OMO

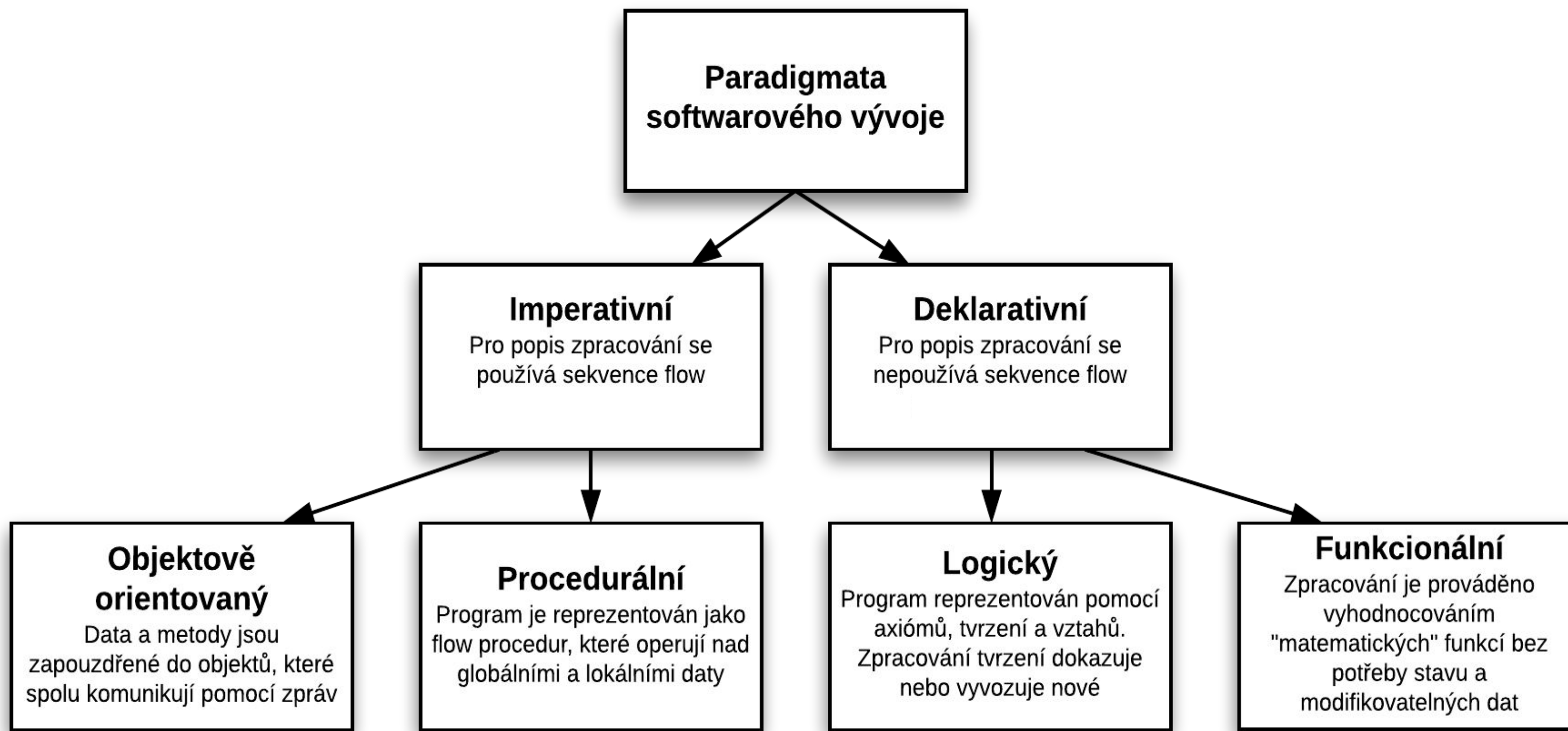
1 - Klíčové koncepty modelování systémů I

