

NÁSKOK
DÍKY
ZNALOSTEM

PROFINIT

Architektura a design

Kolektiv autorů

Říjen 2018

Schematický pohled

(Software System) Architecture

- Struktura
- Dokumentace této struktury

Základní typy architektury

- Software architecture
- Business (process) architecture
 - obchodní strategie, řízení, organizace, obchodní procesy
- Information technology (system) architecture
 - HW a SW infrastruktura nutná pro chod organizace
- Information architecture
 - organizace a správa dat (MDM, BI, DWH, ...)

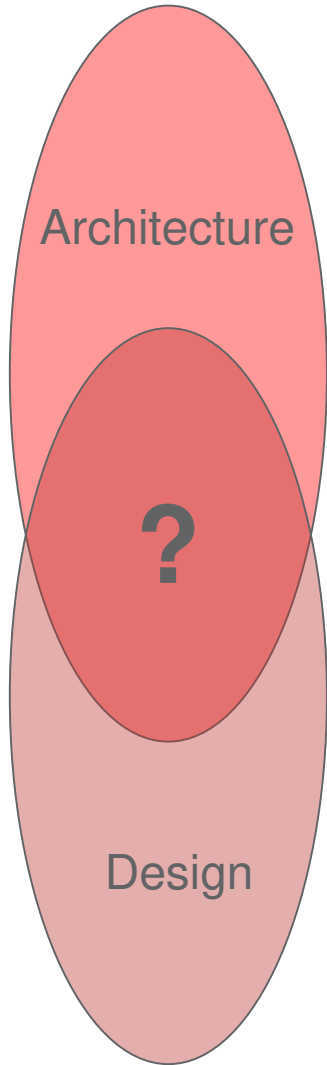


Enterprise architecture

Role a význam architektury

- na projektu?
- v podniku?

Architektura vs. Design



Software architecture

- Realizace **nefunkčních** požadavků
- Strategický design
 - Programovací paradigmatata, architektonické styly, principy, standardy, ...

Software design

- Realizace **funkčních** požadavků
- Taktický design
 - Design patterns, programovací idiomy, refaktoring, ...

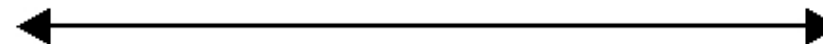
*„Architecture is about the **important** stuff. Whatever that is ...“
Martin Fowler, Who needs an Architect ?*

