Object Oriented Modeling - With Class and StateMaker

by Richard C. Felsinger, RCF Associates, 960 Scottland Dr, Mt Pleasant, SC 29464 Tele 803-881-3648 (Voice & Fax) E-mail - Internet felsingerr@citadel.edu or
CompuServe 71162,755 (Comments and suggestions are requested - 6/9/93)

Introduction

The purpose of this tutorial is to present "step by step" instructions on how to use With Class and StateMaker CASE tools to create object oriented diagrams, fill in forms, and automatically generate C++. With Class and StateMaker are configurable CASE (Computer Aided Software Engineering) tools from MicroGold Software, 698 Birch Hill Drive, Bridgewater, NJ 08807 Tel 908-722-6438 CompuServe 71543,1172. With Class and StateMaker operate in the Microsoft Windows environment. An expanded printed version of this tutorial is provided with the commerial versions of With Class and StateMaker.

O-O modeling means to create object oriented diagrams, text specifications, and code to describe a system, subsystems, and classes. It is to examine a problem from different points of view and to create a software solution to the problem. Just like an automotive design engineer creates drawings, clay models, and text specifications of a new car, we create diagrams, text specifications, and code to describe a new software system. The ultimate aim is to create a new software system (computer program) for the problem that has excellent S/W quality factors, e.g. correct, reliable, and modifiable.

Tutorial Objectives - There are two primary objectives of this tutorial. The first objective is to present how to model and prototype in C++ a simple class using With Class and StateMaker. One example will be provided: a car that a user can start, set the gas quantity, and get the gas quantity.

The second objective is to present how to model and prototype a simple system in C++. One example will be provided: a TV controller to manipulate the volume and channel settings. All examples will use a variety of documentation products, e.g., diagrams, text specifications, and code.

Documentation Products for O-O Modeling

There are three major documentation products for O-O modeling: diagrams, text specifications, and code as described below:

Diagrams - The two major O-O diagrams to be created using With Class are the class diagram and message diagram (event flow diagram). The major diagram to be created using StateMaker is the state diagram. System drawings and interface drawings will be created using With Class or a Windows word processor.

Text Specifications - There are many text forms used in O-O modeling. These include requirements statements, data dictionaries, class specifications, message lists, and message scenarios. The With Class text editor or a Windows word processor will be used to create text specifications.

Code - Source code in C++ will be initially generated using With Class and StateMaker. Then this generated code will be compiled using Borland C++. Next messages and transformations will be added to the source for a full executable C++ prototype.

Windows Tools for O-O Modeling

There are four major windows tools for O-O modeling: windows word processor for text specifications and simple diagrams; CASE tool for O-O diagrams, text specifications, and C++ code generation; GUI interface builder for GUI prototypes and code generation; and C++ development environment. In this tutorial the following windows tools are used:

Windows word processor - Lotus AmiPro and Microsoft Write
CASE tool - MicroGold With Class and StateMaker
GUI interface builder - Protoview Development Protogen +
C++ development environment - Borland C++ with Application Frameworks

Windows Word Processor - The windows word processor is important to create text specifications and simple drawings.   A windows word processor has both text creation and simple drawing creation tools. It has the capability to tie together and integrate text and graphics from several windows tools using the clipboard, dynamic data exchange, or object linking and embedding.   The following are several word processors that are useful for O-O Modeling in the Microsoft Windows environment: Microsoft Word, Lotus AmiPro, Word Perfect, and Frame Technology FrameMaker.

CASE Tool - An O-O CASE tool is important to create O-O diagrams, text specifications, and C++ code generation. An O-O CASE tool is very helpful for O-O modeling because it provides a structure to create diagrams, text specifications, and code. Additionally, an O-O CASE tool saves time because diagrams, text specifications, and code may be quickly created and changed. Finally, an O-O CASE tool is valuable for training and education to understand the various O-O entities, O-O connections, and O-O methodologies. The following are several O-O CASE tools that are useful in the Microsoft Window environment: MicroGold With Class and StateMaker, OOTool from Roman M. Zielinski in Norsborg, Sweden, Mark V ObjectMaker, ProtoSoft Paradigm Plus, Object International OOA Tool, and General Electric OMT Tool.

GUI Interface Builder - A GUI interface builder is important to design and prototype the GUI interface. The GUI interface is an important aspect of an O-O system because it provides direct contact with the user. Often prototyping the GUI interface is extremely valuable to understand the user requirements. GUI interface builders will be used in modeling systems later in this tutorial. The following are several GUI Interface Builders that are useful in the Microsoft Window environment: ProtoView Development Protogen +, Blue Sky WindowsMaker, Borland C++ Resource Workshop, Microsoft Visual C++ App Studio.

C++ Development Environment - The C++ development environment is important to develop working system prototypes and to develop implementation products. The environment has many tools, e.g. class browser, compiler, linker, debugger, GUI builder, and class libraries. Two important C++ development environments in the Microsoft Windows environment are the Borland C++ and Microsoft Visual C++.

Objectives of O-O Modeling - S/W Quality Factors

The primary objective of O-O modeling is to describe and specify a system to lead to a high quality software program. A quality software program has the major S/W quality factors of correctness, reliability, and extendibility as described by Bertrand Meyer in "Object-oriented Software Construction". Other S/W quality factors include understandability, adaptability, reusability, efficiency, portability, traceability, completeness, confirmability, modularity, error handling, uniformity, and ease of use.

Correctness is the ability of a system to perform in accordance with their stated purpose and requirements. Correctness is "doing what it is supposed to do". It ensures that correct values are always computed. It ensures that there is correct stimulus response behavior of a system or object. Correctness is to avoid errors. For example, it is important for a bank automatic teller machine to correctly verify a user card, personal identification number, and to correctly update account balances.