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

Ing. David Kadleček, PhD.
kadlecd@fel.cvut.cz, david.kadlecek@cz.ibm.com

Verze 22.10.2018

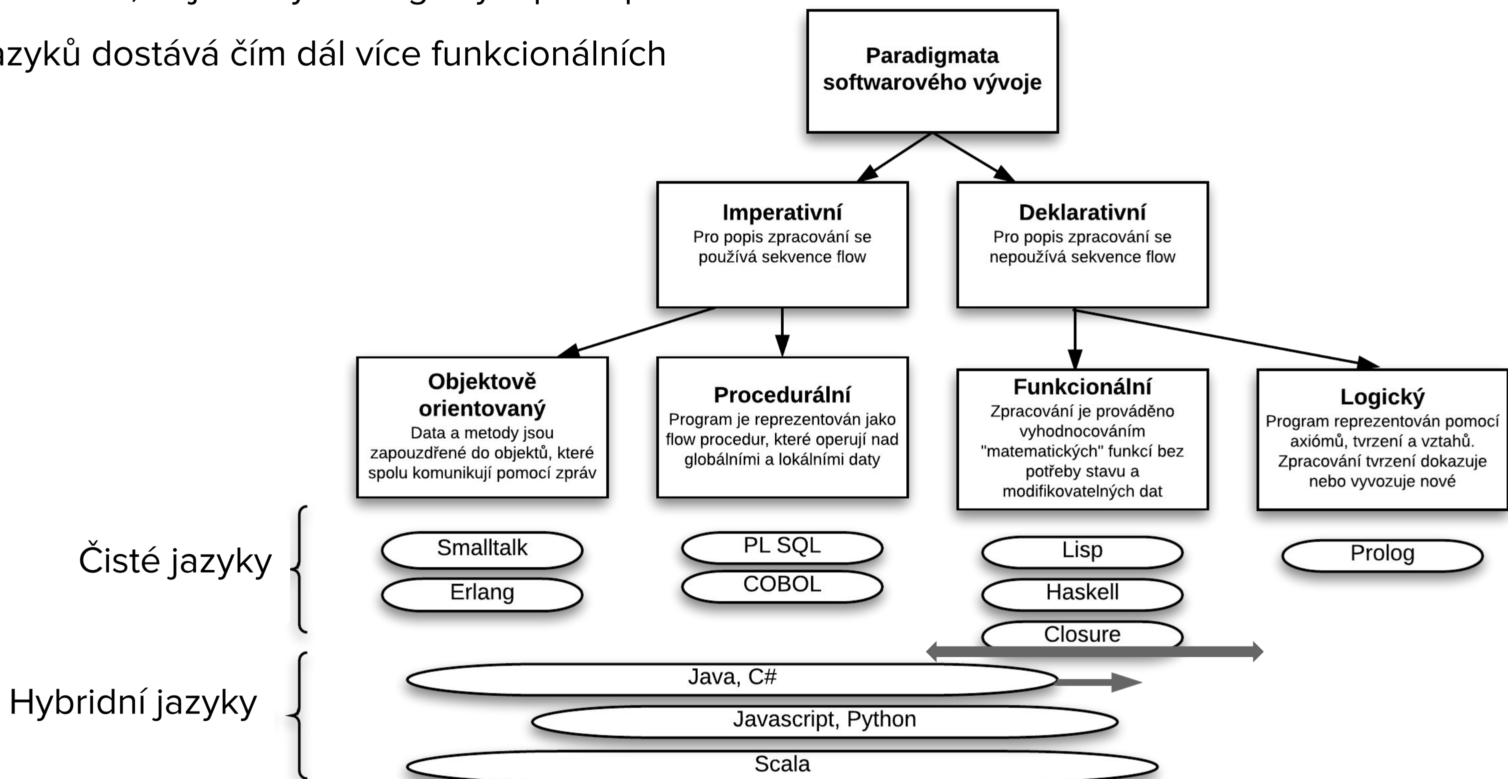
Paradigmata softwarového vývoje



Paradigmata softwarového vývoje

V posledních letech je patrná snaha o nalezení sjednocovací teorie mezi funkcionálním, objektovým a logickým přístupem.

Do objektových jazyků dostává čím dál více funkcionálních konceptů.



Deklarativní versus Imperativní programování

- 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:

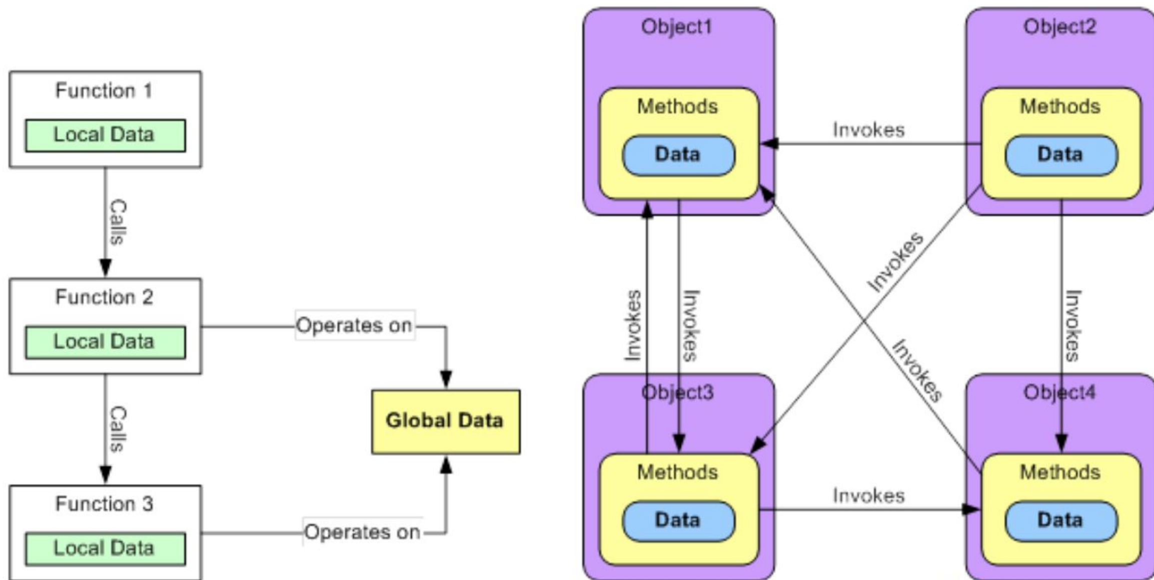
- *Odhadnout výsledek G*
- *Zlepšit odhad zprůměrováním G a x/G*
- *Zlepšovat odhad dokud není dostatečně dobrý*

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

Procedurální versus 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.



