Aplikace

Aplikace se liší tím, k jakému účelu jsou tvořeny. To má vliv na to, jakou mají strukturu i na to, jak pracné je je vyvinout.

Bylo vypozorováno, že aplikace je možné rozdělit do skupin s podobnou náročností a s nějakým charakterizujícím prvkem.

Skupiny obtížnosti tvorby softwaru

Dávkové systémy – systémy (skupiny programů) spolupracujících na složitějších úkolech. Určeno pro zpracování velkých objemů dat se stejnou strukturou. Programy většinou malé, čtou vstupní data záznam po záznamu z několika málo souborů, zapisují nebo tisknou opět několik málo souborů. Programy mají typicky jednu funkci, ale dobře a efektivně zpracovanou.

Vědecko-technické výpočty – opět aplikace na transformaci dat, tentoktát výpočetně náročnější (složitější algoritmy, mnohdy se data procházejí mnohokrát). Stále ještě jednodchá struktura programu.

Skupiny obtížnosti tvorby softwaru (2)

Interaktivní aplikace – systémy (aplikace, programy) určené pro interaktivní spolupráci s uživatelem. Je třeba řádně ošetřit vstupy, uživatel může odbíhat od jedné aktivitě ke druhé, prostředí by mělo být intuitivní.

Distribuované systémy – systémy (skupiny aplikací) zpravidla pracujících na různých počítačích, ale fungujících jako jeden větší celek. Jednotlivé části mohou být schopny práce (byť třeba nějak omezené), i když jié části nejsou dostupné.

Skupiny obtížnosti tvorby softwaru (3)

Systémy pracující přibližně v reálném čase (soft real time) – systémy (aplikace, programy), u nichž je třeba, aby typicky reagovaly na podněty v nějakém poměrně krátkém čase. Často jsou to programy řídící najká nekritická zařízení, či některé interaktivní distribuované informační systémy.

Systémy pracující v reálném čase (hard real time) – systémy (aplikace, programy), u nichž je požadována reakce do určité (zpravidla velmi krátké) doby, jinak hrozí poškození zařízení či ještě větší ztráty.

Je-li úloha kritická (v případě selhání jde o životy nebo obrovské ekonomické ztráty), její tvorba je opět pracnější (více dokazování i testování).

Obtížnost tvorby softwaru

- Přechod od jedné skupiny aplikací ke skupině následující znamená násobné zvýšení pracnosti průměrné řádky kódu, zvýšení pravděpodobnosti, že se projekt nepovede dokončit.
- Přechod o více než jeden stupeň (vůči nejtěžší skupině, co jsme dosud dělali) je téměř žádost o neúspěch. (Aneb potřebujeme někoho, kdo už něco takového nebo skoro takového dělal, aby nám pomáhal, případně aby to vedl).

Co by se nám mohlo hodit

- ošetření vstupu z klávesnice
- trošku lepší zacházení s obrazovkou
- schopnost reagovat na podněty

Vstup z klávesnice

- Je třeba rozlišovat znaky a klávesy – systém dokáže přiřadit mnoha tlačítkům znaky, ale stisk některých kláves je reprezentován sekvencí znaků (například #0 + něco dalšího)
- Umíme rozpoznat, že byla stisknuta klávesa (pomocí funkce `KeyPressed`). Je však třeba ošetřit, zda to nebyla klávesa posílající sekvenci znaků (a pak je třeba zpracovat všechny znaky patřící dané klávese) – zajistit příslušný počet volání funkce `ReadKey`

- Mapování kláves na znaky se liší dle systému a nastavení.

Vstup z klávesnice (2)

```
USES CRT;
FUNCTION ReadKey:Char;
FUNCTION KeyPressed:Boolean;
```

`http://freepascal.org/docs-html/rtl/crt/`

Najdeme tam i prostředky na ovládání kursoru a vlastností textu.

Vstup z klávesnice – Proč?

- Pro některé aplikace (např. výukové programy a hry) se nám může hodit kontrola nad tím, co a kam uživatel píše
- Někdy se může hodit, když můžeme po určité době uživatelovy nečinnosti nějak reagovat (například zobrazit nápovědu).

