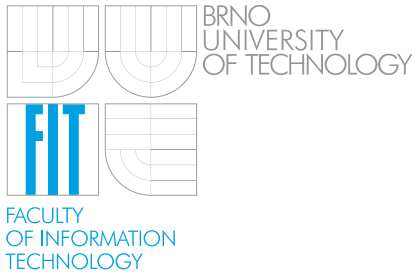
Praktická aplikace unixové filosofie při programování

Ústav počítačové grafiky a multimédií
Fakulta informačních technologií
Vysoké učení technické v Brně



Úvod

- Unix je jednoduchý, ale pouze génius dokáže porozumět jeho jednoduchosti. $\sim$ Dennis Ritchie

Motivace

- Unix vznikl v roce 1969 a od té doby je stále v provozu. Počítače se systémem Unix mají za sebou již více než stovky tisíc provozních hodin.
- Programátoři v systému Unix mají za sebou desítky let zkušeností, to co museli v historii řešit, my dnes pokládáme za samozřejmé.
- Správné porozumění unixové tradici vám pomůže tvořit lepší software dokonce i když neprogramujete v Unixu.

Unixová filosofie v kostce

- Navrhujte programy od spoda nahoru a snažte se, aby byly co nejjednodušší
- Při analýze problému používejte přístupu *rozděl a panuj*
  - Vytvářejte jednoduché nástroje, které vzájemnou spoluprací dokáží plnit složité úlohy
- Pište programy, které spoléhají na inteligenci uživatele
  - Programy by měly dělat to, co po nich uživatelé chtějí, tiše a bez překážení uživateli (neobtěžujte uživatele hláškami typu *"Jste si opravdu jistý? (A|N)"*)
  - Založeno na předpokladu, že uživatel ví co chce a jak toho dosáhnout (Unix věří uživateli)
  - S velkou mocí přichází velká zodpovědnost (pokud uživatel neví co dělá, může to mít fatální následky)

Několik dobrých osvědčených rad

- Návrh algoritmů
  - Rekurzivně rozdělujte problém na menší podproblémy, dokud nejsou tak jednoduché, že je lze řešit přímo
- Funkcionální programování
  - Pište malé funkce, které se snaží řešit jeden specifický problém
  - Zapomeňte na komplikované vnořené bloky, hlídejte velikost každého modulu, pokud je příliš velký, rozdělte ho na několik podmodulů
- Objektově orientované programování
  - Je snadnější rozumět chování velkých rozsáhlých systémů, když jsou popsány jako navzájem komunikující objekty
  - Objekt by měl zastřešovat právě jednu entitu a měl by ji zastřešovat úplně; metoda by měla dělat právě jednu věc a měla by ji dělat úplně

Pravidla unixové filosofie podle Douga McIlorye

1. Tvořte programy, které dělají pouze jednu věc a dělají ji dobře.
   - Pokud potřebujete zpracovat novou úlohu, raději vytvořte nový program, než abyste komplikovali starý přidáváním nových vlastností.
2. Počítejte s tím, že výstup každého programu může být použit jako vstup jiného programu.
   - Nezatěžujte výstup programu zbytečnými informacemi.
   - Vyhýbejte se binárním formátům.
   - Nikdy netrvejte na interaktivním vstupu.

3. Navrhujte software (i OS) tak, aby mohl být vyzkoušen co nejdříve, ideálně během několika týdnů.
   - Nerozpakujte se zahodit špatné části kódu a vytvořit je znovu a lépe.
4. Používejte přednostně takové nástroje, které vám programátorskou činnost usnadní.
   - To platí i v případech, kdy budete muset některé nástroje vytvořit a po dokončení práce je zahodit.

Pravidla unixové filosofie podle Roba Pika

1. Nikdy nemůžete předem vědět, kde bude váš program trávit nejvíce času.
   - Výpočetně náročnější místa programu se vyskytují vždy tam, kde je nejméně čekáte. Nesnažte se proto tato místa odhadnout a optimalizovat dokud se dostatečně neprokáže, že se jedná o daná místa.
2. Před optimalizací vždy měřte spotřebovaný strojový čas.
   - Neoptimalizujte dokud jste neměřil spotřebovaný strojový čas. Optimalizovat se vyplatí jen v případě, že spotřebovaný strojový čas byl změřen a vyplynulo z toho, že jedna část programu spotřebovává znatelně více času než ostatní.
3. Efektivní algoritmy jsou pomalé, pokud `n` je malé a `n` bývá obvykle malé.
   - Efektivní algoritmy mívají velké konstanty.
   - Buďte efektivní pouze v případě kdy víte, že `n` bude velké. (Dokonce i když bude `n` velké, použijte pravidlo 2)

