Object-Oriented Modeling Using With Class

by Richard C. Felsinger, RCF Associates,
960 Scottland Dr, Mt Pleasant, SC 29464 Tele 803-881-3648 (Voice & Fax)
E-mail - Internet 71162.755@compuserve.com or CompuServe 71162,755
Comments and suggestions on this tutorial are solicited.

1. Introduction

The purpose of this tutorial is to present "step by step" instructions on how to use With Class CASE tool to create object-oriented diagrams, fill in forms, and automatically generate C++ to model an object-oriented system. This tutorial presents the Object Modeling Technique (OMT) presented in “Object-Oriented Modeling and Design” by James Rumbaugh, Michael Blaha, William Premerlani, Frederic Eddy, and William Lorensen. All specifications are based upon “Object-Oriented Design with Applications” by Grady Booch. Due to With Class support for multiple methodologies and languages, there are a few minor differences between the specifications in this tutorial and the With Class fill in specification forms .

To create the various diagrams, text specifications, and C++ code in this tutorial, you should have access the following Windows programs:
- With Class CASE tool from MicroGold Software to create class diagrams, text specifications, and C++ code files,
- Any Windows drawing tool to create drawings, block diagrams, event flow diagrams, and other diagrams,
- StateMaker State Diagramming Tool from MicroGold Software to create state diagrams and C++ code files,
- Any C++ compiler environment to compile, link, and run C++ code files.

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 as 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.

Inputs	**Process of O-O Modeling**	**Outputs**
System	Object Model	Diagrams
Requirements	Dynamic Model	Text Specifications

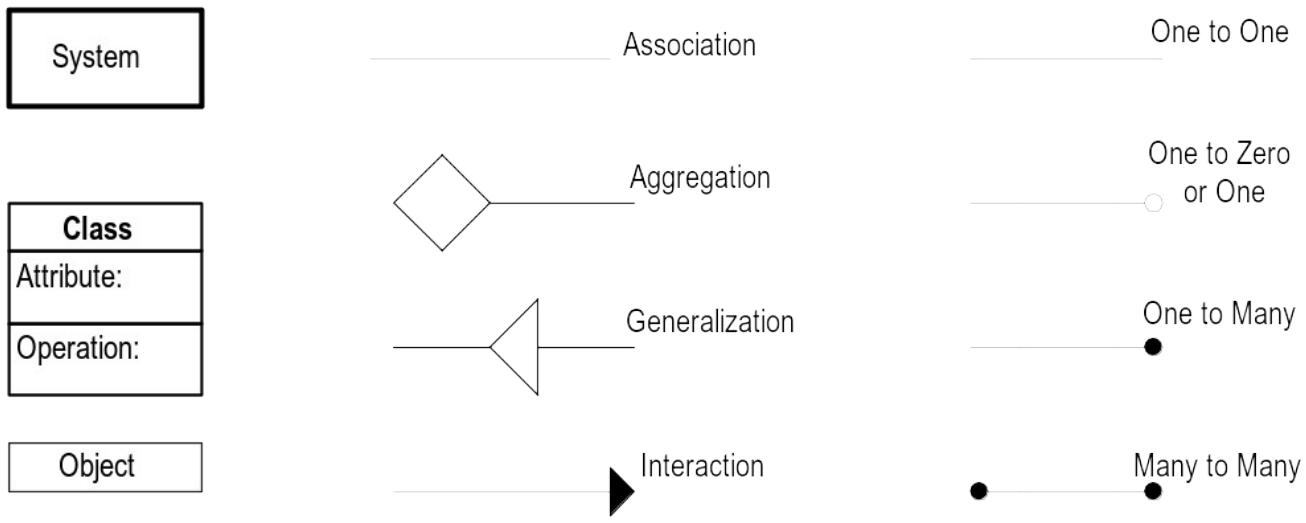
Functional Model Code & Prototypes

Tutorial Objectives - The primary objectives of this tutorial is to present how to model and prototype in C++ a simple system using With Class. The example is a Transportation System consisting of a simulated car. With the car, a user can start, operate, stop, add a passenger, remove a passenger, make a cellular phone call, and receive a cellular phone call. A simulation of a physical car is used since it is easy to understand the basic concepts of O-O modeling.

Three Primary Models (Views) in O-O Modeling - 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 will view the class from the three points of view. In the object model we will create the following: system requirements statement, drawing, class diagram, and class specification. In the dynamic model we will create the object interaction diagram (messages) and state diagram. 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. Using these three points of view, we'll first model a car to provide a physical example. A physical example is easiest to understand the basic concepts. The following are the major graphic symbols used to model the system, classes, and objects in this tutorial.

Entity **Relationship** **Cardinality**



2. Modeling Classes in the Object Model

2.1. Description of the Object Model

The object model corresponds to the OMT Object Model. The major question is "What are the classes in the system?" We are concerned with **O-O entities**, basic building blocks inside a system, e.g. class and object. We are concerned with **O-O relationships**, links or connections between O-O entities, e.g. association, aggregation, and generalization specialization. As described below the object model consists of the following documentation products: 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 relationships between classes, e.g. association, aggregation, and generalization specialization. The interaction relationship is specified in the dynamic model. We specify **constraints** which are rules that restrict or limit the values that objects, attributes, and relationships can have. A constraint is that a car can only have one motor and that the gasQuantity 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 transportation system consists of a class diagram, class specification, and data dictionary.

