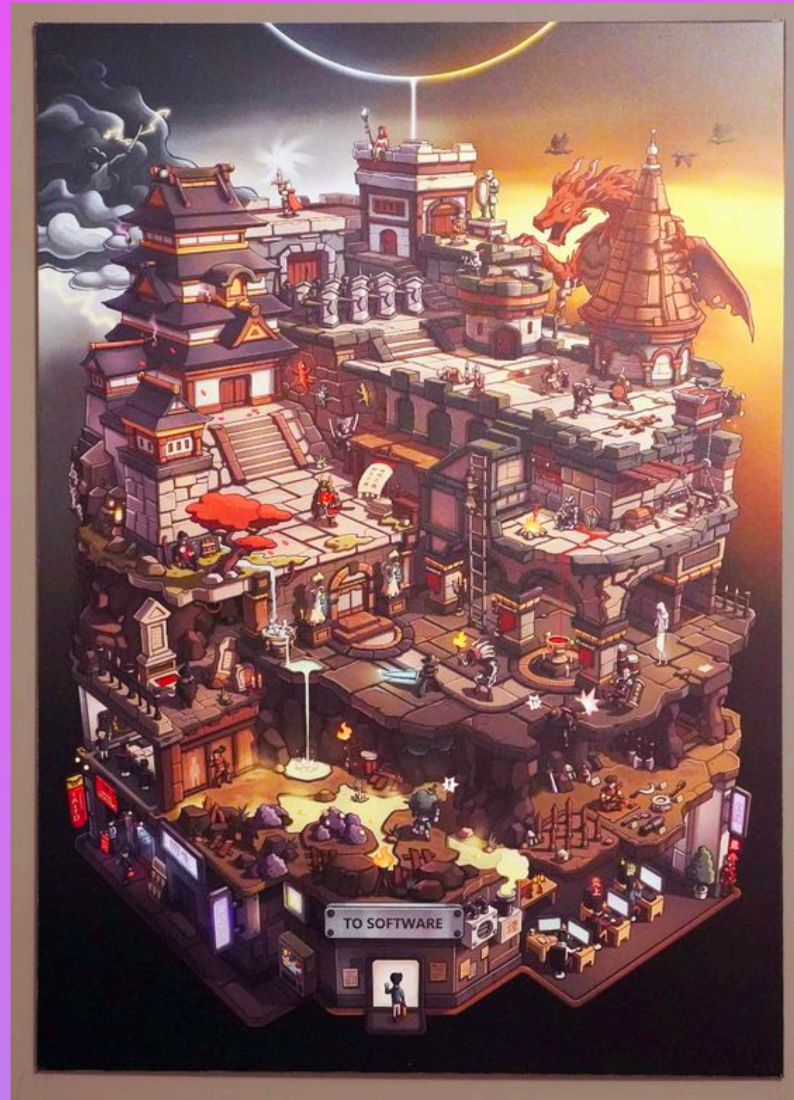
czm

Search . . .



01

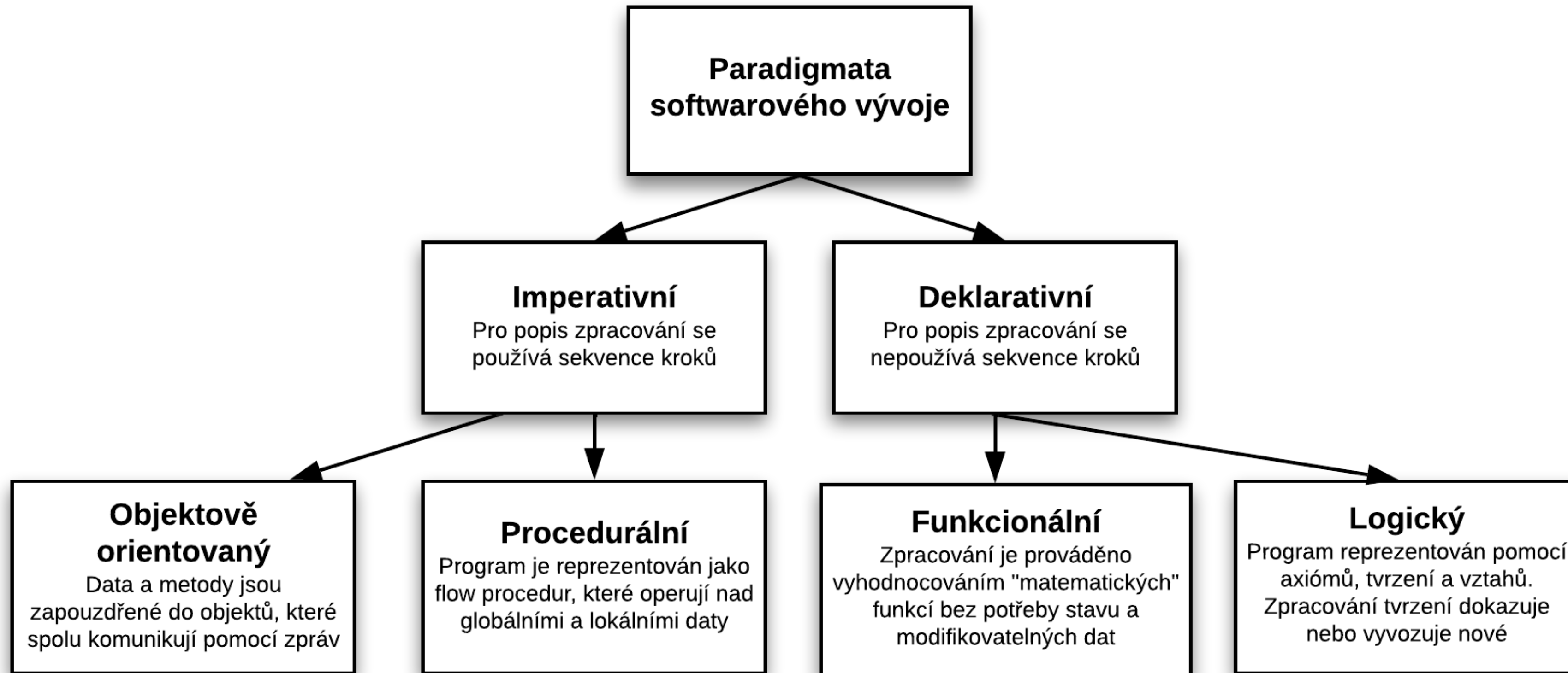
Klíčové koncepty modelování systémů

- Programovací paradigmata
- Deklarativní versus imperativní reprezentace
- Práce s komplexitou
- Dekompozice
- Hierarchie
- Základní abstrakce (jmenné, datové, funkcionální)

