Co je to AppKit (16.)

V minulém dílu našeho volného seriálu jsme si ukázali trochu rozsáhlejší příklad, jak pomocí standardní programátorské aplikace InterfaceBuilder sestavit uživatelské rozhraní aplikace. Přitom jsme samozřejmě "implicitně" používali řadu standardních tříd, reprezentujících prvky grafického uživatelského rozhraní: okna, menu, tlačítka...

Dnes a v několika následujících pokračováních se na tyto třídy, jež jsou sdruženy ve společné knihovně jménem AppKit, podíváme trochu blíže a ukážeme si, jak se s nimi pracuje: naučíme se např. pracovat se schránkou, zobrazovat tabulky a seznámíme se s podporou drag&drop. Nebojte se ale žádných velkých a složitých programů; jak je v Cocoa zvykem, tohle všechno je a bude na pár řádků...

Přehled

Pro začátek je vhodné se jen velmi rychle, letem světem, seznámit s většinou tříd AppKitu - jen orientačně, abychom měli představu, co zde vůbec je k dispozici a jak se to jmenuje. Všechny třídy, které tvoří programátorské rozhraní AppKitu, vidíme na obrázku. Ačkoli jich není tak málo jako ve Foundation Kitu, pořád jde o velmi příjemně přehledný systém - aplikační knihovny, které nabízejí alespoň vzdáleně podobné služby jako AppKit a které jsou založeny na C++, typicky obsahují stovky a stovky tříd; zde jich máme pouze něco málo přes stovku pro pokrytí celé komplexní sady služeb od "zobrazení okna na obrazovceŽ až po "převod obrázku z TIFF na JPEGŽ nebo "zkopírování souboru z jednoho disku na druhý s využitím služeb a rozhraní standardního správce souborůŽ. AppKit samozřejmě zahrnuje i služby pro práci se zvuky, s videozáznamy nebo s třírozměrnou grafikou OpenGL... My si ani zdaleka nebudeme popisovat všechny třídy AppKitu. Ukážeme si jen ty nejdůležitější a nejzajímavější jako ilustraci toho, jak se v objektově orientovaném prostředí s grafickým uživatelským rozhraním pracuje pohodlně. V dnešním přehledu se nebudeme zmiňovat o třídách, jejichž jméno jasně napovídá, jaký je jejich smysl: že třeba NSWindow reprezentuje okno nebo NSFont písmo, je asi každému jasné; případné podrobnější údaje o důležitějších třídách si ukážeme později v samostatných dílech.

Základní grafika

Skupina tříd označená Graphics nabízí služby pro kreslení na obrazovku. NSImage je například libovolný obrázek, přičemž NSImageRep (...representation) odpovídá jedné z jeho reprezentací. Jeden a tentýž obrázek může být třeba reprezentován vektorově jako EPS a zároveň jako několik bitových map TIFF v různých rozlišeních. Za běhu pak aplikace programově nebo automaticky vybere tu nejvhodnější reprezentaci. Zajímavější jsou třídy NSBezierPath a NSAffineTransform. Místo mnoha proprietárních grafických služeb, s jakými se setkáváme ve starších grafických systémech, nabízí Cocoa plnohodnotný grafický systém postscriptového typu - jakoukoli vektorovou grafiku snadno a pohodlně vyjádříme jako řadu Bézierových křivek s využitím zcela obecných maticových transformací. Do této skupiny patří i práce s videem nebo třírozměrnými objekty (OpenGL).

Co je to View

Zatímco služby, o nichž jsme se bavili v minulém odstavci, se starají o vlastní zobrazení dat, třída NSView a řada jejích podtříd reprezentují konkrétní místo na obrazovce (nebo lépe v okně), kde se data zobrazí, a způsob jejich zobrazení. NSView dokáže zajistit základní transformace. Samozřejmě se stará o ořezávání, dokáže svůj obsah vytisknout (a ještě v OpenStepu jej tato třída uměla i odfaxovat, což se společnost Apple z naprosto nepochopitelného důvodu rozhodla zrušit) a podobně. Díky spolupráci objektů třídy NSView je například možné doslova "naprogramovat scrollování bez programováníŽ - připravíme-li jakékoli vlastní View (jehož obsah kreslíme pomocí libovolných služeb z minulého odstavce), stačí jej prostě uzavřít do NSScrollView a už nemusíme dělat vůbec nic jiného. Obsah našeho "obrázkuŽ bude automaticky zobrazen v okénku s posuvníky, které budou korektně fungovat... Za samostatnou zmínku stojí také třída NSCell a třídy od ní odvozené: View je nesmírně výkonné a flexibilní, ovšem každá legrace něco stojí, a proto je také poměrně "těžkéŽ - zabere hodně paměti. Z toho důvodu je velmi šikovné, jestliže u složených objektů, jako je třeba tabulka, použijeme jen jediné View pro celý objekt, zatímco pro zobrazení jednotlivých prvků poslouží jen jiná, daleko "lehčíŽ třída, jež bude pro zobrazování využívat veškerý komfort nadřízeného View a sama bude obsahovat jen nepatrně více než například textový řetězec, který se má zobrazit na daném řádku v daném sloupci.

Správa událostí

V aplikačním programování říkáme "událostŽ tomu, na co aplikace nějak reaguje: stisknutí klávesy, pohyb myší, ale i třeba příchod nějakých dat na sériové rozhraní. Tradičně programátor musel jako jádro aplikace připravit cyklus, který pomocí služeb operačního systému zjišťoval, je-li k dispozici nějaká událost, která patří dané aplikaci, a podle toho se choval - asi nějak takto:

```
for (;;) { // event-driven programování v tradičním prostředí
struct Event evt;
if (get_next_event(&evt))
if (evt.type==EvtKeyboard) {
....
} else if (evt.type==EvtMouse) {
....
```

Vždy šlo o poměrně nepohodlnou a nepříjemnou práci, ve které se snadno udělala řada těžko odhalitelných chyb. V Cocoa je tomu zcela jinak: tento cyklus sice samozřejmě existuje, ale je skryt ve standardních knihovnách a programátor se jím vůbec nemusí zabývat. Události se automaticky převádějí na standardní zprávy Objective C; ty jsou odesílány objektům třídy NSResponder. Chceme-li tedy například reagovat na stisknutí klávesy, nemusíme se vůbec nijak zabývat událostmi a jejich správou - jen ve vhodné instanci třídy NSResponder (nejspíš, ale ne nutně, v nějakém NSView, protože obvykle chceme zadaný text zároveň zobrazovat) implementujeme metodu -keyDown:. Pozorní čtenáři si možná pamatují na akce InterfaceBuilderu: ano, jde vlastně přesně o tentýž mechanismus: ovšemže je vhodné - a díky objektovému systému také snadno možné - použít stejný mechanismus pro "chceme reagovat na stisknutí tlačítka myší" a "chceme reagovat na stisknutí klávesy".

Ondřej Čada