Reliability is the ability of a system to perform correctly for long periods of time. Reliability is sometimes referred to as "robustness", i.e. the ability of a system or object to perform correctly even under abnormal conditions. It ensures that a system or object has the capability to detect errors and faults and to take appropriate action. For example, it is important for a bank automatic teller machine to operate for long periods of time without problems and errors.

Extendibility is the ability to easily change and update a system. It deals with making both minor changes to the existing entity and making major enhancements to the entity. Changes and enhancements should not cause a "ripple effect" of undesired problems in other entities. It means that a system can be easily modified and extended. For example, it is important for a bank to easily make changes and modifications to a bank automatic teller machine without causing undesired side effects and problems.

In summary, to create a high quality software system, we model the system from various points of view with various documentation products. This is to lead to a software system with excellent S/W quality factors of correctness, reliability, and extendibility.

Object Oriented Methodologies

An object oriented methodology has a defined set of entities (system, class, object, etc.), graphic symbols, diagrams, forms, rules, and procedures. It is a process that generally covers the entire software lifecycle from requirements to implementation. With Class supports a subset graphic notation for several object oriented methodologies, e.g., Rumbaugh Object Modeling Technique (OMT), Coad/Yourdon Object Oriented Analysis and Design (OOA/OOD), Booch Object Oriented Design (OOD), and Shlaer Mellor OOA/OOD. The following is a very brief overview of these methodologies.

Rumbaugh Object Modeling Technique (OMT) - This O-O methodology is documented in "Object-Oriented Modeling and Design" by James Rumbaugh et al. It has a strong data modeling origin. It has a very comprehensive graphic notation. Analysis in OMT consists of the Object Model, Dynamic Model, and Functional Model which will be described shortly. Analysis is followed by system design and detailed design.

Coad-Yourdon Object Oriented Analysis and Design (OOA/OOD) - This O-O methodology is documented in "Object Oriented Analysis" and "Object Oriented Design" by Peter Coad and Ed Yourdon. It has a strong data modeling origin. It has a simple graphic notation. OOA consists of identifying and organizing classes from different points of view called layers. OOD consists of refining the OOA model for the human interface, problem domain, data management, and concurrent task management.

Booch Object Oriented Design (OOD) - This O-O methodology is documented in "Object Oriented Design with Applications" by Grady Booch. It has a very strong real time and message passing origin. It has an extensive, comprehensive graphic notation. OOD consists of modeling a system by identifying, organizing, and specifying subsystems, classes, objects, implementation modules and processes.

Shlaer-Mellor Object Oriented Analysis and Design (OOA/OOD) - This O-O methodology is documented in "Object Lifecycles - Modeling the World in States" by Sally Shlaer and Stephen Mellor. OOA consists of information (data) models, state models, and process models similar to the OMT object model, dynamic model, and functional model. OOD consists of refining the OOA models for an implementation environment.

In this tutorial we will use the Object Modeling Technique (OMT). OMT supports the entire software development lifecycle from requirements to implementation source code. However, we will only cover the early portion of the software lifecycle.   In O-O modeling in this tutorial, we will identify, organize, and prototype initial classes and objects in a system. We will not discuss specifics of system design or detailed design. In this tutorial select Rumbaugh from the With Class methodology menu.

Basic O-O Entities and Connections

Three major entities in O-O modeling are system, class, and object as described below.

A **system** is an entity that can be treated as a unit that is composed of simpler components that work together to perform a function. The term system is a general term that may refer to a very large system with smaller subsystems or to a small system without subsystems. A small system (or subsystem) consists of classes and objects that work together as a unit. A system may consist of 10 to 100 classes as a rough order of magnitude. In Borland C++ a **project** groups together a set of classes as a system. For execution, a system is an executable program (.EXE file), a dynamic link library, a process, or other executable entity. We model a system with a class diagram showing all the classes in the system. In this tutorial we will use the term system to refer to either a small system or to a subsystem. A sample system in this tutorial is the TV Controller System which consists of approximately 15 classes in a single executable program.

A **class** is a description of a group of objects with similar attributes, common operations, common connections (association, aggregation, message, and generalization specialization) and a common semantic purpose. In S/W a class is a module. An **attribute** is a characteristic or property of an object. An attribute is typically an atomic literal, e.g. integer, float, character, etc or a set of literals, e.g. a string of characters. For example, an attribute of a car is gas_quantity which is a float. An **operation** is a function, action or set of actions. For example, an operation of a car is to "set gas quantity" and "start". A **connection** is a relationship or link between classes or between objects. The primary connections are association ("has a"), aggregation ("part of"), message ("interacts"), and generalization specialization ("is a"). For example, the Car Class has an association and message connection with the User Class. The Car Class has a generalization specialization connection with the Vehicle Class. The Car Class has an aggregation connection with the Motor Class. The semantic purpose of a class is the reason for being or existence of objects of the class. For example, the semantic purpose of objects of the Car Class is to provide transportation to carry users from one location to another. We model a class with a class diagram and a class specification.

An **object** is an instance of a class. An object has an object name (ID) and a value for all attributes and associated objects. For example an object is Car11 with the attribute value of gas_quantity of 14.2 gallons. In execution an object responds to messages. We model objects in message scenarios.

As described below there are four major O-O connections: association, aggregation, message, and generalization specialization.

Association - Association is a link between classes that is a general mapping between the classes. An association connection may be described as "has a", "associated with", or "knows about". For example a user "has a" car. An association has a cardinality or multiplicity. This is the number of objects on one side of an association that may be associated with an object on the other side of the association. For example, one user has one car. This is an example of a 1 to 1 association. If a user has many cars, then there is a 1 to many association between user and car. The association symbol is a solid line between classes with the cardinality shown on each end of the sold line.