Pravidla unixové filosofie podle Roba Pika

4. Efektivní algoritmy obsahují více chyb než jednoduché algoritmy. Kromě toho je složité implementovat efektivní algoritmy.
   - Používejte jednoduché algoritmy a jednoduché datové struktury.
5. Data vládnou.
   - Pokud zvolíte vhodné datové struktury a správně je použijete, algoritmy budou vždy naprosto jasné.
   - Ústředním bodem při programování jsou vždy data, ne algoritmy.
6. 6. pravidlo neexistuje!

Pravidlo modularity

Pište jednoduché součásti propojené přes bezchybná rozhraní.

- Vrátí správný počet slov latexového dokumentu rozděleného do více souborů

  ```
  detex *.tex | wc -w
  ```

- Vrátí seznam velkých souborů v SVN repozitáři

  ```
  svn status | awk 'print $2' | xargs du | sort -n
  | tail
  ```

- Nalezne slovo vyskytující se ve slovníku po slovu "trivia", přičemž se řadí ne od začátku slova, ale od konce

  ```
  cat /usr/share/dict/british-english | rev | sort
  | rev | grep -A 1 '^trivia$' | tail -1
  ```

Pravidlo srozumitelnosti

Srozumitelnost je lepší než chytrost.

správně (C)

```
int Sum = 0;
for (int i = 1; i < N; ++i)
{
    Sum = Sum ^ i;
}
```

špatně (C)

```
int Sum = (N & (N % 2 ? 0 : ~0) |
(((N & 2)>>1) ^ (N & 1)));
```

Pravidlo kompozice

Navrhujte programy tak, aby je bylo možné připojit k dalším programům.

správně (python)

```
import sys

for row in sys.stdin:
   print row.lower()
```

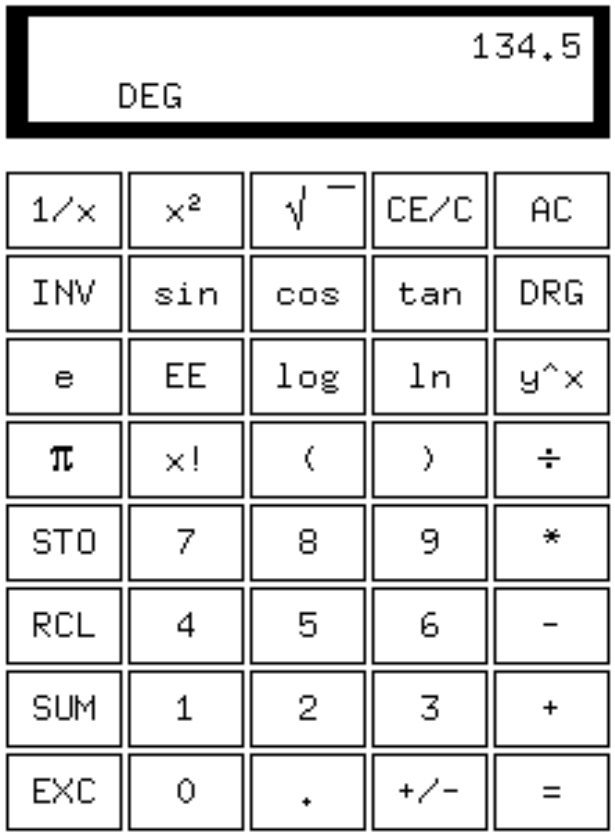
špatně (python)

```
file = open("input.txt", "r")
for row in file:
   print row.lower()
file.close()
```