Softwarový proces



PROJECT MANAGEMENT / QUALITY ASSURANCE / DOCUMENTATION / CONFIGURATION MANAGEMENT / RELEASE MANAGEMENT / DEVOPS

Time



LCO

LCA

IOC

Stages

Process Activities

Requirements Capture

Analysis & Design

Implementation

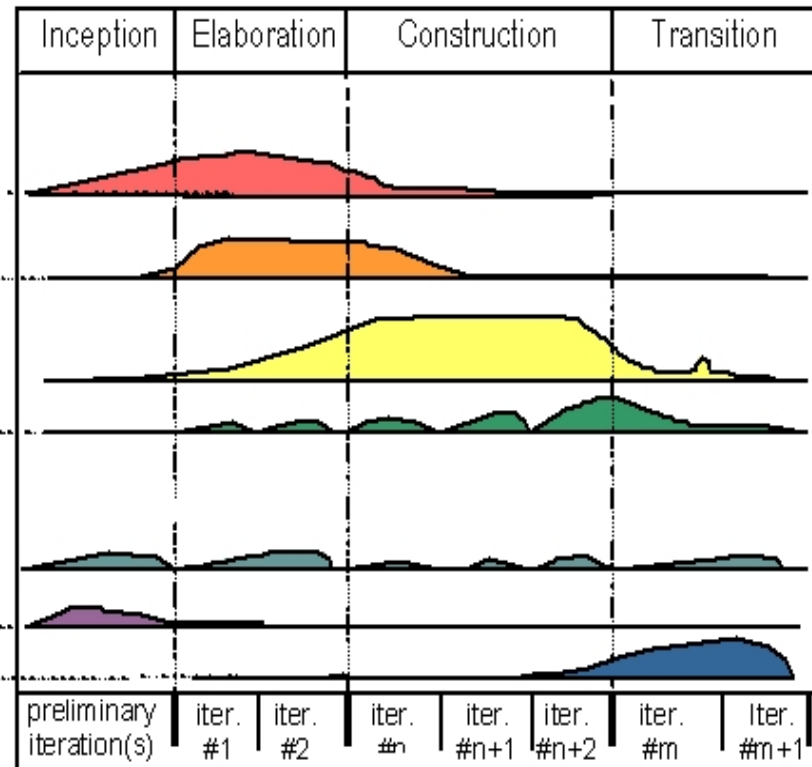
Test

Supporting Activities

Management

Environment

Deployment



Iterations

Dokumentace architektury

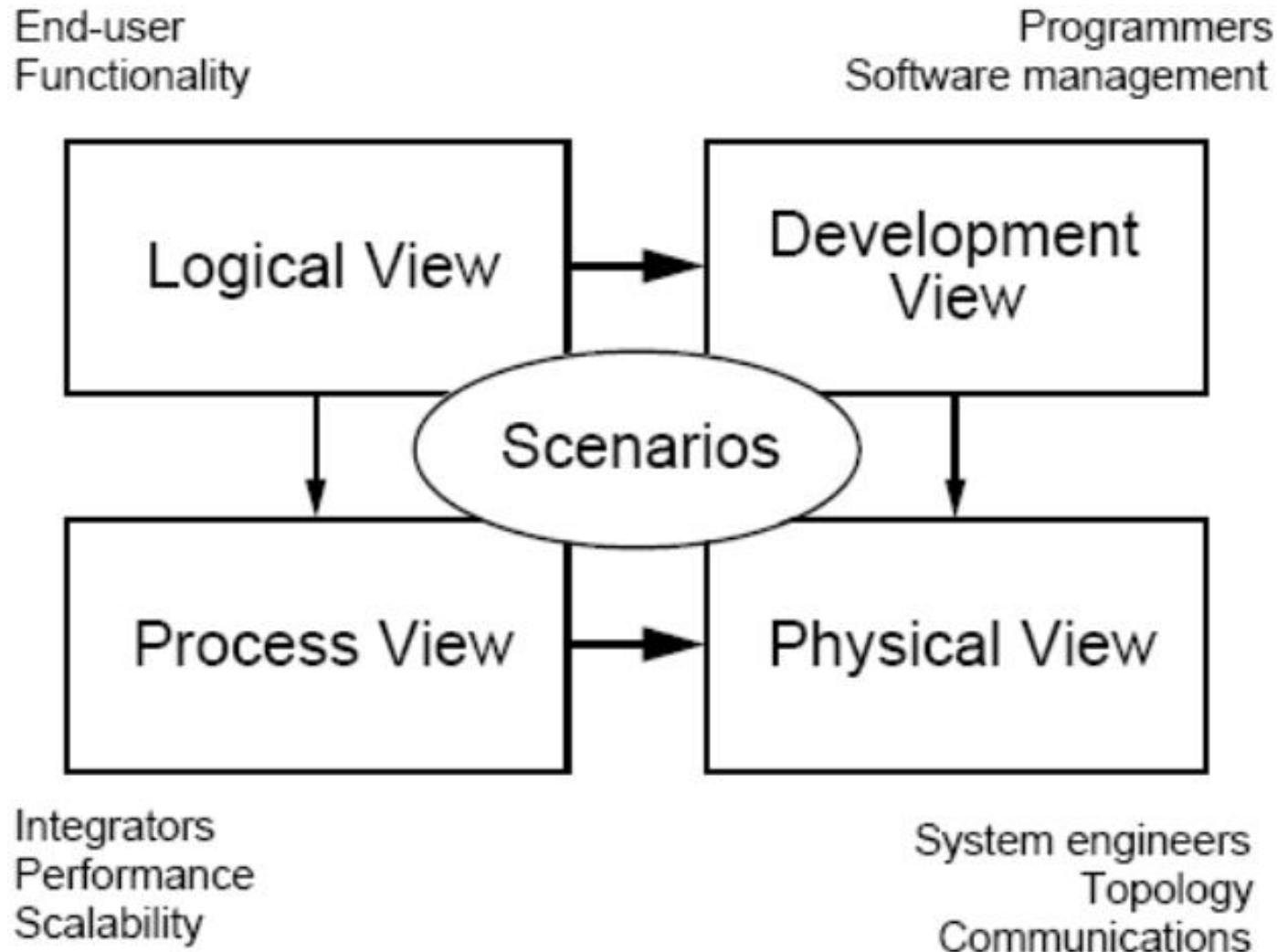


Figure 1 — The “4+1” view model

Softwarová architektúra dle IEEE 1471

- Functional / logic view
- Code / module view
- Development / structural view
- Concurrency / process/thread view
- Physical / deployment view
- User action / feedback view
- Data view

Vliv kontextu na architekturu

- databázový systém / subsystém
- web systém / subsystém
- (tlustý) klient systém / subsystém
- OO systém / subsystém
- data warehouse systém
- integrační systém / subsystém
- ...

