Úvod do softwarového inženýrství

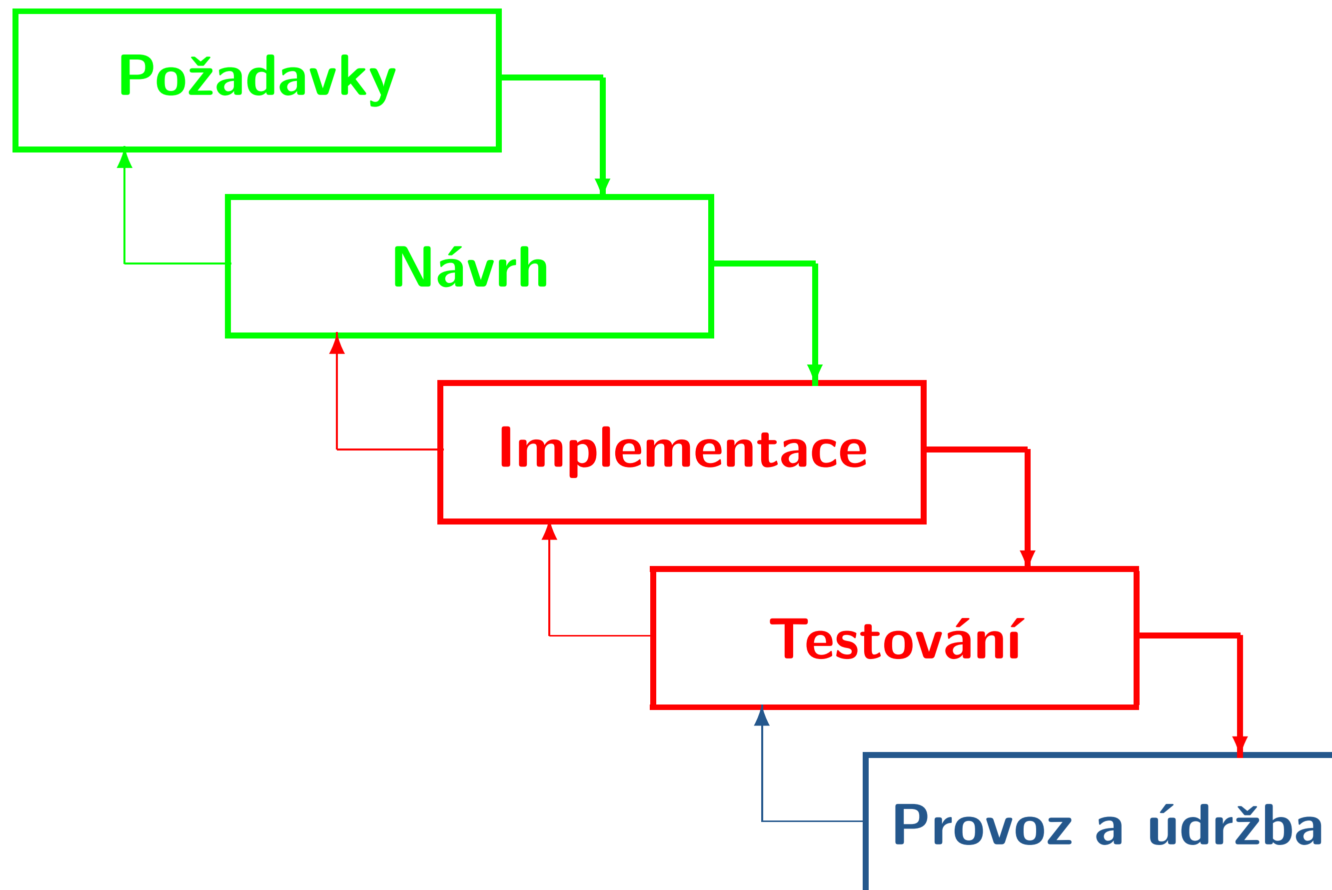
IUS 2024/2025

7. přednáška

Ing. Radek Kočí, Ph.D.
Ing. Bohuslav Křena, Ph.D.

1. a 4. listopadu 2024

Téma přednášky



Implementace softwaru

- Vlastnosti sofwarového produktu
- Výběr programovacího jazyka
- Strategie implementace
- Systémy pro správu verzí
- Dokumentace

Implementace softwaru

... je proces transformace návrhu jednotlivých modulů (návrhových podsystémů) a jejich vzájemných vazeb do programové realizace.

Výstupem etapy implementace je **spustitelný software** (softwarový produkt).

Podíl implementace na celkovém objemu prací
v životním cyklu softwaru **se snižuje**:

- zavedením vysokoúrovňových jazyků (větší míra abstrakce)
- využíváním integrovaných vývojových prostředí
- využíváním pokročilých prostředků trasování a ladění programů
- generováním aplikací z modelů
- vývojem prostředků spolupráce aplikací (middleware)
- rozšířením a *rozvojem* OO/AO/komponentního přístupu
- znovupoužitelností (využití internetu)

Vlastnosti softwarového produktu

Pro implementaci potřebujeme jasné cíle – všechna kritéria nelze splnit.
např. použitelnost × bezpečnost; efektivnost × udržovatelnost

Kromě funkčnosti nás zajímají další vlastnosti softwaru,
které se projeví až při nasazení softwaru.

Použití

- **Správnost** – míra, do jaké software vyhovuje specifikaci.
- **Použitelnost** – úsilí, které je nutné vynaložit na to, aby se dal software používat. Zahrnuje i **srozumitelnost výstupu programu**.
- **Efektivnost aplikace** – doba odezvy, požadavky na paměť, . . .
- **Efektivnost procesu tvorby programu** – čas potřebný na vývoj, náklady
- **Bezpečnost** – míra odolnosti vůči neoprávněným zásahům do systému.
- **Spolehlivost** – pravděpodobnost, že software bude v daném čase vykonávat zamýšlenou funkci.

Vlastnosti softwarového produktu

Přenos

- **Přenositelnost** – úsilí, které je nutné pro přenos softwaru z jedné platformy na jinou.
- **Interoperabilita** – úsilí, které je potřebné k zajištění spolupráce systému s jinými systémy.
- **Znovupoužitelnost** – míra, do jaké je možné jednotlivé části softwaru znovu použít v dalších aplikacích.

Změny

- **Udržovatelnost** – úsilí, které je potřeba vynaložit na další vývoj a údržbu softwaru podle měnících se potřeb zákazníka a také v důsledku měnícího se okolí (např. změna legislativy). Zahrnuje i **čitelnost a pochopitelnost** zdrojového kódu programu.
- **Testovatelnost** – úsilí nutné pro testování vlastností softwaru, např. zda se chová správně.
- **Dokumentovanost** – míra, do které jsou všechna rozhodnutí při vývoji zdokumentována a kontinuita dokumentace v průběhu všech etap vývoje.

