POKROČILÉ SOFTVÉROVÉ INŽINIERSTVO

Poznámky k prednáškam

Jaroslav Porubän, Štefan Korečko, Sergej Chodarev

2025

ISBN XXX-XX-XXX-XXXX-X

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky
Katedra počítačov a informatiky

Jaroslav Porubän
Štefan Korečko
Sergej Chodarev

POKROČILÉ SOFTVÉROVÉ INŽINIERSTVO

Poznámky k prednáškam

Vydanie týchto materiálov bolo podporené projektom APVV-23-0408 *Vyvíjanie architektonických znalostí v edge–cloud kontinuu.*

ISBN XXX-XX-XXX-XXXX-X

Obsah

Kapitola 1

Úvod

Tento dokument je pracovnou verziou poznámok k prednáškam z predmetu Pokročilé softvérové inžinierstvo. Zatiaľ obsahuje kapitoly k týmto prednáškam:

6. Modularita (Sergej Chodarev)
7. Architektúra softvéru (Sergej Chodarev)

Kapitola 2

Modularita

Témou tejto kapitoly je modularita. Asi ste ten pojem počuli, možno ste sa aj snažili robiť svoj kód modulárny. Tu sa na modularitu pozrieme systematickejšie, trochu aj na historický vývoj a na nástroje, ktoré umožňujú modularitu systému zlepšiť.

> **Modularity** is the degree to which a system's components may be separated and recombined, often with the benefit of flexibility and variety in use.
>
> – Wikipedia

Modularita je miera, v ktorej môžu byť komponenty systému od seba oddelené a rôzne kombinované, najmä kvôli flexibilite systému. Prečo vlastne potrebujeme náš systém deliť na časti? Veď mohli by sme všetko napísať do jednej obrovskej funkcie `main()`, ale nerobíme to. Prečo?

2.1 Dôvody

Tri hlavné dôvody sú:

- pochopenie kódu,
- tímová práca a
- flexibilita.

Prvý dôvod je teda **pochopenie kódu**. Kód potrebujeme rozumieť a reálny softvér je veľmi zložitý. Ak by nebol rozdelený na časti, tak by sme ho nedokázali ako celok pochopiť. Musíme si ho deliť na časti, aby sme rozumeli každej zvlášť.

Ďalší dôvod je **tímová práca**. Keď softvér vyvíja viac ľudí, potrebujeme si rozdeliť úlohy tak, aby na jednotlivých častiach mohli pracovať rôzni ľudia či tímy. Veľmi ťažko sa totiž pracuje, ak viacerí robia na tej istej časti kódu. Rozdelenie na moduly umožňuje vyvíjať paralelne, nezávisle na sebe.

Tretí dôvod je **flexibilita**, teda možnosť zmien. Softvér sa neustále mení. Mení sa dokonca aj počas vývoja prvej verzie a neskôr stále potrebujeme aktualizovať a upravovať. Preto sa volá „soft“, lebo sa dá ľahko meniť. Ak by sme mali jednu obrovskú funkciu, nájsť miesto zmeny, pochopiť ho a urobiť zmenu tak, aby neovplyvnila iné časti, by bolo veľmi ťažké.

Testovateľnosť

Ďalší pojem, ktorý s tým súvisí, je testovateľnosť. Potrebujeme sa uistiť, že softvér funguje tak, ako očakávame. Môžeme testovať celý program naraz: spustiť ho a skúšať, či sa správa správne. Pri zložitejších programoch však vzniká problém množstva premenných. Na vstupe môže byť rôzne poradie klikov, rôzne dáta v databáze, rôzny stav aplikácie. Možností je nekonečne veľa a testovanie všetkých kombinácií je nemožné.

Preto potrebujeme testovať moduly samostatne. Keď testujeme jeden modul, zaujímajú nás len premenné, ktoré sa týkajú jeho činnosti. Ostatné môžeme ignorovať. Testovanie sa tak stáva oveľa jednoduchšie, lebo premenných je oveľa menej, a menej je aj ich kombinácií. Keď dôkladne otestujeme malé moduly, máme väčšiu istotu, že aj systém ako celok bude fungovať. Musíme to, samozrejme, otestovať tiež, ale už nie tak dôsledne.

Dôležitý aspekt testovania je **determinizmus**. Testy by mali vždy používať tie isté vstupy a ak sa kód nezmení, mal by byť výsledok rovnaký. Keď testy raz prejdú a raz neprejdu, aj keď sa nič nezmenilo, sú nepoužiteľné[1].

Zdrojom nedeterminizmu je, napríklad, generátor náhodných čísel, ale aj čas. Stretol som sa s testom, ktorý padal len v piatok popoludní, lebo časť kódu mala vtedy iné správanie. Ďalším zdrojom nedeterminizmu je súbežné spracovanie – vlákna a nepredvídateľné poradia operácií.

[1] Náhodné testy tiež majú svoje miesto v softvérovom inžinierstve. Napríklad *fuzz testing* alebo *monkey testing* generuje náhodné vstupy, aby otestoval robustnosť systému. *Property testing* generuje náhodné vstupy na overenie určitých vlastností systému. Tieto prístupy sú užitočné na odhalenie neočakávaných chýb, ale na testovanie funkčnosti počas vývoja softvéru sa používajú deterministické testy.

Aj preto je dôležité, aby sme mohli moduly testovať oddelene, izolovať ich od času či paralelizmu a pracovať s nimi samostatne.

Testovanie sa ukazuje ako dobrá pomôcka pri návrhu modulárneho systému. Najľahšie sa testujú systémy dobre rozdelené na moduly. Ak na testovanie myslíme od začiatku, alebo používame vývoj riadený testmi (test-driven development, TDD), často nás to prirodzene núti navrhovať systém tak, aby bol dobre rozdelený. Ak sa k nejakej časti kódu ťažko píšu testy, často je problém práve v štruktúre kódu.

Nasadenie

Ďalší dôvod modularizácie je nasadenie (deployment). Moduly môžu byť realizované ako samostatné služby, ktoré sa nasadzujú nezávisle od seba. Každý tím môže nasadzovať svoju časť bez koordinácie s ostatnými. To zjednodušuje celý proces a umožňuje automatizáciu pomocou *deployment pipelines*: automatizované testovanie, zostavenie a nasadenie každej služby samostatne.

Zjednodušuje to aj škálovanie – jednotlivé moduly môžu bežať na rôznych serveroch alebo kontajneroch podľa potreby. Prípadne aj na viacerých inštanciách, ak je nejaká časť systému viac zaťažená.

2.2 Dekompozícia

Základná otázka pri delení na moduly je: na základe akých kritérií deliť systém? Kde nakresliť hranicu medzi modulmi? Pričom dekompozícia je viacúrovňová – služby, komponenty, triedy, metódy. Nevystačíme si s nejakým jednoduchým pravidlom. Existujú však metriky, ktoré nám v tom môžu pomôcť. Ani tie však nedávajú jednoznačné odpovede a dekompozícia vyžaduje intuíciu a skúsenosť.

Najčastejšie spomínané pojmy sú **cohesion (kohézia)** a **coupling (prepojenosť)** (Farley 2022). Kohézia sa týka toho, aby boli spolu veci, ktoré spolu súvisia, ktoré sa menia spolu. Coupling zase vyjadruje mieru prepojenia modulov medzi sebou. Ideálne chceme vysokú kohéziu a nízku prepojenosť.

