# Opora IPP-I

## Kapitola 2 – Klasifikace jazyků

Pojem **programování** (proces tvorby programu) budeme rozumět takovou činnost, která jistý postup, algoritmus, typicky myšlenkový, převádí na posloupnost elementárních úkonů nějaké stroje, typicky počítače. Přitom dochází k uložení tohoto postupu tak, aby jej stroj mohl opakovat periodicky.

**Programátor** je ten, kdo proces tvorby programu realizuje.

**Programovací jazyk** je prostředník mezi běžnou řečí a posloupností typicky binárních číslic.

**Programovací jazyk** je konečná množina příkazů (převáděných do binární formy), která má specifickou syntaktickou strukturu a pevně a přesně vymezenou sémantiku.

**Počítačový program** je zápis v programovacím jazyce, který je abstrakcí reality. Implementuje abstraktní model a abstrahuje model počítače/procesoru.

Dělení jazyků dle abstrakce dat:

* Strojové/assemblery
* Vyšší úrovně (než strojové) – Fortran, Cobol
* Univerzální jazyky
* Blokově strukturované – Pascal a C (pokud pomineme modularitu a existenci knihoven)
  + Poprvé umožňuje použít a využít návrhové metodologie a prvky softwarového inženýrství
* Modulární blokově strukturované jazyky (umožňují navíc skrývyat implementaci v modulech)
  + Umožňují oddělit definici typu od operací, kt. ho mainupulují
* OOJ
* Jazyky rozšiřující datové paradigma (logické, funkcionální, pro sazbu textu, atd.)

**Imperativní** (procedurální) je takový jazyk, kde programátor musí řešit tyto otázky řízení běhu programu:

* Co za operace má být provedeno,
* V jakém pořadí to má být provedeno

**Deklarativní** je takový jazyk, kde programátor musí řešit tuto otázku řízení běhu programu:

* Co za operace má být provedeno

Podprogramy umožňují vnořené zpracování určitých logických funkcí/operací. Podprogram je volán skrze své rozhraní. Rozhraní definuje předávané parametry a výsledek.

Bloky, na rozdíl od podprogramů, nemají jméno, takže je nelze volat z různých míst a jejich kód se uplatňuje pouze v místě, kde je vložen (nemůže tedy volat sama sebe).

Ko-programy jsou pojmenované bloky kódu, které mají vzhledem k sobě symetrický vztah. Jejich kód je zpracováván současně, případně prokládaně (podle architektury). Jejich rozhraní a identifikace je shodné jako u podprogramů.

Paralelní programy/procesy jsou oproti ko-programům mnohem větší logické celky. Jejich vztahy nejsou nutně symetrické, může mezi nimi docházet k synchronizaci. V současnosti na nejvyšší úrovni abstrakce hovoříme o procesech, v rámci procesů potom o vláknech (threads).

Odložené zpracování – pokud výsledek operace není potřeba, jeho vyhodnocení je odloženo.

Volání podprogramů podle strategií:

* Volání hodnotou (call-by-value) – parametry se vyhodnocují před voláním podprogramu
* Volání jménem (call-by-name) – parametry se předávají nevyhodnoceny a zástupným jménem a až to vyžaduje podprogram, pak dojde vyhodnocení, může dojít k opakovanému vyhodnocení
* Volání v případě potřeby (call-by-need) – podprogram je prvně zavolán, až poté když je potřeba parametr, tak se vyhodnotí a hodnota se uchová (nedochází k opakovanému vyhodnocení), také označení jako lazy

**Syntaxe** jazyka definuje strukturu programu, tj. to, jakým způsobem je dovoleno jednotlivé konstrukce řadit za sebe.

**Sémantika** je popis/definice významu jednotlivých syntaktických konstrukcí, způsobu jejich vyhodnocení, zpracování, atd.

Statická sémantika – popisuje vlastnosti, které mohou být studovány a ověřovány v době analýzy/překladu programu, např. typová kompatibilita, existence proměnných apod.

Dynamická sémantika – popisuje vlastnosti, jejichž splnění lze ověřit až v době běhu programu, např. velikost indexu pole daného výrazem, velikost výsledku, apod.

**Deklarace** úplně vymezuje atributy dané entity. Může být explicitní i implicitní.

**Definice** úplně vymezuje atributy dané entity a dále u proměnných způsob alokace paměti a u funkcí/procedur navíc tělo funkce.