The object model is important for several reasons. First, it identifies the key entities and relationships in a system. Classes and relationships between classes are the building blocks and connections in a system. Second, it provides the framework of classes to build the system. Third, it ties together all the models. Fourth, a CASE tool like With Class can generate an extensive

amount of code based upon the object model. The CASE tool generates code from the class diagram and class specifications in the object model.

2.2. Steps to Create the Object Model:

Step 1 - To start, create the **system requirements statement, system block diagram**, **system drawing,** and **system specification** to describe the system as a whole.

Step 2 - Create the **class diagram**. The class diagram is a visual, graphic representation of classes showing attributes, operations, and relationships.

Step 2a - Identify **classes**. A class defines objects that represent people, places, and things in the system.

Step 2b - Identify **attributes**. An attribute is a characteristic, property, or component of an object. An attribute consists of an attribute name, type/class, and value.

Step 2c - Identify **operations**. An operation is an action, algorithm, or set of steps that typically uses or modifies an attribute value.

Step 2d - Identify the **relationships** for association, aggregation, and generalization specialization. These relationships represent a connection or link between two classes. Defer identification of interaction relationships (events or messages) until the dynamic model.

Step 3 - Create the **text specifications** for each class, attribute, operation, and relationship. Generate **reports** to display or print out the text specifications.

Step 3a - Create the **class specification**. The class specification provides text information about the class. Class information should include class description, superclasses, access, cardinality, concurrency, transformations and correctness assertions and other information necessary for programming or documentation. Document constraints in the class specification.

Step 3b - Create the **attribute specification**. The attribute specification provides text information about each attribute. Attribute information includes type (integer, float, string, etc.), access (private, protected, public), initial value, minimum value, maximum value, constraints (rules), and description.

Step 3c - Create the **operation specification**. The operation specification provides text information about each operation. Operation information includes the return type or class, input parameters, access (private, protected, public), preconditions, postconditions, transformation, exceptions, and description.

Step 3d - Create the **relationship specification**. The relationship specification provides text information on each relationship such as its name, constraints (rules), and implementation features. For association and aggregation relationships, the cardinality (1 to 1, 1 to many, many to many) is shown.

Step 4 - Create or update the **data dictionary**. The data dictionary lists the terms and a brief description of each term used in the model, e.g. classes, attributes, and operations.

Step 5 - Check and update consistency with other models, e.g. the dynamic model and functional model. Check all diagrams and text specifications for the consistent spelling, capitalization, and meaning of all names.

Step 6 - Generate the **prototype** - **C++ source code** using With Class. Compile and execute the program in the C++ development environment.

2.3. Creating the System Requirements Statement

The system requirements statement describes the system and its functionality. It is often referred to as the charter, functional requirements, or problem statement. The system 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 System requirements statement" is as follows: The system shall store car information such as gasQuantity. A user shall be able to start the car, operate the car, stop the car, add a passenger, remove a passenger, make a cellular phone call and receive a cellular phone call.

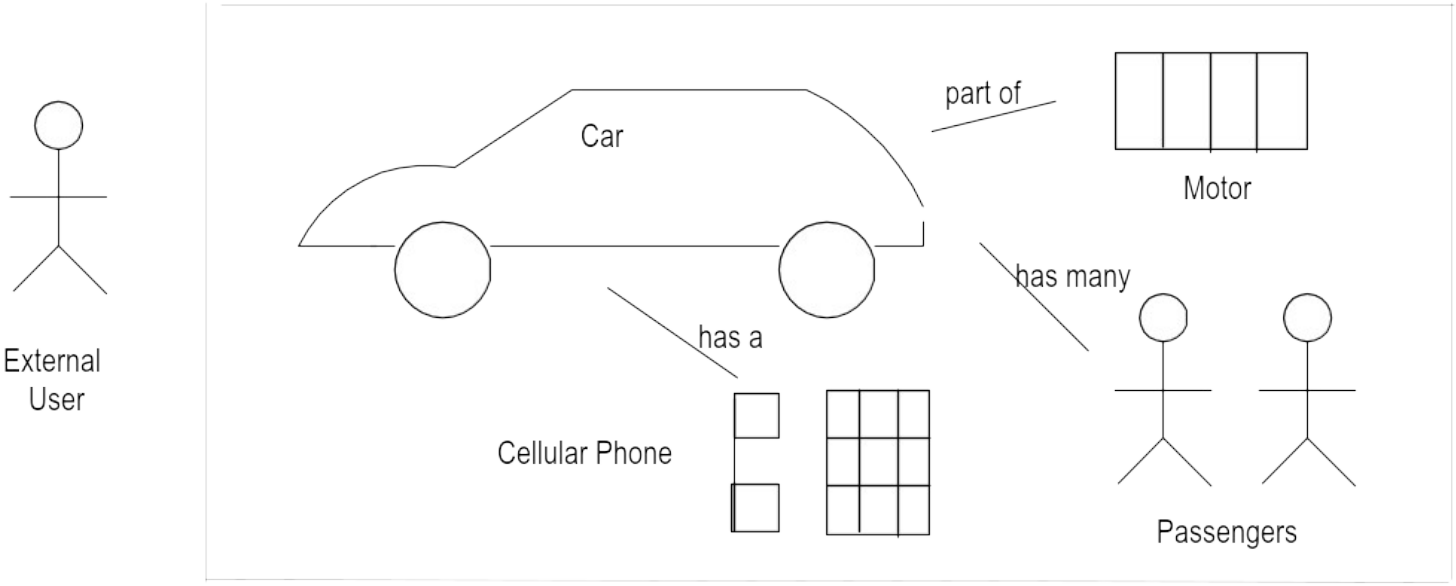
The steps to create the system requirements statement using With Class are listed below.