2.3 Kohézia

> **Cohesion** – the degree to which the elements inside a module belong together.
>
> – Wikipedia

> Pull the things that are unrelated further apart, and put the things that are related closer together.
>
> – Kent Beck

Môžeme si predstaviť extrémny prípad, kde všetko je v jednom module. Ten má paradoxne nízku kohéziu, lebo sa tam miešajú aj nesúvisiace časti. Opačný extrém – každý malý fragment v samostatnom module – má tiež nízku kohéziu, lebo časti, ktoré úzko súvisia, sú zbytočne rozdelené. Navyše, medzi takýmito modulmi bude veľmi vysoká prepojenosť. Potrebujeme teda nájsť rovnováhu.

Ideálny stav z hľadiska kohézie opísal Larry Constantine:

> Attempting to divide a cohesive module would only result in increased coupling and decreased readability.
>
> – Larry Constantine

2.4 Oddelenie záujmov

> **Separation of concerns (SoC)** is a software engineering principle that allows software engineers to deal with one aspect of a problem so that they can concentrate on each individually.
>
> – Wikipedia

Jedným z nástrojov ako zabezpečiť dobrú kohéziu je **„oddelenie záujmov“ (separation of concerns)**. Kód riešiaci jednu vec má byť na jednom mieste. Čo je „tá jedna vec“, je ťažká otázka, ale princíp je jasný: nemiešať dohromady nesúvisiace záujmy. Napríklad v kóde, ktorý rieši pridanie tovaru do košíka, nechceme mať detaily databázy alebo používateľských práv.

Jednou vecí, ktoré chceme oddeliť sú rôzne úrovne zdrojov zložitosti. Pojmy *essential complexity* a *accidental complexity*, teda základná zložitosť riešeného problému a náhodná zložitosť, zaviedol článok *No Silver Bullet* (Brooks 1987).

Článok je o tom, že vo vývoji softvéru sa stále snažíme nájsť nástroje, ktoré nám pomôžu všetko riešiť jednoduchšie. Problémom, ktorému čelíme, je však zložitosť. Softvér je veľmi zložitý. Frederick Brooks, autor článku, argumentuje, že nie všetku zložitosť vývoja softvéru sa dá odstrániť.

Máme náhodnú zložitosť, ktorá súvisí s tým, že to, čo sa snažíme realizovať, beží na počítači. Musíme pracovať s databázou, so súborovým systémom, so sieťou, riešiť transakcie, a tak ďalej. To je tzv. náhodná zložitosť.

Ale existuje aj základná (essential) zložitosť, ktorá je daná samotným problémom, ktorý riešime. Ak robíme systém pre obchodovanie na burze alebo pre zdravotníctvo, tak tie problémy sú samy o sebe veľmi zložité a tejto zložitosti sa nezbavíme. Vieme minimalizovať tú náhodnú, vieme vytvárať nové jazyky, nové rámce a ďalšie veci, ktoré ju znižujú, ale stále nám zostane základná zložitosť samotného problému.

V rámci oddelenia záujmov je užitočné rozdeliť záujmy na *základné* (vychádzajúce z podstaty riešeného problému) a *náhodné* (vychádzajúce z technológií, ktoré používame).

Napríklad, ak ukladáme dáta do databázy, tak v kóde, ktorý rieši doménovú logiku, nechceme riešiť detaily databázového systému. Tieto vrstvy chceme oddeliť.

S tým súvisí aj metodológia *Domain Driven Design* (Evans 2004). Jeho základná myšlienka je sústrediť sa pri návrhu softvéru na doménu a jej základnú zložitosť. Moduly by mali vychádzať z domény: keď je v doméne niečo oddelené, malo by to byť oddelené aj v softvéri. Keď je v doméne nejaký pojem, mala by existovať časť systému, ktorá mu zodpovedá. A zároveň chceme odizolovať doménu od technických záležitostí: či je to webová alebo mobilná aplikácia, či používame Postgres alebo MongoDB. Technické detaily nechceme miešať s jadrom systému.

Ak napríklad robíte informačný systém pre školstvo a máte výpočet známky, tak funkcia, ktorá rieši výpočet, by nemala vedieť detaily o databáze ani o HTTP komunikácii. Tieto úrovne zložitosti majú byť rozdelené, aby zmena v jednej časti neovplyvňovala ostatné.

Jedným z nástrojov na to je *dependency injection*. Ak funkcia na výpočet známky potrebuje pristupovať k databáze, mala by dostať modul, ktorý to zabezpečí, ako závislosť. Nemala by si ho vytvárať sama, lebo by tým musela poznať detaily realizácie. Ak dostane len rozhranie, je od neho nezávislá.

2.5 Kritéria dekompozície

Teraz sa vrátim k otázke: podľa akého kritéria rozdeliť program na moduly? Použijem príklad z vedeckej literatúry – článku „On the Criteria To Be Used in Decomposing Systems into Modules“ (Parnas 1972).

Článok rieši otázku, podľa čoho systém rozdeliť. Modul nechápe ako podprogram, ale ako jednotku, ktorej priraďujeme určitú zodpovednosť. Ako príklad používa systém, ktorý v knižniciach vytváral KWIC index – zoznam cyklicky posunutých názvov kníh.

Napríklad názov „Classification of Books in University Library“ môžeme cyklicky posúvať, aby sa dal vyhľadať podľa ľubovoľného slova. Výsledkom je množstvo posunutých riadkov, usporiadaných podľa abecedy, ku ktorým je priradené číslo knihy:

```
BOOKS in a University Library/Classification of 1279
CLASSIFICATION of Books in a University Library 1279
LIBRARY/Classification of Books in a University 1279
UNIVERSITY Library/Classification of Books in a 1279
```

Takéto riadky sa môžu zaradiť do katalógu knižnice pre jednoduchšie vyhľadávanie.

Úlohou je vytvoriť systém, ktorý takéto indexy generuje. Vstupom je zoznam riadkov (reťazcov), každý riadok je zoznam slov. Výstupom je zoznam všetkých cyklických posunov, zoradený podľa abecedy.

Parnas vo svojom článku uvažuje o dvoch dekompozíciách.

Prvá dekompozícia je podľa krokov spracovania:

1. Vstup – načíta riadky a uloží ich do pamäte.
2. Vytváranie posunov – ukladajú sa len indexy.
3. Triedenie podľa abecedy.
4. Výstup – vypísanie výsledku.
5. Riadiaci modul.

Druhá dekompozícia je založená na skrývaní informácií:

1. Modul na ukladanie riadkov, poskytuje abstraktné rozhranie (get/set char).
2. Vstupný modul, ktorý používa rozhranie modulu 1.
3. Modul vytvárajúci posuny, ktorý používa modul 1 a poskytuje svoje vlastné abstraktné rozhranie.
4. Modul triedenia – pracuje s abstrakciami, nie s konkrétnym spôsobom uloženia.
5. Výstup a riadenie.

Zásadný rozdiel je v tom, že v druhej dekompozícii každý modul skrýva konkrétne návrhové rozhodnutie.

Parnas hodnotí tieto dekompozície podľa kritérií:

Zmeniteľnosť: V prvej dekompozícii zmena ukladania dát v pamäti vyžaduje zmeniť všetky moduly, lebo všetky zdieľajú informáciu o tom, v akej forme sú dáta uložené. V druhej stačí zmeniť jediný modul, lebo ostatné pristupujú k dátam cez rozhranie poskytované týmto modulom.

Nezávislý vývoj: V prvej dekompozícii sa programátori musia presne dohodnúť na vnútorných formátoch. V druhej stačí dohodnúť sa na abstraktných rozhraniach.

Jednoduchosť pochopenia: V prvej dekompozícii musím pochopiť všetky moduly, lebo sú príliš previazané. V druhej sú moduly oveľa nezávislejšie.

