Instructions to Generate Code Using With Class Scripts

Richard Felsinger, RCF Associates
960 Scottland Dr, Mt Pleasant, SC 29464
Tele 803-881-3648 e-mail 71162.755@compuserve.com

The purpose of these instructions are to provide the steps to generate code for various languages and compilers. These are version 1.0 scripts that are provided as a "starting point" for With Class users. Users are encouraged to try the script out for their language of choice. Scripts are provided for the following languages: ANSI C++, Microsoft Visual C++, Borland C++, Eiffel, Borland Turbo Pascal/Delphi, ANSI Ada, Smalltalk, Actor, SQL, and POET. All scripts should generate compilable code if you use/update the test class diagram. In some cases, the test class diagram must be updated with information such as an include file. Scripts get better with usage and feedback. Except for some Smalltalk and Actor scripts, all scripts were written by Richard Felsinger. The With Class script language was designed and is maintained by Michael Gold of MicroGold Software. Please send suggestions, script enhancements, and new scripts to MicroGold S/W e-mail 71543,1172@compuserve.com info Richard Felsinger 71162,755@compuserve.com

The basic steps to generate code are as follows:
- Open a class diagram, e.g. cppcar.omt file or create a new class diagram.
- Update the class diagram and specification forms for your language, e.g. appropriate attribute types or classes - int.
- Select “File - Edit File” to open the code generation script, e.g. cppfunc.sct and save the script in the directory where you want the generated code files to be placed. Note With Class puts the generated code files into the same directory as the code generation script.
- Select “Generate - Generate Code”.
- Enter the appropriate file extension, e.g. cpp.
- Select the appropriate script, e.g. cppfunc.cpp to generate the code files.
- Select “File - Edit File” to check to generated code files, e.g. car.cpp.
- As required, update the class diagram, class specification forms, and script.
- Compile, link, and execute the generated files in your language compiler.

Suggestions to Update Class Diagrams and Scripts to Generate Compilable Code

When you are creating a new class diagram for a particular language, the following are some guidelines to ensure that compilable code is generated. Missing information, e.g. a missing ATTRIBUTE_TYPE will cause a compilation error.
- **C++ and most languages** - Enter correct ATTRIBUTE_TYPE for each attribute, e.g. int.
- **C++ and most languages** - Enter each parameter in the form OPERATION_PARAMETER_TYPE OPERATION_PARAMETER_NAME, e.g. int aNumber. With Class provides a large number of script variables, e.g. CPP_OPERATION_PARAMETERS and PASCAL_OPERATION_PARAMETERS_WITH_VAR which transform the parameter list into the correct format for each language
- **C++ and most languages** - Update the INCLUDE_FILE for collection classes used to implement 1 : Many relationships, e.g. Borland C++ 3.X enter listimp.h and Borland C++ 4.X enter classlib\listimp.h.
- **C++ and most languages** - Update the CLASS_LIBRARY_BASE_CLASS to have subclasses of a library class. For Visual C++ 1.5, enter : public CObject for the highest base class of any class that is an element of a CObList or CPtrList collection class.
- **C++** - Always enter an OPERATION_RETURN_TYPE, e.g. int, void, etc. Only enter a return type for other languages if there actually is a return type.
- **Pascal** - Enter each unit name as the CLASS_FILENAME and enter each unit name required in a uses clause as INCLUDE_FILE. Enter a semicolon after the last unit name in the INCLUDE_FILE list. Turbo Pascal requires a termination semicolon in the uses clause.

To generate compilable code, you can manipulate information on the class diagram, in the class specification forms, and the scripts themselves. If you have a specific problem generating a compilable code statement for your language or library, try the following in this order. For example, you may need a trailing semicolon in a particular statement or a new piece of information.

(1) **Update Specification** - Put the trailing semicolon in the appropriate With Class text box.
(2) **Use a Substitute Variable** - Put the information into a substitute variable, e.g. CLASS_DESCRIPTION or OPERATION_COMMENT3
(3) **Update the Script** - Put the trailing semicolon in the script.
(4) **Generate Comment Statement** - Generate a commented out statement and then update manually update the line for compilable code.
(5) **Request New Special Script Variable** - Contact MicroGold S/W and request a new special script variable that generates the compilable statement. You may require a new text box or check box for the new information.
(6) **Request New Special Operator** - Contact MicroGold S/W and request a new special operator, e.g. IF or COMMENT.