>> Create a directory for Car products, e.g. c:\md Car
>> Run With Class or a text editor from Windows
>> Select "File - 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"

2.4. Creating the System Drawing

The drawing is a graphic, visual representation showing user and other entities listed in the system requirements statement, interviews, and other information.

The drawing for the transportation system is shown below**.**



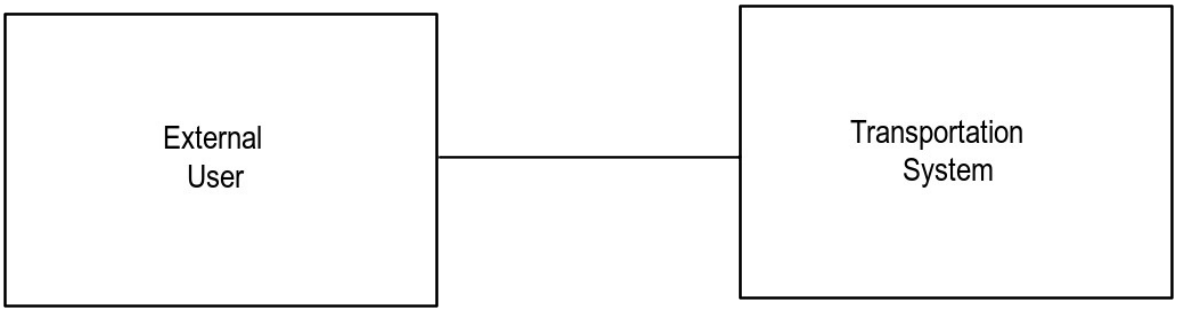
Transportation System

The steps to create the drawing using a Windows drawing program are listed below.

>> Run the Windows drawing program 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 "File - Print" to print the diagram

2.5. Creating the System Block Diagram

The system block diagram is a graphic, visual representation showing the system and interacting systems. The block diagram has a box for the system being modeled and a box for each interacting system. The system block diagram shows a very high level view of the system.



The steps to create the system block diagram using a Windows drawing program are listed below.

>> Run the Windows drawing program from Windows
>> Select "File - New"
>> Select "File - SaveAs" e.g. c:\car\carblock.xxx
>> 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 "File - Print" to print the diagram

2.6. Creating the System Specification

The system specification lists key information about the system as a whole. Grady Booch in Object-Oriented Design with Applications presents a useful system specification. His name for this specification is the “Class Category Specification”. The following specification is based upon the Booch specification.
- **System Name** states the name of the system.
- **Enclosing System** states the name of an enclosing system if the system is a subsystem within a larger enclosing system.
- **Responsibilities**, states the system operations.
- **Access** states the visibility of the system, e.g. public or private.
- **Imports** states the name of any supporting libraries.
- **Remarks** states other relevant information.

The system specification of the Transportation System is shown below.

- **System Name**: Transportation System
- **Enclosing System**: None
- **Responsibilities**: System responds to the following user commands: start, operate, stop, addPassenger, removePassenger, makePhoneCall, and receivePhoneCall.
- **Access**: public access.
- **Imports**: None.
- **Remarks**: The Transportation System is a simulation of a physical car. This is

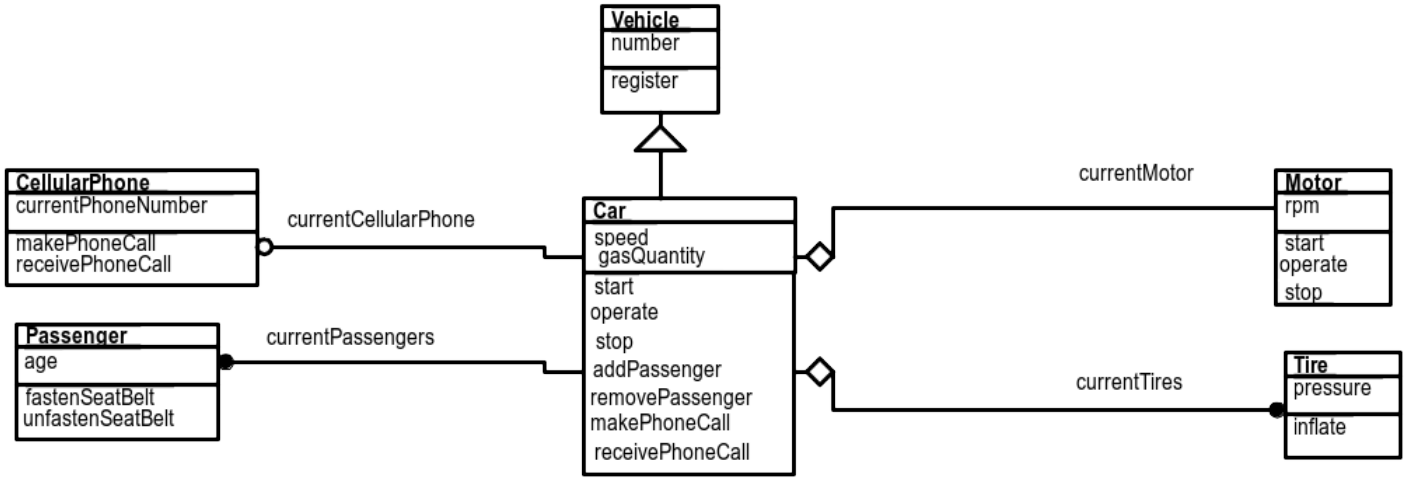
a very simple system to present an easy to understand object-oriented system.