Aggregation - Aggregation is a link between classes that is an association with "part of" semantics.   An aggregation connection may be described as "part of" or "bill of materials" between an assembly object and part objects. For example a car "has a part" motor. An aggregation association has a cardinality or multiplicity. This is the number of objects on one side of an aggregation that may be associated with an object on the other side of the aggregation. For example, a car has one motor. This is an example of a 1 to 1 aggregation.   The aggregation symbol is a diamond with a solid line between classes with the cardinality shown on each end of the solid line.

Message - A message connection is a call or interaction between objects. A message connection may be described as "calls", "interacts", or "communicates with". For example a user "calls" a car. The full specification of a message consists of the receiver object name, the operation name, the input parameters, and the output (return) parameters. Messages may be shown on a class diagram since messages are defined within a class. However, messages are executed between objects. The message symbol is a solid arrow pointing in the direction of the receiver.

Generalization Specialization - A generalization specialization connection indicates a commonality between a superclasses and subclasses. It indicates that the superclass and subclass have common attributes, operations, and connections.   It indicates an "is a" or "type of" connection. For example, a vehicle superclass may define common attributes, operations, and connections for a car subclass. A car "is a" vehicle. The generalization specialization connection is implemented in programming languages with inheritance.   The generalization specialization symbol is a triangle on a solid line.

Three Primary Models

The three primary views in O-O modeling are the object model (OMT Object Model), the dynamic model (OMT Dynamic Model) and the functional model (OMT Functional Model). These three views give us a framework to model systems and classes. They require us to examine a problem from different points of view. They help us achieve quality software that is correct, reliable, and modifiable. All O-O methodologies use these three views in some form with different diagrams and text specifications.

To model and prototype a class, we are going to apply the three primary views of O-O modeling: object, dynamic, and functional. The training objective is to model a car where a user can start the car and can set and get the gas_quantity. From the object model, we will make a drawing of the user and the car. Then we'll make a class drawing showing attributes and operations. From the dynamic model, we'll create a message diagram (event flow diagram) and a message scenario listing messages from the user to the car. From the functional model, we'll list the transformations and correctness assertions for the start operation in the car class.

To model and prototype a class we will view the class from the three points of view. In the object model we will create the following: requirements statement, drawing, class diagram, and class specification. In the dynamic model we will create the following: message diagram (event flow diagram), message scenario, state diagram (if required). In the functional model, we will update class specifications with transformations and correctness assertions. We will automatically generate C++ source code using With Class and StateMaker and then update the source code with messages, transformations, and correctness assertions for an executable prototype.

Modeling Classes - The Object Model

Description of the Object Model

The object model corresponds to the OMT Object Model. The major question is "What is in the system or class?" We are concerned with **O-O entities**, basic building blocks in a system, e.g., class and object. We are concerned with **O-O connections**, links or relationships between O-O entities, e.g., association, aggregation, and generalization specialization. As described below the Object Model consists of the following documentation products: requirements statement, drawing, class diagram, and class specification. A prototype may be created.

In the object model, we define the static structure of classes. We define what is the composition of a class in terms of attributes and operations. We specify the connections between classes, e.g., association, aggregation, and generalization specialization. The message connection is specified in the dynamic model.   We specify **constraints** which are rules that restrict or limit the values that objects, attributes, and connections can have. A constraint is that a car can only have one motor and that the gas_quantity has a minimum and maximum value.

The object model is "snapshot" of a class at a point in time. For example the object model of a car is a physical drawing or a block diagram of a car. In this view we create drawings, class diagrams, and class specifications.

Steps to Create the Object Model

In the object model we specify classes with their attributes, operations, connections, and other relevant information. Follow these steps.

Step 1 - Draft the **requirements statement**. The requirements statement describes the system and its functionality.

Step 2 - Make a **drawing**. The drawing provides a physical, real world view of the objects that we are modeling. The drawing will help us to visualize the attributes and to walk through a scenario of messages.

Step 3 - Create the **data dictionary**. The data dictionary lists and describes key O-O entities and terms. Information on classes includes purpose, attributes, and operations.

Step 4 - Create the **class diagram**. The class diagram is a visual, graphic representation of classes showing attributes, operations, and connections.

Step 4a - Identify **attributes**. An attribute is a characteristic, property, or component of an object. An attribute consists of an attribute name, class, and value.

Step 4b - Identify **operations**. An operation is an action, algorithm, or set of steps that typically uses or modifies an attribute value.

Step 4c - Identify the **connections** for association, aggregation, and generalization specialization. These connections represent a relationship or link between two classes. Defer identification of message connections until the dynamic model.

Step 5 - Create the **class specification**. The class specification provides text information about the class. Class information should include class description, superclasses, visibility, cardinality, concurrency, transformations and correctness assertions and other information necessary for programming or documentation. Document constraints in the class specification.

Step 6 - Generate the **prototype - C++ source code**. Compile and execute the program.

Creating the Requirements Statement

The requirements statement describes the system and its functionality. It is often referred to as the charter, functional requirements, or problem statement. The requirements statement may be created with any text editor or word processor. With Class includes a simple text editor that can be used to create the requirements specification.

The "Car Requirements Statement" is as follows: The system shall store car information such as gas_quantity. A user shall be able to start the car and get and set car information.

The steps to create the requirements statement using With Class are listed below.