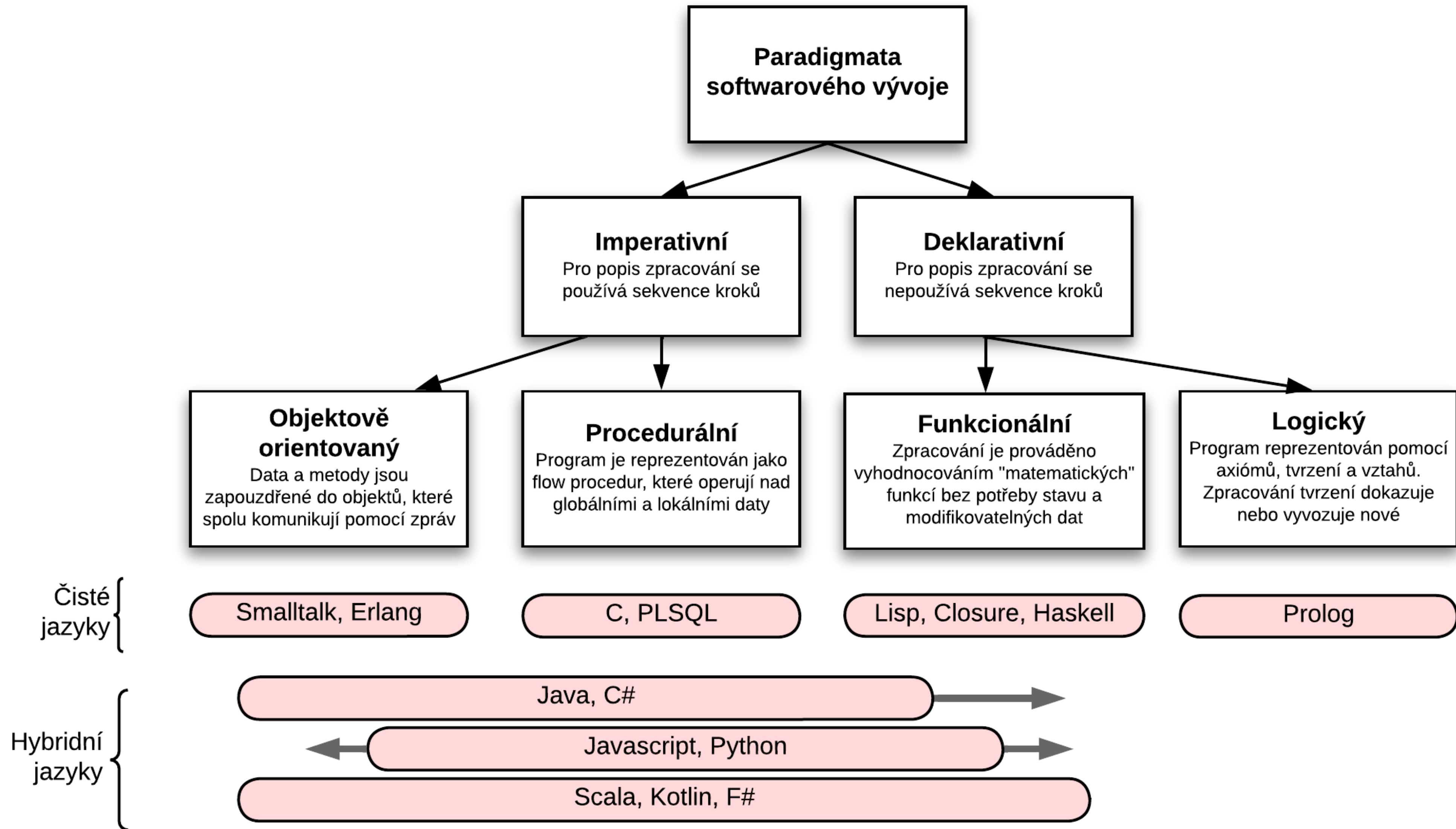
Objektové Modelování - podzim 2022

david.kadlecek@cvut.cz, www.czm.fel.cvut.cz

03 PARADIGMATA SOFTWAROVÉHO VÝVOJE



PARADIGMATA SOFTWAROVÉHO VÝVOJE



03 DEKLARATIVNÍ vs IMPERATIVNÍ PŘÍSTUP

- Výkres v geometrii - popisuje útvary a vztahy mezi nimi = “**What Is**” znalost, **Deklarativní reprezentace**
- Výměna oleje v autě - popisuje postup jako sekvenci činností = “***How To***” znalost, **Imperativní reprezentace**

Odmocnina ze dvou má deklarativní vyjádření: $\sqrt{x}$ je y takové, že $y^2 = x$ a $y \geq 0$

To samé lze imperativně vyjádřit ve formě postupu, kterým odmocninu ze dvou získám.

Například pomocí opakování následujícího postupu:

- *Odhadnout výsledek G*
- *Zlepšit odhad zprůměrováním G a x/G*
- *Zlepšovat odhad dokud není dostatečně dobrý*

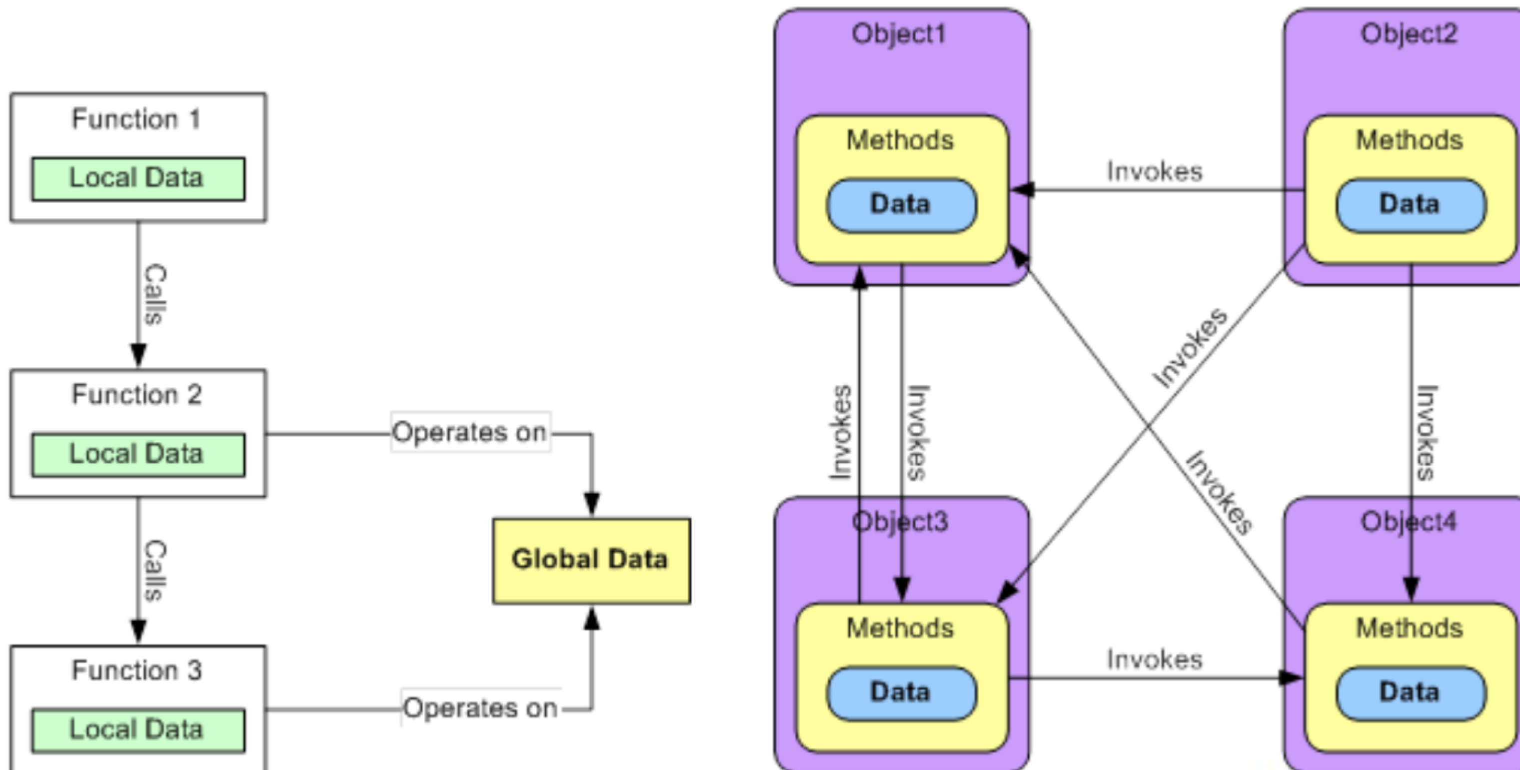
x = 2	G = 1
x/G = 2	G = ½ (1 + 2) = 3/2
x/G = 4/3	G = ½(3/2 + 4/3) = 17/12
x/G = 24/17	G = ½ (17/12 + 24/17) = 577/408 = 1.4142