Pravidlo oddělení

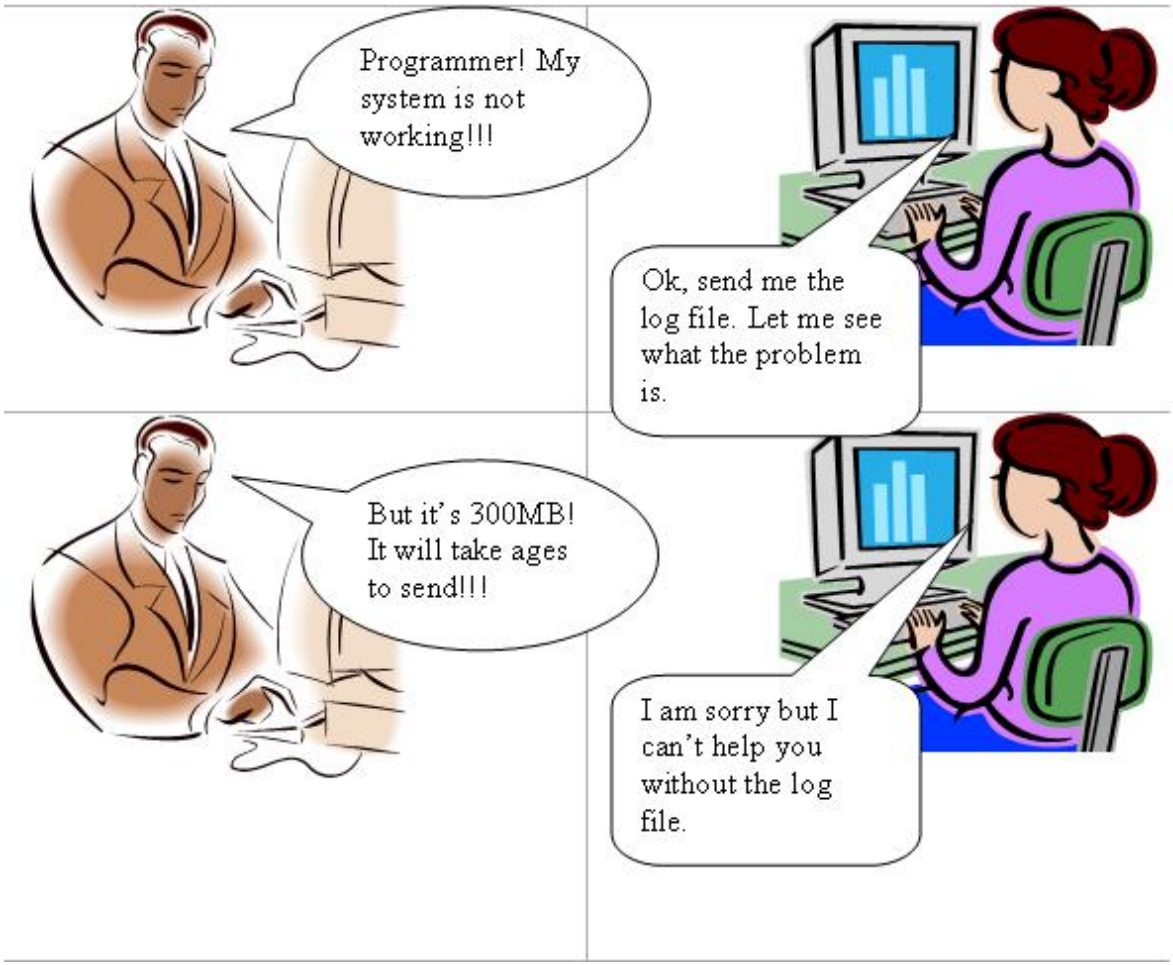
Oddělujte politiku od mechanismu, rozhraní od toho, co je za ním.

```
$ echo "56.8 + 77.7" | bc
$ 134.5
```