Vlastnosti entity programu:

* Jméno – rozlišuje jednotlivé entity v programu, efektivní délka, jazyk C (case sensitive) X Pascal (case insensitive), klíčová slova, rezervovaná slova
* Adresa a umístění/lokace v paměti

**Rozsah platnosti** proměnné určuje tu část programu, kdy je možné s proměnnou pracovat.

Jazyky z hlediska typovosti rozdělujeme:

* Beztypové
* Netypované
* Typované – dále můžeme rozdělit přiřazení typů na explicitní nebo automaticky odvozené

**Doba života** proměnné je časový interval, po který je pro danou proměnou alokována paměť.

## Kapitola 3 Nestrukturované jazyky

* Například jazyk Fortran, Basic, ...

**Formální báze** je takový formální prostředek (kalkulus, algebra, …), který umožňuje exaktně popsat všechny konstrukce daného jazyka. Ovšem nestrukturované jazyky formální bázi nemají.

**Otevřený podprogram** je uložen v rámci hlavního (často jediného) zdrojového textu. Nemá definované pevné rozhraní, tzv. vstupní a výstupní bod, parametry, výsledek apod. Vstup se děje skokem na příkaz, jímž má výpočet podprogramu začít, ukončení podprogramu je dáno vyvoláním příslušného příkazu (nikoliv doběhnutím výpočtu do/za určité místo). Parametry i výsledky jsou předávány jen jako globální proměnné – neexistují lokální proměnné, tudíž je jednoduché udělat chybu, kdy dojde k přespání jiných dat.

Pseudomodul – podprogram uložený v jiném souboru.

Analyzátor je program, který analyzuje vstupní text v nějakém programovacím jazyce a provádí jeho důkladnou kontrolu pouze na základě jeho textu.

Vyhodnocovací program/interpret je takový program, který jakmile rozpozná nějaký příkaz ve vstupním programu, který má na vstupu, tak jej ihned provede. Převádí tak vstupní program na posloupnost okamžitě prováděných akcí (např. Basic).

Překladač (někdy slangově označován jako kompilátor) je program, který vstupní text programu převádí na posloupnost příkazů jiného jazyka či stroje. Cílem může být např. binární soubor, který je přímo spustitelný na dané architektuře (cílovým jazykem překladu jsou instrukce procesoru dané platformy).

## Kapitola 4 Strukturované jazyky

* Například Algol, Pascal (původní návrh)

Floyd-Hoare logika – pro každý programový prvek je definována podmínka splněná před a po provedení operace popsané daným prvkem.

Odvozené typy – vznikají na bázi již existujících typů a to jak základních tak již existujících odvozených

Ukazatel – umožňují vznik rekurzivních datových struktur tím, že umožňují odkazovat určitému typu dat na sebe sama. Nese adresu libovolné paměťové buňky.

Pervasivní funkce – funkce, které jsou přítomny pro manipulaci s daty a jsou v jazyce přímo definovány, nicméně uživatel může vytvořit funkci stejného jména (čímž původní od místa definice pozbývá).

Důležité vlastnosti:

* Uzavřené podprogramy – dramatický milník, který zjednodušuje: rekurzi, ukrytí implementace a odluku od hlavního toku programu.
* Lokální proměnné v zanořených blocích – proměnné stejného jména, různého významu, různých možností přístupu k nim

Datové struktury jejichž tvorba je plně pod kontrolou uživatele, jsou revolucí strukturovaných programovacích jazyků na úrovni tvorby a definice dat. Z formálního pohledu je typ popisující datovou strukturu n-tice, která je jednoznačně identifikována svým jménem. Variantní datová struktura je sjednocením jednotlivých složek struktury.

N-tice ukládají jednotlivé složky v paměti za sebou, zpravidla bez mezer tak, aby struktura zabrala co nejméně paměti. Variantní datová struktura ukládá jednotlivé své složky v paměti přes sebe a velikost obsazené paměti je určena největší složkou struktury.

Adresa složky struktury se skládá ze dvou údajů: z adresy datové struktury jako takové a z posunutí (offset) v rámci ní.

Datový typ pole specifikuje homogenní datovou strukturu, kdy každá složka je pevně indexována určitou hodnotou a všechny složky jsou stejného typu.

Předávání struktur jako parametrů :

* Nelze
* Předání odkazem – jazyk manipuluje datové struktury pouze prostřednictvím odkazu
* Předání hodnotou – může být limitována velikost předávané struktury

Statické zanoření – vnoření na úrovni textu.