3

03 PROCEDURÁLNÍ vs OBJEKTOVÉ PROGRAMOVÁNÍ

V procedurálním přístupu jsou základním stavebním kamenem programu procedury, které pracují nad lokálními a globálními daty, data mohou být organizována do záznamů (Record).

V objektovém přístupu jsou základním stavebním kamenem objekty, které zapouzdřují (a také kontrolují) volání metod a práci s daty, komunikují spolu pomocí zasílání zpráv.



4

03 PROCEDURÁLNÍ (IMPERATIVNÍ) vs FUNKCIONÁLNÍ

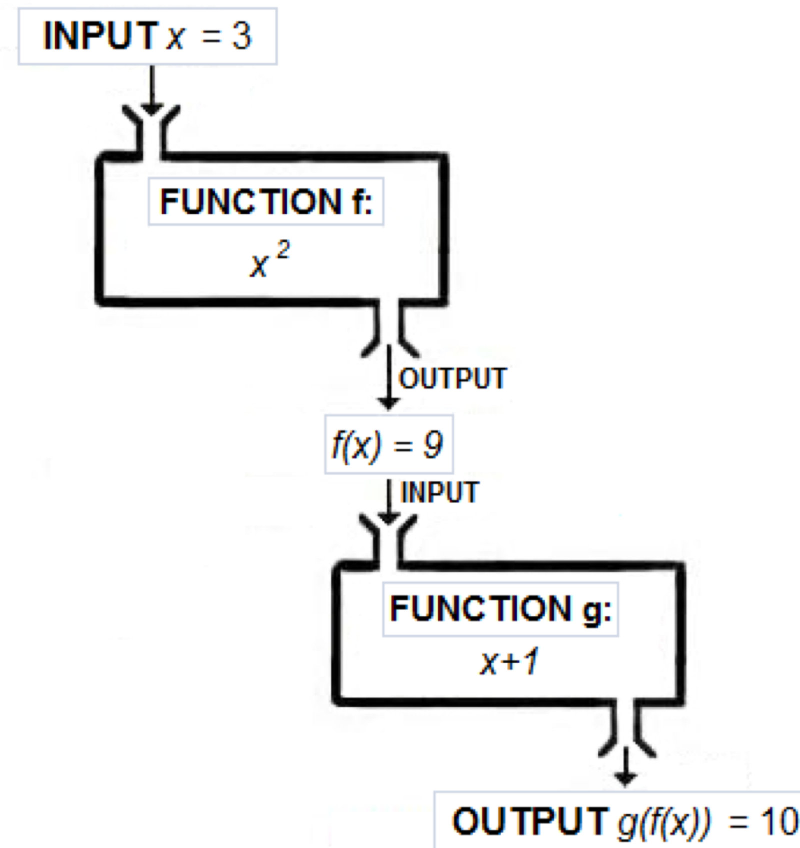
Procedurální přístup

Příklad výměna oleje:

1. Dojdi k autu
2. Otevři kapotu
3. Zkontroluj hladinu oleje
4. Jestliže je ho málo, tak vyměň olej
5. Zavři kapotu
6. ...

Funkcionální přístup

$$f(g(h(j(k(l(x))))))$$



01 PROCEDURÁLNÍ (IMPERATIVNÍ) vs FUNKCIONÁLNÍ

Ve funkcionálním přístupu jsou základním stavebním kamenem programu funkce, se kterými se pracuje jako s hodnotami.

Procedurální (imperativní) přístup

Mutable data, manipuluje se se stavem a objekty, iterace

- \+ Jednoduché porozumět kódu
- \+ Jednoduchý debugging
- \- Delší kód
- \- **Side efekty** při volání procedur
- \- Horší škálování a multithreading

Funkcionální přístup

Immutable data, funkce vyššího řádu, manipulace s funkcemi a datovými množinami, rekurze

- \+ Kratší kód
- \+ Lepší škálování
- \+ **Žádné side efekty** při volání funkce
- \- Horší porozumění kódu
- \- Pomalejší pro jednoduché volání
- \- Pomalejší učící křivka

01 LOGICKÉ PROGRAMOVÁNÍ

Vytvořím takový logický popis problému ze kterého je řešení logicky odvoditelné.

Logický program je deklarativní zápis posloupnosti příkazů (logických vět), které vyjadřují:

- Pravidla (jsou podmíněná)
- Fakta (jsou nepodmíněná)
- Dotazy (cílové klauzule)

Programátor tedy popíše problém, zadá otázky a stroj (problem solver) na ně nalezne odpovědi.

7

PROCEDURÁLNÍ (IMPERATIVNÍ) vs FUNKCIONÁLNÍ

Úloha:

Všichni studenti jsou mladší než Petrova matka. Karel a Mirka jsou studenti.

Kdo je mladší než Petrova matka?

Zápis v PL1:

$\forall x\,[\mathrm{St}(x) \supset \mathrm{Ml}(x, f(\mathrm{a}))]$

$\mathrm{St}(\mathrm{b})$

$\mathrm{St}(\mathrm{c})$

$\Rightarrow \exists y\, \mathrm{Ml}(y, f(\mathrm{a}))$???