>> Create a directory for Car Products, e.g., c:\md Car
>> Run With Class from Windows
>> Select "Generate - Edit File"
>> Enter the file name, e.g., c:\car\carreqs.txt
>> In the Edit Box, enter the text for the requirements
>> In the Edit Box, select "File - Save"
>> In the Edit Box, select "Exit"

Creating the Drawing

The drawing is a graphic, visual representation showing user and other entities listed in the requirements statement, interviews, and other information. The drawing for the car class is shown in file **cardraw.omt.**

The steps to create the drawing using With Class are listed below.

>> Create a directory for Car Products, e.g., c:\md car
>> Run With Class from Windows
>> Select "File - New"
>> Select "File - SaveAs" e.g., c:\car\cardraw.omt
>> Select and place drawing icon, e.g., rectangle, circle, line, and arrow.
>> For labels, select the text icon, enter the text, and place the text
>> Select "File - Save" to save the diagram
>> Select "Print" to print the diagram

Creating the Data Dictionary Listing Classes

The data dictionary lists and describes key O-O entities and terms. One form of the data dictionary lists and describes the classes in a system. The data dictionary listing classes may be created using With Class or any word processor. The data dictionary for the car is as follows:

Data Dictionary Listing Classes
User - Sends messages to the car.
Car - Manages car information.

The steps to create the data dictionary listing classes using With Class are listed below.

>> Create a directory for Car Products, e.g., c:\md Car
>> Run With Class from Windows
>> Select "Generate - Edit File"
>> Enter the file name, e.g., c:\car\carinfo.txt
>> In the Edit Box, enter the text for the data dictionary
>> In the Edit Box, select "File - Save"
>> In the Edit Box, select "Exit"

Creating the Class Diagram

The class diagram is a graphic, visual representation showing classes with their attributes, operations, and connections as described below. Using With Class, it is helpful to understand C++ to be able to enter attributes (C++ data members) and operations (C++ functions) in C++ format as described below. However, you may enter attribute names and operation names and accept the With Class defaults. The class diagram is shown in file **car.omt**.

The steps to create the class diagram using With Class are listed below.

>> Create a directory for Car Products, e.g., c:\md car
>> Run With Class from Windows
>> Select "File - New"
>> Select "File - SaveAs" e.g., c:\car\car.omt
>> Select Class Icon
>> Enter the class name, e.g., Car
>> Enter each attribute in the form <Class/Type> <Attribute Name>, e.g., int gasQty and select Add
>> Enter each operation in the form <return class/type> <operation name> <argument class/type argument name>, e.g., int getGasQty () and void setGasQty (int aGasQty) and select Add
>> Double click OK to create the class
>> Place the class symbol on the page
>> Select "File - Save" to save the diagram
>> Select "Print" to print the diagram

Creating the Class Specification

The class specification lists important text information for each class. It is a key documentation product to document a class. Its purpose is to state adequate information to document and to program each class. The form of the class specification used in this tutorial is adapted from the Booch class specification presented in "Object Oriented Design with Applications" by Grady Booch. He presents the key aspects of each class. **Description/Responsibility** provides general class information, purpose, roles, essential behavior, and responsibilities. **System/Subsystem** states the enclosing system or subsystem. **Superclasses** states the superclasses of the class. **Visibility** of a class indicates whether the class is exported, private, or imported relative to the enclosing system or subsystem. **Cardinality** of a class indicates how many objects (instances) of the class are permitted, i.e. 0, 1, or N (a number).   **Qualifications** denote language specific information such as names of keys or generic parameters. **Concurrency** documents if objects of the class are sequential or concurrent (blocking or active). **Persistence** documents if objects of the class will retain their values when the program is not running, e.g., transitory and persistent. **Space** documents the execution size of the objects of the class, e.g., relative units (small, large) or actual memory units (bytes). **Applicable Documents** states file names and other references for drawings, block diagrams, class diagrams, state diagrams, source code files, etc. **Remarks** includes other relevant information. Transformations and correctness assertions may be placed in the description or remarks section.

The class specification holds information on each attribute and operation. For each attribute, state the attribute name, attribute type or class, initial value, minimum value, maximum value, constraints (limits or restrictions), access (public, protected, private), qualification (language specific information e.g. C++ friend, const, static), and a narrative description. For example, the following is the information on the attribute gas_quanity: attribute name - gas_quantity, attribute type or class - float, initial value - 0.0, minimum value - 0.0, maximum value - 99.0, constraints - must be within minimum and maximum value, description - gas_quantity is the amount of gasoline in the car available for use.

The Car Class documentation from With Class is shown below. This is an updated class specification referred to as a data dictionary using With Class.

---- Class Information ----
- Car -
Description - Car Class stores car information for a user to start a car and get and set car information.
System/Subsystem - User - Car System (Program)
Superclasses - Vehicle
Visibility -- private
Cardinality -1
Concurrency - sequential
Persistence - transitory
Space - small
Applicable Documents - Block Diagram - c:\car\carblock.omt; Class Diagram - c:\car\car.omt; Class Specification - c:\car\carspec.dic; State Diagram - c:\car\carstate.sm; C++ source code c:\car\car.cpp
Remarks - Transformation and correctness assertions are to be added.

----Attributes (C++ Data Members)
float gasQty
float maxGasQty
float minGasQty

----Operations (C++ Functions)
float getGasQty ()

```
void setGasQty (float aGasQty)
void start ()

------Relation Information-----
Superclass Vehicle
```

The steps to create the class specification using With Class are listed below.