Program řízený událostmi

- Některé aplikace přijímají podněty z různých zdrojů – nelze pak čekat, než se na jednom z nich něco stane (zatím by mohlo být třeba řešit nějaký podnět z jiného zdroje).
- Takovéto aplikace pak mají jinou strukturu: Jádrem je ošetřování nastálých událostí
- Ošetření události je také událost (ať už pozitivní nebo negativní ošetření) a může tak spouštět další akce (ošetření těchto událostí).

Program řízený událostmi (2)

```
CASE Co_se_stalo OF
  probuzeni_systemu: ...
  stisknuta_klavesa: ...
  utekla_chvilka: ...
  ...
  END;
```

Program s parametry

Někdy může být výhodné, kdybychom mohli aplikaci při jejím spuštění sdělit, s jakými daty má pracovat, případně jinak ovlivnit její chod. Hodilo by se nám, kdybychom měli přístup k údajům, co jsou na příkazové řádce.

Počet parametrů zjistíme zavoláním funkce `ParamCount`, k i-tému parametru se dostaneme pomocí `ParamStr(i)`.

Efektivní ukládání dat

Někdy pracujeme pouze s klíčem (identifikátorem) reprezentujícím objekt reálného světa, jindy potřebujeme i další údaje o tomto objektu.

Hodilo by se nám mapování klíč→data, které by bylo rychlé a nevyžadovalo zbytečně mnoho paměti.

Efektivní ukládání dat – přímý přístup

Je-li objektů dostatečně mnoho ve srovnání s velikostí domény klíče (množiny všech možných klíčů), je výhodné použít klíč přímo jako odkaz na místo, kde se data nacházejí. Data tak máme velmi rychle dostupná (jeden přístup do pole).

Pozor! Pro velké domény a malá data je to prostorově velmi neefektivní!

Efektivní ukládání dat – tabulka s hodnotami

Mám-li velmi málo místa, mohu záznamy uložit do tabulky. Tu pak musím prohledávat vždy, když ke klíči potřebuji patřičný záznam. Nesetříděná tabulka vyžaduje sekvenční prohledávání (vhodné jen pro velmi málo dat), případně další pomocnou tabulku usnadňující vyhledání potřebného záznamu (index).

Setříděná tabulka, kde by se dalo hledat půlením intervalu, se hůře udržuje (její údržba může vyžadovat mnoho přesunů dat).

Toto řešení zabere málo místa, ale může být pro velká data nepříjemně pomalé.

Efektivní ukládání dat – hašování

Kompromisem je upravená varianta přímého přístupu: budeme jako adresu do tabulky využívat upravený klíč místo klíče samotného + ošetříme případné kolize.

Víme-li, že budeme potřebovat místo pro n záznamů, vytvoříme si tabulku pro $k \cdot n$ záznamů (nebo ještě lépe: pro prvočíslo blízké $k \cdot n$), kde k budeme volit například mezi 2 a 20. (Malé k zvyšuje pravděpodobnost, že budeme muset řešit kolize (zpomalení přístupu), velké k přináší neefektivní nakládání s pamětí.)

Hašování – hašovací funkce

- Transformuje je klíč na odkaz do tabulky
- Měla by množinu očekávaných klíčů rozdělovat na množinu odkazů do tabulky rovnoměrně

  Pozor na domény typu rodné číslo nebo datum narození
- Situaci, kdy hašovací funkce vrátí pro různé klíče stejný výsledek nazýváme *kolize*. Je třeba je ošetřit (například hledáním správného záznamu na dalších políčkách tabulky).

Hašování – kolize

- Situace, kdy hašovací funkce pro různé klíče vrací tutéž hodnotu
- Možné řešení: kolidující záznam uložíme na první volné místo za místem doporučeným, je-li takové); při hledání hledáme za doporučeným místem do nalezení záznamu, volného místa, nebo návratu na původní pozici; dojde-me-li na konec tabulky, pokračujeme od jejího začátku
- Důsledek: vždy musíme kontrolovat, zda máme ten záznam, co jsme hledali, nebo nějaký jiný (kolizní).