Dynamické zanoření – v době běhu programu, když se jednotlivé funkce navzájem volají, případně rekurzivně sami sebe.

## Kapitola 5 Modulární jazyky

Modul –

Knihovny – poskytují kompletní sadu funkcí pro manipulaci s určitým abstraktním typem, případně poskytují jakési zázemí pro danou výpočetní tématiku.

Skrývání implementace – moduly se o svá data starají v plném rozsahu

Rozhraní modulu – rozhoduje o tom jaká data, datové typy a funkce/procedury jsou vyváženy z modulu.

Závislost mezi moduly – moduly lze skládat do hierarchií, přičemž mezi typické struktury patří stromová hierarchie.

Předávání parametrů a výsledků mezi moduly – volaný i volající dodržují určité konvence

Předávání parametrů:

* Hodnotou, výsledkem – vytváří se kopie proměnné
* Odkazem – předává se ukazatel na danou proměnnou
* Jménem – předává se dovjice přístupových metod k dané entitě: pro zápis a čtení hodnot

Linker – spojovací program, který spojuje všechny moduly a části knihoven do jediného bloku, proveditelného souboru – sestavuje cílový (spustitelný) kód.

Na vstupu linkeru je relativní kód, který je výstupem překladu, má vazby na proměnné uskutečněné přes tabulky: tabulka symbolů, tabulka funkcí, tabulka lokálních proměnných.

Absolutní kód – linker určí adresu všech entit v programu.

# Opora IPP-II

## Kapitola 2

**Objektově orientované programování** je způsob abstrakce, kdy algoritmus implementujeme pomocí množiny zapouzdřených vzájemně komunikujících entit, z nichž každá má plnou výpočetní mocnost celého počítače.

**Objektově orientovaný systém** (program, aplikace) se skládá z jednoho či více objektů, které spolu komunikují a interagují při spolupráci na řešení daného problému.

**Objekt** je entita zapouzdřující stavové informace (data reprezentovaná dalšími objekty) a poskytující sadu operací nad tímto objektem nebo jeho částmi.

Zpráva je komunikační jednotka mezi dvěma libovolnými objekty, má odesílatele i příjemce, reakcí na ní je vyvolání patřičné metody objektu. Objekty vzájemně interagují a komunikují pomocí tzv. mechanismu zasílání zpráv.

Rozhraní – množina všech zpráv, kterým objekt rozumí.

Základní koncepty OOP:

* Objekty – základní jednotka modularity i struktury v OO programu, která umožňuje problém intuitivně rozdělit na přímo realitě korespondující podčásti.
* Abstrakce – pohled na vybraný problém reálného světa a jeho počítačové řešení. Detaily se skrývají do černé skříňky (black box), která je pro okolí definovaná pouze svým rozhraním, přes které komunikuje s okolím, a nikoli vnitřními detaily implementace.
* Zapouzdření - uživatel nemůže měnit interní stav objektů libovolný způsobem, ale musí k tomu využívat poskytované rozhraní. Každý objekt tedy nabízí protokol, který určuje, jak s ním mohou ostatní objekty interagovat.
* Polymorfismus – ta samá proměnná programu může během jeho běhu obsahovat či odkazovat různé objekty, a tak může zaslání stejné zprávy objektu ve stejné proměnné při provádění stejné části kódu vyvolat během různých kontextů rozdílné reakce.
* Dědičnost – nové objekty mohou sdílet a rozšiřovat chování těch již existujících bez nutnosti vše znovu reimplementovat.

Identita – porovnává zda jsou dva objekty totožné (jde o tentýž objekt).

Shoda – provádí porovnání objektů podle jejich obsahu. Za shodné mohou být označeny i neidetické objekty.

Model výpočtu obsahuje vždy alespoň dvě základní sémantické konstrukce:

1. Přiřazení – pojmenování objektu neboli přiřazení objektu do proměnné
2. Zaslání zprávy – samotná zpráva může kromě jména (identifikátoru) obsahovat také parametry,

Výhody OOP: vyšší míra abstrakce, přirozenější práce se zapouzdření a moduly, jednodušší dekompozice problémů, udržovatelnost a rozšiřitelnost, znovupoužitelnost

Nevýhody OOP: v některých oblastech neexistuje analogie s reálnými objekty, složitější sémantika, výsledný generovaný kód je pomalejší, režite na uložení objektů v paměti

Dělení OOJ:

1. Třídně orientované jazyky (class-based)
2. Objektově založené jazyky (object-based)

**Třída** je šablona (otisk), podle níž mohou být vytvářeny objekty (instance této třídy). Třída se také stará o správu protokolu objektu, směrování zpráv a obsahuje implementace některých metod.