To create the system specification, start with the system requirements statement and system block diagram. The steps to create the system specification using With Class are listed below.

>> Create a directory for Car products, e.g. c:\md Car
>> Run With Class or a text editor from Windows
>> Select "File - Edit File"
>> Enter the file name, e.g. c:\car\carreqs.txt
>> Enter the text for the requirements
>> Select "File - Save"

2.7. Creating the Class Diagram

The class diagram is a graphic, visual representation showing classes with their attributes, operations, and relationships as described below. Creating a class diagram is an iterative process of starting with a very basic diagram and then adding more and more details. The class diagram with the OMT graphic notation is shown below.



The following are the general steps to create a class diagram.

Step 1 - Identify classes, e.g. car, motor, passenger, cellular phone.

Step 2 - Enter a few attributes, e.g. speed and gasQuantity in the Car class.

Step 3 - Enter a few operations, e.g. start, operate, and stop in the Car class.

Step 4 - Enter a few relationships, e.g. aggregation between the Car class and Motor class.

Step 5 - Repeat the process adding more classes, attributes, operations, and relationships.

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

>> 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, e.g. gasQuantity and select Add
>> Enter each operation, e.g. start and operate 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 "File - Print" to print the diagram

2.8. 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.
- **Class Name** states the name of the 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.
- **Access** of a class indicates whether the class is exported, private, or imported relative to the enclosing system or subsystem. Access is also known as export control or visibility.
- **Cardinality** of a class indicates how many objects (instances) of the class are permitted, i.e. 0, 1, or N (a number).
- Implementation denotes language specific implementation information such as names of keys or generic parameters.
- **Concurrency** documents if objects of the class are sequential or concurrent. Alternatives are sequential, concurrent, blocking or active.
- **Persistence** documents if objects of the class will retain their values when the program is not running, e.g. transitory or persistent.
- **Space** documents the execution size of the objects of the class, e.g. relative units (small, large) or actual memory units (bytes).
- Abstract/Concrete documents the form of the class, e.g. abstract or concrete. An abstract class has no direct instances (objects). A concrete class has direct instances (objects).

- Generic Parameterized (Yes/No) documents if the class is a generic parameterized (template) class. A generic parameterized class serves as a template to create other classes. A generic parameterized class must have a generic parameter before it is instantiated, e.g. ParameterizedStack <int>.
- Generic Parameter documents the generic parameter of a generic class, e.g. int, float, String, etc.
- **Indexed (Yes/No)** documents if the class is indexed. An indexed class is a collection class with a lookup index, e.g. EmployeeList with the index of employeeNumber.
- Index Attributes (Type and Name) documents indexes.
- Constraints (Invariants) are rules or expressions that restrict or constrain an object, attributes, or relationships.
- **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. If applicable, include requirements, user interface, storage, and remote connectivity information.

For example, the Car Class documentation is shown.

Class Name: Car
Description: Car Class stores car information for a user to start a car and get and set car information.
System/Subsystem: Transportation System (Program)
Superclasses: Vehicle
Access: private
Cardinality: 1
Implementation: C++ class
Concurrency: sequential
Persistence: transitory
Space: small
Abstract/Concrete: abstract
Generic (Yes/No): no
Generic Parameter: N/A
Indexed (Yes/No): no
Index Attributes (Type and Name): N/A
Index: N/A
Constraints (Invariants): None
Applicable Documents: Block Diagram - c:\car\carblock.omt; Class Diagram - c:\car\car.omt; State Diagram - c:\car\carstate.sm; C++ source code c:\car\car.cpp and c:\car\car.h
Remarks: Transformation and correctness assertions are to be added.

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 "Spec"
>> Enter class documentation, e.g. description, access, concurrency, cardinality, persistence, etc.
>> Select "Generate - Generate Report and select a report script
>> Select "File - Edit File" and enter the report file name
>> Select "File - Exit"

2.9. Creating the Attribute Specification

The attribute specification holds information on each attribute. For each attribute, state attribute information as described in “Object-Oriented Design with Applications” by Grady Booch. State the following:
- **Class name** provides the name of the class.
- **Attribute name** provides the name of the attribute.
- **Attribute type or class** provides the attribute type or class.
- **Form (object/class)** states whether the attribute is an object attribute or class attribute.
- **Initial value** provides the initial value of the attribute.
- **Minimum value** provides the minimum value of the attribute.
- **Maximum value** provides the maximum value of the attribute.
- **Constraints** (limits or restrictions) provides rules, limits, or restrictions of the attribute.
- **Access** states the visibility of the attribute, e.g. public, protected, private, or implementation.
- Key (Yes/No) states if the attribute is a lookup key.
- Qualifier (Yes/No) states if the attribute is qualifier that distinguishes among the set of objects at the “many” side of an association.
- Derived (Yes/No) states if the attribute is a derived attribute that is computed from other attributes.
- **Implementation** states language specific information, e.g. C++ friend, const, static.
- **Size** states the estimated amount of memory consumed.
- **Description** provides other information about the attribute.