Parnas píše, že prvá dekompozícia je prirodzená pre začínajúcich programátorov (a väčšina z nás tak začínala), ale je nevhodná. Druhá je založená na skrývaní informácií – *information hiding*. To znamená **skrývanie návrhových rozhodnutí*** (ako sa ukladajú dáta, ako sa vypočítavajú posuny...), aby ich zmena ovplyvnila čo najmenej modulov.

To pripomína objektovo orientované programovanie, hoci Parnas písal článok ešte pred jeho masovým rozšírením. Samozrejme, aj objektový kód môžeme dekomponovať nevhodne – ak ho rozdelíme podľa krokov, dopadneme rovnako zle.

Pri dekompozícii ide najmä o zmeniteľnosť. Ak musíme pri jednej zmene upravovať 10 modulov, je to znak zlej dekompozície. Ak zmenu urobíme na jednom mieste, je to dobré znamenie.

2.6 Abstrakcia

Abstrakcie a skrývanie informácií sú úzko prepojené pojmy – sú to odlišné pohľady na tú istú vec. V obidvoch prípadoch sa snažíme skryť niektoré návrhové rozhodnutia.

Dnes používame množstvo abstrakcií bez toho, aby sme sa nad tým zamýšľali. Dátový typ *string* skrýva detaily ukladania znakov, kódovania, dĺžky; dátový typ *list* skrýva implementáciu ukladania prvkov. Nemusíme implementovať vlastné spájané zoznamy. Toto všetko sú abstrakcie, ktoré skrývajú detaily a umožňujú sústrediť sa na základnú zložitosť.

Pri vytváraní vlastných abstrakcií môžeme využívať abstrakcie z domény problému. To je myšlienka *domain-driven design*: skúmame doménu, ktorú realizujeme, identifikujeme abstrakcie, ktoré sú dané samotnou podstatou problému, a tie prenášame do kódu. Extrémnym príkladom sú doménovo-špecifické jazyky,

kde vytvoríme celý jazyk umožňujúci vyjadrovať sa pomocou pojmov a abstrakcií danej domény.

Ďalšia vec, ktorú sa oplatí abstrahovať, je náhodná zložitosť (*accidental complexity*). Ide o rôzne implementačné detaily a detaily viazané na konkrétnu technológiu. Oplatí sa izolovať externé systémy a služby. Ak používame databázu na ukladanie dát, je vhodné to abstrahovať a vytvoriť vlastnú perzistenčnú vrstvu, ktorá poskytuje rozhranie vhodné pre naše použitie. Konkrétnu realizáciu potom vieme vymeniť.

> All non-trivial abstractions, to some degree, are leaky.
>
> – Joel Spolsky: The Law of Leaky Abstractions (Spolsky 2002)

Problém abstrakcií však je, že všetky netriviálne abstrakcie sú takzvané „leaky“, teda „pretiekajú“. Pojem *leaky abstractions* zaviedol Joel Spolsky. Ako príklad uvádzal čísla s pohyblivou desatinnou čiarkou – typy `double` a `float`. Ide o základnú abstrakciu pre prácu s reálnymi číslami, ale správanie týchto typov nie je totožné s reálnymi číslami v matematike.

Dôvodom je spôsob ich implementácie a obmedzená presnosť ukladania. Preto napríklad nie je vhodné ukladať údaje o peniazoch do typu `double`, lebo pri výpočtoch môžu vznikať malé rozdiely na úrovni tisícin alebo menších hodnôt. `Double` nie je skutočné reálne číslo, ale len abstrakcia, ktorá sa tak správa vo väčšine prípadov. Občas však implementačné detaily „pretečú“ a výsledky nie sú také, aké by sme očakávali.

Aj v iných oblastiach nám abstrakcie poskytujú pekné rozhranie bez nutnosti poznať detaily. V okrajových situáciách však musíme rozumieť aj tomu, čo je pod povrchom. Abstrakcie nikdy úplne neskrývajú realitu a určitá znalosť implementácie je stále potrebná.

2.7 Prepojenosť (coupling)

Coupling je miera, v akej sú jednotlivé časti systému medzi sebou prepojené. Keď systém delíme na časti, tie nikdy nebudú úplne nezávislé. Keby boli, nešlo by o jeden systém. Aby tvorili jeden celok, musia medzi sebou komunikovať.

Coupling sa snažíme minimalizovať, aby sme dosiahli výhody modularizácie. Moduly by mali byť čo najviac nezávislé, hoci nikdy nie úplne. Je to jedna z najťažších úloh návrhu systému.

Základným nástrojom je definícia rozhraní. Medzi modulmi by mali byť jasne definované a čo najmenšie rozhrania. Rozhranie je miesto, kde sa moduly

dotýkajú. Čím je väčšie a čím viac detailov odhaľuje, tým viac o sebe moduly vedia a tým sú viac previazané.

Moduly by mali skrývať informácie o svojom vnútornom fungovaní. Ak rozhranie odhaľuje interné detaily, všetok kód, ktorý modul používa, sa stáva závislým nielen od rozhrania, ale aj od jeho implementácie. Preto by sme mali definovať abstraktné rozhrania, ktoré vychádzajú z povahy problému a z domény.

Jednou z techník je *ports and adapters*. Modul, ktorý používa nejakú službu, si definuje port – jasné rozhranie služieb, ktoré potrebuje. Napríklad modul nákupného košíka potrebuje ukladať položky do databázy. Zlé riešenie by bolo riešiť priamo v tomto module SQL príkazy a prácu s databázou, pretože ide o náhodnú zložitosť.

Namiesto toho definujeme abstraktný modul, ktorý len povie „ulož položku nákupného košíka“. Následne vytvoríme adaptér, ktorý túto operáciu realizuje pre konkrétnu databázovú technológiu. Tým oddelíme doménovú logiku od konkrétnej implementácie pomocou rozhrania a adaptéra.

Tento prístup sa nazýva **Dependency Inversion Principle**. Závislosť sa obráti: modul nákupného košíka nezávisí od konkrétnej databázy, ale len od rozhrania, ktoré si sám definuje. Nízkoúrovňový modul, ktorý pracuje s databázou, potom závisí od tohto rozhrania.

Hoci počas behu programu nákupný košík databázu používa, v kóde je závislosť opačná. Tým sa závislosti oslabia a časti patriace k podstatnej zložitosti systému nezávisia od náhodnej zložitosti.

Tieto princípy sú dôležité aj pre testovateľnosť. Modulárny kód sa testuje ľahšie. Ak oddelíme implementačné detaily a externé technológie, pri testovaní môžeme namiesto reálnej databázy alebo služby použiť *mock*. Tým sa tvorba testov výrazne zjednoduší a zároveň dosiahneme lepšiu modularizáciu celého systému.

Kapitola 3

Architektúra softvéru

> Architecture is about the important stuff... whatever that is.
>
> — Ralph Johnson

Definovať softvérovú architektúru je veľmi zložité. Jednoznačná definícia vlastne ani neexistuje. Napríklad jeden citát hovorí, že architektúra je o dôležitých veciach, čo samo o sebe veľa nehovorí. Vo všeobecnosti teda to, čo je pri návrhu softvéru dôležité, často nazývame architektúrou.

Trochu konkrétnejšia je definícia z Wikipédie: „softvérová architektúra je o najdôležitejších štrukturálnych voľbách alebo rozhodnutiach, ktoré je drahé meniť po tom, ako už boli urobené.“ Túto definíciu som počul viackrát, napríklad aj v prezentáciách Martina Fowlera. Architektúra je teda to, čo je drahé zmeniť – základné rozhodnutia, ktorých zmena je neskôr nákladná.