Instance – objekt, který obsahuje naplněny instanční proměnné a odkaz na třídu, ze které vznikl.

Konstruktor – metoda, která slouží k inicializování datových složek instance třídy.

Kopírování objektů:

* Hluboká kopie – kromě objektu samého jsou kopírovány i objekty, které referencují instanční proměnné
* Mělká kopie – instanční proměnné obsahují odkazy na původní objekty (hluboká kopie do hloubky nula)

Rušení objektů v paměti

* Automaticky – existuje garbage collector, který jednou za čas vyhledá objekty, na které již neexistuje žádný odkaz a ty zruší. (finalizace – speciální metoda volaná pro úklid a uvolnění alokovaných zdrojů mimo objekt)
* Manuálně – pro likvidaci objektů je vyčleněna speciální metoda destruktor, kterou když programátor zavolá, tak na jejím konci je objekt uvolněn z paměti.

Klasifikace dědičnosti:

* Podle počtu potomků
  + Jednoduchá dědičnost – každý potomek má nejvýše jednoho přímého předka
  + Vícenásobná dědičnost – případ, kdy může třída dědit od více přímých předků (více jak jednoho)
* Podle toho co se dědí
  + Dědičnost implementace – kromě atributů jsou do dědičnosti zahrnuty celé metody včetně jejich implementace.
  + Dědičnost rozhraní – předpis nebo seznam metod, které je nutné v potomkovi implementovat.

Statické proměnné a metody –

Přístupy k typům:

1. Čistě objektový – vše je objekt a existuje třída, nebo případně objekt, od kterého jsou všechno ostatní odvozeny
2. Hybridně objektový – k dispozici sada základních, neboli primitivních typů, které lze případně skládat do homogenních nebo heterogenních struktur.
   1. Existuje kořenová třída – je předkem každé existující nebo nové třídy (Java, C#)
   2. Nemají žádnou kořenovou třídu (C++)

Staticky typovaný jazyk – určuje množinu operací, které objekt podporuje, již v době překladu programu. V případě, že se zjistí že po objektu je požadována nepodporovaná operace, neproběhne překlad úspěšně.

Dynamicky typovaný jazyk – provádí kontrolu až v době běhu programu.

Slabé typování – při chybě u konverze dochází k vyvolání chyby programu

Silné typování – může dojít k logické chybě při konverzi

Časná vazba – již při překladu je známo, která metoda bude vyvolána.

Pozdní vazba – identifikace potřebné metody probíhá až za běhu programu.

Redefinice metod (method overriding) je možnost jazyka definovat pro metodu podtřídy novou, specifičtější implementaci, než je obsažena v její nadtřídě.

Prototypově orientované jazyky – nerozlišuje se mezi instancemi a třídami

Rys – označení pro objekt, který obsahuje sdílené chování, a tím tak nahrazuje nebo doplňuje třídy.

## Kapitola 3 – Formalismy a jejich užití

## Kapitola 4 – UML

**UML** je jednotný grafický (vizuální) jazyk pro jednotnou specifikaci, vizualizaci, konstrukci a dokumentaci při objektově orientované analýze a návrhu i pro modelování organizace.

Pohled – nazýváme projekci systému (či jeho modelu)na jeden z jeho klíčových aspektů a znázorňujeme jej jedním či více diagramy UML (základní pohledy: strukturální, datový, pohled na chování a rozhraní).

Diagram je struktura podobná obecnému grafu obsahující množinu grafických prvků (vrcholy) propojených vztahy (hrany).

**Diagram tříd** je graf symbolů tříd, rozhraní, seskupení a dalších strukturních prvků propojených statickými vztahy (asociace, závislost, agregace, kompozice, generalizace, realizace).

* Asociace – je obecná relace mezi dvěma či více třídami zakreslená úsečkou spojující tyto třídy.
* Realizace je vztah mezi rozhraním a implementační třídou.
* Generalizace je statický vztah mezi obecnější a specifičtější entitou (v případě tříd mezi rodičovskou třídou a třídou potomka)
* Agregace vyjadřuje složení entity ze skupiny komponentních entit.
* Kompozice je speciální případ agregace, kdy každá komponentní třída smí náležet pouze jednomu celku.
* Závislost je vztah mezi prvky, v níž změna jednoho elementu má vliv na závislý element.