For example, the following is the information on the attribute gasQuantity:
- **Class name** - Car
- **Attribute name** - gasQuantity,
- **Attribute type or class** - float,
- **Form of attribute** - object attribute
- **Initial value** - 0.0,
- **Minimum value** - 0.0,
- **Maximum value** - 99.0,
- **Constraints** - must be within minimum and maximum value,
- **Access** - private,

- Key - no,
- Qualifier - no,
- Derived - no,
- **Implementation** - C++ private data member,
- **Size** - small,
- **Description** - gasQuantity is the amount of gasoline in the car available for use.

The steps to create the attribute 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
>> Double click on an attribute in the class, e.g. gasQuantity
>> Enter attribute information, e.g. description, access, etc.
>> Select "Generate - Generate Report and enter the report script

2.10. Creating the Operation Specification

The operation specification holds information on each operation. Operation information may include the information based upon the operation specification in “Object-Oriented Design with Applications” by Grady Booch.

- **Class name** states the name of the class.
- **Operation name** states the name of the operation.
- **Return type or class** states the return type or class, e.g. integer, float, string, Passenger, etc.
- **Input parameter type/class and name** state input parameters, e.g. integer anInteger, float aFloat, string aString. An operation may have several input parameters,
- **Access** states the visibility of the operation, e.g. public, protected, private, or implementation
- Abstract/Normal states whether an operation is normal or abstract. A normal operation, e.g. not abstract, has an implementation. An abstract operation is declared but not implemented in an abstract class, e.g. a C++ pure virtual function.
- **Category of operation** states the category of the operation e.g. modifier, selector, iterator, constructor, destructor.
- **Implementation** states the implementation language feature such as function or procedure.
- **Transformation** states the formula or expression of the operation.
- **Precondition** states a condition that must be satisfied for the correct transformation
- **Postcondition** states a condition that ensures that the correct transformation has occurred.
- **Exception Type and Name** states the type or class and the name of the

exception that is raised (thrown) by the operation.
- **Applicable Invariant** states the name of the class invariant (rule) that may be violated by the operation. Invariants are a part of the Eiffel language.
- **Concurrency** states whether the operation is sequential, guarded, concurrent, or multiple. See “Object-Oriented Design with Applications” by Grady Booch.
- **Time** states the estimated time to execute the operation.
- **Space** states the estimated size in memory of the operation.
- **Description** states a description of the operation.

For example, a sample operation specification for the “operate” operation is as follows:

- **Class name**: Car
- **Operation name**: operate
- **Return type or class**: none
- **Input parameter type/class and name**: integer aSpeed
- **Access**: public
- Abstract/Normal - normal
- **Category of operation**: modifier
- **Implementation:** C++ function
- **Transformation**: rpm = 50 * aSpeed
- **Precondition**: aSpeed must be between 0 and 120 mph
- **Postcondition**: rpm must be between 0 and 6000 rpm
- **Exception Type and Name Exception**: String speedError or String rpmError
- **Applicable Invariant**: None
- **Concurrency**: Sequential
- **Time**: Low
- **Space**: Low
- **Description**: operate calculates the rpm and sends the rpm to a Motor object

The steps to create the operation 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
>> Double click on an operation in the class, e.g. start
>> Enter operation information, e.g. description, access, etc.
>> Select "Generate - Generate Report and select the report script

2.11. Creating the Relationship Specification

The relationship specification holds information on each relationship. Relationship information includes the following:
- **Source Class** states initial class to define the relationship.

- **Destination Class** states the second class to define the relationship.
- Traversal Attribute Class/Type and Name states the class/type and name of the attribute which implements the relationship and which is used to traverse (get) an object of the associated or part class.
- Order by Attribute Class/Type and Name states the class/type and name of the attribute which orders the objects of the associated or part class.
- Type of Relationship (Association/Aggregation) states the type of relationship, e.g. association or aggregation.
- **Relationship Constraints** states rules or expressions that restrict or constrain a relationship such as a motor weight must be less car frame weight.
- **Relationship Cardinality** states the relationship cardinality, e.g. 1 to 1, 1 to 0 or 1, 1 to many, or many to many
- Inverse Traversal Attribute Class/Type and Name states class/type and name of the inverse traversal attribute..
- **Implementation** states the language dependent implementation of the relationship, e.g. a Motor data member implements the aggregation relationship between Car class and Motor class.
- **Remarks** states any additional information about the relationship.

For example, the following is a sample relationship specification.
- Source Class - Car
- Destination Class - Motor
- Traversal Attribute Class/Type and Name - Motor currentMotor
- Order by Attribute Class/Type and Name - N/A
- Type of Relationship (Association/Aggregation) - Aggregation
- Relationship Constraints (rules) - motor weight must be less car frame weight
- Relationship Cardinality - 1 to 1
- Inverse Traversal Attribute Class/Type and Name - Car currentCar
- Implementation - aMotor data member implements the aggregation relationship between Car class and Motor class. The inverse relationship is implemented with currentCar as a pointer to the Car object.
- Remarks.

