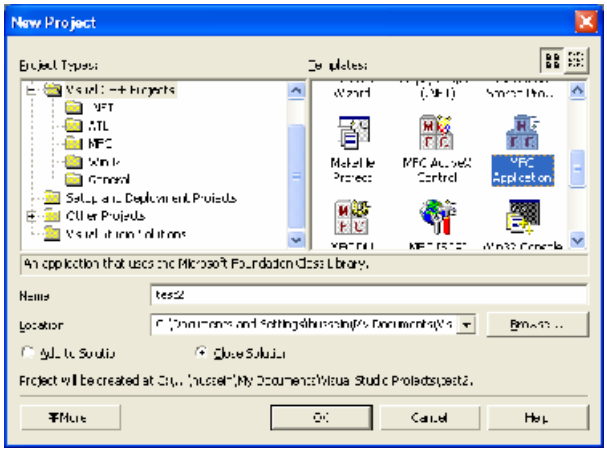
Comparative Programming Languages

hussein suleman
uct csc304s 2003

Visual Languages and IDEs: Visual C++

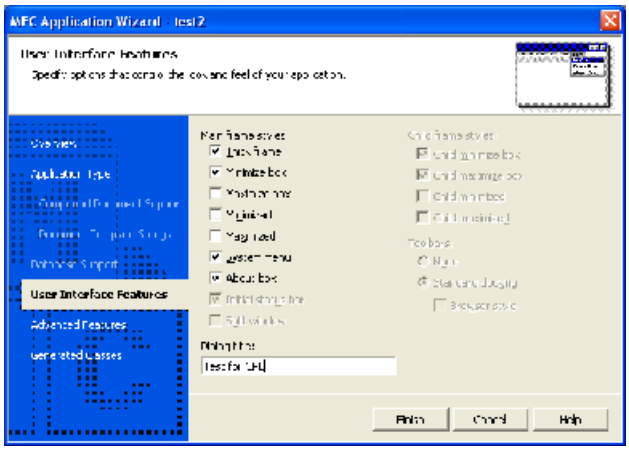
Program Layout

- Wizard selection of type of application to create templates.



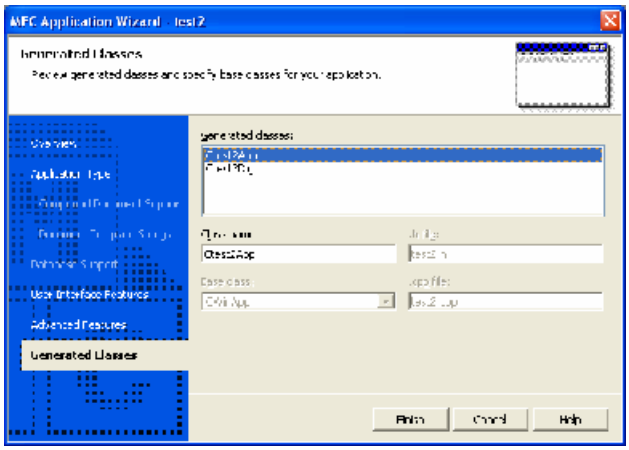
Basic Feature Selection

- Common features of visual applications are simple options.

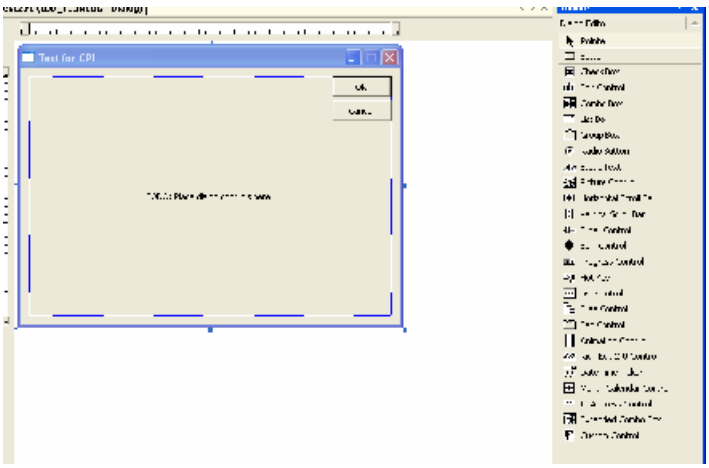


Code generation

- Use visual controls to present options and generate code based on those.