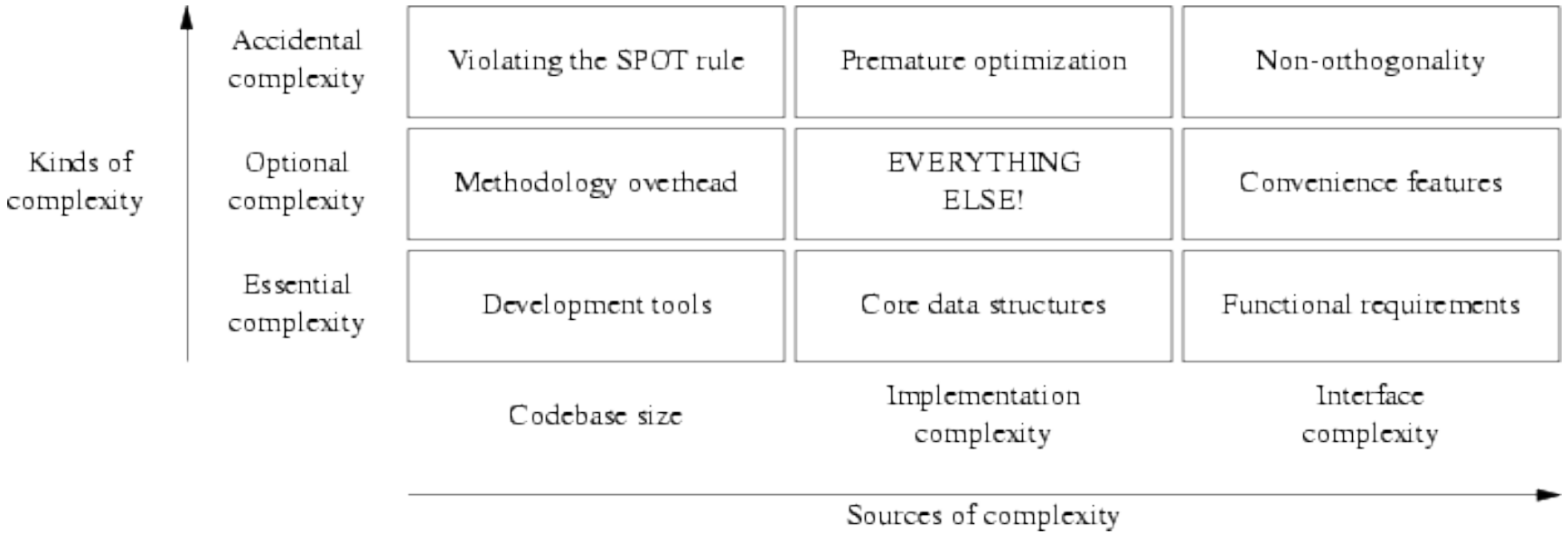
Pravidlo jednoduchosti

Navrhujte jednoduché programy, složitější pište pouze, není-li jiného východiska.



Pravidlo šetrnosti

Velké programy pište pouze v případě, když odzkoušíte, že nic jiného nefunguje.



Pravidlo transparentnosti

Navrhujte programy tak, aby bylo co nejsnazší jejich ladění a kontrola.

správně (python)

```
deckSize = 52
for i in range(deckSize)
    j = i+random.randInt(deckSize-1-i)
    a.swapEntries(i, j)
```

špatně (python)

```
for i in range(52):
    j = i+random.randInt(51-i)
    a.swapEntries(i, j)
```

Pravidlo robustnosti

Robustnost je dítětem transparentnosti a jednoduchosti.

assert (C)

```
int mySqrt (int x)
{
    /* make sure it's positive! */
    assert(x >= 0);

    return x*x;
}
```

kontrola vstupu (python)

```
while True:
    num = raw_input("Enter an integer:")
    if num.isdigit():
        num = int(num)
        break
    print "Try it again."
```

Pravidlo reprezentace

Zahalte znalosti do dat tak, aby logika programu mohla být hloupá a robustní.

správně (python)

```
messages = {
   0:"you typed zero",
   1:"n is a perfect square",
   2:"n is an even number",
   3:"n is a prime number",
   4:"n is a perfect square",
   5:"n is a prime number",
   6:"n is an even number",
   7:"n is a prime number",
   8:"n is an even number",
   9:"n is a perfect square"}
print messages[n]
```

špatně (python)

```
if n == 0:
   print "you typed zero"
elif n == 1 or n == 9 or n == 4:
   print "n is a perfect square"
elif n == 2 or n == 6 or n == 8:
   print "n is an even number"
elif n == 3 or n == 5 or n == 7:
   print "n is a prime number"
```

Pravidlo nejmenšího překvapení

Při návrhu rozhraní se snažte, aby dělalo vždy co nejméně překvapující věci.

správně (C)

```
int multiply(int a, int b)
{
   return a * b;
}
```

špatně (C)

```
int multiply(int a, int b)
{
   return a + b;
}
```

Pravidlo mlčenlivosti

Nemá-li program co překvapivého říci, neměl by říkat nic.

správně

```
$ ls empty_dir
$
```

špatně

```
$ ls empty_dir
$ Vítejte v programu ls. Vámi požadovaný
adresář je prázdný, chcete vypsat obsah
jiného adresáře? [Y|N]: N
$
```