Zápis v Prologu:

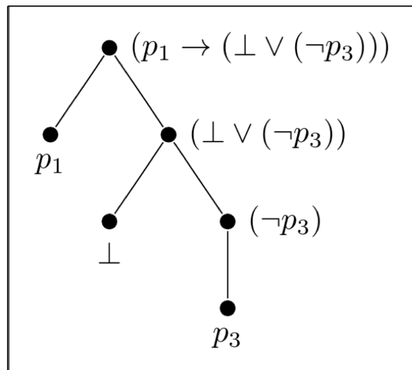
`mladsi(X, matka(petr)):- student(X).`	pravidlo
`student(karel).`	fakt
`student(mirka).`	fakt
`?- mladsi(Y, matka(petr)).`	dotaz

(PL1 - Predikátová logika prvního řádu)

8

01 SHRNUTÍ

Přístup logického programování

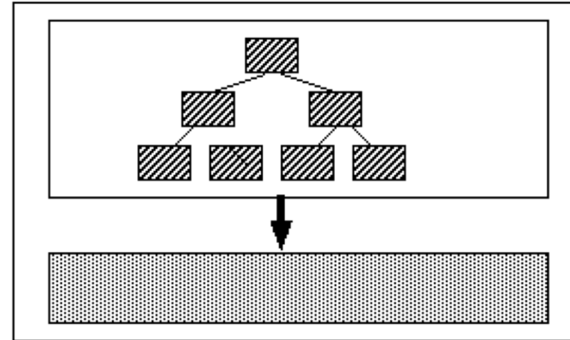


dotaz

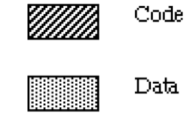
odpověď

Problem solver

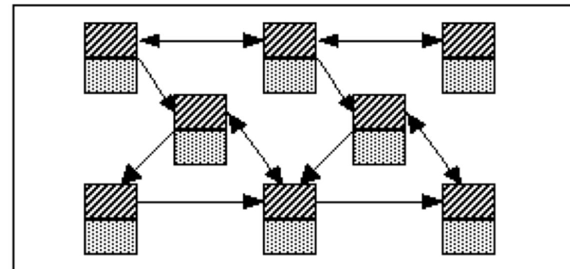
Procedurální přístup



Exekuce zahrnuje provádění kódu, který operuje nad daty



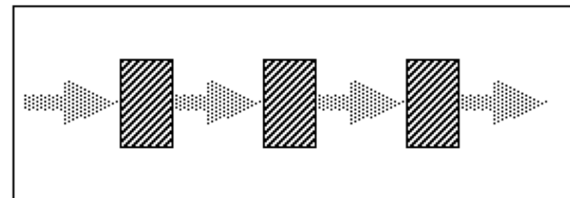
Objektový přístup



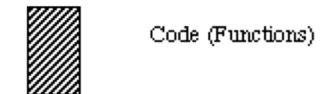
Objekt zapouzdřuje kód i data

Výpočet zahrnuje interakci mezi objekty

Funkcionální přístup

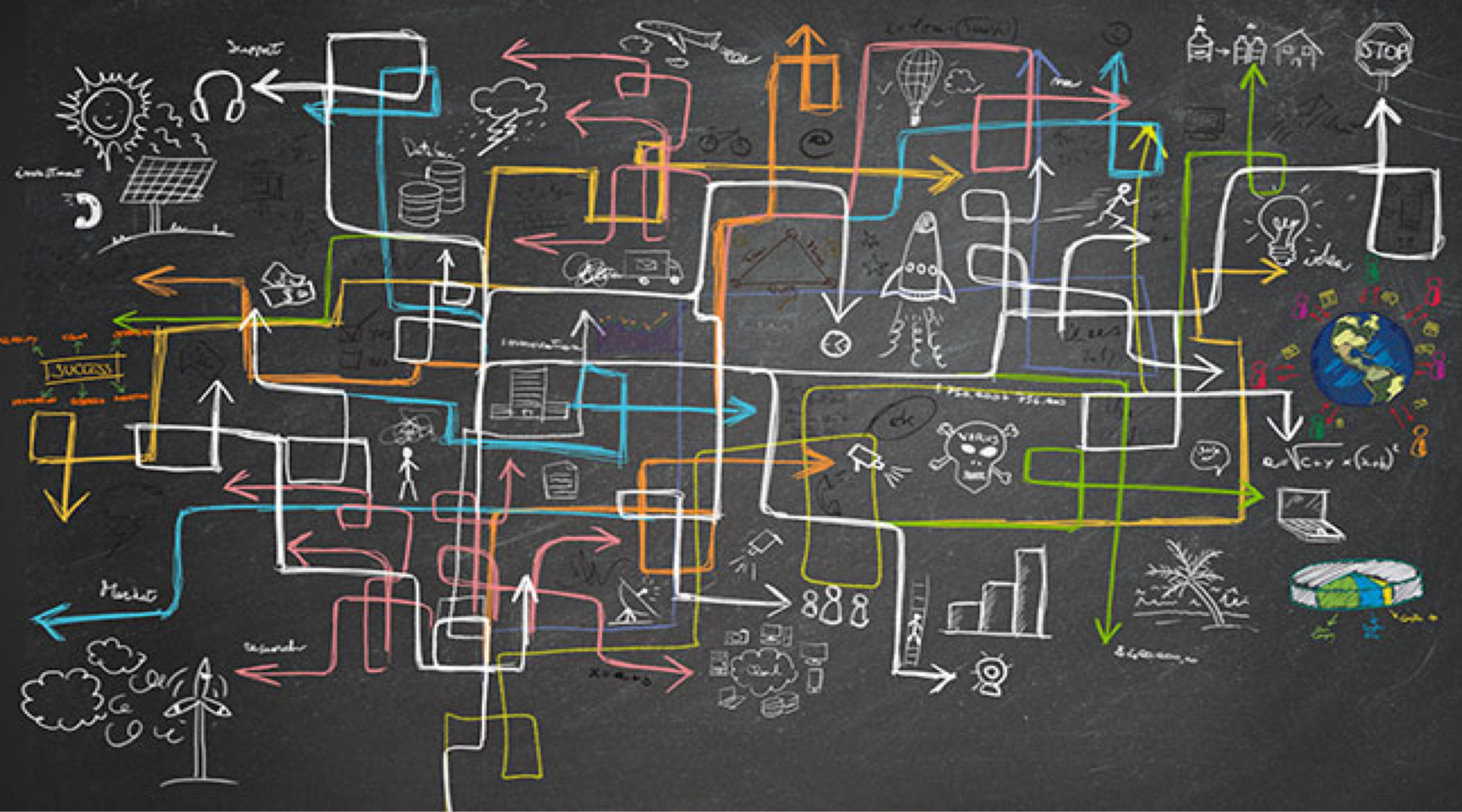


Data neexistují nezávisle



Exekuce zahrnuje zřetězené volání funkcí

Jak komplexní mohou být systémy, pro které píšete softwarovou aplikaci ...



STOP
SUCCESS
idea
Market

01 KOMPLEXITA

Esenciální komplexita – nelze se jí zbavit, vyplývá z podstaty problému, který řeším. Např. poskytnutí úvěru vyžaduje ověření nároku na velikost úvěr podle profilu klienta. Má komplexitu, kterou nelze snížit.

Incidentní komplexita – zavedli jsme si ji sami řešením, které jsme implementovali. Např. jsme údaje o stavu účtu implementovali do dvou systémů, které vyžadují konsolidaci a synchronizaci.

03 KOMPLEXITA

Co dělám, když mám problém (softwarový, matematický nebo i osobní) se kterým si nemohu poradit?

=> snažím se **problém strukturovat**

Jak mohu problém strukturovat?