>> Run With Class from Windows
>> Select "File - Open" e.g., c:\car.omt
>> Double click on a class, e.g., Car
>> Select "Info"
>> Enter class documentation, e.g., description, visibility, concurrency, cardinality, persistence, etc.
>> Select "Generate - Make Data Dictionary" and enter the dictionary file name, e.g., c:\car\carspec.dic
>> Select "Generate - Edit File" and enter the dictionary file name, e.g., c:\car\carspec.dic
>> In the Edit Box, select "File - Exit"

Generating C++ Code Using With Class

When the class diagram is completed then C++ may be generated. The generated C++ code from With Class is shown below.

```
class  Car
{
        float minGasQty;
        float maxGasQty;
        float gasQty;

public:
        void start ();
        void setGasQty (float aGasQty);
        float getGasQty ();
        Car(){}
        ~Car(){}
};

#include "Car.h"

void   Car::start ()
{};
```

```
void   Car::setGasQty (float aGasQty)
{};

float  Car::getGasQty ()
{};
```

The steps to generate C++ code using With Class are listed below.

>> Run With Class from Windows
>> Select "File - Open", e.g., c:\car\car.omt
>> Select "Generate - Options" and select C++ code generation options
>> Select "Generate - Generate Code"
>> Select "Generate - Edit File" and select a ".h" or ".cpp" file for review

The following are the steps to compile the C++ source files in Borland C++.

>> Run Borland C++ (BCW) from Windows
>> Select "Project - Open Project"
>> Enter a project file name, e.g., carproj.prj
>> Select "Project - Add Item"
>> In the Directories Box, change the directory where the C++ files are located, e.g., Car Directory
>> Enter *.cpp in the File Name Box
>> Select the .cpp files, e.g., main.cpp and car.cpp and click on the "Add Button" to add each file to the project
>> Select "Done"
>> Select "Options - Directories" to update the Include Directories List
>> In the Include Directories Box, add the directory where the C++ files are located, e.g., c:\car
>> Select "OK"
>> Select "Compile - Compile" to compile the C++ source code
>> Select "Run - Run" to compile, link, and execute

To execute the C++ source code with messages, you must update the main module with a car object declaration and messages to the car object.   A sample C++ main is as follows:

```
#include "car.h"
main ()
{
        Car car11;                                      //object declaration
        car11.setGasQty (10.0);                                 //function call
        car11.start ();                                         //function call
        int aGasQty = car11.getGasQty ();                       //function call
        return (0);
}
```

The following are the steps to update the main function and the compile and run the whole program from within Borland C++.

>> Select "File - Open"
>> Select the main function, e.g., c:\car\main.cpp
>> Enter the C++ statements for the main, e.g., statements shown above
>> Select "File - Save"
>> Select "Compile - Compile" to compile the main function
>> Select "Run - Run" to compile, link, and execute the program

Reverse Engineering a Class Diagram from C++ Code Using With Class

After C++ code has been generated from a class diagram, many additions must be made to the C++ code for messages, transformations (rules, expressions, equations, algorithms), correctness assertions (preconditions, postconditions, invariants), comments, etc. Later in a project you may desire to create a class diagram from the code you're working on. This is called reverse engineering. It is the creation of a diagram from source code. Both With Class and StateMaker have this capability.

Using reverse engineering the steps to create a class diagram from C++ source code is as follows:

>> Ensure all applicable C++ .h and .cpp files are in a single directory
>> Run With Class from Windows
>> Select "Generate - Reverse"
>> Choose the directory of the C++ files
>> Select "File - SaveAs" xxx.omt to save the new diagram

Creating a Class Diagram Showing Association

Association is a link between objects. The basic questions are "What are the association connections? For each object, what are associated objects? Our goal is to identify association connections for an understandable class structure.

The following are the steps to identify association connections.

1. Make a drawing or physical representation, e.g. a car, a registration form.

2. Identify the objects and their classes, e.g. a car, a motor.

3. For each object, identify association connections with the question "For each object are the associated objects?"

4. Identify the multiplicity of each association connection with the questions "This object is associated with zero, one or many of the other object?" and "This association is either optional or required?"

5. Identify if the association is an association ("has a") or an aggregation ("part of"). When in doubt assume ("has a").

6. If available, check message connections because objects with message connections generally have association connections.

7. Create a class diagram, data dictionary listing classes, class specification, and prototype.

The association connection is important for the following reasons.
- Association is the most fundamental connection of real world physical things and S/W entities.
- Helps develop highly cohesive objects where all associated objects are linked to accomplish a purpose.
- Implements dependency among objects - a object may be dependent upon its associated objects to accomplish its functions. - Implements visibility among objects - an object has visibility for (sees) its associated objects so that it can send messages to its associated objects. An assembly has visibility for its parts so that it can send messages to its parts.
- Implements encapsulation to reduces the number of global objects which may be called from other objects.

The file **carassoc.omt** is an example of a class diagram showing association using With Class.

The steps to create the class diagram showing association using With Class are listed below.

>> Create a directory for Car Products, e.g., c:\md car
>> Run With Class from Windows
>> Select "File - New"
>> Select "File - SaveAs" e.g., car.omt
>> Select Class Icon
>> Enter the class name, e.g., Car. Enter class information
>> Double click OK to create the class
>> Place the class symbol on the page
>> Enter the class name, e.g., User. Enter class information
>> Double click OK to create the class
>> Place the class symbol on the page
>> Select Relationship Icon
>> In the Relationship Dialog Box, select Association and either One to One or One to Many. Enter a variable name, e.g., aCar
>> Attach the relationship from one class to another
>> Select "File - Save" to save the diagram
>> Select "Print" to print the diagram

Creating a Class Diagram Showing Aggregation

Aggregation is the strong form of association that is a "part of" or "bill of materials" connection which shows the following:
- transitivity (if A is part of B and B is part of C, then A is part of C),
- antisymmetric (if A is part of B, then B is not part of A),
- propagation (sharing of common operations and attribute values from the aggregate to the part possibly with modification).