Procedurální (a obecně imperativní) versus Funkcionální přístup

Procedurální přístup

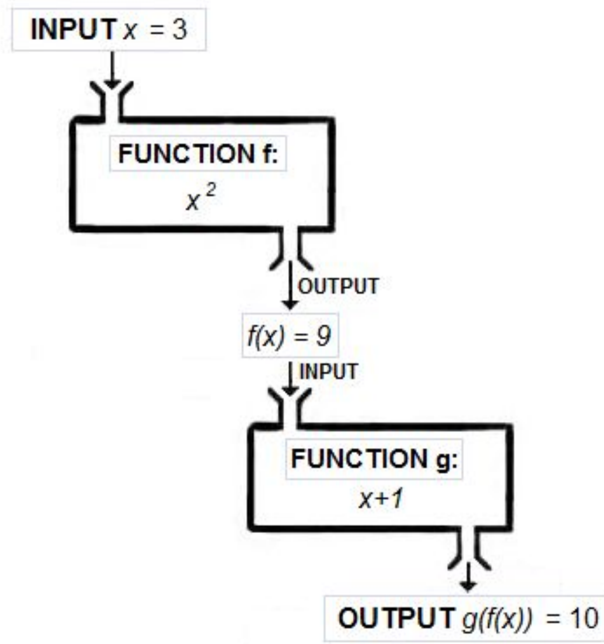
Příklad výměna oleje:

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

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

Příklad výpočtu matematického výrazu:



Procedurální (a obecně imperativní) versus Funkcionální přístup

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

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 program, zadá otázky a stroj (problem solver) na ně nalezne odpovědi.

Logické programování

Úloha:

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

Kdo je mladší než Petrova matka?

Zápis v PL1:

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

$St(b)$

$St(c)$

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

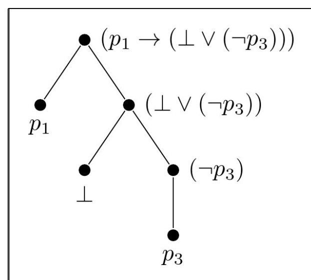
Zápis v Prologu:

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

(PL1 - Predikátová logika prvního řádu)