Autori knihy *Fundamentals of Software Architecture* (Richards a Ford 2020) neprichádzajú s jednou vetou, ale definujú architektúru pomocou štyroch oblastí.

1. **Štruktúra systému**: rozdelenie na komponenty a vzťahy medzi nimi. Ide o základný spôsob, ako je systém členený.
2. **Architektonické charakteristiky**, často nazývané aj „-ilities“ – testovateľnosť, udržiavateľnosť, spoľahlivosť a podobne. Ide o vlastnosti systému a spôsoby, ako ich dosahujeme.
3. **Architektonické rozhodnutia**, teda rozhodnutia o tom, ako bude systém fungovať počas vývoja a prevádzky.

4. **Návrhové princípy**, ktoré používame pri návrhu jednotlivých komponentov. Softvérové systémy sú fraktálne – komponenty sa delia na menšie časti, tie na ešte menšie a podobne. Spôsoby tohto delenia a interakcie medzi časťami sa riadia princípmi, ktoré chceme v systéme dodržiavať. Aj tieto princípy patria do architektúry.

3.1 Kto je softvérový architekt?

Zaujímavá je aj otázka, kto je vlastne softvérový architekt a čím sa líši od programátora. Očakáva sa od neho, že bude robiť architektonické rozhodnutia o najdôležitejších vlastnostiach systému a bude architektúru priebežne analyzovať a upravovať.

Mal by mať prehľad o aktuálnych trendoch. Nemal by to byť človek, ktorý prestal programovať a uviazol pri technológiách, ktoré boli aktuálne pred mnohými rokmi.

Architekt zabezpečuje, že prijaté rozhodnutia sa skutočne dodržiavajú. Môže to robiť manažérsky, ale aj technicky, napríklad vytváraním nástrojov na automatickú kontrolu architektonických pravidiel. Napríklad, že komponenty z jednej vrstvy nemajú priamo pristupovať ku komponentom inej vrstvy.

Architekt by mal mať aj znalosti domény, ktorú softvér rieši. Ak ide o bankovníctvo, musí rozumieť bankovníctvu; ak o medicínu, aspoň základom medicíny. Nemôže navrhovať architektúru systému, o ktorého oblasti nič nevie.

Dôležité sú aj tzv. „soft skills“, najmä komunikácia. Architekt robí rozhodnutia, ktoré realizujú iní ľudia, a komunikuje s tímom aj so zákazníkom. Musí sa vedieť pohybovať aj v organizačnej a politickej štruktúre firmy, pretože ide čiastočne o manažérsku rolu.

3.2 Zákony softvérovej architektúry

Existujú dva zákony softvérovej architektúry.

> Everything in software architecture is a trade-off.
>
> – First Law of Software Architecture (Richards a Ford 2020)

Prvý hovorí, že všetko v softvérovej architektúre je kompromis. Každé rozhodnutie má výhody aj nevýhody. Vždy treba zvážiť konkrétny kontext a situáciu. Neexistujú rozhodnutia, ktoré by boli jednoznačne správne bez nevýhod. Ak sa zdá, že niečo nemá nevýhody, pravdepodobne sme ich len ešte neobjavili.

> *Why* is more important than *how*.
>
> – Second Law of Software Architecture (Richards a Ford 2020)

Druhý zákon hovorí, že „prečo“ je dôležitejšie než „ako“. Keďže všetko je kompromis, je dôležité vedieť, prečo sme sa rozhodli určitým spôsobom. Ak sa neskôr ukáže, že naše predpoklady boli nesprávne, môžeme sa k rozhodnutiu vrátiť a zmeniť ho, pretože rozumieme dôvodom, ktoré k nemu viedli.

3.3 Ciele

Aký je cieľ softvérovej architektúry? Podľa Roberta C. Martina je cieľom minimalizovať ľudské zdroje potrebné na vytvorenie a údržbu systému (Martin et al. 2018). Robíme to preto, aby bol vývoj, údržba a zmeny čo najmenej náročné.

Zmena je nevyhnutná a drahá, preto sa snažíme robiť rozhodnutia tak, aby bola čo najjednoduchšia. Celá architektúra je o minimalizácii ceny budúcich zmien.

Hoci architektúra sa často definuje ako súbor rozhodnutí, ktoré je drahé zmeniť, cieľom je práve opak: navrhnúť systém tak, aby zmena bola čo najlacnejšia.

> The important decisions that a Software Architect makes are the ones that allow you to NOT make the decisions about the database, and the webserver, and the frameworks.
>
> — Robert C. Martin: A Little Architecture (Martin 2016)

Dôležité architektonické rozhodnutia podľa Roberta C. Martina nie sú o tom, akú databázu, webový server alebo framework použijeme. Sú o tom, aby sme tieto rozhodnutia nemuseli robiť napevno. Ideálne by architektúra mala zabezpečiť, že technické detaily, ako databáza či framework, sa dajú vymeniť.

Drahé rozhodnutia by mali byť v oblasti domény – v tom, čo systém rieši. Doména je stabilnejšia než technológie, ktoré sa rýchlo menia. Architekt by teda nemal rozhodovať o konkrétnom rámci, ale navrhnúť systém tak, aby na ňom nebol závislý.

3.4 Princípy SOLID

No a poďme si to pozrieť postupne. Prejdeme rôzne princípy, ktoré by nám mali pomôcť pri návrhu architektúry a dobrej modularity.

Konkrétne pri objektovo-orientovanom návrhu je známa sada piatich princípov, pomenovaná podľa začiatočných písmen – SOLID. Poďme si ich prejsť.

Single Responsibility Principle

> A module should have one, and only one, reason to change.

Single Responsibility Principle je vlastne základným princípom modularity: modul by mal robiť len jednu vec, alebo by mal existovať iba jeden dôvod na jeho zmenu. Toto je trochu ťažšie na pochopenie – čo presne znamená „jeden dôvod na zmenu“?

Tu citujem Roberta C. Martina (Martin et al. 2018): dôvod na zmenu by mal byť jeden konkrétny aktér. Aktérom sa myslí nejaký typ používateľa alebo skupina, ktorá so systémom interaguje.

Napríklad pri podnikovom systéme, ktorý používajú rôzne oddelenia – finančné, marketingové, HR – môžeme mať modul, ktorý rieši zamestnancov. Niektoré vlastnosti zamestnancov zaujímajú HR oddelenie, iné zasa finančné oddelenie, napríklad platy.

Aktérmi sú teda jednotlivé oddelenia, ktoré systém používajú. Nemali by sme miešať v rámci jedného modulu to, čo potrebuje HR, s tým, čo potrebuje finančné oddelenie, pretože tieto skupiny budú mať rôzne požiadavky na zmeny. Inak sa môže stať, že zmena kvôli HR pokazí výpočet platov pre financie. Tieto veci by mali byť oddelené.

Open–Closed Principle

> A software artifact should be open for extension but closed for modification.

Klasická definícia Open–Closed Principle hovorí, že softvérový artefakt má byť otvorený pre rozširovanie, ale uzavretý pre modifikáciu.

Ide o to, že v ideálnom prípade by sme mali meniť správanie modulu jeho rozšírením, nie úpravou existujúceho kódu. V objektovo-orientovanom programovaní to môže znamenať napríklad vytvorenie podtriedy, ktorá rozšíri správanie triedy, alebo pridanie nového komponentu.

Modifikovanie existujúceho komponentu so sebou vždy nesie riziko – komponent môže byť previazaný s inými a zmena môže porušiť ich očakávania. Rozšírenie je preto bezpečnejší spôsob pridávania funkcionality.