The key questions to find an aggregation connection from "O-O Modeling and Design" by Rumbaugh et al are:

- Would you use the phrase "part of"? A paragraph is "part of" a chapter.

- Are some operations on the whole automatically applied to its parts? Copy applies to chapter and paragraph.

- Are some attribute values from the whole applied to all or some parts? Chapter title applies to paragraph.

- Is their intrinsic asymmetry where one object is subordinate to other objects? A paragraph is subordinate to a chapter.

An aggregation connection requires defining additional semantic rules particularly for the creation, copy, and deletion of objects, e.g. do you automatically delete the part when the assembly is deleted?

The file **caraggr.omt** is a class diagram showing aggregation using With Class.

The steps to create the class diagram showing aggregation using With Class are listed below.

>> Create a directory for Car Products, e.g., c:\md car
>> Run With Class from Windows
>> Select "File - New"
>> Select "File - SaveAs" e.g., car.omt
>> Select Class Icon
>> Enter the class name, e.g., Car. Enter class information
>> Double click OK to create the class
>> Place the class symbol on the page
>> Enter the class name, e.g., Motor. Enter class information
>> Double click OK to create the class
>> Place the class symbol on the page
>> Select Relationship Icon
>> In the Relationship Dialog Box, select Aggregation and either One to One or One to Many. Enter a variable name, e.g., aMotor
>> Attach the relationship from one class to another
>> Select "File - Save" to save the diagram
>> Select "Print" to print the diagram

Creating a Class Diagram Showing Generalization Specialization

Generalization specialization indicates a commonality between superclasses and subclasses. The basic questions are "What are the generalization specialization connections?" "As a generalization class (superclass) what are the specialization classes (subclasses)?" "As a specialization class (subclass) what are the generalization classes (superclasses)?" Our goal is the effective use of generalization specialization for reusability, code sharing and extendibility.

The following are the steps to identify generalization specialization connections.

1. Make a drawing or physical representation, e.g. a car, a customer form

2. Identify the classes, e.g. Car, Customer.

3. For each class, identify generalization specialization connections with the questions "As a superclass what are the subclasses?" and "As a subclass what are the superclasses?"

4. Ask the following questions. "What are the general attributes, operations, and exceptions that may be shared (inherited) in subclasses?" " What are the specific attributes, operations, and exceptions that ought to be refined, added, or removed from the subclasses?" "Is there a single or multiple superclasses?" , "Is association more appropriate than generalization specialization?" "Is there a single "cosmic" superclass named Object?"

5. Identify polymorphic (same name) operations. For example, a superclass and subclass may have an operation named "start ()". The implementation of the start operation is refined and specialized in the subclass.

6. For each generalization specialization connection, identify if the connection is sharing with implementation inheritance or supertype inheritance (compatible behavior).

7. Create or update the class diagram, data dictionary listing classes, class specifications, and prototype.

An example of a class diagram showing generalization specialization using With Class is shown in **cargen.omt**.

The steps to create the class diagram showing generalization specialization using With Class are listed below.

>> Create a directory for Car Products, e.g., c:\md car
>> Run With Class from Windows
>> Select "File - New"
>> Select "File - SaveAs" e.g., car.omt
>> Select Class Icon
>> Enter the class name, e.g., Vehicle. Enter class information
>> Double click OK to create the class
>> Place the class symbol on the page
>> Enter the class name, e.g., Car. Enter class information
>> Double click OK to create the class
>> Place the class symbol on the page
>> Select Relationship Icon

>> In the Relationship Dialog Box, select Inheritance
>> Attach the relationship from one class to another, e.g., from Vehicle to Car
>> Select "File - Save" to save the diagram
>> Select "Print" to print the diagram

Modeling Classes in the Dynamic Model

Description of the Dynamic Model

The dynamic model corresponds to the OMT Dynamic Model. The major questions are "What is the stimulus response behavior of a system or object?" and "What occurs over time in the system or object?" In the dynamic model, we define stimulus response behavior and the time oriented sequence (scenario) of messages.

An **event** is something that happens at a point in time that is a stimulus for action. Sample events are "user presses the start button", "temperature reaches 100 degrees", "user selects a menu item - print". An event has no time duration. An event is a stimulus for an action. Generally events are expressed as nouns or noun phrases. The above sample events are expressed as noun phrases. Sometimes events are expressed as commands or requests for action using verbs or verb phrases. Either expressed as a noun or verb, an event happens at a point in time that is a stimulus for action. An **action** is an operation in response to an event. Sample actions based upon the sample events are "start motor", "sound temperature alarm", and "execute menu item - print".

Stimulus response behavior may be expressed in terms of events and actions. In simple cases, stimulus response behavior can be expressed in a decision table where each event always results in the same action. For example, in an operating car, pressing the horn button always result in activating the horn sound. In more the complex cases, stimulus response behavior is "state based". This means that a system or object has modes or states in which an event has different actions based upon the current mode or state. For example the stimulus response behavior of a car is "state based". When a user presses the start button, the resulting action is to start the car in the "gas OK state". When a user presses the start but in the "gas Not OK state", then nothing occurs. State based behavior is modeled using state diagrams.

In the dynamic model, we are particularly interested in identifying and describing messages and the time ordered sequence of messages through a system. A **message** invokes an operation. Event messages are particularly important. An **event message** is a stimulus command or message that is associated with an event. Sample event messages are "start ()", "evaluate_temperature ()", and "print_document (). Event messages are the stimulus for the system or object to respond. Event messages represent the basic commands and functionality of a system.