Diagram objektů – ukazuje objekty a jejich propojení, včetně identifikace význačných objektů v určitém čase.

Diagram seskupení je vhodný především pro modelování rozsáhlých systémů prezentaci vazeb mezi jeho podsystémy a dalšími moduly.

Diagram případů použití – graficky znázorňuje účastníky, případy použití, interakce a hranice systému.

Diagram datových toků (data flow diagram) je hierarchický model používaný při strukturované analýze a návrhu pro specifikaci chování systému.

Stavový diagram – se váže vždy na konkrétní třídu a ukazuje stavy, do kterých mohou její instance vstupovat a přechody mezi nimi.

## Kapitola 5 – Vlastnosti objektově orientovaných jazyků

Ortogonalita je nezávislost vlastnosti jazyka na programovacím paradigmatu. Obecněji je to nezávislost jisté technologie na kontextu použití. (například šablony a výjimky jsou ortogonální vůči OO paradigmatu, z čehož plnye, že je lze uplatnit i v jazycích, jež nejsou OO)

Datový typ popisuje reprezentaci, interpretaci a především strukturu hodnot používaných objekty v paměti počítače. Typová chyba je ohlášení chyby, která označuje použití nepovolených/nekompatibilních typů operandů při libovolné operaci.

Silně typová kontrola je takový typ kontroly, kdy jsou vždy detekovány všechny typové chyby, což vyžaduje schopnost určit typy všech operandů, ať již při kompilaci (staticky typované jazyky) nebo při běhu programu (dynamicky typované jazyky).

Klasifikace jazyků:

* Podle určování typů při zápisu programu:
  + Beztypové jazyky – v podstatě se jedná pouze o teoretické a formální jazyky nebo jazyky s jediným univerzálním typem (např. sigma kalkul a lambda kalkul)
  + Netypované – proměnná nemá určen ve zdrojovém kódě typ a tuto informaci nemá programátor k dispozici, interpret/kompilátor zajišťuje implicitní typové konverze automaticky (např. BASIC, shell, PHP především starší verze)
  + Typované – typ je určen ve zdrojovém programu u každé entity. Proměnná může mít typ určen explicitně (uvedením typu) nebo odvozením, kdy je výsledný typ výrazu odvozen z typů jeho operandů.
* Podle doby vytvoření vazby proměnné na typ:
  + Staticky typované – během programu, tj. v době návrhu kompilátoru, kompilace programu či zavádění programu. Např. Delphi, C++, Java
  + Dynamicky typované – typ proměnné je určen až za běhu programu. Např. Smalltalk. Všechny proměnné obsahují objekty, ale konkrétní typ záleží až na konkrétních přiřazených instancích.
* Podle způsobu typové kontroly:
  + Statická typová kontrola – kdy je většina typových kontrol prováděna během překladu. Např. C++, Pascal.
  + Dynamická typová kontrola – veškeré typové kontroly mohou být prováděn až za běhu programu, což však ovlivňuje i výkon programů v těchto jazycích a často vyžadují běh na virtuálním stroji. Například Smalltalk, JavaScript, Python nebo SELF.
* Podle důkladnosti typové kontroly:
  + Silně typované jazyky
    - Téměř silně typované jazyky – neexistuje možnost jak implicitně způsobit typovou chybu/nekonzistenci s výjimkou explicitní typové konverze s vypnutou typovou kontrolou
    - Středně silně typované jazyky – např. ML
    - Absolutně silně typované jazyky – silně typované a neobsahují naprosto žádné implicitní konverze – některé implementace Smalltalk, SELF
  + Slabě typované jazyky – Např. Pascal, Fortran, kde je možné i po úspěšné kompilaci dojít k typové chybě. Dokonce velmi slabě typované jazyky (C/C++) umožňují volnější práci s variantním typem i možnost nekontrolovat typy parametrů funkcí/metod.

Jmenný prostor – je kontejner pro identifikátory (či jiné entity jazyka). V tomto prostoru platí pravidlo, že uvnitř se nevyskytují žádné dva stejné identifikátory. Důvodem zavádění je zabránění kolizí jmen v rozsáhlých zdrojových textech.