Napadne mi príklad z menších projektov, ktoré robím na katedre – systém pre záverečné práce. Ten rozširuje funkcionalitu GitLabu. Nevytvárali sme vlastný systém správy úloh, ale použili sme existujúce funkcie GitLabu a iba sme ich rozšírili o špecifické vlastnosti, bez toho, aby sme GitLab menili. Vytvorili sme

samostatný systém, ktorý využíva jeho API. Keby GitLab neposkytoval rozhranie na rozšírenie, porušoval by Open–Closed princíp.

Liskov Substitution Principle

> Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.
>
> – Barbara Liskov a Jeannette Wing (Liskov a Wing 1994)

Liskov Substitution Principle sa týka vzťahov typov a podtypov. V OOP ide o vzťah medzi nadtriedou a podtriedou, prípadne medzi rozhraním a triedou.

Princíp hovorí, že ak platí nejaká vlastnosť pre všetky objekty typu T, musí platiť aj pre všetky objekty typu S, kde S je podtyp T. Inak povedané, ak niekde môžeme použiť objekt nadtypu, mali by sme ho vedieť nahradiť objektom podtypu bez toho, aby sa správanie systému pokazilo.

Tento princíp sa často porušuje pri neuváženom používaní dedičnosti len na znovupoužitie implementácie. Typickým príkladom sú staršie GUI knižnice, kde ste dedili triedu okna len preto, aby ste mohli použiť jej metódy, ale výsledná trieda už nespĺňala všetky vlastnosti pôvodného okna.

Dedičnosť by mala vyjadrovať vzťah „je typu“, nie len „používa implementáciu“. Ak potrebujete iba časť implementácie, je lepšie použiť kompozíciu – vytvoriť si inštanciu triedy, ktorú potrebujete, a tú používať interne.

Interface Segregation Principle

Interface Segregation Principle hovorí, že rozhrania sú pre klientov, nie pre implementácie.

Ak máme triedu, ktorá poskytuje viac funkcionalít, mali by sme rozhrania navrhovať podľa toho, ako ju budú používať klienti. Klient by nemal závisieť od metód, ktoré nepotrebuje. Preto je rozumné rozdeliť funkcionalitu do viacerých menších rozhraní a každý klient nech závisí len od toho, ktoré potrebuje.

Tento princíp pekne ilustrujú jazyky ako Go alebo TypeScript, ktoré používajú štrukturálne typovanie. Trieda nemusí explicitne deklarovať, že implementuje rozhranie – stačí, že má správne metódy. Klient si môže definovať vlastné rozhranie ako podmnožinu funkcionality a jazyk automaticky overí, že trieda mu vyhovuje.

Rozhranie teda navrhujeme z pohľadu klienta: čo z danej triedy skutočne potrebuje. Ak existuje viac typov klientov, vytvoríme viac rozhraní.

Dependency Inversion Principle

Posledným princípom SOLID je Dependency Inversion Principle. Je veľmi dôležitý pre riadenie previazanosti komponentov.

Hovorí, že vysokoúrovňové moduly by nemali priamo závisieť od nízkoúrovňových modulov. Obe by mali závisieť od abstrakcií.

Vysokoúrovňové moduly sú tie, ktoré riešia doménovú logiku – napríklad use case z našej oblasti. Nízkoúrovňové moduly sú technické detaily, napríklad konkrétne API volania.

Použijem príklad vytvárania odovzdávok záverečných prác. Doménová logika vytvárania odovzdávky je vysokoúrovňový modul. Volania GitLab API sú nízkoúrovňové detaily. Vysokoúrovňový modul síce tieto služby potrebuje, ale nemal by na nich priamo závisieť. Riešením je vložiť medzi ne rozhranie, ktoré definuje služby v pojmoch domény. Vysokoúrovňový modul závisí len od rozhrania, nízkoúrovňový ho implementuje.

Platí teda: abstrakcie nemajú závisieť od detailov, ale detaily majú závisieť od abstrakcií. Aj preto potrebujeme rozhrania a polymorfizmus. Umožňujú nám mať viac implementácií, napríklad reálne a testovacie (fake) objekty.

3.5 Kohézia, prepojenosť a abstraktnosť

V kapitole o modularite som spomínal pojmy kohézia a prepojenosť (coupling). Pridám pár odporúčaní, ako tieto princípy aplikovať.

Kohézia

Kohézia je o tom, že súvisiace veci majú byť spolu. *Common Closure Principle* hovorí, že triedy, ktoré sa menia z rovnakých dôvodov a v rovnakom čase, majú byť spolu. Naopak, veci, ktoré sa menia z rozdielnych dôvodov a v rozdielnom čase, by nemali byť v jednom komponente.

Common Reuse Principle je, že používatelia by nemali byť závislí od vecí, ktoré nepotrebujú. Časti funkcionality, ktoré sa často používajú spolu, môžu byť v jednom module. Ak však modul obsahuje viac častí a rôzni používatelia potrebujú vždy len niektoré z nich, tieto časti by možno nemali byť spolu v jednom module. Mali by byť oddelené tak, aby každý používateľ závisel len od tej konkrétnej funkcionality, ktorú využíva.

Prepojenosť (coupling)

Pri couplingu rozlišujeme dva smery prepojenia medzi modulmi. Prvým sú vstupujúce závislosti (incoming dependencies), teda keď iné komponenty závisia od nás. Tomu sa hovorí *afferent coupling*. Druhým sú vystupujúce závislosti (outgoing dependencies), teda keď daný komponent závisí od iných komponentov. To je *efferent coupling*.

Tieto závislosti sa dajú merať. V praxi môžeme počítať napríklad importy v triedach. Keď spočítame všetky outgoing závislosti (Ce) a vydelíme ich súčtom incoming (Ca) a outgoing závislostí, získame metriku nazývanú *nestabilnosť* (*instability*). Tá má hodnotu od 0 do 1.

$$I = \frac{Ce}{Ca + Ce}$$

Ak je nestabilnosť 0, ide o stabilný modul. To znamená, že nemá žiadne outgoing závislosti, teda na ničom nezávisí, ale iné moduly závisia na ňom. Je stabilný preto, že zmeny v iných moduloch ho neovplyvnia a zároveň jeho zmena by mohla ovplyvniť veľa iných modulov, takže sa mení ťažko.

Ak máme modul, na ktorom závisí celý zvyšok systému, budeme ho chcieť meniť čo najmenej, lebo zmena môže ovplyvniť celý systém. To však nemusí byť vždy výhoda. Ak taký modul potrebujeme zmeniť, zmena je náročná a riziková.

Opačným extrémom je modul, ktorý závisí na iných moduloch, ale nikto nezávisí na ňom. Ten má nestabilnosť 1, je extrémne nestabilný, ale zároveň sa veľmi ľahko mení, pretože jeho zmena neovplyvní žiadny iný modul.

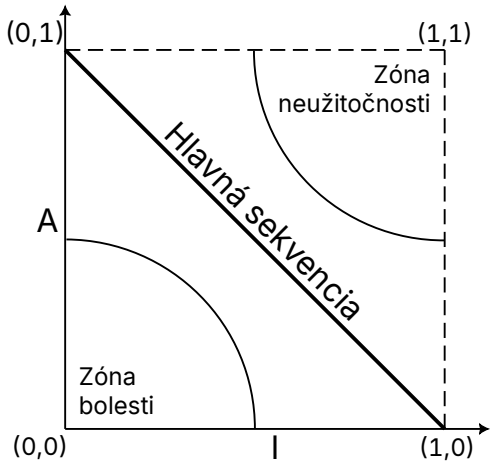
Abstraktnosť

Ďalším pojmom je abstraktnosť. Tá sa dá merať pomerne jednoducho ako podiel počtu abstraktných tried a rozhraní (Nc) k počtu konkrétnych tried (Na). Získame tak mieru abstraktnosti:

$$A = \frac{Na}{Nc}$$

Vzťah nestabilnosti a abstraktnosti

Tieto dve metriky vieme zobraziť v grafe, kde na jednej osi je nestabilita a na druhej osi abstraktnosť. Každý modul alebo komponent sa nachádza niekde v tomto priestore.