Výběr programovacího jazyka

Kritéria výběru vhodného programovacího jazyka:

- zkušenosti programátorů s daným jazykem
- vhodnost jazyka pro příslušnou aplikaci, rozsah projektu
- dostupnost podpůrných prostředků pro vývoj systémů v daném jazyku
- rozšířenost jazyka
- požadavky na přenositelnost
- použitelnost na vybraném výpočetním prostředí
- existující knihovny a možnosti znovupoužití
- cena vývojového prostředí
- budoucí strategie, orientace organizace na určité vývojové prostředí
- požadavky zákazníka

Důležité vlastnosti

Při programování v malém se zajímáme o:

- **vlastnosti jazyka**: jednoduchost, srozumitelnost, side effects
- **syntax jazyka**: konzistentnost, jednoduchost, možnost tvorby čitelných a snadno udržovatelných programů
- **datové typy**: statické $\times$ dynamické, elementární $\times$ strukturované
- **řídicí konstrukce**: posloupnost, výběr, cyklus, rekurze, backtracking, ...
- **čitelnost**: možnosti formátování, syntax, pojmenování identifikátorů

Při programování ve velkém se zajímáme o:

- **podporu abstrakce**: procedury, funkce, generické typy údajů
- **plánovací mechanizmy**: dělení práce, harmonogram, zdroje
- **prostředí**: týmová tvorba softwaru, efektivní kompilace, podpora integrace systému, uchovávání a identifikace verzí

Generace programovacích jazyků

- **1. generace**
  - programování přímo v binárním kódu
- **2. generace**
  - asemblery, symbolické vyjádření binárních instrukcí (1 ku 1)
- **3. generace**
  - strukturované programování
  - strojově nezávislé jazyky
  - jeden příkaz se transformuje do 5-10 instrukcí v binárním kódu
  - procedurální jazyky: Fortran, Pascal, C, ...
- **$3\frac{1}{2}$. generace** (objektově orientované jazyky)

Generace programovacích jazyků

- **4. generace**
  - neprocedurální jazyky, vizuální jazyky, doménově specifické jazyky
  - snaha o zjednodušení programování, využívají vestavěné funkce/komponenty (definuje se, co je třeba vykonat, ne jak)
  - nemožnost ovlivnit zabudovaný způsob realizace funkcí
  - jeden příkaz se přeloží do cca 30-50 instrukcí v binárním kódu (často méně efektivní realizace kódu)
  - SQL, MATLAB, doménově specifické jazyky, . . .
  - *"End-User Programming"* – např. Microsoft Excel
- **5. generace**
  - neprocedurální jazyky
  - definují se objekty, pravidla, omezení, kritéria pro řešení, postup řešení pak hledá stroj
  - umělá inteligence, neuronové sítě, . . .

Paradigmata programovacích jazyků

- **imperativní** × **deklarativní**
  - imperativní (Fortran, Algol, Ada, C, Pascal, Java, C++)
  - deklarativní (Prolog, Lisp, Haskell)
- **procedurální** × **funkcionální**
  - procedurální (Algol, Ada, C)
  - funkcionální (Lisp, Haskell, Scheme)
- **objektově orientované**
  - class-based (Simula, Smalltalk, Java, C++, C#)
  - prototype-based (Self, Io, Prothon)
- logické (Prolog)
- paralelní (MPI, Shared-Memory, CUDA)
- ...

Typy, kontrola typů

Význam typování

- určit sémantický význam elementů (hodnoty v paměti)
  $\Rightarrow$ víme jaké operace je možné provést, můžeme provádět kontrolu typové konzistence atp.

Staticky typované jazyky

- k typové kontrole dochází v době kompilace
- jazyky `C++`, `Java`, ...

Dynamicky typované jazyky

- k typové kontrole dochází v době běhu programu
- jazyky Smalltalk, Self, Python, Lisp ...
- *dynamická typová kontrola probíhá u všech jazyků, avšak jako dynamicky typované se označují ty, které nemají statickou kontrolu*
- *některé staticky typované jazyky (`C++`, `Java`) umožňují dynamické přetypování, čímž částečně obcházejí statickou typovou kontrolu*

Typy, kontrola typů

I. Ukázka chování staticky a dynamicky typovaných systémů

```
var x;          // (1)
x := 5;         // (2)
x := "hi";      // (3)
```

- staticky typované: řádek č. 3 je ilegální
- dynamicky typované: řádek č. 3 je OK (není požadovaná typová konzistence pro proměnnou x)

II. Ukázka chování staticky a dynamicky typovaných systémů

```
var x;          // (1)
x := 5;         // (2)
5 / "hi";       // (3)
```

- staticky typované: řádek č. 3 je ilegální
- dynamicky typované: řádek č. 3 vyvolá chybu za běhu programu

Typy, kontrola typů

Silně a slabě typované jazyky

- tyto pojmy dostávaly různé významy
- bývá obtížné porozumět, co konkrétní autor míní užitím těchto pojmů

Silně a slabě typované jazyky (interpretace)

- silně typované $\Rightarrow$ silná omezení na kombinace typů, zamezení kompilace či běhu kódu, který může obsahovat nekorektní data (nekompatibilní typy)
- slabě typované $\Rightarrow$ slabá omezení na kombinace typů (obsahují např. implicitní přetypování)
- *silně typované jazyky bývají nazývány typově bezpečné (type safe)*

Příklady

- Haskell $>$ Java $>$ Pascal $>$ C
- $3 + "27" \Rightarrow 30$, $"327"$, *nemožné*

Strategie implementace

- postup, jakým se realizují jednotlivé softwarové součásti a odevzdávají na testování
- částečná závislost na architektuře a strategii návrhu
- potřeba inkrementálního (postupného) vývoje
- strategie implementace zpravidla podmiňuje strategii testování

Implementace zdola-nahoru

- systém je možné předvádět až po jeho úplném dokončení
- možnost přímého použití odladěných modulů nižších úrovní
- chyby v logice se identifikují až v etapě integračního testování
- testování modulů na nižších úrovních: potřeba speciálních modulů (simulace chování/dat vyšších úrovní)
- testování modulů jednotlivě je jednodušší než testování logiky celého systému

Strategie implementace

Implementace shora-dolů

- možnost demonstrace systému poměrně brzy
- včasná identifikace najzávažnějších chyb
- logika systému se ověřuje několikrát (testování celého systému)
- testování systému: potřeba simulačních modulů (simulace práce podsystémů)
- nedá sa použít, pokud se požaduje implementace některých modulů nejnižší úrovně na začátku (např. výstupní sestavy)
- testování logiky systému je náročnější než testování modulů jednotlivě

Strategie implementace

Implementace shora-dolů

- možnost demonstrace systému poměrně brzy
- včasná identifikace najzávažnějších chyb
- logika systému se ověřuje několikrát (testování celého systému)
- testování systému: potřeba simulačních modulů (simulace práce podsystémů)
- nedá sa použít, pokud se požaduje implementace některých modulů nejnižší úrovně na začátku (např. výstupní sestavy)
- testování logiky systému je náročnější než testování modulů jednotlivě

V praxi se požívá kombinace přístupu zdola-nahoru a shora-dolů.

Dobré programátorské praktiky

- **Komentáře**
- **Jednoduchost**
- **Přenositelnost**
  - žádné magické konstanty (cesty, soubory, adresy, atd.)
- **Jednotný programátorský styl (house style)**
  - pojmenování (např. proměných)
  - odsazování
  - bílé znaky (mezery, tabelátory, volné řádky)
  - délka řádků
  - způsob ošetření chyb
  - ...

Nástroje pro správu verzí

= **inteligentní sdílení a zálohování**

- Usnadňují souběžný vývoj softwaru více lidmi.
- Sledují historii změn zdrojových textů i dokumentů.
- Středem celého systému je datové skladiště (*repository*).
- Metody ukládání: changeset vs. snapshot
- Dostupné nástroje:
  - Revision Control System (RCS), 1982
  - Concurrent Versions System (CVS), 1986
  - Apache Subversion (SVN), 2000, `https://subversion.apache.org/`
  - Git, 2005, `https://git-scm.com/`
  - ...

Subversion – typický pracovní cyklus

- Aktualizace lokální (pracovní) kopie
  `svn checkout`
  `svn update`
- Práce (provádění změn)
  `svn add`
  `svn delete`
  `svn copy`
  `svn move`
- Kontrola vlastních změn
  `svn status`
  `svn diff`
  `svn revert`
- Připojení změn od ostatních
  `svn update`
  `svn resolved`
- Zapsání vlastních změn
  `svn commit`

Dokumentace programu

- **interní dokumentace**
  - současně se čtením programu
  - slouží pro opravu chyb, k údržbě
- **externí dokumentace**
  - pro ty, kdo se nemusí zabývat vlastním programem
  - např. pro návrháře pro potřeby modifikace návrhu apod.
  - popis problému, algoritmů, údajů, ...
- **hlavičky souborů**
  - co soubor obsahuje
  - kdo a kdy ho vytvořil/upravil
  - závislosti na dalších souborech ...
- **komentáře**
  - komentovat účelně (hlavně nestandardní a neočekávané obraty)
  - stručný popis algoritmů
  - ...