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

>> Run With Class from Windows
>> Select "File - Open" e.g. c:\car.omt
>> Select "File - Edit File" and enter the relationship information into a text file

2.12. Creating the Data Dictionary Listing Terms

The data dictionary lists and describes key O-O entities and terms. The purpose of the data dictionary is to aid the model builder or model user to lookup and understand names and terms used in the model. As a minimum the

data dictionary lists and describes the classes in a system. The data dictionary may list and describe all modeling entities in the system including classes, objects, attributes, operations, relationships, etc. The data dictionary listing terms may be created using With Class or any word processor. The initial data dictionary for the Transportation System is as follows. Updates to the data dictionary will be provided in each of the models.

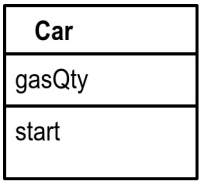
Term	**Description**
Vehicle	Class
Car	Class
Motor	Class
Passenger	Class
CellularPhone	Class
Tire	Class

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

>> Run With Class or text editor from Windows
>> Select "Generate - Generate Report and select the data dictionary report script

2.13. 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. Notice that With Class generated the class declaration and definition including the C++ constructor, destructor, and accessor functions.




```
class  Car
{
        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 - Generate Code and select a C++ code generation script
>> Select "File - 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++. The steps are very similar in Microsoft Visual C++ and other C++ environments.

>> 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

2.14. Creating a Class Diagram Showing Association

Association is a “has a” or “knows about” link between objects. The basic questions are "What are the association relationships? For each object, what are the associated objects? Our goal is to identify association relationships for an understandable class structure. The class diagram shows a 1 to 0/1 association between Car and CellularPhone and a 1 to many association between Car and Passenger.

The following are the steps to identify association relationships.

Step 1. Make a drawing or physical representation, e.g. a car, a passenger.

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

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

Step 4. Identify the multiplicity of each association relationship with the questions "Is this object associated with zero, one or many of the other object?" and "Is this association optional or required?"

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

Step 6. If available, check interaction relationships because objects with

interaction (message) relationships generally have association relationships.

Step 7. Identify operations to attach and detach associated or part objects, e.g. attachCellularPhone and detachCellularPhone.

Step 8. Create a class diagram, data dictionary listing terms, class specification, and prototype.

The association relationship is important for the following reasons.
- Association is the most fundamental relationship 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. An object may depends upon its associated objects to accomplish its functions.
- Implements access among objects - an object has access (visibility for) its associated objects so that it can send messages to its associated objects. An assembly has access to its parts so that it can send messages to its parts.
- Implements encapsulation to reduce the number of global objects which may be called from other objects.

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. CellularPhone. Enter class information
>> Double click OK to create the class
>> Place the class symbol on the page
>> Select the “1 to 0/1” Cardinality Icon
>> Select the “Association Relationship” Icon
>> Drag the relationship (pencil) cursor from the Car class to the CellularPhone class
>> Select the green/gray dot on the relationship line. Enter a traversal variable name, e.g. currentCellularPhone
>> Select "File - Save" to save the diagram
>> Select "File - Print" to print the diagram

2.15. Creating a Class Diagram Showing Aggregation

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

The key questions to find an aggregation relationship from "Object-Oriented 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 relationship 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 class diagram shows a 1 to 1 aggregation between Car and Motor and a 1 to many aggregation between Car and Tire.

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 the “1 to 1” Cardinality Icon
>> Select the “Aggregation Relationship” Icon
>> Drag the relationship (pencil) cursor from the Car class to the Motor class
>> Select the green/gray dot on the relationship line. Enter a traversal variable name, e.g. currentCellularPhone

>> Select "File - Save" to save the diagram
>> Select "File - Print" to print the diagram

2.16. 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 relationships?" "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 class diagram shows a generalization relationship between Vehicle and Car.

The following are the steps to identify generalization specialization relationships.

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

Step 2. Identify the classes, e.g. Car, Passenger.

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

Step 4. Ask the following questions. "What are the general attributes and operations that may be shared (inherited) in subclasses?" "What are the specific attributes and operations 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?"