Obr. 3.1: Vzťah medzi nestabilnosťou a abstraktnosťou (Martin et al. 2018)

Existujú tu extrémy. Jeden z nich je modul s vysokou nestabilitou a nulovou abstraktnosťou, teda obsahuje len konkrétne triedy a nikto na ňom nezávisí. Druhým extrémom je modul, ktorý je veľmi stabilný a zároveň vysoko abstraktný, teda obsahuje len abstraktné triedy a rozhrania a veľa iných modulov na ňom závisí.

Medzi týmito extrémami sa nachádza tzv. *hlavná sekvencia*, ktorá predstavuje ideál. Platí, že čím je modul stabilnejší, tým by mal byť abstraktnejší. Naopak, čím je menej abstraktný, tým by mal byť aj menej stabilný.

Mimo tejto sekvencie sú dve problematické oblasti. *Zóna bolesti* obsahuje moduly, ktoré sú veľmi stabilné, veľa vecí na nich závisí, ale sú veľmi konkrétne a majú málo abstrakcií. Takéto moduly sa veľmi ťažko menia a každá zmena je bolestivá.

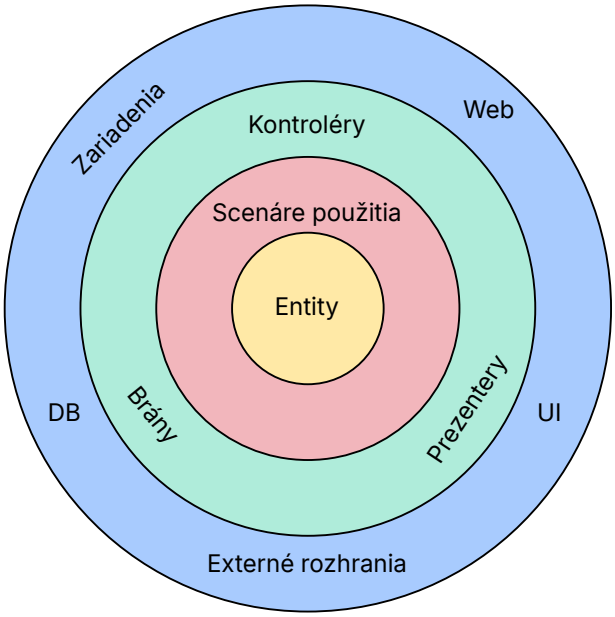
Zóna neužitočnosti obsahuje moduly, ktoré majú veľa abstrakcií, ale nikto na nich nezávisí. Obsahujú množstvo rozhraní alebo abstraktných tried, ktoré sa reálne nepoužívajú, a tým sú v podstate zbytočné.

Týmto extrémom by sme sa mali vyhýbať. Čím viac sa od modulu očakáva stabilita, tým viac by mal byť abstraktný, pretože abstrakcie sú stabilnejšie než konkrétne implementácie. Dá sa dokonca vypočítať vzdialenosť modulu od hlavnej sekvencie a cieľom je, aby bola čo najmenšia.

$$D = |A + I - 1|$$

3.6 Clean architecture

Poďme spojiť tieto princípy do celkovej architektúry. Vychádzam tu z knihy Roberta C. Martina *Clean Architecture* (Martin et al. 2018) a jeho pohľadu na to, aká by mala byť „čistá“ architektúra. Podobné princípy nájdete aj pod inými názvami u iných autorov. Často sa označuje ako hexagonálna architektúra. Tu však budem používať definíciu od Martina a celé sa to dá zhrnúť do jedného obrázka.



Obr. 3.2: Čistá architektúra (Martin et al. 2018)

Snažíme sa architektúru rozdeliť tak, aby sme oddelili doménovú logiku od technických detailov. O tom som hovoril už aj minule. Konkrétne môžeme mať niekoľko kruhov, kde v strede sú *entity*. To sú základné doménové objekty, základné pojmy, ich vlastnosti a operácie, ktoré sa s nimi dajú realizovať.

To je úplné jadro systému, ktoré priamo vychádza z domény, ktorú riešime. Tieto časti by mali byť čo najmenej ovplyvnené technickými detailmi, ako je spôsob implementácie alebo použité databázy. Toto je podstata doménovej logiky. Okolo toho sú *scenáre použitia* (use cases), teda operácie, ktoré sa dajú v danom

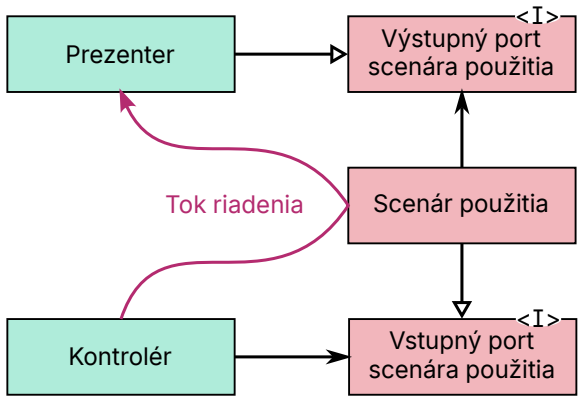
systéme vykonávať, a logika týchto operácií.

Úplne na vonkajšej vrstve sú konkrétne realizácie: prístup ku konkrétnej databáze, používateľské rozhranie, webové rozhranie, prístup k zariadeniam alebo externým službám. Medzi tým je vrstva, ktorá oddeľuje doménovú logiku od technických detailov. Sú v nej *prezentéry*, *kontroléry* a *brány* (gateways).

Každá vrstva by mala byť oddelená od tej vonkajšej tak, že závislosti smerujú vždy iba dovnútra. Šípky znázorňujú smerovanie závislostí v kóde. Vždy iba vonkajšia vrstva závisí od vnútornej.

Vnútorné vrstvy sú vysokourovňové, vonkajšie sú nízkourovňové. To je základný princíp.

Otočenie závislostí



Obr. 3.3: Otočenie závislostí v čistej architektúre (Martin et al. 2018)

Tu je uvedený príklad otočenia závislostí. Máme realizáciu nejakého scenára použitia, ktorý vyžaduje spoluprácu modulov medzi vrstvami. Vnútorná vrstva je samotný scenár, ktorý realizuje logiku operácie. Vo vonkajšej vrstve je kontrolér a prezentér.

Kontrolér je niečo, čo využíva používateľské rozhranie, napríklad stlačenie tlačidla. Spustí sa metóda v kontroléri. Prezentér je zodpovedný za zobrazenie výsledkov používateľovi.

Tok riadenia počas behu programu (runtime) je taký, že používateľ stlačí tlačidlo, aktivuje sa metóda kontroléra, tá zavolá metódu v scenáru použitia,

scenár vykoná potrebnú operáciu a zavolá metódu v prezentéri, aby sa zobrazili výsledky.

Problém je, že scenár nemôže priamo volať prezentér, pretože ten patrí do vonkajšej vrstvy. Preto scenár komunikuje cez rozhranie – takzvaný *vystupný port*. Prezentér toto rozhranie implementuje. Rovnako kontrolér volá *vstupný port* scenáru použitia. Komunikácia medzi vrstvami je teda abstrahovaná, aby boli vrstvy od seba oddelené.