Dokumentace programu

Co by měla dokumentace obsahovat:

- název
- autoři
- datum
- kam je daný modul (soubor) zařazen
- účel
- předpoklady (jaké se očekávají vstupy apod.)
- . . .

Dokumentaci lze automaticky generovat ze zdrojového kódu.

- struktura programu, komentáře, kontrakty
- Doxygen, Javadoc, Sandcastle, . . .

Implementace – shrnutí

Programy se nevytvářejí tak,
aby se lehce psaly,
ale aby se lehce četly a modifikovaly!

Validace a verifikace programu

Zjišťujeme, zda software odpovídá specifikaci a splňuje potřeby uživatele.

- verifikace: **Vytváříme výrobek správně?**
  (podle požadavků, specifikace, . . .)
- validace: **Vytváříme správný výrobek?**
  (Jsou splněny potřeby uživatele? Odpovídá tomu specifikace?)

Sledované vlastnosti:

- správnost
- spolehlivost
- efektivnost
- bezpečnost
- . . .

Validace a verifikace programu

Správnost výrobku nepostačuje!
Dokonce správnost někdy není nevyhnutelná!

Příklad specifikace procedury SORT:

- Vstupní podmínka A: array(1..N) of integer
- Výstupní podmínka B: array(1..N) of integer, přičemž $B(1) \leq B(2) \leq \ldots \leq B(N)$

Implementace:

$procedure\ SORT$
$begin$
$\quad for\ i := 1\ to\ N\ do$
$\quad\quad B[i] := 0;$
end

Verifikace: $\surd$

Validace: $\times$

Validace a verifikace programu

Správnost výrobku nepostačuje!
Dokonce správnost někdy není nevyhnutelná!

Příklad specifikace procedury SORT:

- Vstupní podmínka A: array(1..N) of integer
- Výstupní podmínka B: array(1..N) of integer, přičemž $B(1) \leq B(2) \leq \ldots \leq B(N)$

Implementace:

$$
\begin{array}{l}
procedure\ SORT \\
begin \\
\quad for\ i := 1\ to\ N\ do \\
\qquad B[i] := 0; \\
end
\end{array}
$$

Verifikace: $\surd$

Validace: $\times$

Chyba ve specifikaci: . . . a prvky pole B jsou permutací prvků pole A.

Cíle verifikace a validace

- **odhalit chyby během vývoje**
  Test, který neodhalí nesprávné chování systému, je neúspěšný.
- **prokázat požadované vlastnosti**

Dijkstra:

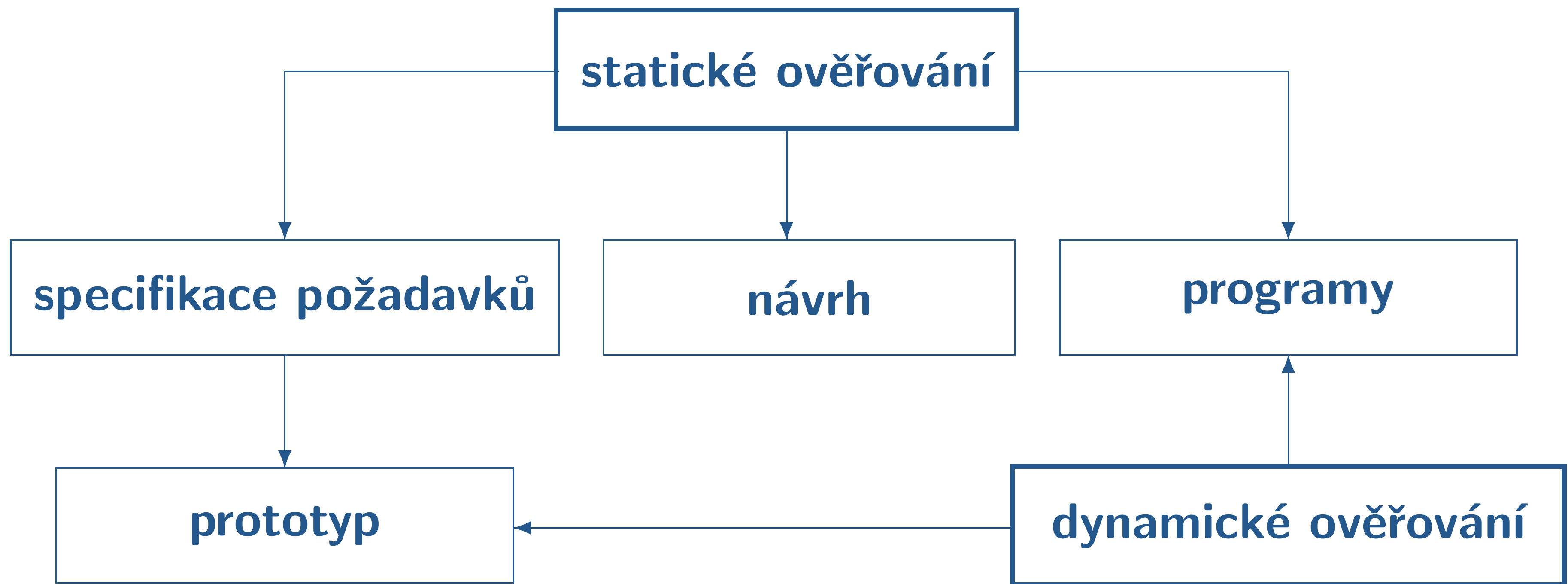
Testování nemůže prokázat, že v programu nejsou chyby.
Může pouze ukázat, že tam chyby jsou!

Murphy:

Když může systém *spadnout*, tak taky spadne,
a to v tom nejnevhodnějším okamžiku.

Typy ověřování

- **statické** – nevyžaduje běh programu, lze v libovolné etapě vývoje SW
- **dynamické** – proces odvození vlastností výrobku na základě výsledků použití (běhu) programu s vybranými vstupy



Statické ověřování – Prohlídka dokumentu

Prohlídka dokumentu je založena na statické prohlídce vytvořených dokumentů (včetně zdrojových textů programů).

Existují různé přístupy

- formální (Inspection)
- neformální (Walkthrough)
- koukání přes rameno (Over-The-Shoulder)
- párové programování (Pair Programming)
- koupací kačenka (Rubber Duck Debugging)

IEEE Std 1028-2008: IEEE Standard for Software Reviews and Audits

Knihu v seriózním vydavatelství před vytištěním také přečte několik osob.