Shrnutí

Přístup logického programování

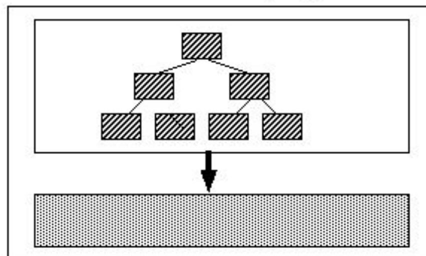


dotaz

odpověď

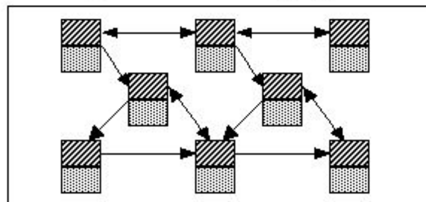
Problem solver

Procedurální přístup

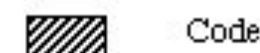


Exekuce zahrnuje provádění kódu, který operuje nad daty

Objektový přístup

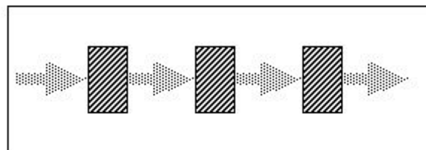


Objekt zapouzdřuje kód i data

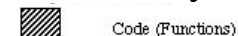


Výpočet zahrnuje interakci mezi objekty

Funkcionální přístup

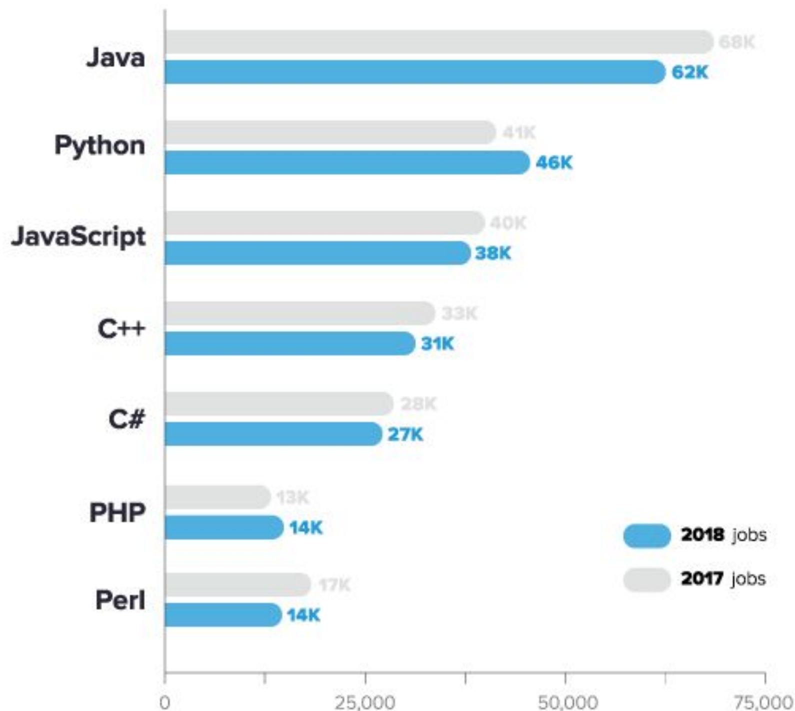


Data neexistují nezávisle



Exekuce zahrnuje zřetězené volání funkcí

Současné trendy - nejvíce nabídek práce



Source: www.indeed.com 2018

Trendy - nejvíce aktivity na GitHub

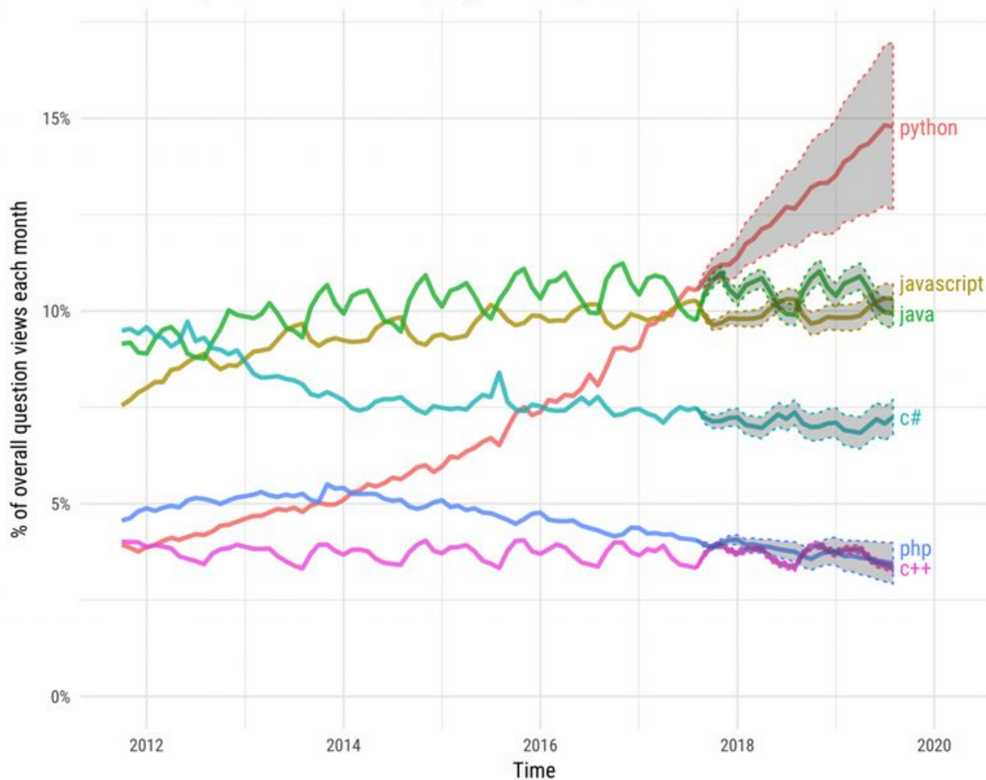
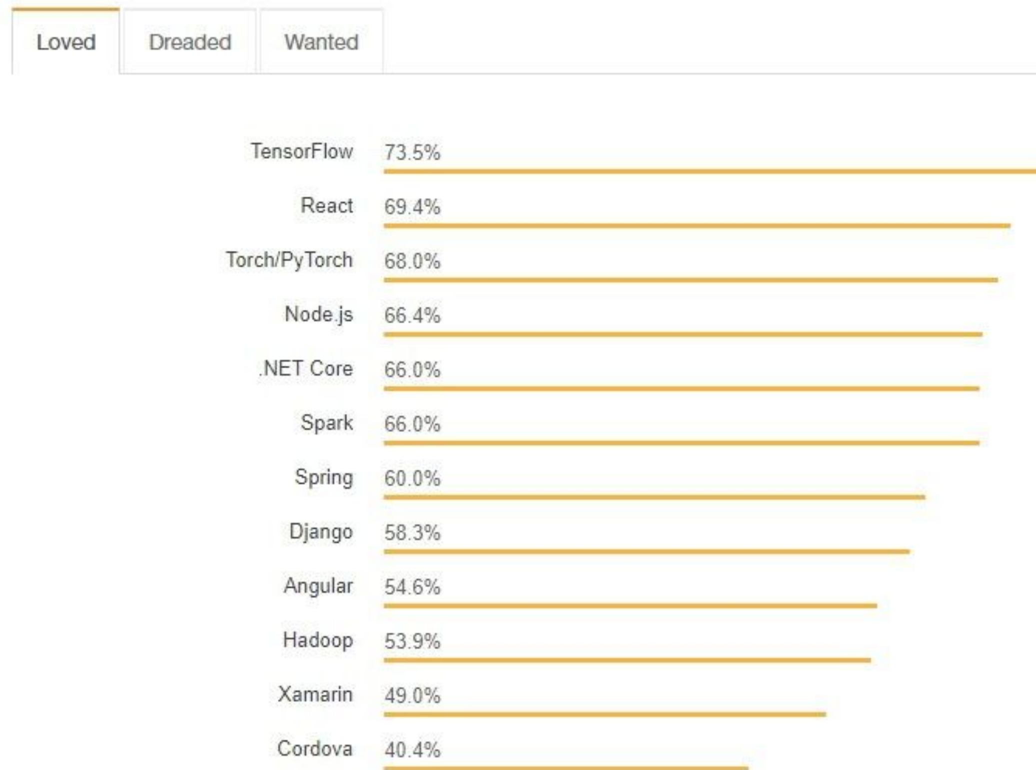