With Class 2.1 supports entering and saving data types from various languages. Data types are saved in the wc2.ini file. WC2.ini is created upon installation and if no wc2.ini exists. The initial wc2.ini has C/C++ data types, e.g. int, float, etc. Whenever you enter a new data type for an attribute, the wc2.ini file is updated. When you begin using With Class, it is recommended that you create a diagram to enter the all the basic data types for attributes for the language and class library that you are using. The following are sample data types that may be used in various languages:

- Microsoft Visual C++: CString	needs #includes "stdafx.h" or <afx.h>
- Borland C++ 3.X: String	needs #include <strng.h>
- Borland C++ 4.X: string	needs #include <string.h>

- Pascal and Delphi: Byte, Integer, LongInt, Word, Real, char, String, Double
- Ada: Boolean, Character, Integer, Float, Positive, Natural, Duration, String
- Eiffel: BOOLEAN, CHARACTER, INTEGER, REAL, DOUBLE, ARRAY, STRING

You may directly update the wc2.ini file to change a spelling or capitalization.
If your wc2.ini file becomes corrupted, delete or rename wc2.ini and a new wc2.ini will be regenerated the next time you start With Class.

Steps to Generate Code in Various Languages

With Class supports code generation for a large number of object-oriented languages. Provided are a test class diagram (.omt file), one or two code generation scripts, and (if required) a main program. It is best to copy the .omt file, scripts, and main program into a separate directory, e.g. CAR. For each code generation script With Class generates a file for each class in the directory where the code generation script resides. For example, if the code generation script cppfunc.sct is in the CAR directory, then all the generated .cpp files will be placed in the CAR directory.

1. **ANSI C++** - Getting Started ANSI C++ Script

Class Diagram: cppcar.omt
Script Files: cpphead.sct and cppfunc.sct
Main File: cppmain.cpp

The ANSI C++ scripts and most of the C++ scripts generate the following C++ statements: #indefs, #includes, default constructor, constructor with arguments, copy constructor, default destructor, =operator, ==operator, get and set accessor functions for data members, pointer data members for association relationships, data member for aggregation relationships, C-arrays for 1 to many relationships, operator<< insertion for cout statements, and operator>> for cin statements. You must manually update the default constructor to initialize base class data members. If you are not using IOSTREAM's then delete the #include <iostream.h> statement and the operator<< and operator>> code generation functions in the scripts.

The steps to generate ANSI C++ and to compile the generated files are as follows:

>> Launch With Class from Windows
>> Open cppcar.omt
>> Select "Generate - Generate Code"
>> Select h as the file extension

>> Select cpphead.sct to generate the header files (.h)
>> Select "Generate - Generate Code"
>> Select cpp as the file extension
>> Select cppfunc.sct to generate the source code files (.cpp)
>> Select "File - Edit File" to examine the generated files, e.g. car.h
>> Run your C++ environment
>> Create the Car Project, e.g. carproj.mak
>> Add the .cpp files to the project. e.g. cppmain.cpp, car.cpp etc
>> Compile all .cpp files
>> Compile cppmain.cpp
>> Execute the project

2. **Microsoft Visual C++ 1.5 with QuickWin** - Getting Started Visual C++ Script

Class Diagram: cppcar.omt
Script Files: vc15head.sct and vc15func.sct
Main File: vc15main.cpp

The easiest way to use With Class and Visual C++ is to create a QuickWin project. This will permit the use of the C++ iostream objects, e.g. cin and cout. The steps to generate Microsoft Visual C++ 1.5 code with QuickWin and to compile the generated files are as follows:

>> Launch With Class from Windows
>> Open cppcar.omt
>> Double click on the Tire and Passenger classes to add the following in the Library Base Class box
: public CObject. These are on the 1 : Many side. The objects are elements of CObList and CPtrList Collection classes
>> Select "Generate - Generate Code"
>> Select h as the file extension
>> Select vc15head.sct to generate the header files (.h)
>> Select "Generate - Generate Code"
>> Select cpp as the file extension
>> Select vc15func.sct to generate the source code files (.cpp)
>> Select "File - Edit File" to examine the generated files, e.g. car.h
>> Run Visual C++ environment
>> Create QuickWin Car Project, e.g. carproj.mak and select "Use Microsoft Foundation Classes"
>> Add the .cpp files to the project
>> Compile all .cpp files
>> Compile vc15main.cpp
>> Execute the project