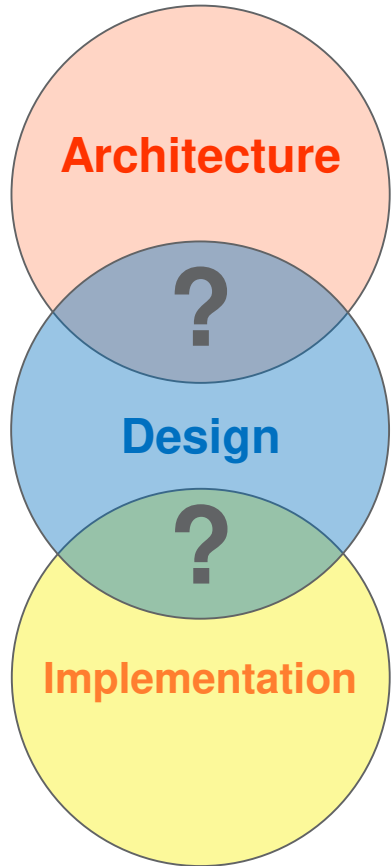
Odbočka - Enterprise architektura

- Architektura na úrovni celé společnosti (organizace).
 - Enterprise architecture zahrnuje popis cílů organizace, způsobů jak jsou tyto cíle dosahovány pomocí podnikových procesů a způsobů, jak mohou tyto procesy být podpořeny technologiemi (A Better Path to Enterprise Architectures, Roger Sessions)
- Proč je potřeba?
 - Velké společnosti mají tisíce aplikací => těžké udržet pořádek
- Archimate
 - Business vrstva – produkty a služby zákazníkům
 - Aplikační vrstva – aplikace (aplikační komponenty)
 - Technologická vrstva – infrastruktura (Node, Device, System Software, ...)
 - Příklad: https://www.msk.cz/cz/verejna_sprava/korporatni-architektura-moravskoslezskeho-kraje-83244/

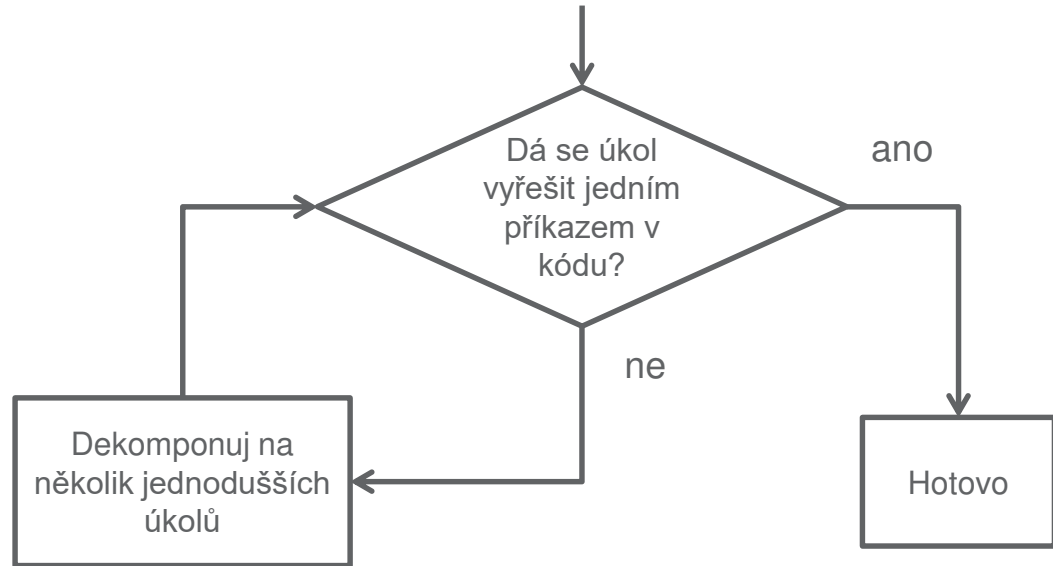


Design

Architektura vs. návrh vs. implementace



Návrh v kostce:



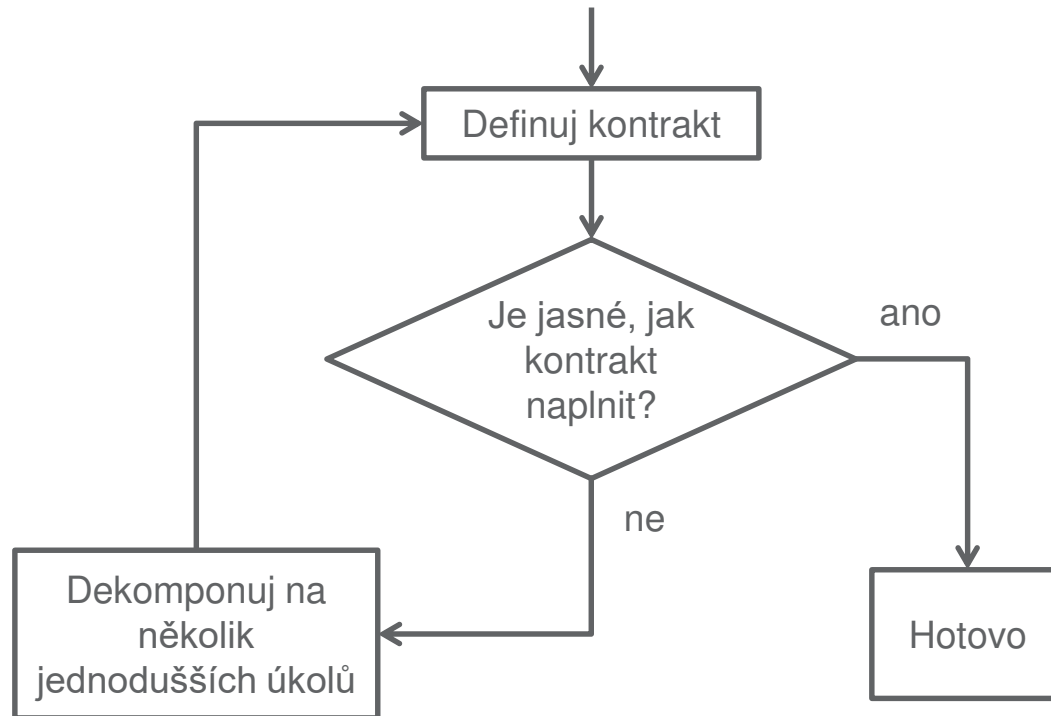
Kontrakt



Kontrakt

```
public class MyClass extends MyBaseClass implements MyInterface {  
    public static final int THE_ANSWER = 42;  
  
    protected MyClass() {  
        System.out.println("Hello World!");  
    }  
  
    public final ArrayList<String> handleStuff(ArrayList<Object> input)  
        throws IOException {  
        return myHandleStuff(input);  
    }  
  
    protected abstract ArrayList<String> handleStuff(  
        ArrayList<Object> input) throws IOException;  
  
}
```

Návrh v kostce – lépe



Základy dobrého (OOP) návrhu

- › Decomposition
- › Abstraction
- › High cohesion
- › Loose coupling
- › Encapsulation



Cohesion (soudržnost)

Service1
+ overHeslo () : boolean + tiskniFakturu () : void

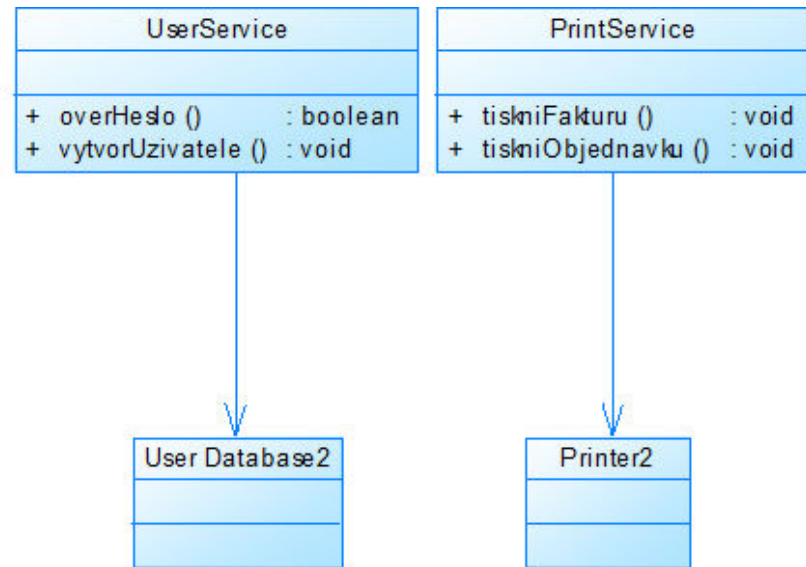
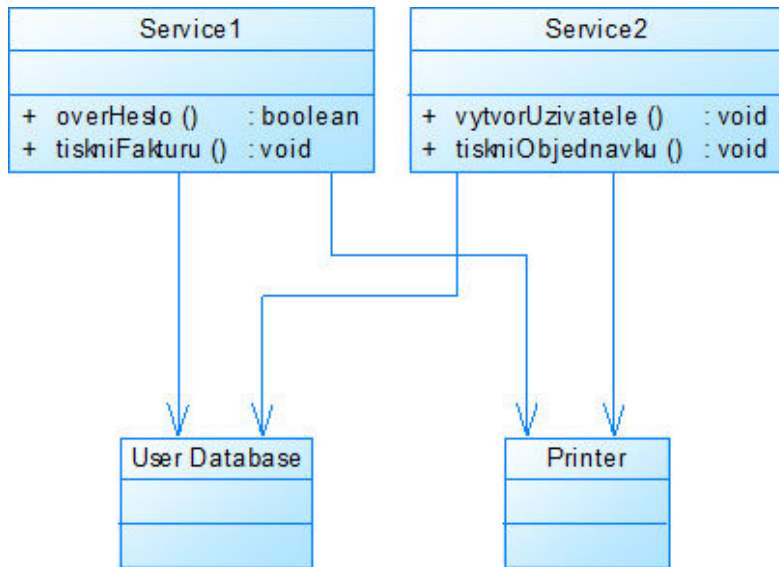
Service2
+ vytvorUzivatele () : void + tiskniObjednavku () : void