Pravidlo opravy

Pokud se program musí zhroutit, ať to udělá hlasitě a co nejdříve.

správně

```
$ wget www.google.com
$ Resolving www.google.com...
failed: Name or service not known.
$ wget: unable to resolve host
address ‘www.google.com’
```

špatně

```
$ wget www.google.com
$ Resolving www.google.com...
failed: Name or service not known.
$ Trying again.
$ Resolving www.google.com...
failed: Name or service not known.
$ Trying again.
$ Resolving www.google.com...
failed: Name or service not known.
$ Trying again.
$ Resolving www.google.com...
failed: Name or service not known.
```

Pravidlo hospodárnosti

Programátorův čas je drahý, šetřete svůj čas na úkor času počítače.

program v C

```
int wordCount(char *line)
{
     int count = 0;
     while(isspace(*line)) line++;

     int len = strlen(line) ;
     bool onWord = true;
     for(int i = 0; i < len; i++)
     {
          if(isspace(line[i]))
          {
               if(onWord)
                    count++;
               onWord = false;
          }
          else
          {
               onWord = true;
          }
     }

     if( onWord )
     count++ ;

     return count;
}
```

program v pythonu

```
def wordCount(line):
     words = line.strip().split()
     return len(words)
```

Pravidlo generování

Vyhýbejte se ručním improvizacím při psaní kódu. Můžete-li, pište programy, které budou psát další programy.

tisk tabulky pro Latex

```
print "\\begin{table}[ht]"
print "\\begin{center}"
print "\\begin{tabular}|l|"+len(sentences)*"c"+"|"
print "\\hline"
header = ""
for i in range(len(sentences)):
    header = header+" & {\\bfseries $c_"+str(i+1)+"$}"
    print header+"\\\\"
    print "\\hline"
    print "\\hline"
    for word in order:
        results = ""
        for i in range(len(sentences)):
                results = results+" & "+str(syntagmatic[word][i+1])
        print word+results+" \\"
        print "\\hline"
    print "\\end{tabular}"
    print "\\caption{Table for syntagmatic use of context.}"
    print "\\label{tab_syntagmatic}"
    print "\\end{center}"
    print "\\end{table}"
```

Pravidlo optimalizace

Nejprve vytvořte prototyp, zprovozněte jej a pak se teprve pusťte do jeho optimalizace.

správně

```
float f[100],sum;

...

sum = 0;
for(i=0;i<100;i++) {
    sum += f[i];
}
```

špatně

```
float f[100],sum0,sum1;
float sum2,sum3,sum;

...

sum0 = sum1 = sum2 = sum3 = 0;
for(i=0;i<100;i+=4) {
    sum0 += f[i];
    sum1 += f[i+1];
    sum2 += f[i+2];
    sum3 += f[i+3];
sum = (sum0+sum1) + (sum2+sum3);
}
```

Pravidlo rozmanitosti

Nedůvěřujte všem tvrzením typu „existuje jediný správný způsob“.

Pravidlo rozšiřitelnosti

Navrhujte pro budoucnost, protože ta nastane dříve, než si myslíte.

Unixové nástroje

- Všechny nástroje příkazové řádky v Unixu mají tři hlavní komponenty:
  - Parametry příkazové řádky
  - Standardní vstup (`stdin`)
  - Standardní výstup (`stdout`)
- Defaultně je `stdin` bráno z terminálu a `stdout` vypisováno do terminálu. `stdin` a `stdout` lze přesměrovat pomocí operátorů přesměrování:
  - < přesměruje soubor do `stdin`
  - > přesměruje `stdout` do souboru
  - >> přesměruje `stdout` do souboru (přidá obsah, nepřepíše jej)
- Programy, které čtou vstupní data ze `stdin`, modifikují je a následně zapisují na `stdout`, se nazývají *filtry*.