Code Review

Doporučení

- 200 až 400 LOC pro prohlídku
- obvyklá rychlost procházení 300 LOC/h
- délka nejlépe do 60 minut (max. 90 minut)
- intenzita odhalování 15 chyb za hodinu

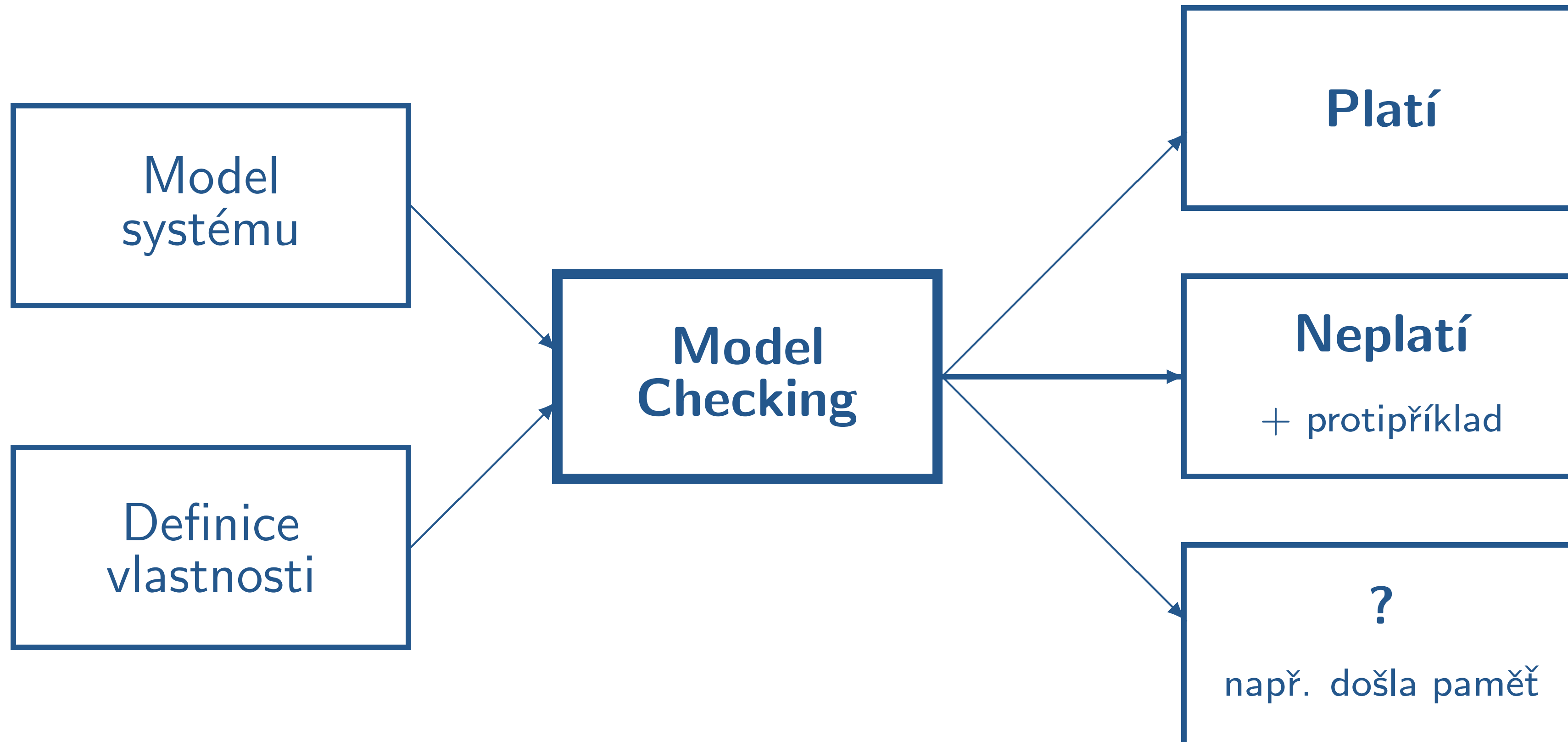
Výhody

- nejefektivnější způsob odhalování chyb v kódu
- zvyšuje i čitelnost a udržovatelnost kódu
- zlepšuje schopnosti méně zkušených programátorů
- úspora nákladů později

Nevýhody

- nemusí to být moc příjemné

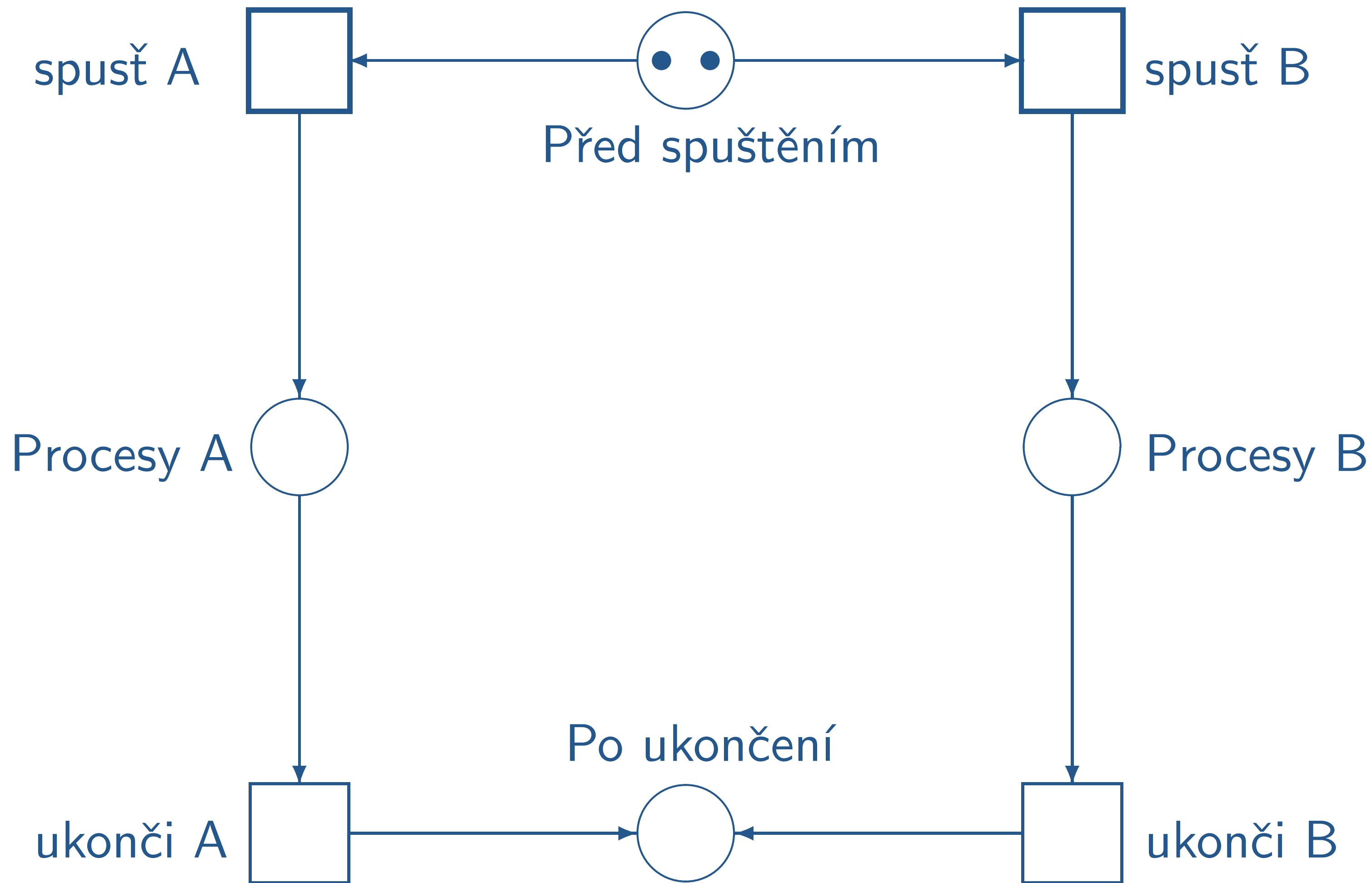
Statické ověřování – Formální verifikace