=> pomocí **abstrakce, dekompozice** a **hierarchie.**

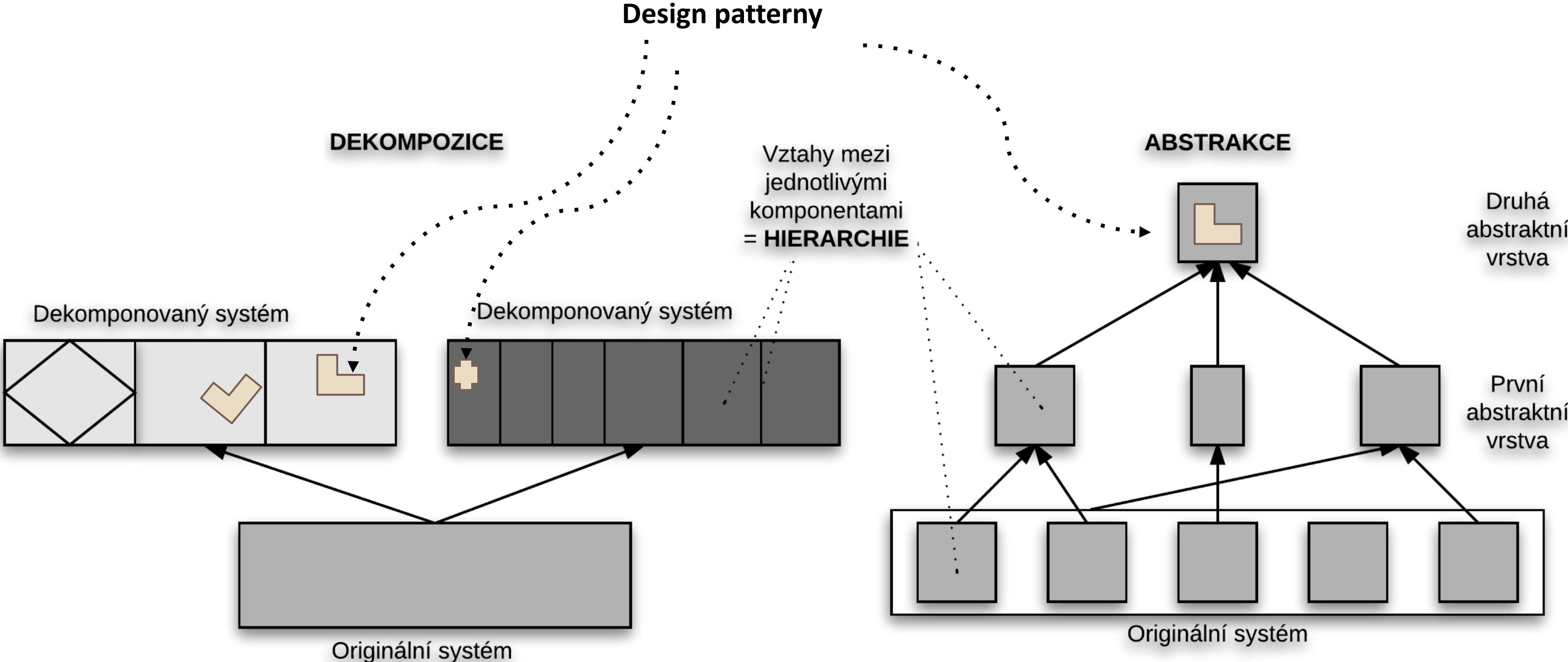
Dekompozicí rozkládám systém na menší komponenty

Abstrakcí skrývám komplexitu komponent do jednodušších

Hierarchií tyto komponenty provazuji mezi sebou

Pro řešení problémů se snažím aplikovat design patterny (návrhové vzory)

01 KOMPLEXITA



01 JAK DEKOMONOVAT SLOŽITÝ SYSTÉM

Pokud dekomponuji systém do modulů, tak aplikuji následující principy:

- ***Cohesion (princip soudržnosti)*** - spojuji funkcionalitu podle podobnosti. Atributy podobnosti volím intuitivně podle toho jakým způsobem a za jakým účelem budu modul používat. Atributem podobnosti může být doména, entita, uživatel atd.

  Příklad: mám skupinu lidí různých profesí a v různých počtech. Chci implementovat systém pro plánování stavby domu. Vzhledem k účelu systému lidi rozdělím na skupiny, kde každá umí realizovat stejné činnosti

- ***Coupling (provázanost)*** - snažím se dosáhnout velmi kohezních (soudržných) modulů, které mají pevné vazby uvnitř a co nejvolnější vazby mezi sebou.

- ***Reusabilty (přepoužitelnost)*** - modul je “opracován” tak, abych ho mohl přepoužívat v různých kontextech. Kontextem rozumím volat z různých modulů, v různých fázích životního cyklu aplikace a jeho fungování pro různé domény - např. použití toho samého modulu pro *spotřební úvěry* pak *hypotéky* a pak *pojištění*.

01 JAK DEKOMONOVAT SLOŽITÝ SYSTÉM

- ***DRY (Don't Repeat Yourself) -*** stejný kód (stejný kus funkcionality) se nenachází na víc jak jednom místě (nenachází se ve více modulech).

- ***Flexibility isolation (izolace flexibility)*** - jestliže budu muset implementovat změnu, tak dopad této změny bude omezen na vazby mezi moduly nebo minimum modulů.

- ***Encapsulation (zapouzdření)*** - data modulů jsou privátní, k modulům přistupuji ne pomocí jejich dat, ale pomocí povolených operací

01 ABSTRAKCE

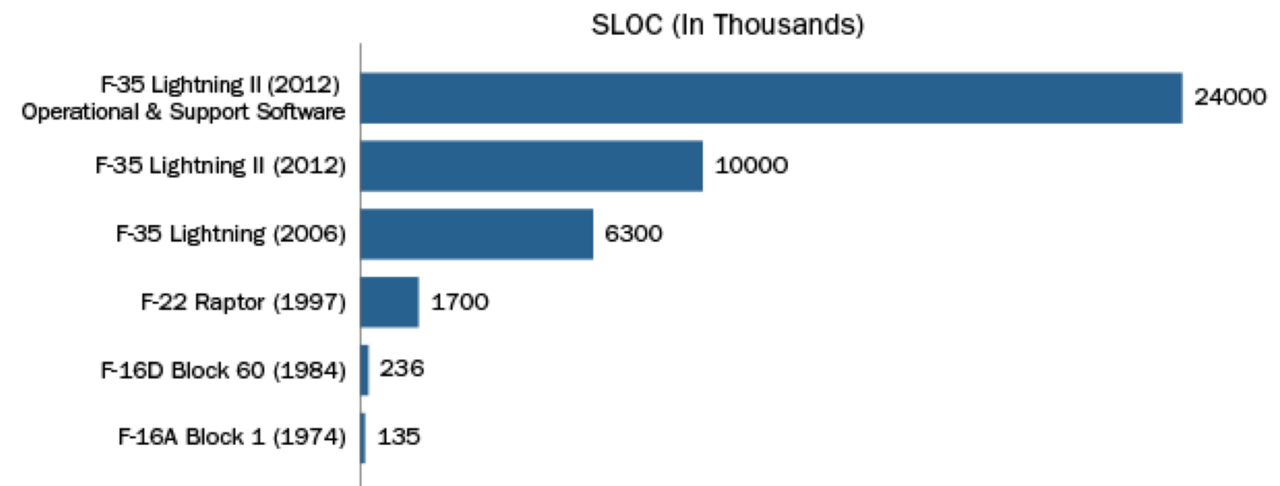
Edsger W. Dijkstra said, “Being abstract is something profoundly different from being vague.…The purpose of abstraction is not to be vague but to create a new semantic level in which one can be absolutely precise.”