Příklady filtrů

- `cat` - nejjednodušší filtr, pouze zkopíruje `stdin` na `stdout`
- `head` - vypíše pouze několik prvních řádků textu
- `tail` - vypíše pouze několik posledních řádků textu
- `tac` - stejné jako `cat`, ale řádky vypisuje odzadu
- `tee` - zkopíruje `stdin` na `stdout` a zároveň jej zapíše do souboru definovaném jako parametr
- `tr` - nahrazuje znaky definované znaky ze `stdin` jinými znaky
- `grep` - filtruje data ze `stdin` pomocí regulárních výrazů
- `cut` - vypisuje pouze definované sloupce textu ze `stdin`
- `wc` - spočítá počet řádků, slov a znaků ze `stdin`
- `sort` - seřadí řádky textu
- `uniq` - odstraní duplicity
- `sed` - filtrování textu, nahrazování textu atd.
- `awk` - programovatelný filtr

Některé další užitečné nástroje

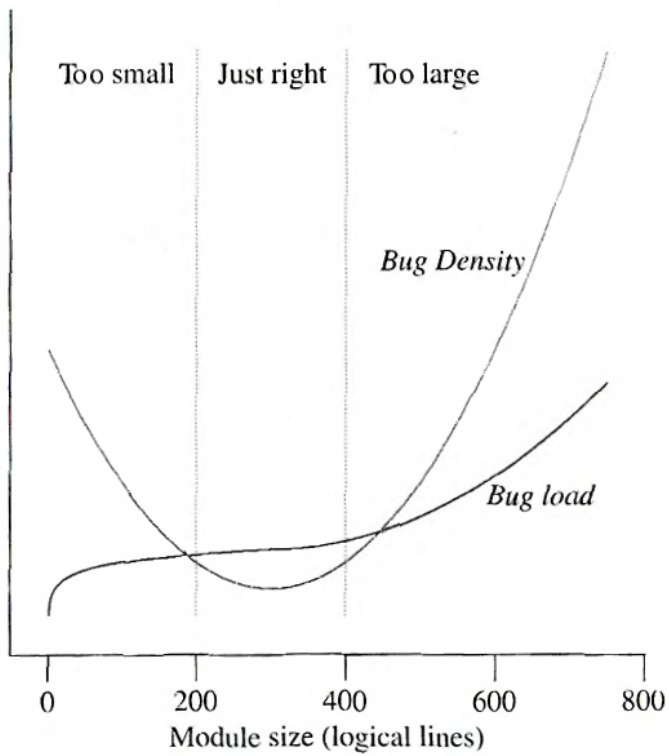
- `echo` - vypíše text v argumentu na `stdout`
- `find` - najde umístění souborů odpovídajících danému vzoru
- `xargs` - vloží data ze `stdin` jako argument nějakému příkazu
- `paste` - spojí data ze dvou souborů do jednoho
- `man, info` - získá nápovědu o daném příkazu

- přesměrovávají `stdout` jednoho programu na `stdin` jiného programu, přičemž oba programy běží ve stejný čas
- rychlejší než předávání dat přes soubory, navíc elegantně řeší problém s přenosem dat při multi-taskingu
- je zajištěna synchronizace a bufferování dat – nedojde ke ztrátě dat
- několik příkladů použití roury:

```
$ cat *.txt | sort | uniq > output.txt
$ cat /usr/share/dict/words | head -23 | tail -1
```

Zásady při tvorbě modulů

- správně zapouzdřené moduly nevolají cizí kód a nedělí se o globální data
- navrhujte jasné a výstižné API pro vaše moduly (návrh předchází vlastní implementaci)
- velikost modulu hraje velkou roli, *Hattonova křivka* říká, že velmi malé a velmi velké moduly obsahují výrazně více chyb než moduly středně velké



Kompaktnost a ortogonalita

Kompaktnost

- kompaktnost návrhu znamená, že jej běžný člověk dokáže snadno pochopit
- pokud i zkušený uživatel potřebuje k ovládání programu běžně manuál, návrh není kompaktní
- příklady nekompaktních nástrojů a jazyků: `autoconf`, `automake`, `Perl`, `Java` (jazyk `C++` je dokonce antikompaktní)

Ortogonalita

- vykonávané operace nemají žádné vedlejší efekty
- každou vlastnost systému lze změnit výhradně jedním příkazem
- ortogonalita zkracuje dobu testování a vývoje softwaru

Pravidlo jediné pravdy

- každá část znalostí v programu musí mít jednu jednoznačnou a spolehlivou podobu
- opakování vede k nekonzistencím a opakovaný kód lze velmi snadno zničit
- konstanty tabulky a metadata by měla být deklarována a inicializována pouze jednou
- kdykoliv spatříte duplicitní kód, je to známka nebezpečí
- *refactoring* je proces zabývající se vnitřním přeuspořádáním kódu, aniž by se změnila jeho funkčnost, mimo jiné zahrnuje i odstraňování duplicit

Jak je důležité používat textový formát

- textový formát mohou snadno číst a upravovat lidé, aniž by k tomu potřebovali speciální nástroje
- textové proudy pomáhají vynucovat zapouzdření
- binární formáty zpravidla omezují maximální velikost ukládaných dat
- binární formáty používejte jen tam, kde není jiného vyhnutí (snaha maximálně využít kapacitu daného média, čas k překladu na textovou reprezentaci dat je neúnosný)

Jak vytvářet transparentní a přístupný kód?