Robert C. Martin odporúča považovať aj rámec (framework) za technický detail. Svoj kód sa snažíme držať čo najviac nezávislý od rámcov, najmä pri väčších systémoch, kde silná závislosť na rámci môže byť nevýhodná. Napríklad v Jave pri Springu je snaha, aby entitné triedy nemuseli dediť od tried rámca, ale používali iba anotácie, čo predstavuje slabšiu závislosť.

Výhody čistej architektúry

Aké sú výhody? Jednou z nich je, že jednotlivé vrstvy majú rôzne dôvody na zmenu. Vnútorné vrstvy sa menia, keď sa menia biznis požiadavky. Vonkajšie vrstvy sa menia z technických dôvodov alebo kvôli interakcii s vonkajším svetom, napríklad pri redizajne používateľského rozhrania.

Zmena používateľského rozhrania by nemala vyžadovať zmenu biznis logiky. Rovnako zmena externého API alebo databázy, napríklad prechod z PostgreSQL na MongoDB, by nemala znamenať prepis doménovej logiky. Doménová logika by nemala obsahovať technické detaily, ako sú konkrétne SQL dotazy.

Ďalšou výhodou je testovanie. Testovateľnosť je jeden zo základných princípov dobrej architektúry. Automatické testy znižujú cenu vývoja, pretože každá zmena je lacnejšia – vieme rýchlo overiť, či sme nepokazili existujúcu funkcionalitu.

Aj keď písanie testov niečo stojí, v dlhodobom horizonte ušetrí veľa času. Pri väčších a dlhodobo vyvíjaných systémoch cena zmien rastie, ak nemajú dobrú architektúru a dobré testy. Bez testov je každá zmena čoraz náročnejšia a pomalšia.

Architektúra s oddelenými vrstvami umožňuje testovať vnútorné vrstvy nezávisle od vonkajších. Funkcie, ktoré dostanú vstup a vrátia výstup, sa testujú veľmi ľahko. Ak však kód pracuje s databázou, súbormi alebo externým API, testovanie je omnoho zložitejšie.

Preto sa snažíme, aby vonkajšie vrstvy boli čo najtenšie a obsahovali minimum logiky. Slúžia len ako jednoduché prevodníky dát medzi rôznymi podobami. Tento prístup sa nazýva *Humble Object Pattern*.

Ideálne je, ak je kód vo vonkajších vrstvách taký jednoduchý, že je na prvý pohľad zrejmé, že sa tam nemá čo pokaziť. Väčšina logiky má byť sústredená vo

vnútorných vrstvách, ktoré sú nezávislé od technických detailov.

Tento princíp sa objavuje aj vo funkcionálnom programovaní, kde sa striktne oddeľujú vstupno-výstupné operácie od logiky. Rovnaký princíp teda funguje v rôznych oblastiach.

Spôsoby rozdelenia systému

To je základný princíp čistej architektúry. Okrem toho existujú aj ďalšie princípy návrhu architektúry. Jedným z nich je spôsob rozdelenia systému – buď podľa technických vrstiev, alebo podľa domény.

Diagram s kruhmi ukazuje rozdelenie podľa technických vrstiev. Zároveň však môžeme systém deliť aj podľa domény. Pri veľkom systéme, napríklad pre finančnú inštitúciu, môžeme mať samostatné časti pre účty, úvery alebo poistenie.

Jeden prístup je delenie podľa technických činností, druhý podľa domény. Tieto prístupy sa väčšinou kombinujú. Otázkou je, ktorý z nich bude hlavný. Či vytvoríme pre každú doménovú časť samostatný „kruh“ prepojený cez API, alebo jeden kruh, kde sú doménové časti rozdelené vo vnútri vrstiev.

Ide o trade-off, ktorý závisí od konkrétnej domény a toho, ako jasne sú jej časti oddeliteľné.

3.7 Architektonické štýly

Následne sa dostávame k architektonickým štýlom, teda ku konkrétnym štýlom realizácie. Predtým sme hovorili o všeobecných princípoch a teraz máme konkrétne štýly, teda spôsoby, ako deliť systém.

Veľmi často sa vyskytuje takzvaný **big ball of mud**, teda „veľká kopa blata“. Existuje dokonca článok, ktorý tento architektonický vzor opisuje (Foote a Yoder 1997). Je o vlastnostiach takejto architektúry, ktorá väčšinou nevznikne zámerne, ale postupne. Začne sa niečo vyvíjať, pridávajú sa funkcie, všetko sa lepí jedno na druhé, až z toho vznikne veľká kopa blata. Cena ďalších zmien potom rastie neúmerne – každá ďalšia zmena je náročnejšia než predchádzajúca. Tomuto by sme sa mali vyhnúť.

Naopak, dobré architektonické štýly sú napríklad tieto:

Monolitic Architectures:

- Layered (n-tier)
- Pipeline
- Microkernel

Distributed Architectures:

- Service-Based
- Event-Driven
- Space-Based
- Orchestration-Driven Service-Oriented Architecture
- Microservices Architecture

Je ich viac a tento prehľad je z knihy *Fundamentals of Software Architecture* (Richards a Ford 2020). Každý z nich je tam opísaný detailnejšie, takže ja ich tu nebudeme rozoberať do hĺbky.

Prvým je **vrstvená (layered) architektúra**. Ide o rozdelenie na vrstvy, napríklad databázu, biznis logiku a používateľské rozhranie. Klasicky hovoríme o troch vrstvách: databáza, serverová časť a klientská časť používateľského rozhrania, ale tých vrstiev môže byť aj viac. Ide o technologické delenie. Keď sa na to pozrieme ako na kruh a urobíme z neho výsek, dostaneme vrstvy ako používateľské rozhranie, logiku a databázu.

Ďalším štýlom je **pipeline**, kde sa dáta spracúvajú postupne v jednotlivých krokoch. V niektorých typoch úloh sa to môže veľmi dobre hodiť.

Mikrokernel je architektúra, kde máme malé jadro a k nemu pluginy alebo rozšírenia. Jadro by malo byť čo najmenšie a väčšina funkcionality by mala byť v rozšíreniach. Klasickým príkladom sú rôzne IDE, ktoré majú jadro zabezpečujúce základnú funkcionalitu a všetko ostatné riešia rozšírenia.

Tieto prvé tri štýly sú v zásade **monolitické**, teda všetko beží v rámci jedného procesu alebo jedného systému. Potom tu máme architektúry **distribuované**, kde máme samostatné procesy, prípadne bežiace na samostatných počítačoch, ktoré medzi sebou komunikujú cez sieť.

Medzi ne patria architektúry **založené na službách**, kde je systém rozdelený na viacero služieb, ktoré zodpovedajú jednotlivým častiam domény. Pre každú časť domény máme samostatnú službu, ktorá ju realizuje. Samotná služba môže byť napríklad viacvrstvová. Služby môžu mať spoločné používateľské rozhranie alebo každá vlastné, môžu mať spoločnú databázu alebo každá vlastnú. Variabilita je tu veľká.

Ďalším typom sú architektúry **založené na udalostiach**, kde jednotlivé časti komunikujú pomocou udalostí a message queues, v ktorých sa udalosti spracúvajú.

Existuje aj **space-based** architektúra, ktorá je založená na práci s dátami bez jednej centrálnej databázy, pričom dáta sú distribuované.

Service-oriented architecture je opäť prístup založený na službach, ktorý bol istú dobu veľmi populárny. Existuje tu centrálny message hub alebo orchestrátor, ktorý riadi komunikáciu medzi jednotlivými službami. Okrem služieb tu teda máme aj orchestračný subsystém, ktorý môže byť pomerne komplexný. Často išlo o komerčné alebo open-source riešenia.