01 ABSTRAKCE

Zjednodušeně řečeno potřebujeme schopnost vzít větší kusy a přistupovat k nim jako k primitivum tak, abychom je mohli kombinovat do větších celků a přitom se nestarali o jejich detail.

Druhy abstrakce v softwarovém vývoji:

- Jmenné abstrakce
- Datové abstrakce
- Procedurální abstrakce (v imperativních jazycích)
- Funkcionální abstrakce (ve funkcionálních jazycích)
- Objektové abstrakce
- A další exotické druhy abstrakcí ...

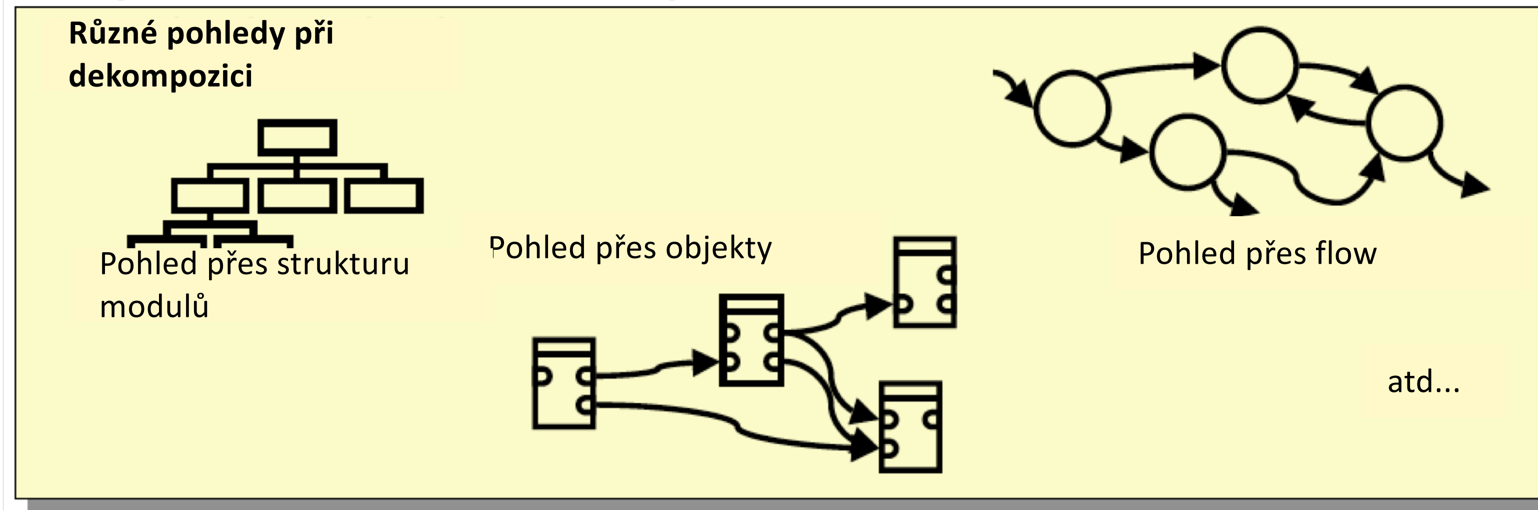


01 ABSTRAKCE

K větším problémům přistupuji způsobem "**Rozděl a panuj**” tak, aby:

- Každý podproblém měl přibližně stejnou úroveň detailu
- Každý podproblém byl řešitelný samostatně
- Řešení podproblémů lze zkombinovat tak, abych tím vyřešil celý problém

Systém zpravidla představuje n rozměrný problém, který nelze popsat jedním pohledem. Místo toho potřebuji **několik pohledů**. Dekompozici tak mohu provést pro tyto různé pohledy. Viz UML definuje několik druhů UML diagramů, každý navržen pro modelování jiného pohledu na systém.



01 DRUHY ABSTRAKCE – JMENNÁ

Jména a ***jmenné prostory*** jsou nejzákladnějším druhem abstrakce. Umožňují odkazovat se na proměnné, konstanty, operace, typy, funkce, moduly atd.. Používají se v ostatních typech abstrakcí.

Složitější příklad: framework *SpringData* ze jména metody generuje kód pro přístup k datům. V názvu vyhledává klíčová slova `find…By`, `read…By`, `query…By`, `count…By`, `get…By` ty propojuje pomocí `And`, `Or`, a propojuje s názvy atributů objektů.

```
public interface PersonRepository extends Repository<User, Long> {
  List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);
  // Enables the distinct flag for the query
  List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
  // Enabling ignoring case for an individual property
  List<Person> findByLastnameIgnoreCase(String lastname);
  // Enabling static ORDER BY for a query
  List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
}
```

01 DRUHY ABSTRAKCE – OBJEKTOVÁ

Využívána hojně v objektově orientovaných jazycích.

Část reality je reprezentována jako objekt, který skrývá komplexitu uvnitř a navenek poskytuje pouze operace, které je možné s objektem provést. Objektové abstrakce jsou přirozené pro uvažování člověka - většinou se sami díváme na objekty jako na celky, jejich celkové vlastnosti a schopnosti/operace

Např. objekty jako:

- Kávovar - poskytuje pouze metody pro ovládání (*udělej presso, lungo…, vypláchni se*) a čtení klíčových atributů kávovaru (*on/off, množství kávy, teplota, množství vody*)
- Auto
- Člověk
- Knihovna
- ...

01 DRUHY ABSTRAKCE – DATOVÁ

Odděluje abstraktní vlastnosti datového typu od jeho implementace. Abstraktní vlastnosti jsou ty, které jsou viditelné a měl bych je brát v potaz v kódu ve kterém abstraktní datový typ používám, zatímco konkrétní implementace je schovaná a mohu ji bez dopadu na tento kód měnit.

Hlavním reprezentantem datové abstrakce je ***Abstraktní Datový Typ (ADT)***. ADT je matematický model pro datový typ definovaný množinou hodnot a operací nad těmito hodnotami, které splňují definované axiomy. Ekvivalentní k algebraické struktuře v abstraktní algebře.

Příklad: Typ Integer je ADT definovaný hodnotami …, −2, −1, 0, 1, 2, … a operacemi + , - , /, <,>,= které splňují axiomy asociativity, komutativity atd. V programovacích jazycích např. typ *boolean, float* atd. reprezentuje tzv. ADT.