Modifikátory viditelnosti – mění možnosti přístupu k různým entitám jazyka, a tak podrobněji konfigurovat koncept zapouzdření. Tři základní modifikátory: soukromý (private), chráněný (protected), veřejný (public).

Přetěžování metod – method overloading – je vlastnost umožňující definovat ve třídě (respektive protokolu objektu) více metod se stejným jménem. Jediným požadavkem bývá, aby se metody odlišovali v typech nebo v počtu parametrů.

Vícenásobná dědičnost

Rozhraní – je schéma, které deklaruje seznam metod (jména, parametry, návratové typy). Použití rozhraní na jistou třídu pak vynucuje implementaci všech metod uvedených v rozhraní. I v případě jednoduché dědičnosti lze každé třídě uvést několik rozhraní. Možnost nahradit vícenásobnou dědičnost, zachovává výhody a vylučuje nevýhody.

Výjimky – mechanismus, který popisuje způsob šíření informace o chybě, způsob zastavení/přeskočení výpočtu a umožňuje provést ošetření chyby až za samotným algoritmem a tím zlepšit čitelnost kódu i samotného algoritmu. Umožňuje také spouštění finalizační sekce, jejíž provedení je garantováno i v případě vyvolání vyjímky.

Šablona – mechanismus, který umožňuje parametrizaci definic datových typů (a potažmo i tříd). V definici nového šablonového typu potom daný parametr využíváme jako proměnnou, která obsahuje identifikaci jiného typu. Šablony mohou obsahovat i více parametrů.

Šablony mohou být implementovány v zásadě 3 způsoby:

* Staticky – šablona je zpracovávána a využívána pouze při kompilaci zdrojového kódu, obdoba preprocesoru jazyka
* Dynamicky – v případě hybridních jazyků nejčastěji pomocí virtuálních fcí, metod
* Ad hoc

Role objektu (objekt s více typy)

Objekt s více typy je objekt, který může mít v jeden okamžik více typů. Původ tohoto pojmu je v objektově orientovaných databázích, kde pak hovoříme o rolích objektu. Systémy s rolemi používají většinou perzistentní objekty, což jsou objekty, které přežívají nezávisle na běhu nebo ukončení obsluhující aplikace.

Perzistentní objekt – objekt nebo instance, která přežívá rámec aplikace (čas kdy aplikace běží) a při dalším spuštění aplikace je tentýž objekt opět k dispozici v přesně stejném stavu jako měl při posledním ukončení aplikace.

Implementace perzistentních objektů:

* Ukládat pouze stav objektu (hodnoty atributů) – metody objektu musí být dosažitelné jiným způsobem, z teoretického pohledu se však nejedná o plnohodnotnou perzistenci, jelikož lze pohodlně zajistit knihovnami pro ukládání a načítání stavu objektů
* Ukládat stav objektu i jeho metody (případně třídu objektu)

Virtuální stroj – speciální softwarová vrstva, jejíž primárním účelem je odstínit pro běžící aplikaci hardwarová specifika počítače, na němž je vykonávána.

Virtuální stroje se v přístupu k vykonání kódu líší a volí jeden z následujících přístupů:

* Přímá interpretace kódu ze syntaktického stromu zdrojového kódu
* Kompilace do mezikódu a jeho následná přímá interpretace
* Kompilace do mezikódu, jeho následný překlad do nativního strojového kódu a vykonání (SELF, Java)

Jako mezikód se ve většině případů využívá tzv. bytekód (bytecode), což je binární forma mezikódu s členěním po bytech.

Obrazy objektové paměti – při spuštění nedochází k inicializaci objektů, ale je obnoven naposledy uložený stav. Celá řada výhod (např. rychlý start aplikací). Smalltalk, SELF.

Nevýhoda virtuálního stroje je podstatně vyšší režie, než v případě nativních programů.

Návrhový vzor – systematicky nazývá, vysvětluje a vyhodnocuje důležitý a v objektově orientovaných systémech se opakující návrh.

Návrhové vzory lze rozdělit:

1. Vytvářecí (instanciační) – Jedináček (Singleton) – omezuje možnosti vytvářet z třídy více jak jednu instanci.
2. Strukturální – Dekorátor – pomáhá přidávat vlastnosti objektům aniž bychom rušili jejich stávající vlastnosti
3. Vzory chování – Pozorovatel, Návštěvník