Microservice-oriented architecture je tomu blízka, ale ide ešte viac do extrému. Systém je rozdelený na veľké množstvo menších služieb. Delenie by malo vychádzať z domény. Zoberieme doménu problému a snažíme sa nájsť čo najmenšie, relatívne nezávislé časti, ktoré vieme oddeliť. Každá z nich je samostatná služba a služby medzi sebou komunikujú. Prepojenie medzi službami by malo byť naopak čo najjendnouchšie.

Každý architektonický štýl má svoje výhody aj nevýhody a svoje vhodné použitie. Vyžaduje si to samostatné štúdium a rozhodovanie, čo použiť a prípadne ako jednotlivé prístupy kombinovať.

3.8 Monolitické alebo distribuované architektúry

Ešte pár poznámok k monolitickým versus distribuovaným architektúram. Distribuované architektúry majú viacero výhod, napríklad flexibilitu. Jednotlivé služby sú oddelené procesy. Môžeme ich implementovať v rôznych programovacích jazykoch podľa toho, čo sa hodí pre danú časť – používateľské rozhranie, napríklad, v JavaScripte, prácu s umelou inteligenciou v Pythone, biznis logiku v Jave a podobne. Súvisí s tým aj to, že na jednotlivých službách môžu pracovať nezávislé tímy. Distribuovanosť tak často kopíruje rozdelenie tímov.

Veľmi zaujímavou vlastnosťou je vynútenie hraníc. V monolitickom systéme, kde sú moduly len na úrovni balíkov, je veľmi ľahké porušiť hranice medzi nimi. Je jednoduché zavolať metódu, ktorá mala byť interná, a ani si to nevšimneme. Programovacie jazyky síce ponúkajú nástroje ako prístupové modifikátory, ale nie vždy je to dostatočná ochrana. Výsledkom môže byť príliš silné previazané komponentov.

Ak máme systém rozdelený na samostatné procesy komunikujúce cez sieť, komunikačný protokol je jasne daný. Nemôžeme len tak zavolať metódu, musíme použiť definovaný protokol. Táto komunikácia je však drahšia – HTTP požiadavka je náročnejšia než volanie metódy. Pridáva sa latencia, možnosť zlyhania a ďalšie komplikácie. To nás núti komunikáciu minimalizovať, čo môže byť výhoda, ale aj nevýhoda. Ak zle navrhneme rozdelenie a služby musia komunikovať príliš často, zistíme, že to bolo nevhodné a zmena je už náročná.

Výhodou distribuovaných architektúr je aj škálovateľnosť. Ak je nejaká časť

výkonovo náročná, vieme ju spustiť viackrát na rôznych počítačoch. Veľkou nevýhodou je však zložitosť. Namiesto jednoduchých volaní metód riešime sieťovú komunikáciu a množstvo súvisiacich problémov.

Existuje celý súbor tzv. klamných predpokladov distribuovaného výpočtu (Deutsch 1994).

Fallacies of distributed computing:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

V skutočnosti to neplatí. Sieť je nespoľahlivá, latencia existuje, prenosové pásmo je obmedzené, sieť nemusí byť bezpečná a topológia sa môže meniť. Toto všetko zvyšuje komplexnosť distribuovaných architektúr.

Distribuované architektúry teda majú svoje výhody aj nevýhody. Opäť ide o trade-off a o rozhodnutie v konkrétnej situácii.

3.9 Architektonické rozhodnutia

A ešte jedna poznámka k architektonickým rozhodnutiam. Tie sa dajú zaznamenávať a existuje dokonca aj taký formát, ktorému sa hovorí Architecture Decision Records (ADR) (Nygard 2011).

Väčšinou sú to čisto textové Markdown súbory alebo niečo podobné. Odporúča sa mať ich buď v Markdown formáte priamo v Gite spolu s kódom aplikácie, alebo ako wiki zdieľané pre celý tím.

Takýto záznam by mal obsahovať Tieto položky:

1. Title
2. Context
3. Decision
4. Status
5. Consequences

Začína to názovom rozhodnutia. Ďalej by tam mal byť kontext: čo vlastne riešime a prečo. Aké sú požiadavky, v akej situácii sa nachádzame, aké sú výhody a nevýhody rôznych riešení. Nasleduje samotné rozhodnutie a prečo sme sa rozhodli pre konkrétne riešenie. Môže ísť napríklad o spôsob dekompozície, architektonický štýl, konkrétne rozdelenie systému alebo princíp, ktorý budeme dodržiavať. Potom je tam jeho stav – teda či ide o niečo, čo je len diskutované, prijaté, zamietnuté alebo neskôr nahradené iným rozhodnutím. Nasleduje časť o následkoch. Každé rozhodnutie má svoje dôsledky – pozitívne aj negatívne – a je dobré ich stručne zhrnúť.

Takéto dokumenty vieme počas vývoja systému vytvárať, diskutovať o nich v rámci tímu, prijímať rozhodnutia a ukladať ich pre budúcnosť. Vďaka tomu sa k nim vieme spätne vrátiť, zistiť, prečo bolo rozhodnutie urobené, a ak sa kontext zmení, rozhodnutie aj prehodnotiť alebo zmeniť. Preto je dobré mať ich zaznamenané.

Je to bežná prax. Odporúčajú to aj veľké firmy – napríklad Microsoft, Red Hat a ďalšie – a majú tento formát uvedený vo svojej dokumentácii ako odporúčaný postup.

Literatúra

Brooks, Jr., F. P. 1987. "No Silver Bullet: Essence and Accidents of Software Engineering". *Computer* 20 (4): 10–19. https://doi.org/10.1109/MC.1987.1663532.

Deutsch, Peter. 1994. "The eight fallacies of distributed computing". https://nighthacks.com/jag/res/Fallacies.html.

Evans, Eric. 2004. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley.

Farley, David. 2022. *Modern Software Engineering: Doing What Works to Build Better Software Faster*. Boston: Addison-Wesley.

Foote, Brian, a Joseph Yoder. 1997. "Big ball of mud". *Pattern languages of program design* 4: 654–92. http://www.laputan.org/mud/.

Liskov, Barbara H., a Jeannette M. Wing. 1994. "A Behavioral Notion of Subtyping". *ACM Transactions on Programming Languages and Systems* 16 (6): 1811–41. https://doi.org/10.1145/197320.197383.

Martin, Robert C. 2016. "A Little Architecture". 2016. http://blog.cleancoder.com/uncle-bob/2016/01/04/ALittleArchitecture.html.

Martin, Robert C., James Grenning, Simon Brown, a Kevlin Henney. 2018. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Prentice Hall.

Nygard, Michael. 2011. "Documenting architecture decisions". 2011. https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions.

Parnas, D. L. 1972. "On the Criteria to Be Used in Decomposing Systems into Modules". *Communications of the ACM* 15 (12): 1053–58. https://doi.org/10.1145/361598.361623.

Richards, Mark, a Neal Ford. 2020. *Fundamentals of Software Architecture: An Engineering Approach*. First edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly.

Spolsky, Joel. 2002. "The law of leaky abstractions". 2002. https:

//www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/.

Názov:	Pokročilé softvérové inžinierstvo
Podnázov:	Poznámky k prednáškam
Autori:	prof. Ing. Jaroslav Porubän, PhD. doc. Ing. Štefan Korečko, PhD. Ing. Sergej Chodarev, PhD.
Vydavateľ:	Technická univerzita v Košiciach
Rok:	2025
Vydanie:	prvé
Náklad:	50
Rozsah:	28 strán

ISBN XXX-XX-XXX-XXXX-X