- Jaká je maximální hloubka hierarchie zanoření procedur?
- Jsou volání funkcí ve vašem API ortogonální?
- Obsahuje váš kód vlastnosti, jež jsou nejen silné, ale také viditelné?
- Máte k dispozici několik datových struktur nebo jen jednu globální tabulku výsledků?
- Lze snadno najít část kódu odpovědnou za určitou funkci?
- Existuje jasné mapování mezi vytvořenými objekty a entitami v reálném světě?
- Vyskytují se ve vašem kódu speciální případy?
- Kolik magických čísel obsahuje váš kód?

Konfigurace programu

- nikdy nenabízejte konfigurační přepínače proto, co může být spolehlivým způsobem zjistitelné automaticky (např. autodetekce formátu vstupního souboru)
- uživatelé by neměli mít přístup k optimalizačním přepínačům
- nesnažte se přidat přepínačem to, čehož by šlo dosáhnout zapouzdřením programu do skriptu nebo jeho začleněním do zřetězení (např. program `ls` neobsahuje přepínač pro stránkování)
- přemnožení zbytečných přepínačů může mít pro program negativní výsledky, mimo jiné např. problémy při testování všech možných variant chování

Použití přepínačů

Původní unixový styl

- používá jednopísmenové přepínače uvozené spojovníkem `-a`
- jde–li o přepínače změny režimu `-a`, `-b`, umožněte jejich slučování `-ab` nebo `-ba`
- má–li přepínač argumenty, následují za přepínačem (volitelně za mezerou) `-a5` (`-a 5`)
- chcete-li jako přepínač použít velké písmeno, mělo by jít o variantu přepínače se stejným malým písmenem

GNU styl

- jako přepínače používá celé slova uvozená dvěma spojovníky, `--help`
- přepínače GNU nelze spojovat
- argument přepínače lze od samotného přepínače oddělit mezerou nebo znakem = (`--max=10`)

Použití přepínačů 2

- některé nástroje (např. `-X window`) používají nestandardní přepínače ve tvaru `-geometry`, tyto přepínače slouží jako filtry, které upravují příkazový řádek
- mnohé nástroje používají také pouhý spojovník `-` jako speciální přepínač, který přesměruje vstup aplikace ze `stdin`
- argument `--` pak značí pozastavení interpretace argumentů, veškeré argumenty za tímto symbolem budou brány jako doslovné hodnoty
- používejte zažité hodnoty přepínačů pro různé funkce: `-a` (all), `-f` (file), `-o` (output), `-v` (verbose), ...
- používejte knihovny pro parsování přepínačů (např. `getopt` pro Python)

Opětovné použití kódu

- čas programátora je drahý a měl by být využíván pro řešení nových problémů, ne těch již vyřešených
- pište transparentní a dobře komentovaný kód, aby bylo snadné jej použít a v případě potřeby i upravit
- nejlepším příkladem opětovného použití kódu představuje *Open Source*

Užitečné techniky pojmenování distribučních souborů

1. předpona projektu
2. spojovník
3. číslo verze
4. tečka
5. src nebo bin (nepovinné)
6. tečka nebo spojovník (upřednostňuje se tečka)
7. binární typ a možnosti (nepovinné)
8. přípona archivu nebo kompresního formátu

```
battalion-1.2.src.tgz
grep-2.0.bin.ELF.tar.gz
linberto-1.0.5-1.i386.rpm
```

Děkuji za pozornost!

Použitá literatura


[Raymond, 2003] Raymond, E.S.
The art of unix programming, *ISBN-0131429019*, *Pearson Education*


[Balzarotti] Davide Balzarotti
Software Development The UNIX Philosophy, *Eurecom – Sophia Antipolis, France*


[Cantin, 2010] Claude Cantin
Basic Introduction to UNIX/linux, *2010*


[Lande] Joshua Lande
Unix Tutorial, *2010*