Notes for Visual C++ 1.5. To use iostream.h then create a VC15 QuickWin project. Create a VC15 QuickWin project with vc15main.cpp. Add the following files to the project: vehicle.cpp, car.cpp, cellular.cpp, passenge.cpp, tire.cpp, motor.cpp, and vc15main.cpp.

Select "Options Project Linker
Miscellaneous" and enter /NOE. Select "Options Project
Linker Windows-Library" and select COMMDLG and SHELL.

Copy stdafx.h and stdafx.cpp into your directory.

This script uses CPtrList ASSOCIATION_MANY_NAME and CObList AGGREGATION_MANY_NAME for 1 to Many Relationships. For your highest base class of any class whose objects are elements of the CObList or CPtrList collection classes, enter : public Cobject in the Class Specification Library Base Class.

For Visual C++ 2.0 1 : Many relationships substitute a template collection class for the CObList and CPtrList.

3. **Microsoft Visual C++ 1.5 for a Windows Application** - Getting Started Visual C++ Script

Class Diagram: cppcar.omt
Script Files: vc15hdex.sct and vc15fuex.sct
Main File: N/A

The steps to generate Microsoft Visual C++ 1.5 code for a Windows EXE and to compile the generated files are as follows:

>> Launch With Class from Windows
>> Open vc15car.omt
>> Double click on the Tire and Passenger classes to add the following in the Library Base Class box
: public CObject. These are on the 1 : Many side. The objects are elements of CObList and CPtrList Collection classes.
>> Select "Generate - Generate Code"
>> Select h as the file extension
>> Select vc15hdex.sct to generate the header files (.h)
>> Select "Generate - Generate Code"
>> Select cpp as the file extension
>> Select vc15fuex.sct to generate the source code files (.cpp)
>> Select "File - Edit File" to examine the generated files, e.g. car.h
>> Run Visual C++ environment
>> Select "App Expert"
>> Create Windows Application Car Project, e.g. XXX.mak
>> Add the .cpp files to the project
>> Select "File - Open" XXXdoc.h file to add #include "car.h" as an include file and add Car car1; as a data member
>> Compile all .cpp files
>> Update the XXXdoc with functions to invoke the car functions
>> Execute the project

Note: This script uses CPtrList ASSOCIATION_MANY_NAME and CObList AGGREGATION_MANY_NAME for 1 to Many Relationships. For your highest base class of any class whose objects are elements of the CObList or CPtrList collection classes, enter : public CObject in the Class Specification Library Base Class.

For Visual C++ 2.0 1 to Many relationships substitute a template collection class for the CObList.

4. **Borland C++ Version 3.X** - Getting Started BC 3X Script

Class Diagram: cppcar.omt
Script Files: bc3head.sct and cppfunc.sct
Main File: cppmain.cpp

The steps to generate ANSI C++ and to compile the generated files are as follows:

>> Launch With Class from Windows
>> Open cppcar.omt
>> Double click on the Car class and select Class Spec dialog box, and enter the include file listimp.h. This is for the BI_ListImp Class to implement 1 to Many relationships
>> Select "Generate - Generate Code"
>> Select h as the file extension
>> Select bc3head.sct to generate the header files (.h)
>> Select "Generate - Generate Code"
>> Select cpp as the file extension
>> Select cppfunc.sct to generate the source code files (.cpp)

>> Select "File - Edit File" to examine the generated files, e.g. car.h
>> Launch Borland C++ Version 3.X
>> Create the Car Project, e.g. carproj.mak
>> Add the .cpp files to the project
>> Compile all .cpp files
>> Compile cppmain.cpp
>> Execute the project

Notes for Borland C++ 3.X. Enter include file names in the include box in With Class, e.g. strng.h for String class. For 1 : Many relationships enter listimp.h in the Class Spec include box for BI_ListImp class. If desired change BI_ListImp in this script to BI_ArrayAsVector with arrays.h to BI_StackAsVector with stacks.h or to BI_BagAsVector with bags.h.