Image: Stack Overflow

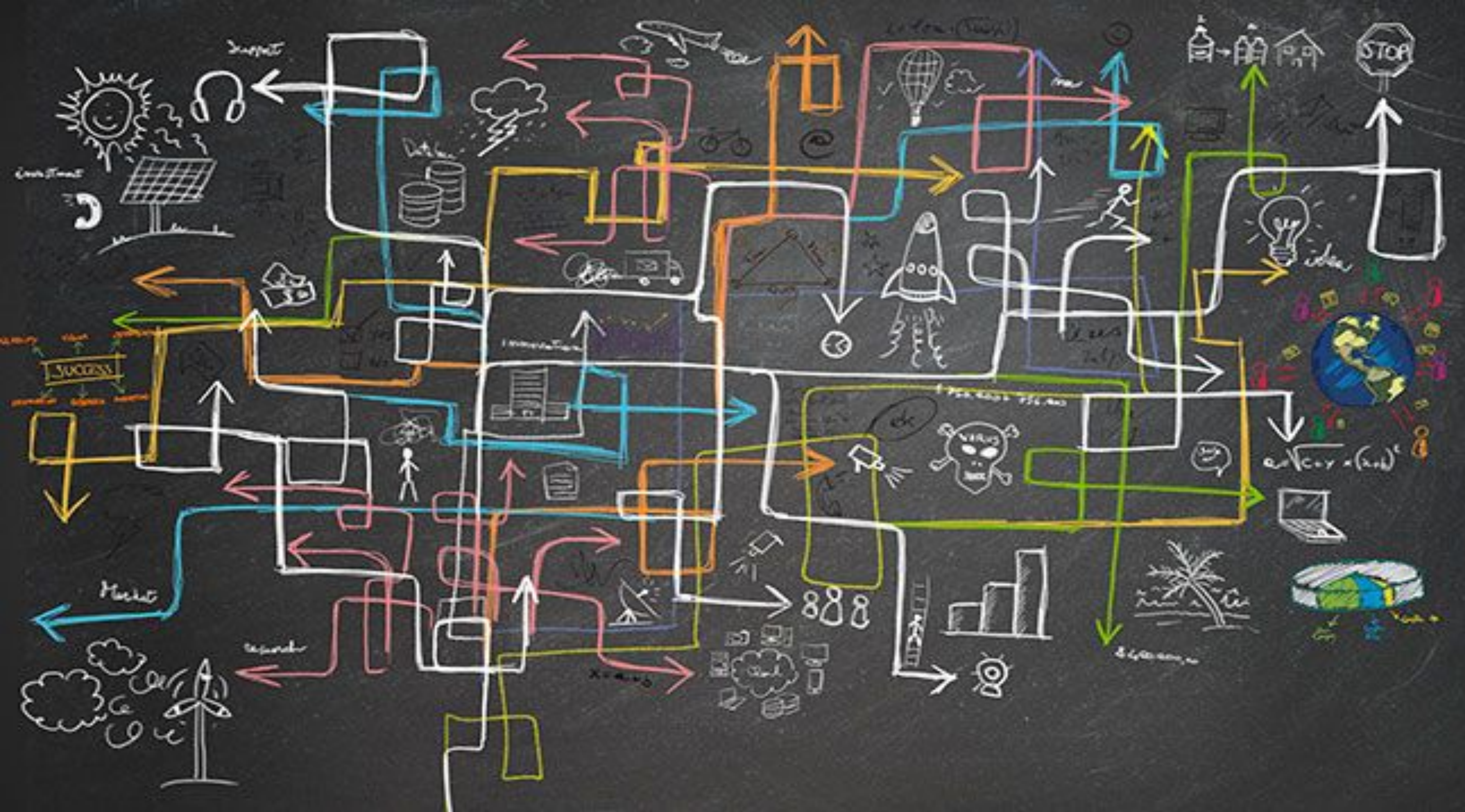
Source: Stack Overflow 2018

Současné trendy - nejoblíbenější frameworky



Source: StackOverflow 2018

První úloha, kterou budete programovat v OMO …



Support
STOP
SUCCESS
idea
VIRUS
Market

Souboj s komplexitou

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.**

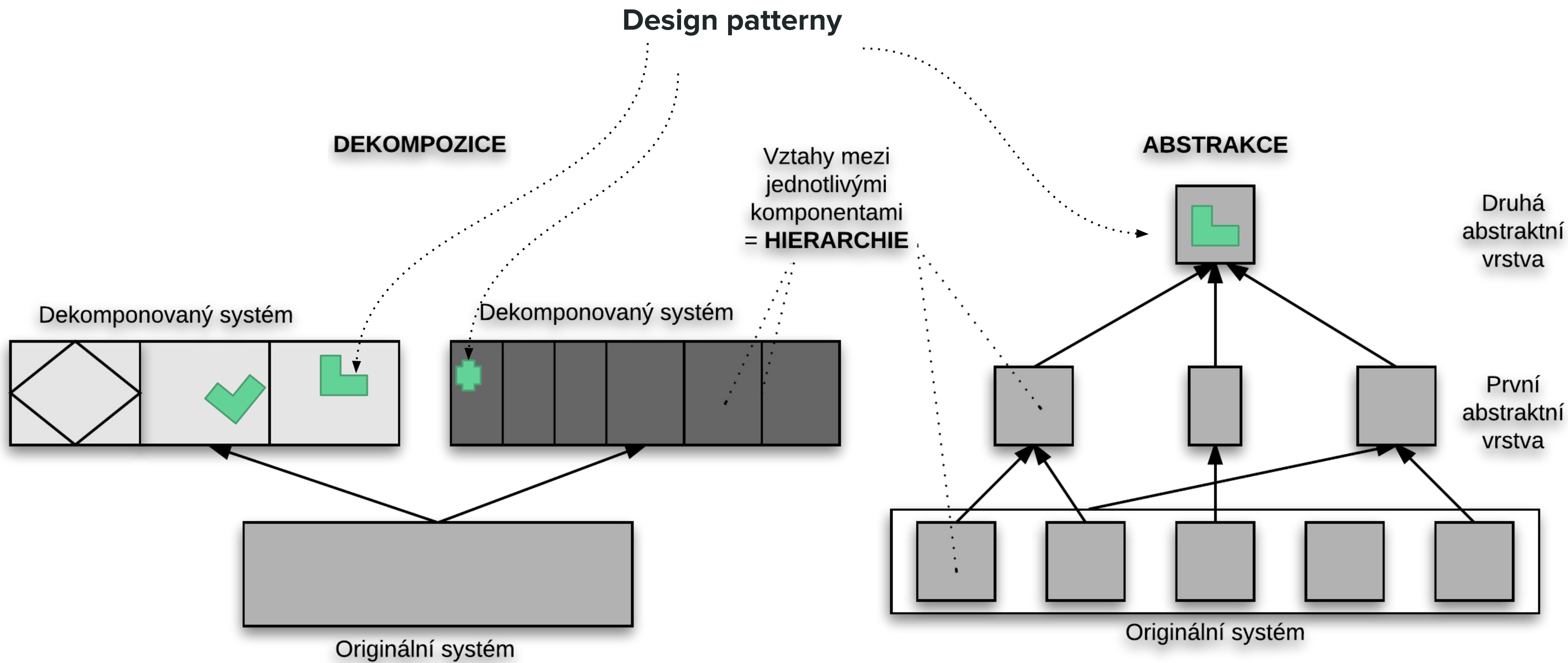
Dekompozicí rozkládám systém na menší komponenty

