystém vyvinou natolik, aby na nich bylo možné stavět dobrá řešení, neodkladně zastarají nebo je výrobce přestane podporovat nebo se stanou příliš složitými. V mé paměti se již změnilo více než pět takových technologických ekosystémů.

1.7. Přehled programovacích paradigmat

No translation

2. Základní pojmy

No translation

```
// Single-line comment
```

```
/*
  Multi-line
  comments
*/
```

```
# Single-line comment
```

```
"""
  Multi-line
  comments
"""
```

```
select name from PERSON -- comments in sql
```

```
<!-- commented block in xml and html -->
```

```
; Single-line comment in Assembler and LISP
```

2.1. Hodnota, identifikátor, proměnná a konstanta, literál, přiřazení

No translation

2.2. Datové typy, skalární, referenční a strukturované typy

No translation

2.3. Operátor a výraz, blok kódu, funkce, smyčka, podmínka

No translation

2.4. Kontexty a lexikální rozsah

No translation

2.5. Procedurální paradigma, volání, zásobník a halda

No translation

2.6. Funkce vyššího řádu, čistá funkce, vedlejší účinky

No translation

2.7. Uzávěry, funkce zpětného volání, zabalení a události

No translation

2.8. Výjimky a řešení chyb

No translation

2.9. Úkoly

No translation

3. Stav aplikace, datové struktury a kolekce

No translation

3.1. Stavové a bezstavové přístupy (stateful and stateless)

No translation

3.2. Struktury a záznamy

```
#include <stdio.h>

struct date {
  int day;
  int month;
  int year;
};

struct person {
  char *name;
  char *city;
  struct date born;
};

int main() {
  struct person p1;
  p1.name = "Marcus";
  p1.city = "Roma";
  p1.born.day = 26;
  p1.born.month = 4;
  p1.born.year = 121;

  printf(
    "Name: %s\nCity: %s\nBorn: %d-%d-%d\n",
    p1.name, p1.city,
    p1.born.year, p1.born.month, p1.born.day
  );
```

```
  return 0;
}
```

Pascal

```
program Example;

type TDate = record
  Year: integer;
  Month: 1..12;
  Day: 1..31;
end;

type TPerson = record
  Name: string[10];
  City: string[10];
  Born: TDate;
end;

var
  P1: TPerson;
  FPerson: File of TPerson;

begin
  P1.Name := 'Marcus';
  P1.City := 'Roma';
  P1.Born.Day := 26;
  P1.Born.Month := 4;
  P1.Born.Year := 121;
  WriteLn('Name: ', P1.Name);
  WriteLn('City: ', P1.City);
  WriteLn(
    'Born: ',
    P1.Born.Year, '-',
    P1.Born.Month, '-',
    P1.Born.Day
  );
  Assign(FPerson, './record.dat');
```

```
  Rewrite(FPerson);
  Write(FPerson, P1);
  Close(FPerson);
end.
```

Rust

```
struct Date {
    year: u32,
    month: u32,
    day: u32,
}

struct Person {
    name: String,
    city: String,
    born: Date,
}

fn main() {
    let p1 = Person {
        name: String::from("Marcus"),
        city: String::from("Roma"),
        born: Date {
            day: 26,
            month: 4,
            year: 121,
        },
    };

    println!(
        "Name: {}\nCity: {}\nBorn: {}-{}-{}\n",
        p1.name, p1.city,
        p1.born.year, p1.born.month, p1.born.day
    );
}
```

TypeScript: Interfaces

```
interface IDate {
  day: number;
  month: number;
  year: number;
}

interface IPerson {
  name: string;
  city: string;
  born: IDate;
}
```

```
const personToString = (person: IPerson): string => {
  const { name, city, born } = person;
  const { year, month, day } = born;
  const fields = [
    `Name: ${name}`,
    `City: ${city}`,
    `Born: ${year}-${month}-${day}`,
  ];
  return fields.join('\n');
};
```

```
const person: IPerson = {
  name: 'Marcus',
  city: 'Roma',
  born: {
    day: 26,
    month: 4,
    year: 121,
  },
};

console.log(personToString(person));
```

```
class DateStruct {
  day: number;
  month: number;
  year: number;
}

class Person {
  name: string;
  city: string;
  born: DateStruct;
}
```

JavaScript: Classes

```
class DateStruct {
  constructor(year, month, day) {
    this.day = day;
    this.month = month;
    this.year = year;
  }
}

class Person {
  constructor(name, city, born) {
    this.name = name;
    this.city = city;
    this.born = born;
  }
}
```

```
const personToString = (person) => {
  const { name, city, born } = person;
  const { year, month, day } = born;
  const fields = [
    `Name: ${name}`,
    `City: ${city}`,
    `Born: ${year}-${month}-${day}`,
  ];
```

```
  return fields.join('\n');
};
```

```
const date = new DateStruct(121, 4, 26);
const person = new Person('Marcus', 'Roma', date);
console.log(personToString(person));
```