In Borland C++ 3.X enter the following options:
Application: Large Memory Model
Linker Link Libraries: Static for all libraries
Include Directories: c:\bc\include;c:\bc\owl\include; c:\bc\classlib\include
Library Directories: c:\bc\lib;c:\bc\owl\lib;c:\classlib\lib;
Compiler - Code Generation - Defines WIN31

5. **Borland C++ Version 4.X with EasyWin** - Getting Started BC 4X Script

Class Diagram: cppcar.omt
Script Files: bc4head.sct and cppfunc.sct
Main File: cppmain.cpp

The steps to generate Borland 4.X C++ and to compile the generated files are as follows:

>> Launch With Class from Windows
>> Open cppcar.omt
>> Double click on the Car class and select Class Spec dialog box and enter the include file classlib\listimp.h. This is for the TListImp <XXX> Class to implement 1 to Many relationships
>> Select "Generate - Generate Code"
>> Select h as the file extension
>> Select bc4head.sct to generate the header files (.h)
>> Select "Generate - Generate Code"
>> Select cpp as the file extension
>> Select cppfunc.sct to generate the source code files (.cpp)
>> Select "File - Edit File" to examine the generated files, e.g. car.h
>> Run your C++ environment
>> Create the Car Project, e.g. carproj.mak
>> Add the .cpp files to the project
>> Compile all .cpp files
>> Compile cppmain.cpp
>> Execute the project

Notes for Borland C++ 4.X. Enter include file names in the include box in With Class, e.g. cstring.h for string class. For 1 : Many relationships enter classlib\listimp.h in the include box for TListImp <XXX> class. XXX represents the class names of the objects in the collection, e.g. Passenger. If desired change TListImp in this script with TArray with classlib\arrays.h, TStack with classlib\stacks.h, TBag with classlib\bags.h.

6. **Eiffel with ISE Personal Eiffel** - Getting Started Eiffel Script

Class Diagram: eiffcar.omt
Script Files: eiffel1.sct
Other Files: carsys.e and ACE

The steps to generate Eiffel code and to compile the generated files are as follows:
>> Create a wc2prog directory and copy ACE, eiffel1.sct, carsys.e, and eiffcar.omt into it
>> Launch With Class from Windows
>> Open eiffcar.omt
>> Select "Generate - Generate Code"
>> Select e as the custom file extension
>> Select eiffel1.sct to generate the eiffel classes
>> Select "File - Edit File" to examine the generated files, e.g. car.e
>> Launch your Eiffel environment
>> Check your ACE file to ensure it has the correct directory
>> Compile the .e files
>> Execute

Eiffel Notes:

(1) Enter Basic Eiffel Types: BOOLEAN, CHARACTER, INTEGER, REAL, DOUBLE, ARRAY, STRING in the With Class Attribute and Operation specification forms.

(2) In With Class, enter the operation parameters in the form operation_type operation_name. Then use the keyword PASCAL_OPERATION_PARAMETERS to print the parameters in Eiffel form.

(3) In Notepad update the ACE file with the current directory. If you place these files in another directory update the following line in the ACE file to show the correct directory: ROOT_CLUSTER: "$EIFFEL3\wc2prog";

(4) The script eiffel1.sct generates several commented out lines, e.g. invariant. If you put invariant information into the Class Specification INVARIANT box, then it will be inserted in the generated .e file. It is commented out since some classes do not have invariants.

(5) To provide ISE Personal Eiffel with maximum memory, it is best to quit With Class before running ISE Personal Eiffel.

Question - What additional With Class script variables or additional dialog boxes, text boxes, or check boxes are required to fully support Eiffel. Please submit new Eiffel scripts.