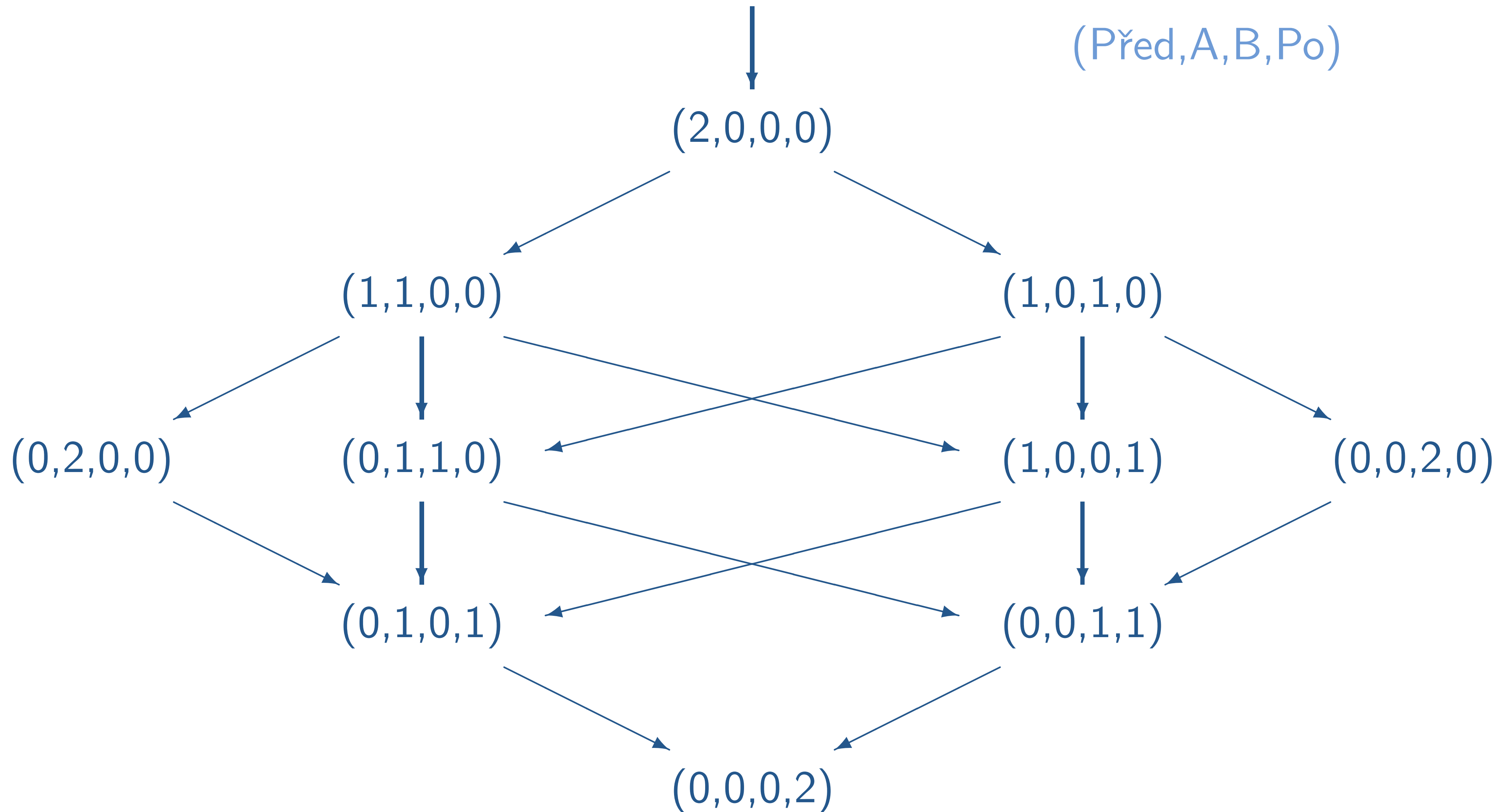
Statické ověřování – Formální verifikace

- formální matematický důkaz
- ověřovaný dokument musí být formálně reprezentovaný (přesná definice sémantiky)
- **modelování systému**
  - automaty
  - Petriho sítě
  - procesní algebry
  - programovací jazyky
  - ...
- **specifikace vlastností**
  - obecné vlasnosti (např. bez uváznutí)
  - tvrzení (assertions)
  - zakázané stavy (bad states)
  - temporální logiky
  - ...