Visual Metaphors

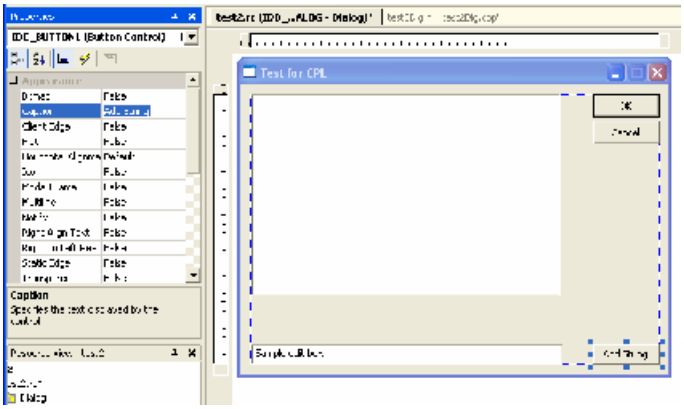


Canvas

Widgets/Tools/Components

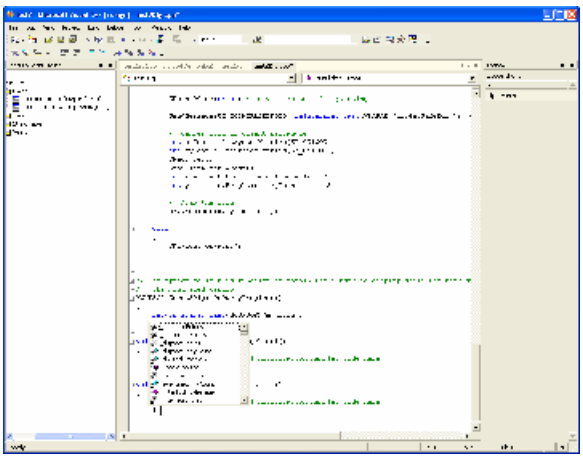
Property Lists

- Wherever possible, initial values of variables are set visually – some languages treat such variables differently.



Code Correspondence

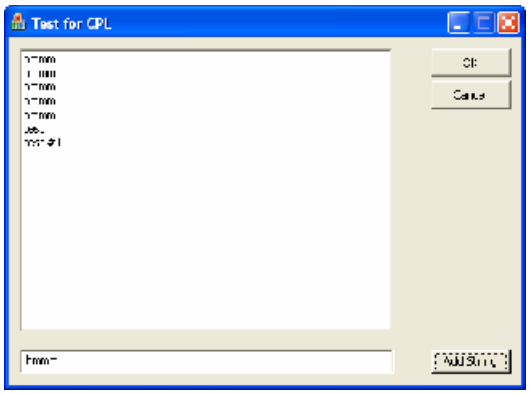
- "Make the simple things simple and the difficult things possible". It is still possible to edit code.



Minimalist Programming

```
void Ctest2Dlg::OnBnClickedButton1()
{
    // TODO: Add your control notification handler code here
    char c[1024];
    e.GetWindowText(c, 1024);
    l.AddString(c);
}
```

Only three lines of actual code to get from “go” to application!



Runtime and Design-time

- Core widgets/controls are provided by OS.
- Common controls (e.g., JPEG viewers) are distributed as libraries.
- User-defined components can be defined and added to toolbar.
- Components have runtime operation and design-time operation.
  - For example, at runtime a clicked button generates an event. At design-time when a button is scaled the text label recenters itself in the button.

Issues?

- How do you define non-visual parts of the program?
- Is it possible to link together non-visual components in the same way as visual components?
- Can components be language independent? (Builder v1.0 came with both a Pascal and C++ compiler to accomplish this)

Back in the good ol' days

- You wrote 300 lines of code in C for a "Hello World" program.
- Your application had to use code to manually create type definitions and instances of each widget.
- Your application had to do explicit cooperative synchronisation with other applications.