The dynamic model is a "movie" of a system or object showing the stimulus response behavior of a system or class over time. It shows the sequence of messages over time. In the car example, the dynamic model is a movie of the car with the sequence of messages over time. The event message "start ()" to a car object results in message "start_motor ()" to the motor object.   In the dynamic model, we create class diagrams showing messages (event flow diagrams), messages scenarios, and state diagrams.

Steps to Create the Dynamic Model
In the dynamic model we have specify stimulus response behavior and the time sequence of event and response messages. Follow these steps.

Step 1 - Create a **message diagram** (event flow diagram).

Step 2 - Create a **message scenario** starting with some external user action or event. A message scenario lists messages in a time ordered sequence through a system from first message to last message.

Step 3 - Optionally, create a **state diagram** for each class with complex state based behavior.

Step 4 - Generate C++ source code and update the source code with messages. Compile and execute the program.

Creating the Message Diagram (Event Flow Diagram**)**
Once operations have been identified, then a message diagram can be created. It is equivalent to the event flow diagram presented by Rumbaugh et. al. in "Object-oriented Modeling and Design". Generally classes with association connections also have message connections. Most O-O methodologies have an arrow notation for messages, e.g., Booch, Coad/Yourdon, and Shalaer/Mellor. OMT from Rumbaugh et. al. uses the message symbol (arrow) in separate event flow diagrams not on class diagrams. Coad and Yourdon in "Object Oriented Analysis" use the message symbol on class diagrams. One way to show

messages on the class diagram using With Class is to use the arrow symbol in the drawing toolbox. The message diagram (event flow diagram) is shown in file **carmsg.omt**.

The basic questions are "What are the message connections?" "As a requester, what are the servers?" "As a server what are the requesters?" The goal is the effective use of message connections for loose message coupling for independent changeable modules with no undesirable side-effects.

The following are the steps to identify message connections.

1. Make a drawing or physical representation, e.g. a car, a car registration form.

2. Identify the objects and their classes, e.g. a car, a motor.

3. For each object, identify message connections with the questions "As a requester what are the servers?" and "As a server what are the requesters?"

4. If available, check the association connections because objects with association or aggregation connections generally have message connections.

5. Create a class diagram with classes (event flow diagram).

An example of a message diagram (event flow diagram) using With Class is shown in file carmsg.omt.

The steps to create the Message Diagram (Event Flow Diagram) using With Class are listed below.

>> Create a directory for Car Products, e.g., c:\md car
>> Run With Class from Windows
>> Select "File - New"
>> Select "File - SaveAs" e.g., c:\car\carmsg.omt
>> Select Class Icon
>> Enter the class name, e.g., Person. Enter class information
>> Double click OK to create the class
>> Place the class symbol on the page
>> Enter the class name, e.g., Car. Enter class information
>> Double click OK to create the class
>> Place the class symbol on the page

>> Select the Arrow Icon from the Drawing Toolbox
>> Attach the relationship from one class to another
>> Select "File - Save" to save the diagram
>> Select "Print" to print the diagram

Creating the Message Scenario

The message scenario shows a full sequence of messages generally starting with an event message. The message scenario may be created with any text editor or word processor. With Class includes a simple text editor that can be used to create the requirements specification.

The sample "Car Message Scenario" is as follows:

Seq-	Sender	Receiver Object.Invoked Operation	Output Parameter
uence	Object	(Input Parameter Class & Object)	(Class & Object)
1	aUser	aCar.setGasQty (Integer aGasQty)	None
2	aUser	aCar.start ()	None
3	aCar	aMotor.start ()	None

Creating the State Diagram (State Transition Diagram)

Some systems and classes have very simple control in which each event message always results in the same actions or responses. There are no states or modes of behavior. The Customer Class is an example of a class with very simple control. The message "GetCustomer ()" always results in the same action. This can be modeled as a decision table.

However, systems and objects may have states, e.g., modes of behavior. Each event message may result in different actions or responses depending upon the current state. An event message may result in a transition to a new state. A system or class with states is called a **Finite State Machine**.   A simple example of a finite state machine is a toggle switch. The toggle switch has an operation "toggle".   The event message to invoke the toggle operation is "theToggleSwitch.toggle ()". The toggle operation has different actions depending whether theToggleSwitch is in the OnState or in the OffState. In the OnState the action is "turnOn". In the OffState the action is "turnOff".

The car class could also be modeled as a finite state machine. The car has an operation "start". The event message to invoke the start operation is "start ()". The start operation has different actions depending whether the car is in the GasOK state or the GasNotOK state. In the GasOK state, the action of the start operation is to startMotor (). In the GasNotOK state, the action of the start operation is doNothing ().

The purpose of the State Diagram (State Transition Diagram) is to specify stimulus response logic for a system or class of objects. It specifies the pattern of event messages, conditions, actions, and states. The basic steps to create the state diagram are:

1 - Identify a class, e.g., Car (finite state machine) that has states, e.g., modes of behavior.

2 - Identify the states, e.g., GasOK or GasNotOK. A **state** represents a mode of behavior that has a unique combination of event messages, conditions, actions, and next state. A state is static, i.e. waiting for an event message to arrive. While in a state, a defined set of rules, laws, and policies apply. A state is like a manager or coordinator that knows how to respond to each event message according to his rules, laws, and procedures. Identify the **initial state** that is entered upon creation.

3 - Identify the **event messages,** any stimulus messages to an object of a class e.g., start () that results in some action and that may result in a transition to a new state. Identify any **parameters** that are passed in the event message.

4 - Identify the **actions,** e.g., startMotor () and doNothing () in response to an event message. Actions include updating an attribute, sending a message, or similar action.

5 - Identify the conditions that affect the stimulus response logic. A **condition** is a guard or boolean expression signifying OK or NOTOK that are used in IF Condition = True THEN DoSomeAction. Examples of conditions in a Temperature Class might be "temperature high" and "temperature OK".