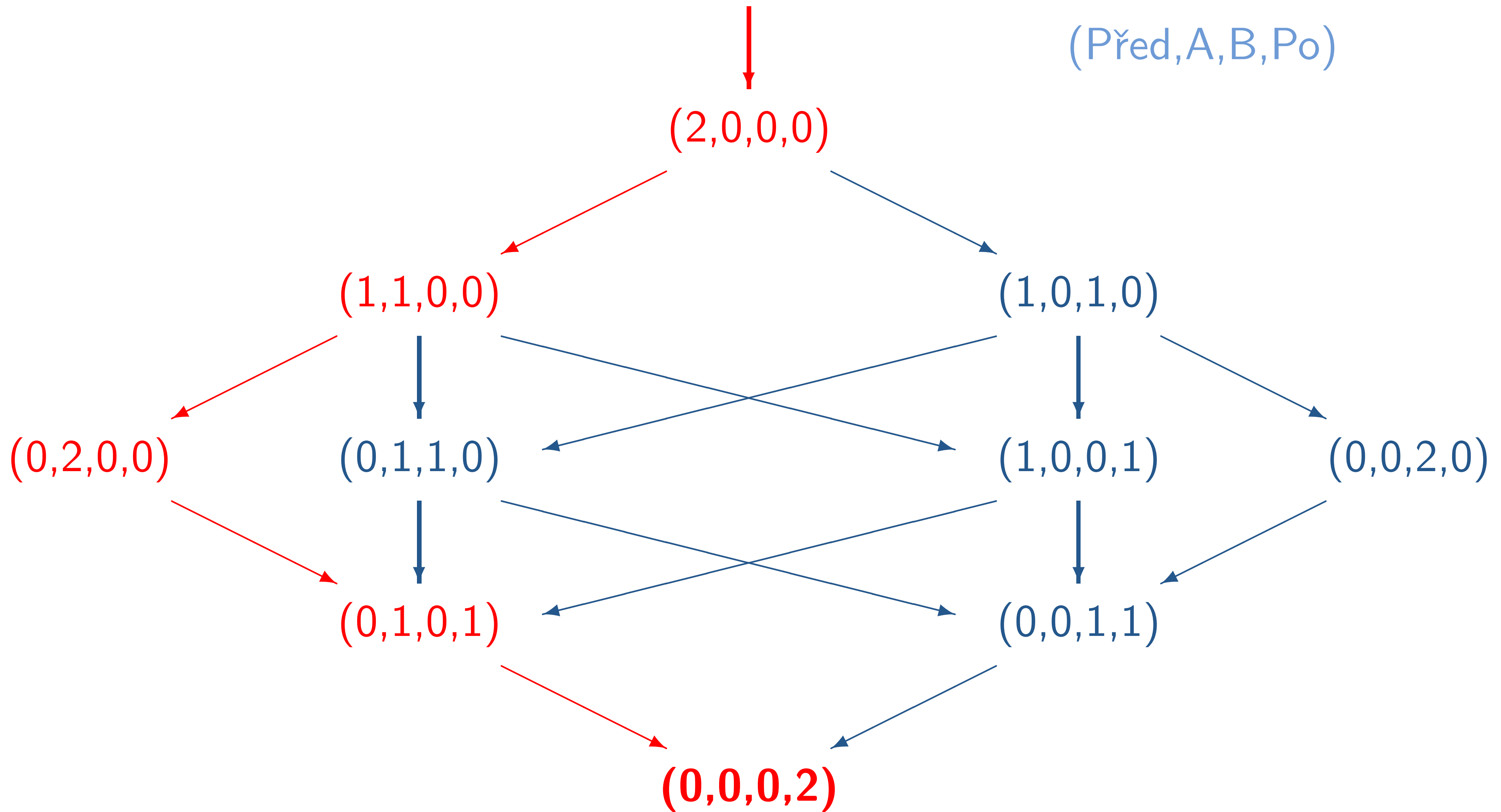
Jednoduchý příklad modelu



. . . a jeho stavový prostor

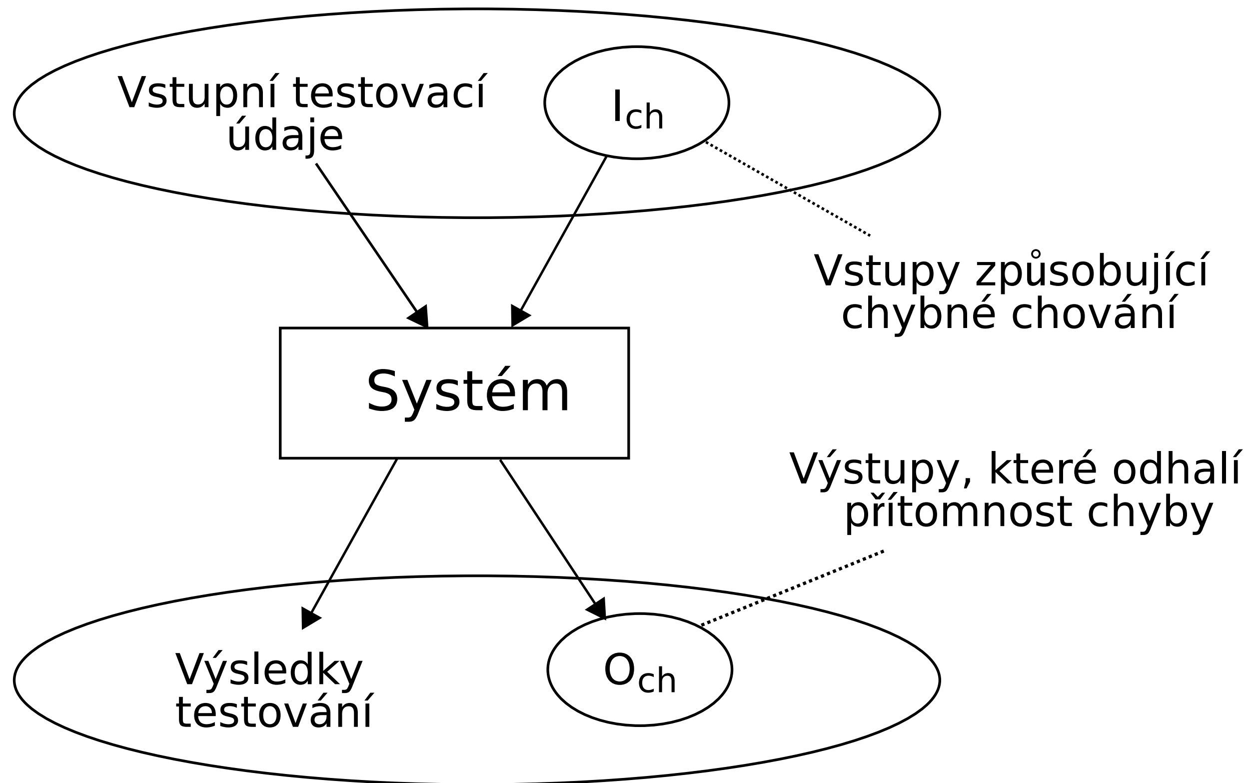


Cesta bez spuštění procesu B

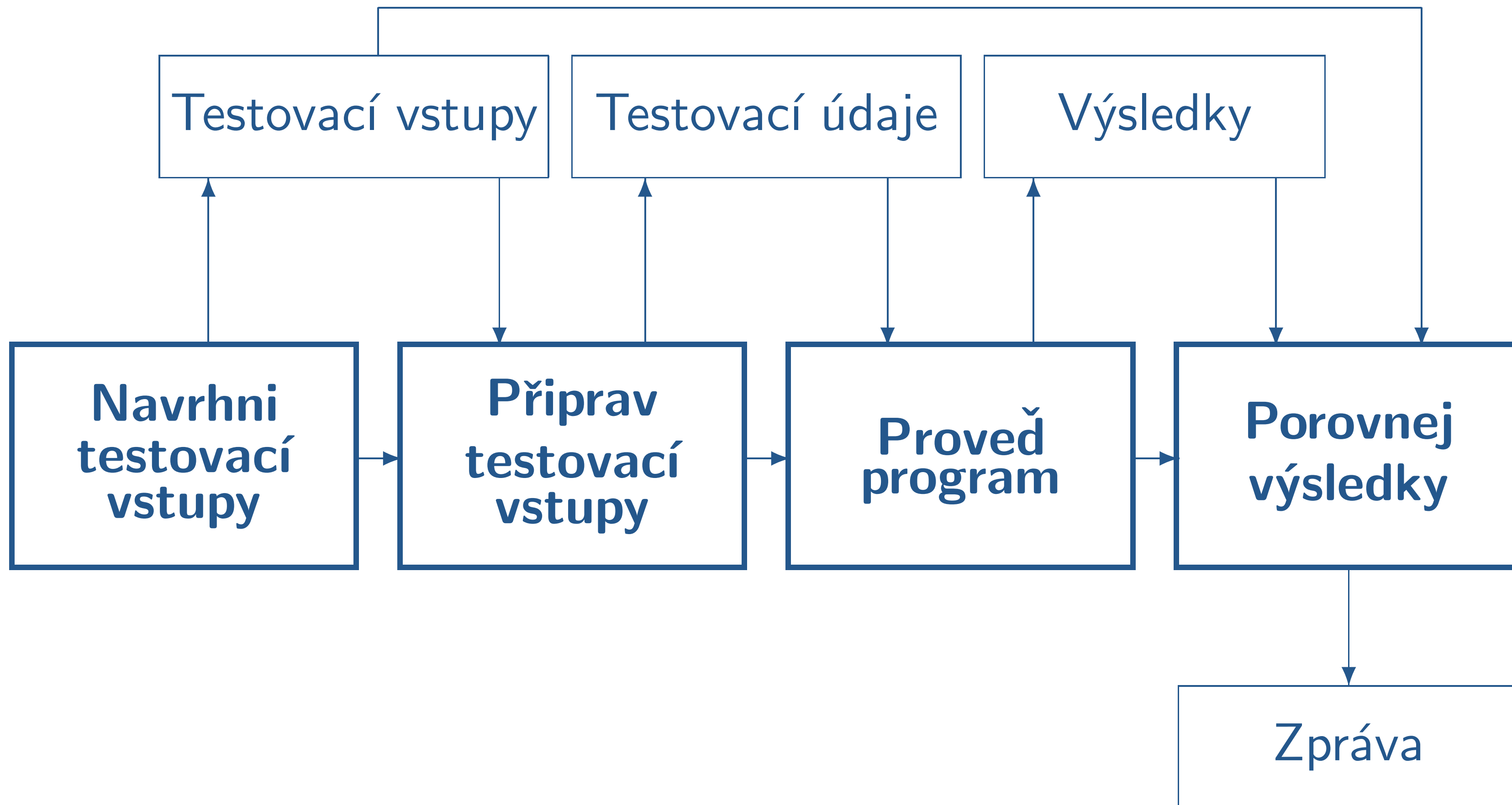


Dynamické ověřování – Testování

- cíl: vybrat takové testovací vstupy, pro které je pravděpodobnost příslušnosti do množiny I_{ch} vysoká



Proces testování



Množina testovacích vstupů

- velikost množiny testovacích vstupů musí být *přijatelná*
- množina testovacích vstupů se vybírá na základě testovacího kritéria
- testovací kritérium určuje podmínky, které musí splňovat množina testovacích vstupů, např. pokrytí všech příkazů v programu
- testovací kritérium může splňovat více množin testovacích vstupů

Vlastnosti testovacího kritéria

- **spolehlivost**: kritérium K je spolehlivé, když všechny množiny testovacích vstupů splňující kritérium K odhalí ty samé chyby $\Rightarrow$ nezáleží na tom, která množina testovacích vstupů se vybere, vždy odhalíme ty samé chyby
- **platnost**: kritérium K je platné, když pro každou chybu v programu existuje množina testovacích vstupů, která splňuje kritérium K a která odhalí chybu

Když je testovací kritérium spolehlivé a platné a množina testovacích vstupů, která splňuje kritérium, neodhalí žádné chyby, tak **program neobsahuje chyby**.