Další příklady:

- Typy z collection API jako *Collection, List, Set, Map* jsou příkladem datové abstrakce - předepisují práci s datovou strukturou pomocí API a definuje princip práce s daty.
- RDBMS používájí abstrakci tabulky, které má záznamy (*Row*) a sloupečky (*Column*). Uživatel je odstíněn od uložení dat

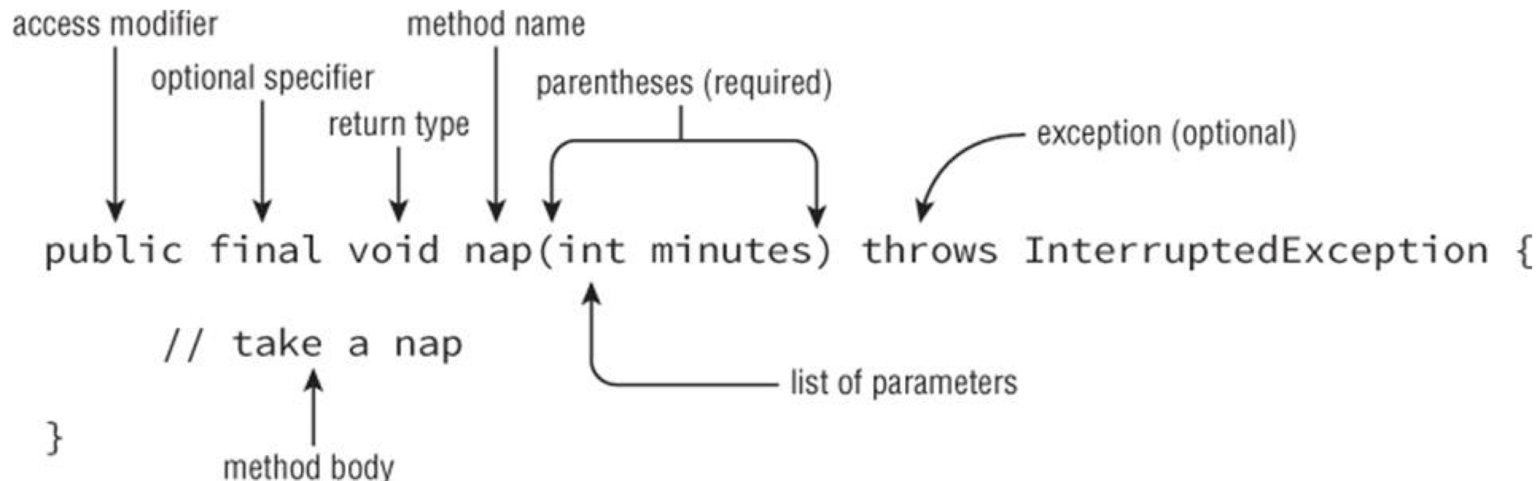
01 DRUHY ABSTRAKCE – PROCEDURÁLNÍ

Cílem je, abych mohl vzít komponentu (část funkcionality svého systému) a bez zásahu do komponenty ji přepoužít na jiném místě v systému. K tomu mi stačí znát pouze rozhraní komponentu a funkcionalitu, kterou realizuje.

Tento mechanismus realizuji pomocí tzv. subrutin.Tento machanismus má následující vlastnosti:

- Izolace použití subrutiny od její implementace.
- Redukce množství duplicit v kódu a tzv. boilerplate kódu
- Možnost kombinování a zanořování

Příklad subrutiny v Java:



01 DRUHY ABSTRAKCE – PROCEDURÁLNÍ

V čem se liší v různých implementacích:

- Syntax, typová kontrola
- **Mechanismus předávání parametrů do a z subrutiny**
- Statická či dynamická alokace a scope lokálních proměnných
- Overloading
- Generika

01 DRUHY ABSTRAKCE – DLE VOLÁNÍ PROCEDURY

Call-by-Value (*a* se kopíruje do *x*)

```
int a = 3;
void foo (int x) {
   //a and x have same value
   //changes to a or x don't
   //affect each other
}
//argument can be an expression
foo (a+a);
//no modifications to a
```

Call-by-Reference (*x* se nastaví na stejné místo v paměti jako *a*)

```
int a = 3;
void foo (int x) {
   //a and x reference same location
   //changes to a and x affect each other
}
//argument can be an expression
foo (a);
//a might be modified
```

Call-by-Result (hodnota *x* se inicializuje uvnitř a na konci x => a)

```
int a = 3;
void foo (int x) {
   //x is not initialized
   //changes to a or x don't
   //affect each other
}
//argument must be variable
foo (a);
//a will be modified by x upon
method call
```

Call-by-Result (a => x a na konci x => a)

```
int a = 3;
void foo (int x) {
   //a and x have same value
   //changes to a or x don't
   //affect each other
}
//argument must be variable
foo (a);
//a might be modified
```

Call-by-Name (x se nastaví na funkci)

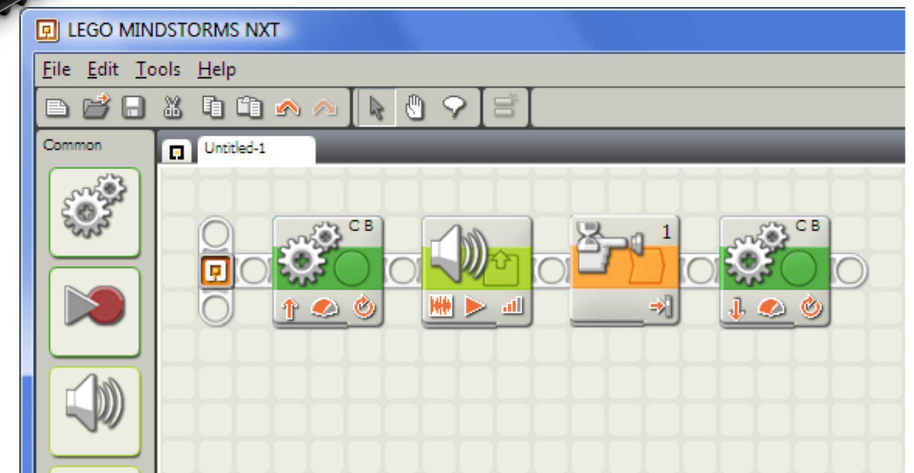
```
int a = 3;
void foo (int x) {
  //x is a function
  //to get value of argument
  //evaluate x() when value needed
}
//argument can be an expression
foo (a + a);
//no modifications to a
```

01 ROZPOZNÁTE DEKOMPOZICI od ABSTRAKCE?

Co je dekompozice a co abstrakce u Lego Mindstorms?

Kostky ze kterých robota sestavuji

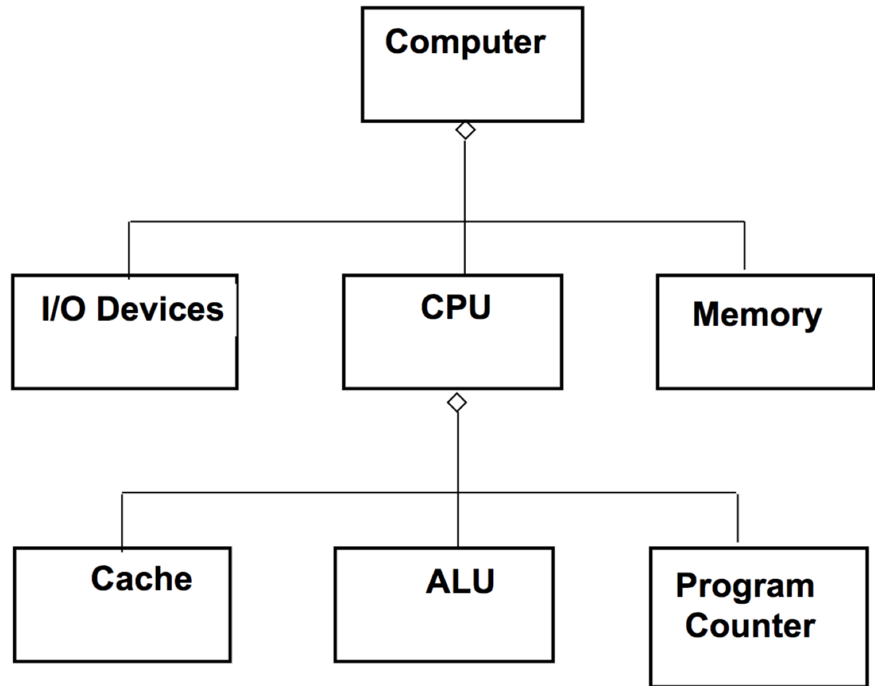
Vizuální editor pro programování chování robota