6 - Identify the transitions. A **transition** is a unique pattern of an event message, conditions, actions, and a destination state. For each state identify applicable event messages. Then for each event

message identify the applicable conditions, actions, and the destination state.

7 - Walk through the state diagram by sending each event message to ensure the correct transitions.

8 - Prototype, test, and iterate.

The state diagram for the Car class is shown in file **carstate.sm**.
Note:You must have StateMaker to view this file.

The steps to create the state diagram using StateMaker are listed below.

>> Create a directory, e.g., c:\md car
>> Run With Class - StateMaker from Windows
>> Select "File - SaveAs" c:\car\carstate.sm
>> Select the State Icon
>> Enter the State Name, e.g., GasOK
>> Double Click OK
>> Select the State Icon
>> Enter the State Name, e.g., GasNotOK
>> Double Click OK
>> Select the Transition Icon
>> Enter the Event, e.g., start ()
>> Enter the Action, e.g., startMotor ()
>> Double Click OK
>> Connect two states together with the transition
>> Select "Compile - To C" to generate C code
>> Enter the file name for the C source code, e.g., c:\car\carstate.c
>> Select "File - Save" to save the diagram

Modeling Classes - The Functional Model

Description of the Functional Model

The functional model is the set of transformations and correctness assertions, e.g., preconditions, postconditions, and invariants. A **transformation** is a description of how a data value may be correctly changed in a formula, expression, table, etc. An **assertion** is a rule or expression for correctness, e.g., a data value must always be greater than zero. An **operation precondition** is a rule or expression that must be satisfied before the execution of an operation for correct results. An **operation postcondition** is a rule or expression that is satisfied upon the correct execution of an operation.   An **invariant** is a general rule or expression that must be satisfied at all times by all applicable operations. An **exception** is an abnormal execution error condition, e.g. list_full_error or gas_empty_error that may be raised to signal that an operation cannot be executed correctly. In the car example, the start operation has a precondition and postcondition. The precondition for the start operation is that the gas_quantity must be greater than zero. The postcondition of the start operation is that the car motor is running. The invariant is that gas_quantity must be equal to or greater than zero and equal to or less than the maximum_gas_quantity. In the functional model, we update class specifications with transformations and correctness assertions.

In the class specification include an operation specification for each operation. This operation specification is adapted from "Object-Oriented Design with Applications" by Grady Booch. State information on each of the following as required: operation name, responsibilities (role, purpose, and essential behavior), classification (constructor, destructor, modifier, selector), access (public, protected, private, or implementation), implementation language qualification (virtual, static, const), input parameters (class and object names), output parameters (return class name), preconditions, postconditions, invariants, exceptions (error conditions to signal an execution problem), time complexity (time budgeted to complete an operation), space complexity (amount of storage consumed by invoking the operation), concurrency (sequential, guarded, or synchronous), transformation rules (formula, expression, table), and remarks.

In the functional model, we specify how data is transformed in a system. Follow these steps.

Step 1 - Update class specifications with transformations, e.g., formulas, expressions, equations and correctness assertions, e.g., preconditions, postconditions, and invariants.

Step 2 - Update the C++ source code to reflect the transformations and correctness assertions. Compile and execute the program.

Steps to Create the Functional Model
In the functional model, we specify how data is transformed in a system. Follow these steps.

Step 1 - Update class specifications with transformations, e.g., formulas, expressions, equations and correctness assertions, e.g., preconditions, postconditions, and invariants for each operation.

Step 2 - Update the C++ source code to reflect the transformations and correctness assertions. Compile and execute the program.

A key documentation product to document a class is the class specification. Its purpose is to state adequate information to document and to program each class. The form of the class specification used in this tutorial is the Booch class specification presented in "Object Oriented Design with Applications" by Grady Booch. He presents the key aspects of each class. **Description** provides general class information, class purpose, and class responsibilities. **System/Subsystem** states the enclosing system or subsystem. **Superclasses** states the superclasses of the class. **Visibility** of a class indicates whether the class is exported, private, or imported relative to the enclosing system or subsystem. **Cardinality** of a class indicates how many objects (instances) of the class are permitted, i.e. 0, 1, or N (a number). **Concurrency** documents if objects of the class are sequential or concurrent (blocking or active). **Persistence** documents if objects of the class will retain their values when the program is not running, e.g., transitory and persistent. **Space** documents the execution size of the objects of the class, e.g., relative units (small, large) or actual memory units (bytes). **Applicable Documents** states file names and other references for block diagrams, class diagrams, state diagrams, source code files, etc. **Remarks** includes other relevant information. Transformations and correctness assertions may be placed in the description or remarks section.

Specifying Transformations and Correctness Assertions

To describe transformations and correctness assertions using With Class, update each class specification. A sample remarks section of the car class specification is shown below:

Class Car - Transformations and Correctness Assertions

```
void start ()
        transformation - gas_quantity = gas_quantity - .1;
        precondition - gas_quantity > 0
        postcondition - car is running
        invariant - qas_quantity must be  equal to or greater than 0 and
                        equal to or less than max_gas_quantity
```

The steps to update the class specification for transformations and correctness assertions are listed below.

>> Run With Class from Windows
>> Double click on a class
>> Select "Info"
>> Enter transformations and correctness assertions in the Description or Remarks section
>> Select "Generate - Make Data Dictionary" and enter the dictionary file name, e.g., c:\car\carspec.dic
>> Select "Generate - Edit File" and enter the dictionary file name, e.g., c:\car\carspec.dic
>> In the Edit Box, select "File - Exit"