Vlastnosti testovacího kritéria

- **spolehlivost**: kritérium K je spolehlivé, když všechny množiny testovacích vstupů splňující kritérium K odhalí ty samé chyby $\Rightarrow$ nezáleží na tom, která množina testovacích vstupů se vybere, vždy odhalíme ty samé chyby
- **platnost**: kritérium K je platné, když pro každou chybu v programu existuje množina testovacích vstupů, která splňuje kritérium K a která odhalí chybu

Když je testovací kritérium spolehlivé a platné a množina testovacích vstupů, která splňuje kritérium, neodhalí žádné chyby, tak **program neobsahuje chyby**.

ALE

Bylo dokázané, že **neexistuje algoritmus**, který určí platné kritérium pro libovolný program.

Techniky testování

- **náhodné testování**
  - množina testovacích vstupů se vybere náhodně
- **funkcionální testování**
  - na základě specifikace programu (vstupy, výstupy)
  - metoda černé skříňky
    black box, data driven, functional, input/output driven, closed box
- **strukturální testování**
  - na základě vnitřní struktury programu
  - metoda bílé skříňky
    white box, glass box, logic driven, path oriented, open box
- **testování rozhraní**
  - na základě znalostí rozhraní mezi moduly a specifikace programu

Funkcionální testování

- zjištění zda vstupně-výstupní chování vyhovuje specifikaci
  např. matematická funkce se specifikuje vstupy a výstupy
- testovací vstupy se odvozují přímo ze specifikace
- neuvažuje se vnitřní struktura, logika modulu
  $\Rightarrow$ velká množina testovacích vstupů (problém)
- úplné funkcionální testování je v praxi nemožné

Příklad: ABS(x)

vstup (x)	výstup
-5	5
-2	2
0	0
5	5

Třídy ekvivalence vstupů/výstupů

- každý možný vstup/výstup patří do jedné z tříd ekvivalence, pro které je chování systému identické (vstup-výstup)
- žádný vstup/výstup nepatří do více tříd ekvivalence
- pokud se při daném vstupu/výstupu zjistí chyba, tak stejnou chybu je možné odhalit použitím jiného vstupu/výstupu z dané třídy ekvivalence

Třídy ekvivalence vstupů/výstupů

Granularita třídy ekvivalence

- rozsah
- hodnota
- podmnožina

Výběr testovacích údajů z třídy ekvivalence

- průměr, medián třídy ekvivalence
- hranice třídy ekvivalence (příp. s okolními hodnotami)
- náhodně (doplnění množiny testovacích vstupů)

Příklad: ABS(x)

třídy ekvivalence	vybrané vstupy/výstupy (x:int)
$(-\infty, 0)$	$(-32767, 32767)$, $(-16384, 16384)$, $(-1, 1)$, ...
0	$(0, 0)$
$(0, \infty)$	$(32768, 32768)$, $(16384, 16384)$, $(1, 1)$, ...

Strukturální testování

- vychází se z vnitřní struktury programu
  testuje se implementace programu
- snaha o pokrytí různých struktur programu – řízení, data
- kritéria:
  - založená na tocích řízení (pokrytí cest, pokrytí rozhodovacích bloků nebo podmínek a pokrytí příkazů)
  - založená na tocích dat
- mutační testování
  - do programu se úmyslně zavedou chyby
  - kontrolujeme, zda navržené testy tyto chyby odhalí (kvalita testu)

Příklad:

```
if x > 0 then
    y := x
else
    y := -x;
```

Testování vícevláknových aplikací (1/3)

Mějme následující jednoduchý paralelní (concurrent) systém:

`x=0` . (`x++` ‖ `x+=2`)

Jaká hodnota bude v proměnné `x` po dokončení výpočtu?

Testování vícevláknových aplikací (1/3)

Mějme následující jednoduchý paralelní (concurrent) systém:

`x=0` . (`x++` ‖ `x+=2`)

Jaká hodnota bude v proměnné `x` po dokončení výpočtu?

Vlákno 1:	Vlákno 2:	x1	x2	x
`load x`		0		0
`inc`		1		0
`store x`		1		1
	`load x`		1	1
	`add 2`		3	1
	`store x`		3	**3**

Testování vícevláknových aplikací (2/3)

Mějme následující jednoduchý paralelní (concurrent) systém:

`x=0 . ( x++ || x+=2 )`

Jaká hodnota bude v proměnné `x` po dokončení výpočtu?

Vlákno 1:	Vlákno 2:	x1	x2	x
`load x`		0		0
`inc`		1		0
	`load x`	1	0	0
`store x`		1		1
	`add 2`		2	1
	`store x`		2	**2**

Testování vícevláknových aplikací (3/3)

Mějme následující jednoduchý paralelní (concurrent) systém:

`x=0` . (`x++` ‖ `x+=2`)

Jaká hodnota bude v proměnné `x` po dokončení výpočtu?

Vlákno 1:	Vlákno 2:	x1	x2	x
`load x`		0		0
`inc`		1		0
	`load x`	1	0	0
	`add 2`	1	2	1
	`store x`	1	2	2
`store x`		1		**1**

Testování vícevláknových aplikací (3/3)

Mějme následující jednoduchý paralelní (concurrent) systém:

`x=0 . ( x++ || x+=2 )`

Jaká hodnota bude v proměnné `x` po dokončení výpočtu?

Vlákno 1:	Vlákno 2:	x1	x2	x
`load x`		0		0
`inc`		1		0
	`load x`	1	0	0
	`add 2`	1	2	1
	`store x`	1	2	2
`store x`		1		**1**

Kolikrát musíme zopakovat test, abychom dostali chybný výsledek?

Testování vícevláknových aplikací (3/3)

Mějme následující jednoduchý paralelní (concurrent) systém:

`x=0` . (`x++` ‖ `x+=2`)

Jaká hodnota bude v proměnné `x` po dokončení výpočtu?

Vlákno 1:	Vlákno 2:	x1	x2	x
`load x`		0		0
`inc`		1		0
	`load x`	1	0	0
	`add 2`	1	2	1
	`store x`	1	2	2
`store x`		1		**1**

Kolikrát musíme zopakovat test, abychom dostali chybný výsledek?

Ze 20 možných proložení vláken jsou obvyklá jenom 2 proložení.
Je **obtížné** najít podobné problémy klasickým (*sekvenčním*) testováním.

Příklad reálné chyby

Vlákno 1:

```
...
if (p != null) {
   ...
   p.doSomeWork();
   ...
}
```

Vlákno 2:

```
...
...
p = null;
...
...
...
```

- Tato chyba se při běžném testování objevila jednou z 10 000 spuštění.
- Proto IBM dodalo software s touto chybou zákazníkům.

Jak tedy testovat vícevláknové aplikace?