7. **Borland Turbo Pascal or Delphi** - Getting Started Turbo Pascal Script

Class Diagram: pascar.omt
Script Files: pascal1.sct
Main Files: pasuser.pas

The steps to generate Borland Turbo Pascal code and to compile the generated files are as follows:

>> Launch With Class from Windows
>> Open pascar.omt
>> Double click on each class to enter information in the Class Specification dialog box. Enter Unit name in the File Name box and enter the Uses Unit names in the Include File box. Add a semicolon to the last unit name
>> Select "Generate - Generate Code"
>> Select pas as the custom file extension
>> Select pascal1.sct or delphi1.sct to generate the Pascal units
>> Select "File - Edit File" to examine the generated files, e.g. car.pas
>> Launch your Pascal environment
>> Compile, make, and build each generated file, e.g. VehicleU.pas, MotorUn.pas, CellulaU.pas, PassengU.pas, TireUnit.pas, and CarUnit.pas.
>> Compile, make, build and execute pasuser.pas

Notes for generating Turbo Pascal (Turbo Pascal 5.5. or greater and Delphi Version 1.0):

(1) In With Class enter the unit name in the class specification dialog box "File Name" field. The file name becomes the unit file name and the unit name, e.g. file name = CarUnit.pas.

(2) For uses clause, enter the unit name in the Include Box. Enter the last unit followed by a ; e.g. UnitZ; Ensure that the unit names entered in the Include Box are consistent with the unit names entered in the CLASS_FILENAME Box.

(3) On the class diagram, insert parameters in the C++ form, e.g. Parameter_Type Parameter_Name - Integer aNumber. The script variable PASCAL_OPERATION_PARAMETERS_WITH_VAR then puts the parameter into the Pascal form, e.g. (Parameter_Name : Parameter_Type ; Var A_Parameter_Type : Parameter_Type) - (A_Number : Integer ; Var A_Integer : Integer).

(4) This version of With Class cannot differentiate between a function and a procedure. This script only generates procedures for operations. Return types become Var parameters, e.g. Var A_Integer : Integer.
With Class provides PASCAL_OPERATION_RETURN_TYPE, e.g. : Integer.

Question - What additional With Class script variables or additional dialog boxes, text boxes, or check boxes are required to fully support Turbo Pascal and Delphi. Please submit new scripts.

8. **ANSI Ada (Ada83)** - Getting Started Ada Script

Class Diagram: adacar.omt
Script Files: ada1.sct
Main File: adauser.ada

The steps to generate ANSI Ada (Ada83) code and to compile the generated files are as follows:

>> Launch With Class from Windows
>> Open adacar.omt. Note that is diagram has no inheritance relationships because inheritance is not supported in ANSI Ada
>> Select "Generate - Generate Code"
>> Enter ada as the custom file extension
>> Select ada1.sct to generate the Ada packages
>> Select "File - Edit File" to examine the generated files, e.g. car.ada
>> Launch your Ada environment
>> Compile and link each generated file, e.g. car.ada
>> Compile, link, and execute adauser.ada

Ada Notes:

(1) Enter the basic Ada types in the attribute and operation specification forms: BOOLEAN, CHARACTER, INTEGER, FLOAT, POSITIVE, NATURAL, DURATION, STRING.

(2) Enter operation parameters in the operation specification form parameter_type parameter_name. Then use the variable ADA_OPERATION_PARAMETERS.

(3) This version of With Class cannot differentiate between a function and a procedure. The Ada script only generates procedures for operations.

Question - What additional With Class script variables or additional dialog boxes, text boxes, or check boxes are required to fully support Ada-83, Ada-9X, and VHDL. Please submit new Ada scripts.

9. **Other Languages**: "Getting Started Scripts" are provided for the following languages: Smalltalk (smtalk1.sct or smtalk2.sct), Actor 4.0 (actor1.sct), SQL (sql1.sct), and POET (poethead.sct and poetfunc.sct). The Smalltalk

and Actor scripts may be tested using the cppcar.omt file or any .omt file. The Smalltalk and Actor scripts were provided by Jim Peterson, e-mail 70730.1602@compuserve.com. The SQL (sql1.sct) and POET (poethead.sct and poetfunc.sct) are very, very simple scripts and require enhancements.

10. **Final Note** - Comments, suggestions, and enhanced scripts are requested. Scripts for new languages are requested. Are there any Ada9X or Object COBOL programmers out there? Suggestions on new specification forms and text boxes are requested to better support code generation. Comments on these instructions are solicited. Please send comments to MicroGold S/W e-mail 71543,1172@compuserve.com info Richard Felsinger 71162,755@compuserve.com.