Step 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.

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

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

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. Car. Enter class information
>> Double click OK to create the class
>> Place the class symbol on the page
>> Enter the class name, e.g. Vehicle. Enter class information
>> Double click OK to create the class
>> Place the class symbol on the page
>> Select the “Inheritance Relationship” Icon
>> Drag the relationship (pencil) cursor from the Vehicle class to the Car class
>> Select "File - Save" to save the diagram
>> Select "File - Print" to print the diagram

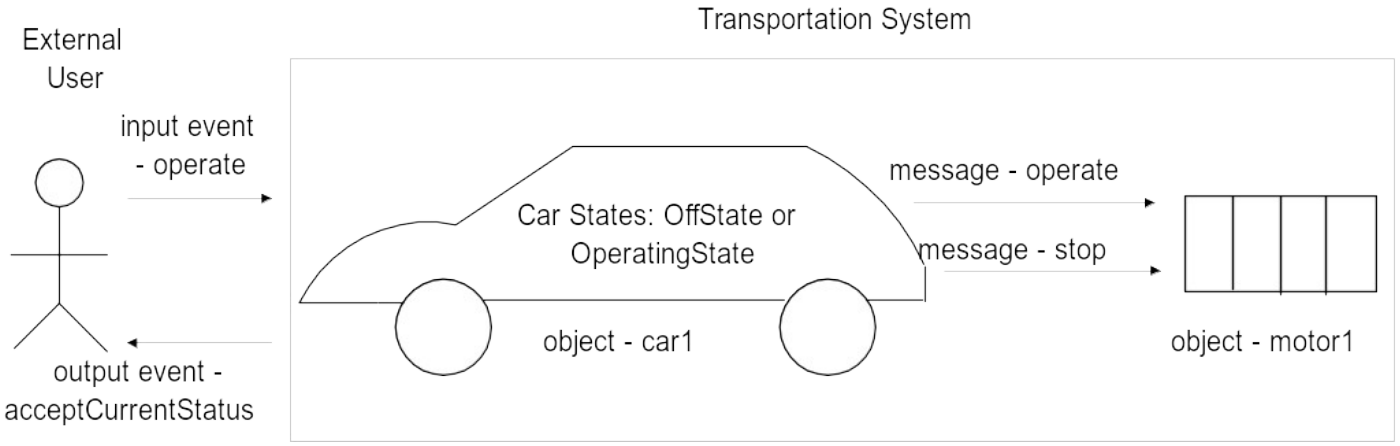
2.17. Summary

The object model is the description of classes, objects, and relationships in a system. A class is a description of a group of objects (instances) with common attributes, operations, and relationships. The object model consists of documentation products, e.g. class diagram, class specification, data dictionary, and code.

3. Modeling Interactions and States in the Dynamic Model

3.1. Description of the Dynamic Model

The dynamic model shows the stimulus response behavior of the system and objects in the system. Behavior is sum of stimuli that an entity (system or object) detects and the responses to those stimuli. As shown below, the stimulus for the system as a whole is an input event. For example, the Transportation System has an input event “start” that it detects and responds to. For an object, a stimulus is a message from one object to another object. For example, the car1 object sends the message “start” to the motor1 object. In the dynamic model we identify the stimuli, e.g. input events and messages. Then we identify the time ordered sequence of input to output events for the system and the sequence of input events to messages to output events for objects. Additionally, we are modeling state based behavior of the system and objects in the system. The major questions are "What is the stimulus response behavior of a system or an object?", "What occurs over time in a system or an object?" “What is the state based behavior of a system or an object?”



Stimulus response behavior may be expressed in terms of input events leading to output events at the system level or in terms of input events leading to messages leading to output events at the object level. In simple cases, stimulus response behavior can be expressed in a decision table where each input event always results in the same output event or message. For example, in the Transportation System, the input event start always results in the output event “acceptCurrentStatus”. 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 input event has different actions based upon the current mode or state. For example the stimulus response behavior of the car1 object is "state based". When the car1 object receives the event “operate” in the “OffState”, then there is no action taken”. However, when the car1 object receives the event “operate” in the “OperatingState”, then messages are sent to the motor1 object. State based behavior is modeled using state diagrams.

After an input event occurs, messages are sent through the system. 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. For example, the input event “start” results in a series of messages through the system.

The dynamic model is a "movie" of a system or object showing the stimulus response behavior of a system or object over time. It shows the sequence of messages over time. In the dynamic model, we create system event lists, system event flow diagrams, object interaction scenarios showing messages, object interaction diagrams showing messages, and state diagrams.

The dynamic model is important for several reasons. First, it is way to describe the stimulus response behavior of the system or object. This shows the basic functionality of the system or object. Second, it necessary to model the stimulus response behavior in terms of a sequence of messages to create an

executable program. The executable program consists of objects sending messages to other objects.

3.2. Steps to Create the Dynamic Model

In the dynamic model we have specify stimulus response behavior and the time sequence of input events, messages, and output events. Follow these steps.

Step 1 - Start with the **system requirements statement** and **system event flow diagram** listing system input and output events.

Step 2 - Create an **object interaction scenario (messages)** starting with some external user action or event. An object interaction scenario lists messages in a time ordered sequence through a system from an input event to an output event.

Step 3 - Create an **object interaction diagram (messages**) to display the time ordered sequence of messages between objects.

Step 4 - Create a **state diagram** for each class with complex state based behavior.

Step 5 - Update the **data dictionary** listing objects and states.

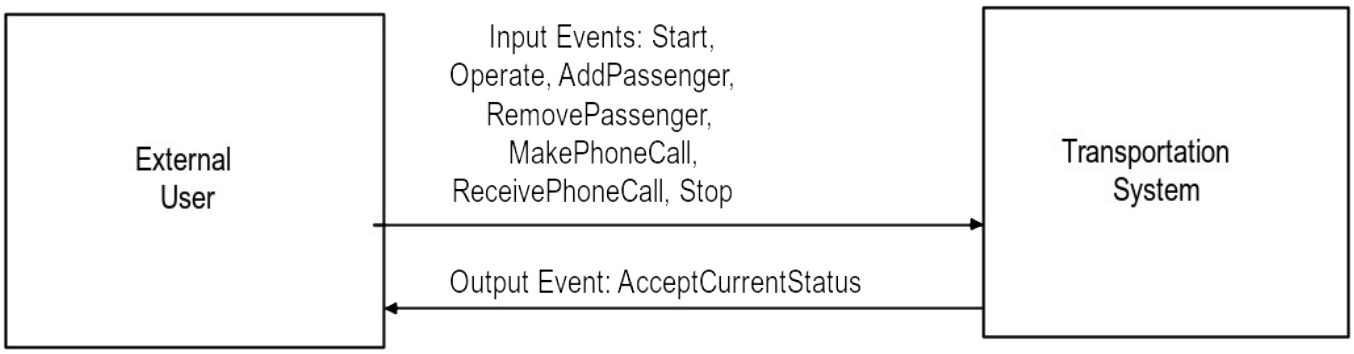
Step 6 - Create **C++ source code** and update the source code with messages. Compile and execute the program in the C++ development environment.

Step 7 - Check and update consistency with other models, e.g. the object model and functional model. Check all diagrams and text specifications for the consistent spelling, capitalization, and meaning of all names.

3.3. Creating the System Event Flow Diagram