Abstrakcí skrývám komplexitu komponent do jednodušších

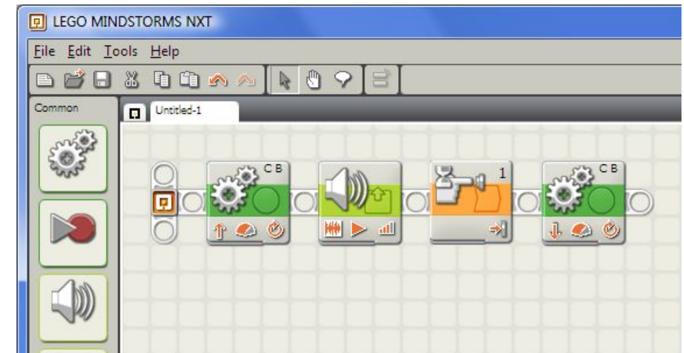
Hierarchií tyto komponenty provazuji mezi sebou

Pro řešení problémů se snažím aplikovat design patterny

Souboj s komplexitou



Co je dekompozice a co abstrakce?

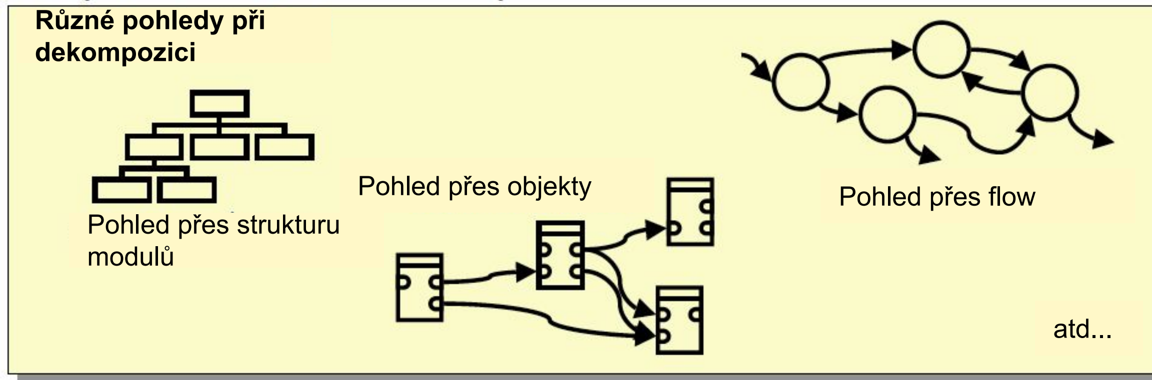


Dekompozice

K větším problémům přístupuji 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

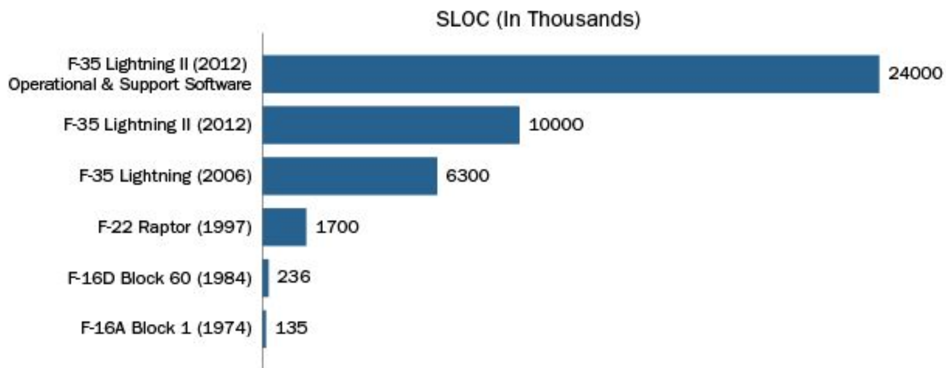
Systém zpravidla představuje n rozměrný problém, který nelze popsat jedním pohledem. Místo toho potřebuji **několik pohledů**. Dekompozici tak mohu provést pro tyto různé pohledy. Viz UML definuje několik druhů UML diagramů, každý navržen pro modelování jiného pohledu na systém.



Abstrakce

Abstrakce je jednodušší reprezentace systému pomocí které na něj nahlížím nebo k němu přistupuji aniž bych znal jeho detaily

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.