JavaScrip: Objects

```
const person = {
  name: 'Marcus',
  city: 'Roma',
  born: {
    day: 26,
    month: 4,
    year: 121,
  },
};

console.log(personToString(person));
```

JavaScrit: strucute serialization

```
const v8 = require('v8');
const fs = require('fs');
```

Take from previous example:

- class DateStruct
- class Person

```
const date = new DateStruct(121, 4, 26);
const person = new Person('Marcus', 'Roma', date);

const v8Data = v8.serialize(person);
const v8File = './file.dat';
```

```
fs.writeFile(v8File, v8Data, () => {
  console.log('Saved ' + v8File);
});
```

File: file.dat

```
FF 0D 6F 22 04 6E 61 6D 65 22 06 4D 61 72 63 75
73 22 04 63 69 74 79 22 04 52 6F 6D 61 22 04 62
6F 72 6E 6F 22 03 64 61 79 49 34 22 05 6D 6F 6E
74 68 49 08 22 04 79 65 61 72 49 F2 01 7B 03 7B
03
```

Nested structures

```
#include <stdio.h>
#include <map>
#include <string>
#include <vector>

struct Product {
  std::string name;
  int price;
};

void printProduct(Product item) {
  printf("%s: %d\n", item.name.c_str(), item.price);
}

void printProducts(std::vector<Product> items) {
  for (int i = 0; i < items.size(); i++) {
    printProduct(items[i]);
  }
}

int main() {
  std::map<std::string, std::vector<Product>> purchase {
    { "Electronics", {
```

```
      { "Laptop", 1500 },
      { "Keyboard", 100 },
      { "HDMI cable", 10 },
    } },
    { "Textile", {
      { "Bag", 50 },
    } },
  };

  std::vector electronics = purchase["Electronics"];
  printf("Electronics:\n");
  printProducts(electronics);

  std::vector textile = purchase["Textile"];
  printf("\nTextile:\n");
  printProducts(textile);

  Product bag = textile[0];
  printf("\nSingle element:\n");
  printProduct(bag);

  int price = purchase["Electronics"][2].price;
  printf("\nHDMI cable price is %d\n", price);
}
```

Python

```
purchase = {
  'Electronics': [
    { 'name': 'Laptop', 'price': 1500 },
    { 'name': 'Keyboard', 'price': 100 },
    { 'name': 'HDMI cable', 'price': 10 },
  ],
  'Textile': [
    { 'name': 'Bag', 'price': 50 },
  ],
}

electronics = purchase['Electronics']
```

```
print({ 'electronics': electronics })

textile = purchase['Textile']
print({ 'textile': textile })

bag = textile[0]
print({ 'bag': bag })

price = purchase['Electronics'][2]['price']
print({ 'price': price })
```

JavaScript

```
const purchase = {
  Electronics: [
    { name: 'Laptop', price: 1500 },
    { name: 'Keyboard', price: 100 },
    { name: 'HDMI cable', price: 10 },
  ],
  Textile: [{ name: 'Bag', price: 50 }],
};

const electronics = purchase.Electronics;
console.log(electronics);

const textile = purchase['Textile'];
console.log(textile);

const bag = textile[0];
console.log(bag);

const price = purchase['Electronics'][2].price;
console.log(price);

const json = JSON.stringify(purchase);
console.log(json);
const obj = JSON.parse(json);
console.log(obj);
```

3.3. Pole, seznam, sada, n-tice

No translation

3.4. Slovník, hashovací tabulka a asociativní pole

No translation

3.5. Zásobník, fronta, deque

No translation

3.6. Stromy a grafy

No translation

3.7. Projekce a zobrazení datových

No translation

3.8. Odhad výpočetní složitosti

No translation

4. Rozšířené koncepty

No translation

4.1. Co je technologický stack

No translation

4.2. Vývojové prostředí a ladění

No translation

4.3. Iterace: rekurze, iterátory a generátory

No translation

4.4. Stavební bloky aplikací: soubory, moduly, komponenty

No translation

4.5. Objekt, prototyp a třída

No translation

4.6. Částečná aplikace a curryfikace, skládání funkcí

No translation

4.7. Řetězení pro metody a funkce (chaining)

No translation

4.8. Mixiny (mixins)

No translation

4.9. Závislosti a knihovny

No translation