VS.

UserService
+ overHeslo () : boolean + vytvorUzivatele () : void

PrintService
+ tiskniFakturu () : void + tiskniObjednavku () : void

Coupling (provázanost)



Loose coupling

- › Cíl: Modul (třída, ...) má co nejméně záviset na svém okolí
- › Výhody:
 - Odolnost proti změnám okolí
 - Snazší pochopení
 - Znovupoužitelnost
 - Testovatelnost
- › Pozor na skryté vazby
 - Časové
 - Sousednost událostí

```
/**
 * ...
 * Pozor! Tato metoda musí být volána až po metodě ...
 */
```

Encapsulation

- › Cíl:
 - Znemožnit okolí záviset na mých implementačních detailech
 - Ochránit vnitřní stav před vnějšími zásahy
- › Výhody:
 - Omezení dopadu změn na okolí („Encapsulate what changes“)
 - Snazší testování a verifikace
 - Snazší debugging
- › Interfaces
 - Ultimátní podoba zapouzdření – implementační detaily tu vůbec nejsou
 - Jasně definují kontrakt a oddělují ho od implementačních detailů
- › Gettery a Settery

Co zbylo z OOP?

- › Encapsulation
 - Rozhodně
- › Polymorphism
 - Ano, ale omezený často na interface inheritance
- › Inheritance
 - Prefer composition over inheritance
 - „Implementation inheritance“ považována za nevhodnou

Z dokumentace `java.sql.Timestamp` (extends `java.util.Date`):