- Thankfully, those days are gone ☺

Scripting Languages: Introduction to Perl

Perl as Scripting Language

- Scripts are usually thought of as “glue” that connects real applications together.
- Scripting languages are sometimes highly specialised (e.g., BASH) or sometimes very general (e.g., Perl).
- Most scripting languages are interpreted, therefore very flexible but not fast.
- Perl is one of the most popular scripting languages because it can be quick-and-dirty.

Simple Example

```
print "Checking chat directory\n";

# get listing of all files
opendir (DIR, "$d/chat");
@files=readdir (DIR);
closedir (DIR);

# iterate throughlist
foreach (@files)
{
   # build filename and get information
   $filename="$d/chat/$_";
   @info=stat ($filename);

   # check age and move file if it has expired
   $daysold=(-M $filename);
   if ($daysold > $maxdaysgame )
   {
      system "mv $filename $d/delgame";
      print "moving ... $_   [$daysold]\n";
   }
}
```

Data Types

- Scalar – prefixed with a $
  - `$t =“Text string”;`
  - `$n = 1234;`
- List – prefixed with a @
  - `@alist = (1, 2, 3);`
  - `$t = $alist[0];  # assigns value of 1`
- Associative Array – prefixed with a %
  - `%parms = (“foam”, 1, “bubbles”, 2);`
  - `$foamvalue = $parms{“foam”};`

Variable Substitution

- Perl can process strings to interpolate values for variables.
- “” strings are preprocessed – ‘’ strings are not.

- Example:
  - `$dollar = 123;`
  - `$currency = ‘$dollar’;`
  - `print “rand - $currency rate”;`
    - rand – $dollar rate
  - `print‘rand – ’.$currency.‘ rate’;`

Loose Typing

- Types of scalars are not declared. Conversion between strings and numerics is implicit as and when needed.
  - Example: `print substr (12+1, 1);`
- Lists can store any scalars.
  - Example: `@t = (1, “123”, 2);`
- Specific semantics are associated with the use of types in “wrong” contexts.
  - Example: `$size = @t;`
  - Returns the number of elements in the list.

Control Structures

- Loosely based on C - statements are semicolon-separated.
- Conditional:

```
if (expression) { … }
  elsif (expression) { … }
  else { … }
```

- Iteration:
  - `while (expression) { … }`
  - `for (initial; test; increment) { … }`
  - `foreach variable (list) { … }`

Regular Expressions

- Regular expressions are used to match strings from a regular language – i.e. context-free with no recursion.
- Perl regular expressions are used for:
  - matching
  - search-and-replacement
- Example:
  - `$t =~ s/[\xa0-\xff]/'&#'.ord ($&).';'/geo;`
  - Converts all extended ASCII characters in the string $t into Unicode.

Regular Expression Syntax

\	Escape for next character
.	Any character
*	Zero or more of preceding element
+	One or more of preceding element
?	Optional preceding element
{}	Occurrence range
[]	Character class – any one of characters
|	Logical OR
()	Grouping operator
^ / $	Matches beginning/end of string

Regular Expression Examples

- `hello.*`
  - Matches “hello world”, “hello gazlum”, …
- `(a|b)?(c|d)*e`
  - Matches “accce”, “bcdcde”, “ae”, “be”, …
- `[a-z]+`
  - Matches “a”, “bed”, “fdsff”, …
- `[hH][eE][lL]{2}[oO]`
  - Matches “HeLlO”, “HELLO”, “hello”, …
- `http:\/\/[a-zA-Z0-9]+(\.[a-zA-Z0-9]+)*(:[0-9]+)?(.*)`
  - Matches simple URLs!
  - http://www.cs.uct.ac.za:80/

Using Regular Expressions

- Boolean expressions
  - `if ($text =~ /[a-z]+/) { … }`
- Search-and-replace
  - s/<source>/<destination>/options
  - `$url =~ s/(http:\/\/[a-zA-Z0-9]+(\.[a-zA-Z0-9]+)*)(:[0-9]+)?(.*)/$1:8080$4/go;`
  - Changes the port number in a URL to 8080 by decomposing the URL into constitutents and re-forming it with the new port number.
  - $1, $2, etc. are used to refer to groups defined by parentheses.