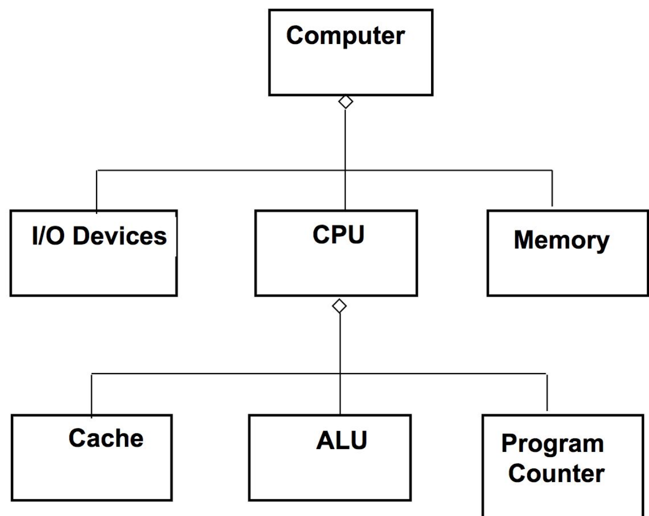
Druhy abstrakce v softwarovém vývoji:

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

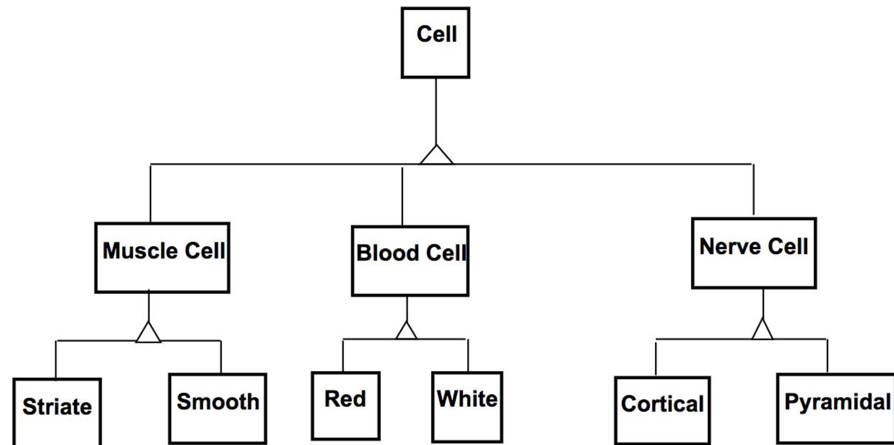
Hierarchie

Hierarchie je určena vazbami mezi komponentami systému

***Part of* hierachie**



***Is kind of* hierachie:**



Design patterny

= návrhové vzory

- 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 ….

Jak dekomponovat

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*.

Jak dekomponovat

- ***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

Kritérium číslo jedna u softwarového vývoje

Schopnost reagovat na změny

<= 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

Kvalita software - multikriteriální optimalizační problém

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

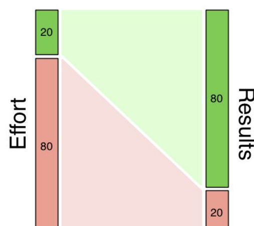
Kvalita software - multikriteriální optimalizační problém

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.

Abstrakce - jmenné abstrakce

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);
}
```

Abstrakce - datová abstrakce

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

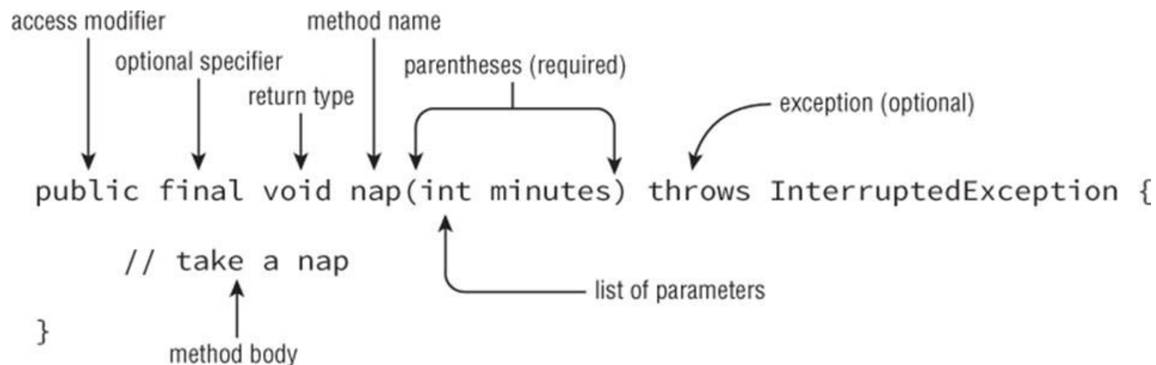
Abstrakce - procedurální abstrakce (Control abstraction)

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:



Abstrakce - procedurální abstrakce (Control abstraction)

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

Mechanismus předávání parametrů do a z subrutiny

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
```

Funkcionální abstrakce - funkce první třídy

Objektem **první třídy** (first-class citizen) v programovacích jazycích je entita, která podporuje následující operace: být předána jako parametr, přiřazená proměnné a být vrácená z funkce. Funkce první třídy je tedy taková funkce, která splňuje výše uvedené vlastnosti.

Pozn. Metody a třídy, jelikož to nejsou hodnoty, tak jsou považovány za objekty druhé třídy.

Klasický přístup:

```
public List filterPersonByAge(List<Person>
list) {
   List result = new ArrayList();
   for (Person person : list) {
       if(p.age > 65){
           result.add(person);
       }
   }
   return result;
}
```

Filtruji a vracím každého, kdo je starší než 65 let. Problém je, že když chci filtrovat podle jiného atributu, tak musím celý tento kód zduplikovat, abych modifikoval pouze jednu řádku kódu.