- systematické testování (řízení plánovače)
- vkládání šumu
- saturační testování

Strategie testování

Testování zdola-nahoru (*bottom-up testing*)

- testují se komponenty na nižší úrovni, poté se integrují do komponenty vyšší úrovně a znovu testují
- vhodné, pokud většina modulů stejné úrovně je připravena

Testování shora-dolů (*top-down testing*)

- testují se integrované moduly nejvyšší úrovně, poté se testují submoduly
- problém s připraveností všech modulů (simulace modulů na nižších úrovních)

Sendvičové testování (*sandwich testing*)

- kombinace strategií bottom-up a top-down testování
- moduly se rozdělí do dvou skupin
  - *logické*: řízení a rozhodování, top-down
  - *funkční*: vykonávání požadovaných funkcí, bottom-up

Strategie testování

Jednofázové testování (*big-bang testing*)

- moduly se otestují samostatně a poté se naráz integrují
- náročná identifikace místa chyby při integraci
- náročné rozlišení chyb v rozhraní modulů od ostatních chyb

Testování porovnáváním (*comparison testing, back-to-back testing*)

- více verzí systému na testování
  - prototyp
  - technika programování N-verzí $\Rightarrow$ vývoj vysoce spolehlivých systémů
  - vývoj více verzí produktu pro různé platformy
- stejné výsledky značí, že verze *pravděpodobně* pracují správně
- problémy:
  - stejné chyby ve verzích
  - nevyhovující specifikace

Testování produktu

Různé způsoby (účely) testování:

- testování funkčnosti celého systému
- testování robustnosti celého systému
- kritické testování (*stress testing*) – testování hraničních podmínek
- testování objemu dat (*volume testing*)
- regresní testování (ověření, že fungují dříve vytvořené funkce systému)
- testování splnění mezí
  - doba odezvy
  - paměťové nároky
  - bezpečnost (jak drahé je proniknutí do systému, . . .)
- testování zotavení (chybný vstup, odpojení napájení, . . .)
- testování dokumentace (dodržení standardů, aktuálnost, použitelnost)

Testování produktu

Způsob testování

- automatické
- ruční

Testovací scénář

- postup testování vlastnosti komponenty
- skript/tester postupuje podle scénáře
- během vývoje se scénáře mohou modifikovat (rozšiřovat)

Zpráva o chybách

- shrnuje, jaké testy (scénáře) byly provedeny, s jakými daty a s jakým výsledkem
- je nutné přesně specifikovat vyvolanou chybu (nějak to špatně počítá . . .)
- je nutné zaznamenat postup vyvolání chyby

Podpora testování

Statická analýza

- analýza programu bez spuštění
- Snaží se najít časté programátorské chyby:
    - syntaktické chyby
    - nedosažitelné části programu
    - neinicializované proměnné
    - nevyužití hodnoty po jejím přiřazení do proměnné
    - odkaz přes NULL ukazatel
    - použití paměti po jejím uvolnění
    - opakované uzavření souboru
    - dělení nulou
    - uváznutí (deadlock)
    - časově závislé chyby (race condition)
    - ...
- Obvykle hlásí řadu falešných chyb.

Podpora testování

Dynamická analýza

- analýza při běhu testovaného programu
- Může detekovat některé chyby:
  - nesprávná práce s dynamickou pamětí (např. Valgrind)
  - uváznutí (deadlock)
  - časově závislé chyby (race condition)
  - ...
  - Obvykle hlásí méně falešných chyb než statická analýza.
- Profiling – zjišťuje např. využití paměti nebo počet vyvolání a čas strávený v jednotlivých funkcích (užitečné pro optimalizace)
- analýza pokrytí

Akceptační testování

- Testuje se na reálných datech.
- Testuje se u uživatele.
- Uživatel určí, zda produkt splňuje zadání.
- Další změny *po* akceptaci systému již představují *údržbu systému*.
- Vztahuje se na zakázkový software.

Alfa a Beta testování

...pro generické softwarové výrobky, kde není možné provést akceptační testy u každého zákazníka (operační systémy, kompilátory, ...)

Alfa testování

- tam, kde se vyvíjí software
- testuje uživatel, vývojáři sledují a evidují chyby
- známé prostředí

Beta testování

- testují uživatelé u sebe
- neznámé prostředí
- výsledkem je zpráva uživatele ⇒ modifikace softwaru ⇒ předání softwaru k používání

Související předměty na FIT

- IPP – Principy programovacích jazyků a OOP (D. Kolář)
- IMS – Modelování a simulace (P. Perginer)

- IVS – Praktické aspekty vývoje software (J. Dytrych)
- ITS – Testování a dynamická analýza (A. Smrčka)
- IAN – Analýza binárního kódu (RedHat)

- ATA – Automatizované testování a dynamická analýza (A. Smrčka)
- SAV – Statická analýza a verifikace (T. Vojnar)
- MBA – Analýza systémů založená na modelech (A. Rogalewicz)

Variantní termíny zkoušky

Tři pevně stanovené termíny

- nekolizně centrálně naplánované termíny s *neomezenou* kapacitou
- jednoduché (není nad čím přemýšlet), ale neflexibilní

Variantní termíny – v IUS letos budou

- 5 termínů, ale s omezenou kapacitou
- celková kapacita min. 1,5 násobek zapsaných studentů
- přípustné kolize s jinými zkouškami
- výsledky mohou být známy až po dalším termínu
- je-li kapacita posledního termínu vyčerpána, nemá student nárok na opravnou zkoušku

V Moodle > Studijní materiály najdete zadání ER diagramů ze zkoušek

- z ak. r. 2011/12 včetně vzorových řešení
- z ak. r. 2015/16
- z ak. r. 2017/18 včetně komentovaných vzorových řešení

Studijní koutek – prospěchová stipendia

- vyšší (nižší) stipendium pro 2 % (5 %) nejlepších studentů v ročníku
- vyšší je 3× vyšší než nižší, nárůst s vyšším ročníkem

FIT – za zimní semestr 2023/24:

Ročník	SP ≤	Částka	SP ≤	Částka
1BIT	1,00	18 180 Kč	1,16	6 060 Kč
2BIT	1,24	23 424 Kč	1,53	7 747 Kč
3BIT	1,28	25 037 Kč	1,52	8 346 Kč
1MITAI	1,10	32 778 Kč	1,39	10 926 Kč
2MITAI	1,00	22 305 Kč	1,08	7 435 Kč

FIT – za letní semestr 2023/24:

Ročník	SP ≤	Částka	SP ≤	Částka
1BIT	1,08	21 001 Kč	1,25	7 000 Kč
2BIT	1,26	21 701 Kč	1,43	7 234 Kč
1MITAI	1,00	25 995 Kč	1,19	8 665 Kč

Studijní koutek – další stipendia

Mimořádná stipendia

- za ukončení studia (červený diplom, cena děkana za BP/DP)
- za zapojení do vědeckých projektů fakulty
- jako podpora zahraničních výjezdů (Erasmus+)
- za pomoc fakultě při organizaci různých akcí (např. Gaudeamus, DoD)
- za reprezentaci fakulty v odborných soutěžích

Ubytovací stipendia

- Stipendijní řád VUT
- Směrnice rektora č. 71/2017 Ubytovací a sociální stipendium
- pro studenty s trvalým bydlištěm mimo Brno-město a Brno-venkov
- aktuální výše (Rozhodnutí rektora č. 7/2017, příloha č. 1 z 2. 9. 2024): 560 Kč měsíčně

Sociální stipendia

- pro studenty z rodin s nejvýše 1,5 násobkem životního minima
- $1/4$ základní sazby minimální mzdy: 4 730 Kč měsíčně (1. 1. 2024)

Motivace k dobrým studijním výsledkům

Variabilní kreditové limity pro akademický rok

- **60 kreditů** nominální roční zátěž
- **65 kreditů** základ pro všechny studenty
- **70 kreditů** při absolvování všech zapsaných předmětů (připouští se jeden neúspěšný P či PV předmět)
- **75 kreditů** a při dosažení studijního průměru do 2,00
- **80 kreditů** a při dosažení studijního průměru do 1,50

Variabilní kreditové limity pro celé studium

- **180 kreditů** nominální zátěž
- **184 kreditů** pro všechny studenty
- **189 kreditů** nejvýše dva neúspěšné předměty
- **194 kreditů** žádný neúspěšný předmět a průměr do 2,00

Přednost při registraci do volitelných předmětů

Prominutí přijímací zkoušky do navazujícího magisterského studia na FIT