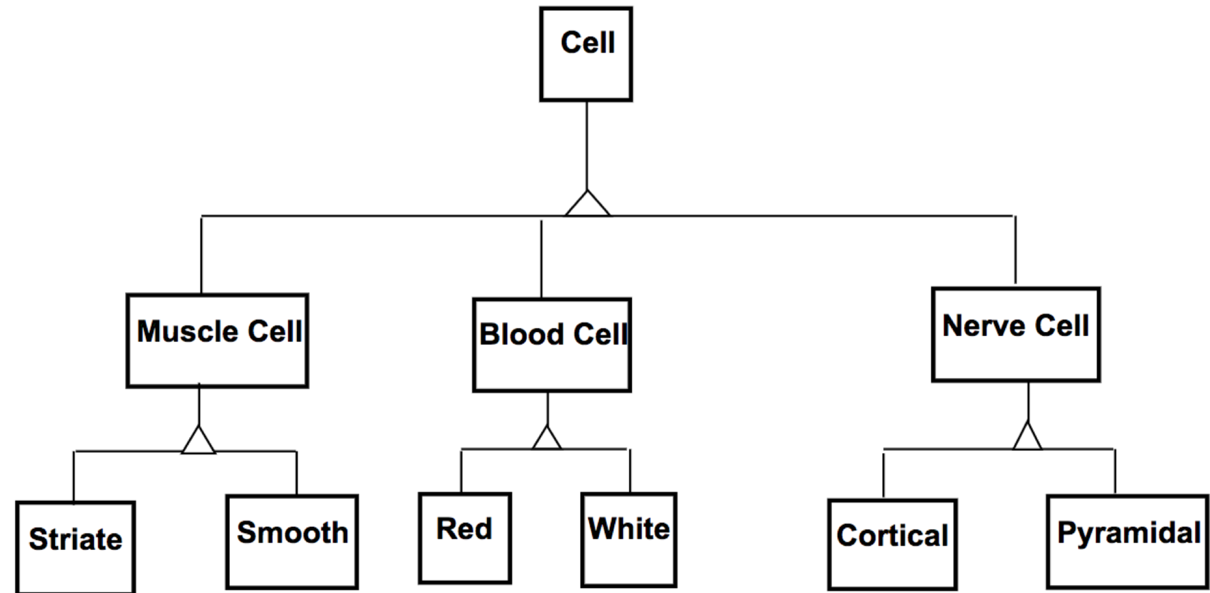
01 HIERARCHIE

Hierarchie je určena vazbami mezi komponentami systému

Part of **hierachie**



Is kind of **hierachie:**



01 DESIGN PATTERNY

Design pattern **je** obecné řešení problému, které má implementace v různých jazycích a doménách

Design pattern **není** není knihovnou nebo částí zdrojového kódu, která by se dala přímo vložit do našeho programu, jedná se o popis řešení problému nebo šablonu, která může být použita v různých situacích

Různé design patterny jsou definované pro:

- Různé typy programovacích jazyků: Objektové, funkcionální, logické…
- Vrstvu aplikace: Frontend (uživatelské rozhraní), Datová vrstva (uložení dat)...
- Typ deploymentu: Centrální aplikace, distribuované aplikace ….

01 KVALITA SOFTWARE

Při vývoji software jsou ***čas, peníze, scope*** a ***kvalita*** navzájem provázány. Jelikož jsme v předmětu OMO, tak se zde budeme zabývat primárně kvalitou.

Kritéria kvality softwarového systému jsou:

- ***Flexibilita*** - množství a složitost změn, které musím v systému provést proto, aby fungoval i pro jiné scénáře
- ***Přepoužitelnost*** - použitelnost systému pro konstrukci v různých aplikacích
- ***Rozšiřitelnost*** - schopnost systému adaptovat se na změny ve specifikaci (rozšiřuji funkcionalitu)
- ***Robustnost*** - schopnost systému reagovat na nepředpokládané situace
- ***Kompatibilita*** - jednoduchost kombinování softwarových komponent mezi sebou
- ***Použitelnost*** - jednoduchost použití systému uživatelem nebo jiným systémem
- ***Efektivnost*** - minimalizace požadavků na zdroje (hardwarové, lidské, finanční)
- ***Škálování*** - schopnost systému fungovat při narůstající zátěži
- ***Portovatelnost*** - náročnost přenesení software do jiného hardwarového a softwarového prostředí

01 KVALITA SOFTWARE

Schopnost reagovat na změny = kritérium číslo jedna u softwarového vývoje

<= nejvíce softwarových bugů je způsobeno změnami v kódu

<= chápání aplikace, kterou píše vývojář se mění

<= do aplikace zasahuje více vývojářů současně a neznají všechny části kódu

<= zákazník a ani analytik nikdy nedá požadavky kompletní, finální a 100% konzistentní

<= systém se bude rozšiřovat i po jeho dokončení, bude se měnit jeho okolí i SW a HW komponenty které využívá

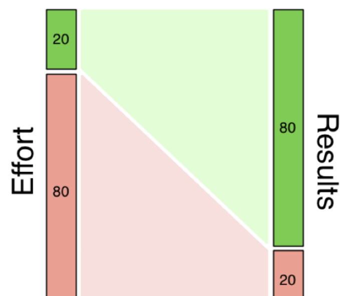
01 KVALITA SOFTWARE

Nelze najít optimální řešení, existují pouze sub optimální řešení, jelikož kritéria jsou ve vzájemné kontradikci, např:

- Zvyšování flexibility zvyšuje komplexitu a tím pádem náklady a čas
- Zvyšování přepoužitelnosti snižuje použitelnost (musím zvýšit obecnost systému)
- Vyšší granularita zvyšuje použitelnost, ale snižuje přepoužitelnost

Jak se tedy rozhodovat? Snažím se upravovat oblasti, kde s minimálním úsilím dosahuji maximálního efektu. Zastavuji ve chvíli, kdy jsou pro mě další zlepšení už příliš drahá.

Optimální poměr cena výkon



Paretovský princip z teorie her

Paretovské zlepšení - alokace může být paretovsky zlepšena pokud existuje jiná alokace při které jeden hráč na tom může být lépe aniž by si žádný další hráč nepohoršil

Paretovsky optimální - alokace je paretovsky optimální jestliže není možné paretovské zlepšení.