Funkcionální abstrakce - funkce první třídy

Přepis pomocí funkce první třídy (Java 1.8+):

```
import java.util.function.Predicate;
public class FirstClassFunctionExample {

    public List filterPerson(List<Person> list, Predicate<Person> p) {
        List result = new ArrayList();
        for (Person person : list) {
            if(p.test(person)) {result.add(person);}
        }
        return result;
    }
    public boolean ageFilter(Person p){
        return p.age > 65;
    }
}
```

Přidali jsme nový parametr typu *Predicate*, který obsahuje podmínku, kterou testujeme. Dále pak metoda *ageFilter*, kterou vkládáme jako parametr *p*.

Funkce je volána následovně:

```
filterPerson(personList, FirstClassFunctionExample::ageFilter);
```

Jestliže chceme filtrovat podle jiného atributu, tak uděláme drobnou změnu do mplementace filtru a vlastní kód na filtrování je přepoužit.

Funkcionální abstrakce - funkce vyššího řádu

Funkce vyššího řádu je funkce, které splňuje přinejmenším jednu z vlastností:

- Jedním či více parametry je funkce
- Vrací funkci jako parametr

```
/* Scala: function compute má dva parametry - funkci f a hodnotu v. V
těle metody je aplikace funkce f na hodnotu v*/
def compute(f: Int => String, v: Int) = f(v)
```

Funkcionální abstrakce - funkce vyššího řádu

```
/* Java 1.8+ : function compute vezme funkci f a hodnotu v a aplikuje funkci f na hodnotu v*/
public static String compute(Function<Integer, Integer> f, Integer v) {
   return f.apply(v);
}
```

Funkci pak použiji takto

```
/* Java 1.8+: function apply vezme funkci f a hodnotu v a aplikuje funkci v na hodnotu v*/
public class AwesomeClass {
   private static Integer invert(Integer value) {
       return -value;
   }
   public static Integer invertTheNumber(){
       Integer toInvert = 5;
       Function<Integer, Integer> invertFunction = AwesomeClass::invert;
       return compute(invertFunction, toInvert);
   }
}
```

Funkcionální abstrakce - lambda expressions

Lambda expression je forma ve tvaru: **(seznam argumentů funkce) -> tělo funkce**

```
/* Java 1.8+ Funkce, která sečte dvě čísla */
(int x, int y) -> x + y
/* Bezparametrická funkce */
() -> 42
/* Procedura */
(String s) -> { System.out.println(s); }
/* Komparátor */
List<Person> personList = Person.createShortList();
Collections.sort(personList, (Person p1, Person p2) -> p1.getSurName().compareTo(p2.getSurName()));
```

- Lambda výrazy se používají především k definování implementace funkčního **rozhraní s jedinou metodou** tzv. **inline formou** což vede k výrazné redukci kódu a přináší např. do Javy některé výhody funkcionálního programování.
- Lambda expression v programovacích jazycích je funkce, kterou je možné definovat a volat bez bindingu s identifikátorem

Pozn. Lambda calculus je formální systém matematické logiky a informatiky pro vyjádření výpočtu pomocí bindingu proměnných a jejich substituce

Funkcionální abstrakce - currying

Currying spočívá ve vyhodnocování argumentů funkce per partes, kdy po každém kroku získám funkci, která má o jeden argument méně.

Např. pro funkci

$f(x, y, z) = x * y + z$

můžeme aplikovat argumenty 3, 4, 5 a dostaneme:

$f(3, 4, 5) = 3 * 4 + 5 = 17$

Současně ale můžeme aplikovat pouze 3 a získáme novou funkci *f*

$(3, y, z) = g(y, z) = 3 * y + z$

currying podruhé pro 4 nám dá:

$g(4, z) = h(z) = 3 * 4 + z$

Procedurální abstrakce - currying

Příklad vytvoření složené funkce při deklaraci:

```
/*Java 1.8+*/
public class Currying {
   public void currying() {
       // Create a function that adds 2 integers
       BiFunction<Integer,Integer,Integer> adder = ( a, b ) -> a + b ;
       // And a function that takes an integer and returns a function
       Function<Integer,Function<Integer,Integer>> currier = a -> b -> adder.apply( a, b ) ;
       // Call apply 4 to currier (to get a function back)
       Function<Integer,Integer> curried = currier.apply( 4 ) ;
       // Results
       System.out.printf( "Curry : %d\n", curried.apply( 3 ) ) ; // ( 4 + 3 )
   }
```

Procedurální abstrakce - currying

Vytvoření složené funkce ex post po jejich deklaraci:

```
public void composition() {
        // A function that adds 3
        Function<Integer,Integer> add3   = (a) -> a + 3 ;
        // And a function that multiplies by 2
        Function<Integer,Integer> times2 = (a) -> a * 2 ;
        // Compose add with times
        Function<Integer,Integer> composedA = add3.compose( times2 ) ;
        // And compose times with add
        Function<Integer,Integer> composedB = times2.compose( add3 ) ;
        // Results
        System.out.printf( "Times then add: %d\n", composedA.apply( 6 ) ) ;  // ( 6 * 2 ) + 3
        System.out.printf( "Add then times: %d\n", composedB.apply( 6 ) ) ;  // ( 6 + 3 ) * 2
}
public static void main( String[] args ) {
        new Currying().currying() ;
        new Currying().composition() ;
        }
}
```