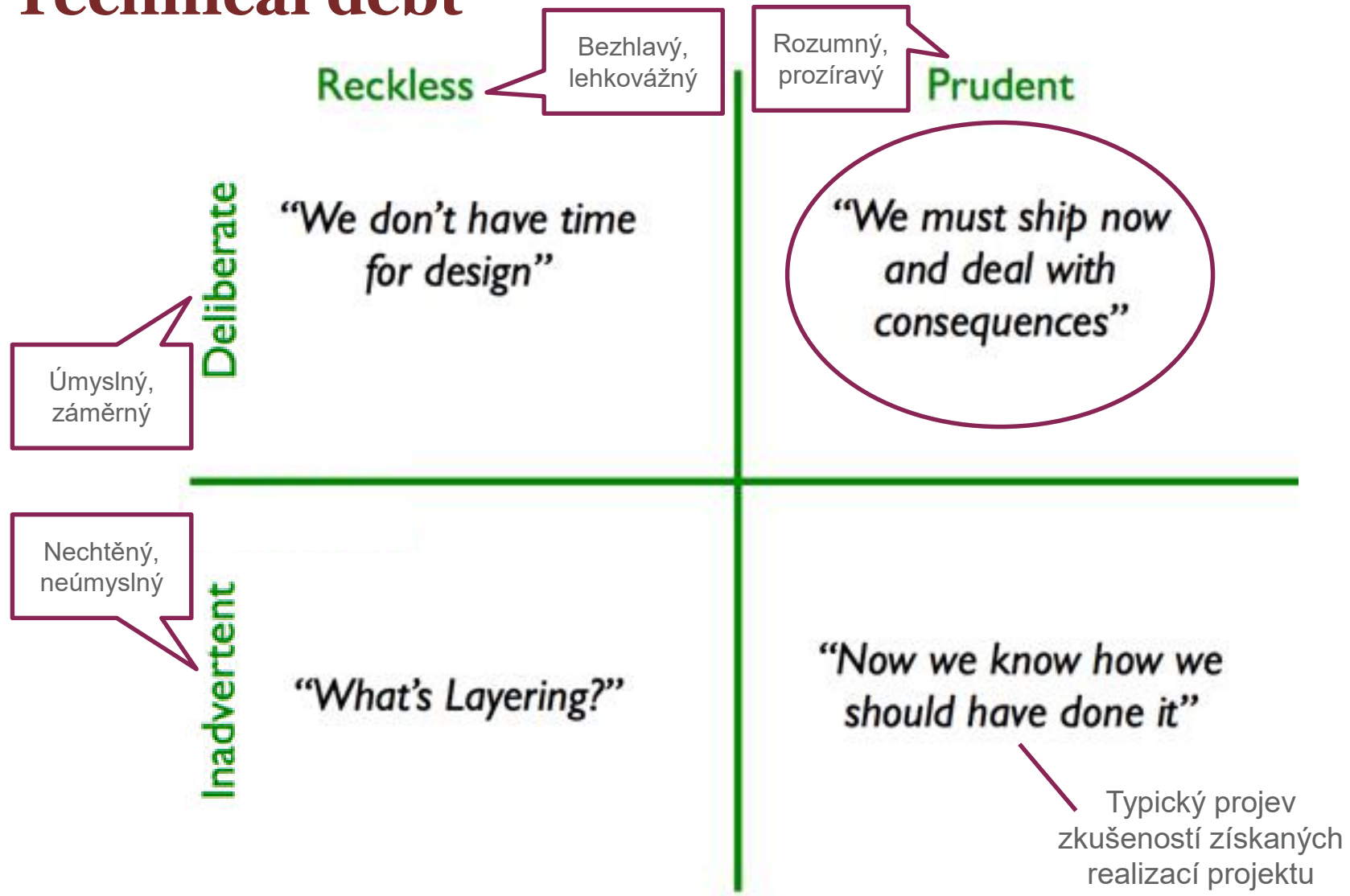
... As a result, the `Timestamp.equals(Object)` method is not symmetric with respect to the `java.util.Date.equals(Object)` method ...

*Due to the differences between the `Timestamp` class and the `java.util.Date` class mentioned above, it is recommended that code not view `Timestamp` values generically as an instance of `java.util.Date`. **The inheritance relationship between `Timestamp` and `java.util.Date` really denotes implementation inheritance, and not type inheritance.***

Technical debt

- Volba snadného řešení namísto koncepčního řešení
 - Často pod tlakem na termín. Snaha rychle „něco“ dodat
- = náklady (pracnost), které bude třeba v budoucnu vynaložit (pokud budeme chtít mít systém bez problémů udržitelný)
- Existují nástroje, které změří na úrovni kódu -
<https://docs.sonarqube.org/display/SONARQUBE52/Technical+Debt>
 - Toto samozřejmě nezahrnuje špatná architektonická rozhodnutí.
- Typické problémy:
 - Tightly-coupled components
 - Zanedbaný refactoring
 - Nedostatečné pokrytí testy
 - Nedostatečné znalosti (zejména bezpečnost, paralelismus, ...)

Technical debt



Design patterns

Katalog

- základní GOF návrhové vzory
- prakticky nekonečné kombinace a variace

Význam

- znovupoužitelnost
- společný jazyk
- ...

Pozor

- na počáteční nadšení
- na nadbytečné užívání patterns
 - indirection, úrovně abstrakce
 - složitost

Jaké znáte patterns ?

Frameworks

- Znovupoužitelný návrh pro SW systém
- Podpora (základna) při vývoji jiných SW aplikací
- Diktuje architekturu systému
- Určuje jak dekomponovat systém a jak budou jeho jednotlivé části komunikovat
- Základní dekompozice
 - **Frozen spots** – definice celkové architektury, neměnné
 - **Hot spots** – zajišťují rozšiřitelnost (abstraktní třídy, anotace)

Co odlišuje framework od knihovny - shrnutí

- Inversion of control
- Rozšiřitelnost
- Nemodifikovatelnost
- Defaultní chování



Zásady elementárního návrhu

DRY – Don't Repeat Yourself

› Controller:

```
...  
params.put("axis_length", axisLength);  
...
```

› Window:

```
...  
Integer axisLength = params.get("axis_length");  
...
```

› Dialog:

```
...  
if (params.get("axis_lenght") != null) {  
    ...  
}  
...
```