Subroutines

- Subroutines are defined as follow:
  - `sub <name> { … }`
- Parameters are passed in the @_ list and are usually assigned in the first line.
- Return value is the last expression.
  - Example:

```
sub concatenate
{
   my ($a, $b) = @_;
   $a.$b;
}
```

- Note: “my” creates local variables.

Perl as Text Processor

- Besides regular expressions, Perl has many built-in functions for efficient text processing:
  - split – splits a string into a list of parts based on a separator string or separator regular expression.
  - join – joins a list into a single string.
  - index/rindex – search for the index of one string within another.
- Example:
  - `@CGIarray=split (/[=&]/);`

Perl as Functional Language

- Being interpreted, Perl easily mixes aspects of procedural, OO and functional paradigms.
- map – applies a "function" to a list.
  - Example:
    - `$prefix = join ('', map { chr (ord('a')+rand(26)) } (1..5));`
    - Creates a random string of 5 characters
- eval – interprets the string or block as Perl statements
  - Example:
    - `while ($name = <STDIN>) { eval $name; }`
    - One-line Perl interactive shell!

References

- References are special scalars that contain the address of a piece of data.
- Obtain reference by prefixing data/variable with "\".
- Dereference variable by prefixing with either "$", "@" or "%", depending on data type.
- Example:

```
my @a = (1,2,3);
my $aref = \@a;
my @b = @$aref;
my $c = $$aref[0];
```

Anonymous Data Structures

- References to anonymous data structures can be created with [] for lists and {} for associative arrays.
- Example:

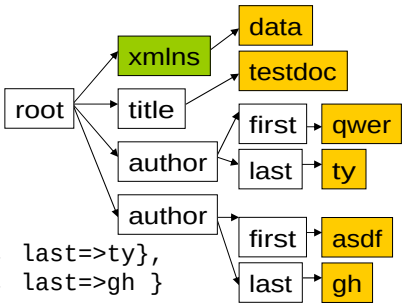
```
my $a = [ "apples", "oranges" ];
my $b = { apples=>1, oranges=>2 };
foreach my $fruit (@$a)
{
   print $b->{$fruit};
}
```

- Note: -> is the dereference and index operator for lists and associative arrays.

Complex Data Structures

- Example:

```
my $xmldata =
   { root =>
      [[
         { xmlns=>data },
         { title=>testdoc,
           author=>
              [
                 { first=>qwer, last=>ty},
                 { first=>asdf, last=>gh }
              ]
         }
      ]]
   };
foreach $auth (@{$xmldata->{root}->[0]->[1]->{author}})
{
   print "$auth->{first} $auth->{last}\n";
}
```



Packages

- Perl subroutines can be stored in separate files with specified names/namespaces (like in C++).
- To define a package:
  - `package Test::CPL;`
    - and a bit of code to export symbols …
- To use a package:
  - `use Test::CPL;`
- CPAN (Comprehensive Perl Archive Network – www.cpan.org) is a clearinghouse for modules for just about everything!

O-O Perl

- Packages can be further abstracted to correspond to classes, similar to Java.
- Perl includes support for
  - class/instance methods,
  - information hiding,
  - constructors and destructors,
  - multiple inheritance,
  - method overloading,
  - polymorphism.
- Most reusable code in Perl is now OO.

Evaluation

- Good for rapid prototyping and glue.
- Easy to write complex code – difficult to understand and maintain unless programmers are disciplined.
- Lots of features but not very efficient because of it is interpreted.
- Can be used for procedural, object-oriented, functional, visual (Glade) and maybe even declarative (Perlv6) programming – the kitchen sink solution!

Which language ?

- What are your basic needs?
  - speed of code? space efficiency? speed of coding?
- What are the problem domain issues?
  - ai-centric? data-centric? parallel? gui?
- What are the environment restrictions?
  - platform? memory? available compilers? company policy?
- What are the cost issues?
  - compiler cost? training of programmers?

go forth and code !