SRP – Single Responsibility Principle

- › Každý logický celek (modul, třída, metoda, ...) má dělat vždy jen jednu věc
 - Co když budu potřebovat přepoužít jen jeden z několika kroků?
- › Symptom: inline komentáře ve stylu: `// krok 2: teď uděláme ...`

```
// Spocítej poradi, ve kterem je nutne parsovat DDL skripty
```

```
...
```

```
// Odstran existujici DDL skripty ve vystupnim adresari
```

```
...
```

```
// Pro kazdy databazovy objekt ve spocitanem poradi ziskej jeho
```

```
// DDL a zparsuj ho
```

```
for (DdlName ddl : extractionOrder) {
```

```
...
```

```
    // Zparsuj DDL
```

```
...
```

```
    // Uloz DDL
```

Broken Windows

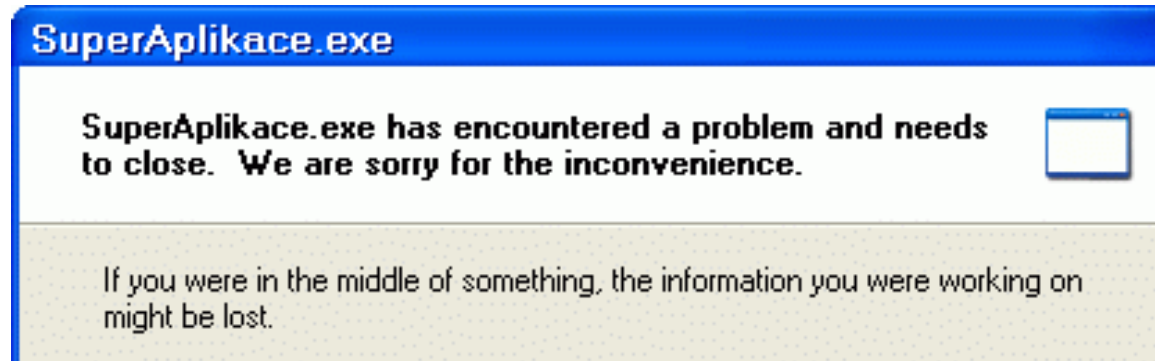


Broken Windows

„Stejně už je to zpravený“

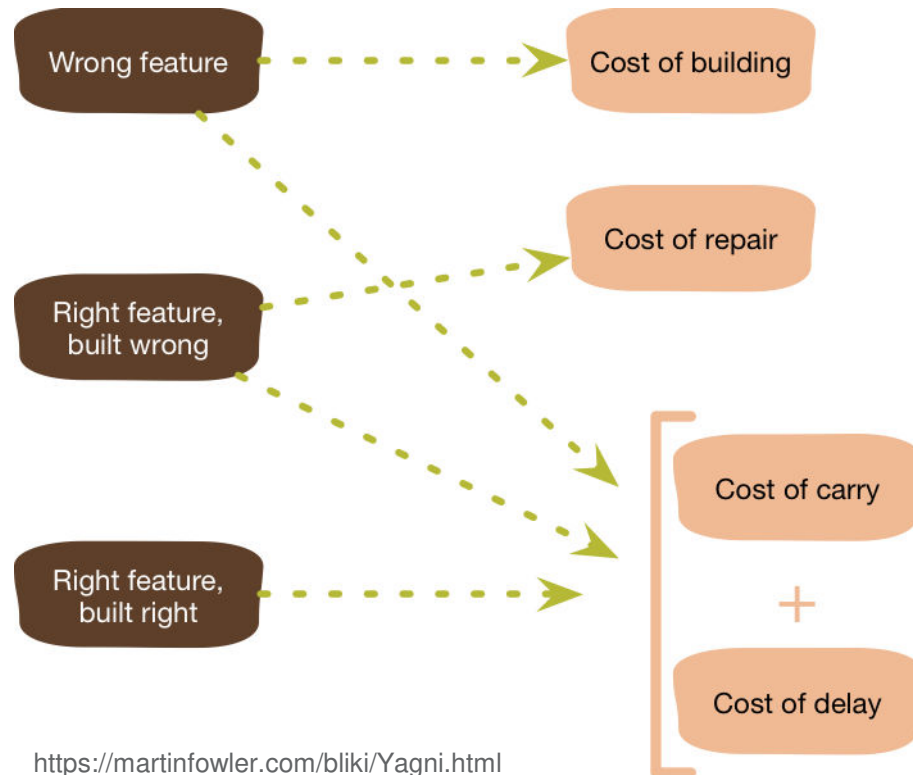
Fail Fast

- › Je-li v programu chyba, měl by selhat co nejdříve
- › Aktivně kontrolujte konzistenci a ošetřujte možné chyby
- › „Dead programs tell no lies“
- › Nejhorší je na chybu přijít až po databázovém commitu
- › Čím dříve chybu odchytíte, tím víc debugovacích informací můžete poskytnout
- › Necháte-li chybu probublat až do obecného error handleru, můžete už říct jen:



YAGNI a KISS

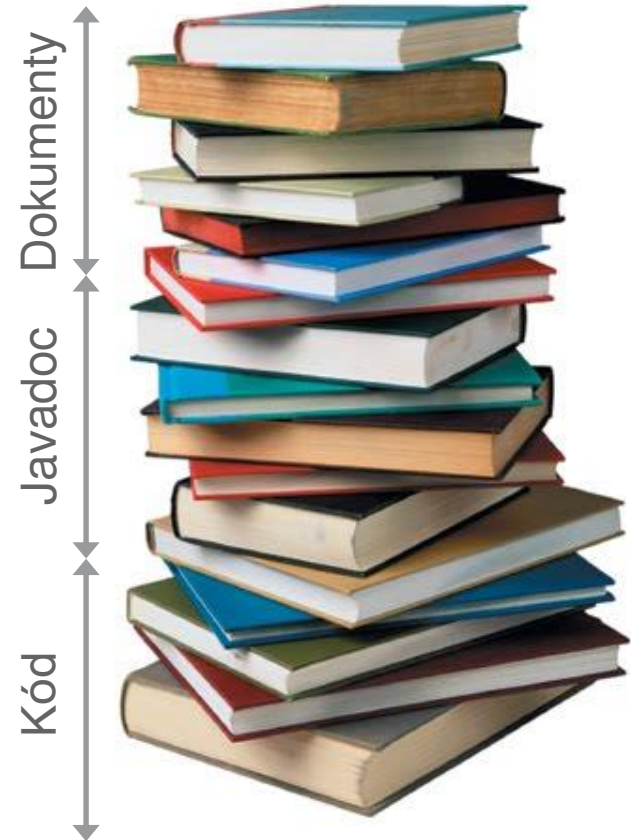
- › YAGNI – You Ain't Gonna Need It
- › KISS – Keep It Simple Stupid
- › POGE - Principle of Good Enough



<https://martinfowler.com/bliki/Yagni.html>

Self-documenting code

- › Kód, který se svou formou (strukturou, jmennými konvencemi apod.) snaží omezit nutnost číst dokumentaci
- › Neznamená úplnou absenci dokumentace
 - čitelný kód nenahradí koncepční dokumentaci (architektura, design moments, ...)
- › Problémy při chybějící dokumentaci
 - Význam větších funkčních celků
 - Pre/post conditions, invariants
 - Inheritance – kontrakt vůči potomkům



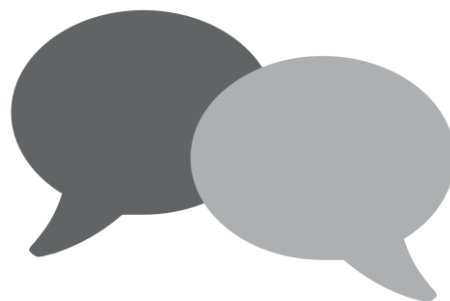
Další dobré rady

- › Premature optimization is root of all evil
- › Jak implementovat vlastní cache – rada první: nedělejte to
- › Jak implementovat vlastní transakce – viz předchozí bod
- › Jak implementovat vlastní lazy fetching, zámky, ... – však víte
- › Bezpečnost nelze do kódu přidat dodatečně
- › Thread-safety nelze do kódu přidat dodatečně

Thread-safety

- › Immutable třídy jsou inherentně thread-safe
- › Třídy bez vnitřního stavu jsou inherentně thread-safe
 - „Immutable once configured“
- › Doporučuji všude, kde to jde:
 - Služby beze stavu (bez instančních proměnných kromě odkazů na jiné služby)
 - Stav předávat v parametrech metod a držet v lokálních proměnných
- › Takto napsaný modul:
 - Nemá režii na synchronizaci (vhodný i pro aplikace nepotřebující thread-safety)
 - Není nezbytně thread-safe, ale půjde to snadno zařídit
 - Je čitelnější – je zřejmé, kudy tečou data

```
public Graph generateDataflow(Tree tree) {  
    return handler.processNode(tree);  
}
```



Dotazy?

Děkuji za pozornost

PROFINIT

NÁSKOK DÍKY ZNALOSTEM

Profinit EU, s.r.o.

Tychonova 2, 160 00 Praha 6 | Telefon + 420 224 316 016



Web
www.profinit.eu



LinkedIn
linkedin.com/company/profinit



Twitter
twitter.com/Profinit_EU



Facebook
facebook.com/Profinit.EU



Youtube
Profinit EU