The system event flow diagram shows the input and output events for a system. An input event is any stimulus that can be detected and responded to. The system event flow diagram shows the input and output events between the system and interacting systems. As shown below, “start” is an input event that the Transportation System detects and responds to. An output event is a stimulus from the system to an interacting system. In this simple example, the only output event is “acceptCurrentStatus”. When an input event occurs, then the Transportation System issues the output event “acceptCurrentStatus”. It is the responsibility of the External User to detect and respond, if necessary, to the output event.

The system event flow diagram is important because it is the “starting point” for the object interaction diagram. The sample system event flow diagram for the Transportation System is shown below.



3.4. Creating the Object Interaction Scenario (Messages)

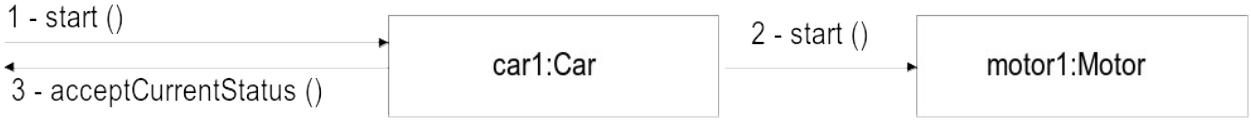
For each system input event there is a sequence of messages through the system. The object interaction scenario shows a full sequence of messages generally starting with a system input event. The object interaction scenario listing messages may be created with any text editor or word processor. With Class provides a simple text editor that can be used to create the object interaction scenario.

The sample Car object interaction scenario (messages) is as follows:

Seq-uence	**Sender Object**	**Receiver Object.Invoked Operation (Input Parameter Class & Object): return Type/Class**
1	External User	car1.start ()
2	car1	motor1.start ()
3	car1	ExternalUser.acceptCurrentStatus(String status)

3.5. Creating the Object Interaction Diagram (Messages)

An object interaction scenario lists messages for some external action or event in a time ordered sequence through a system from first message to last message. The object interaction diagram displays the messages between objects. The basic questions are "What are the input and output events at the system level, e.g. start and operate?" "For each input event, what are the messages through the system leading to output events?" The goal is the effective use of messages for loose message coupling for independent changeable modules with no undesirable side-effects. The object interaction diagram for the start input event is shown below.



The following are the general steps to identify objects and messages to create the object interaction diagram.

Step 1. Start with a drawing or physical representation and the system requirements listing events and system operations, e.g. start and operate.

Step 2. Identify the objects and their classes, e.g. car1:Car, motor1:Motor.

Step 3. For each input event or system operation, identify messages between objects possibly leading to one or more output events.

Step 4. Check the relationships on the class diagram because objects with association or aggregation relationships generally have messages.

Step 5. Create the object interaction diagram showing objects and messages.

The steps to create the object interaction diagram using a Windows drawing tool are listed below.

>> Create a directory for Car products, e.g. c:\md car
>> Run the Windows drawing editor from Windows
>> Select "File - New"
>> Select "File - SaveAs" e.g. c:\car\carmsg.omt
>> Select the rectangle or object icon (if available)
>> Enter the object and class name, e.g. car1:Car
>> Double click OK to create the object symbol
>> Place the object symbol on the page
>> Select the rectangle or object icon (if available)
>> Enter the object and class name, e.g. motor1:Motor
>> Select the arrow icon
>> Attach the arrow from one object to another
>> Select "File - Save" to save the diagram
>> Select "File - Print" to print the diagram

3.6. Creating the State Diagram (State Transition Diagram)

Some systems and objects have very simple control in which each input event always results in the same actions or responses. There are no states or modes of behavior. A customer object is an example of an object 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 may result in different actions or responses depending upon the current state. An event may result in a transition to a new state. A system or object

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 "operate". The event (message) to invoke the operate operation is "operate ()". The operate operation has different actions depending whether the car is in the OffState or the OperatingState. In the OffState, there is no action. In the OperatingState, the action is to send the message motor1.operate () or motor1.stop () depending upon the condition gasStatus.

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 events, conditions, actions, states, and activities. The basic steps to create the state diagram are:

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

Step 2 - Identify the states, e.g. OffState or OperatingState. A **state** represents a mode of behavior that has a unique combination of an event, conditions, actions, and next state. A state is static, i.e. waiting for an event 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 according to his rules, laws, and procedures. Identify the **initial state** that is entered upon creation. Identify the **termination (final) state** that is entered immediately prior to exit.

Step 3 - Identity any **activities** (operations) that commence when the state is entered. Activities include sequential operations that terminate after a period of time, after completion of a calculation or upon receipt of an event. Activities include continuous operations that terminate when the state is exited.

Step 4 - Identify the **events,** any stimuli 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. An event may be modeled as an event object or as an event message. Identify any **parameters** that are passed in the events.

Step 5 - Identify the **actions,** e.g. startMotor () in response to an event. Actions include updating an attribute, sending a message, or similar action.

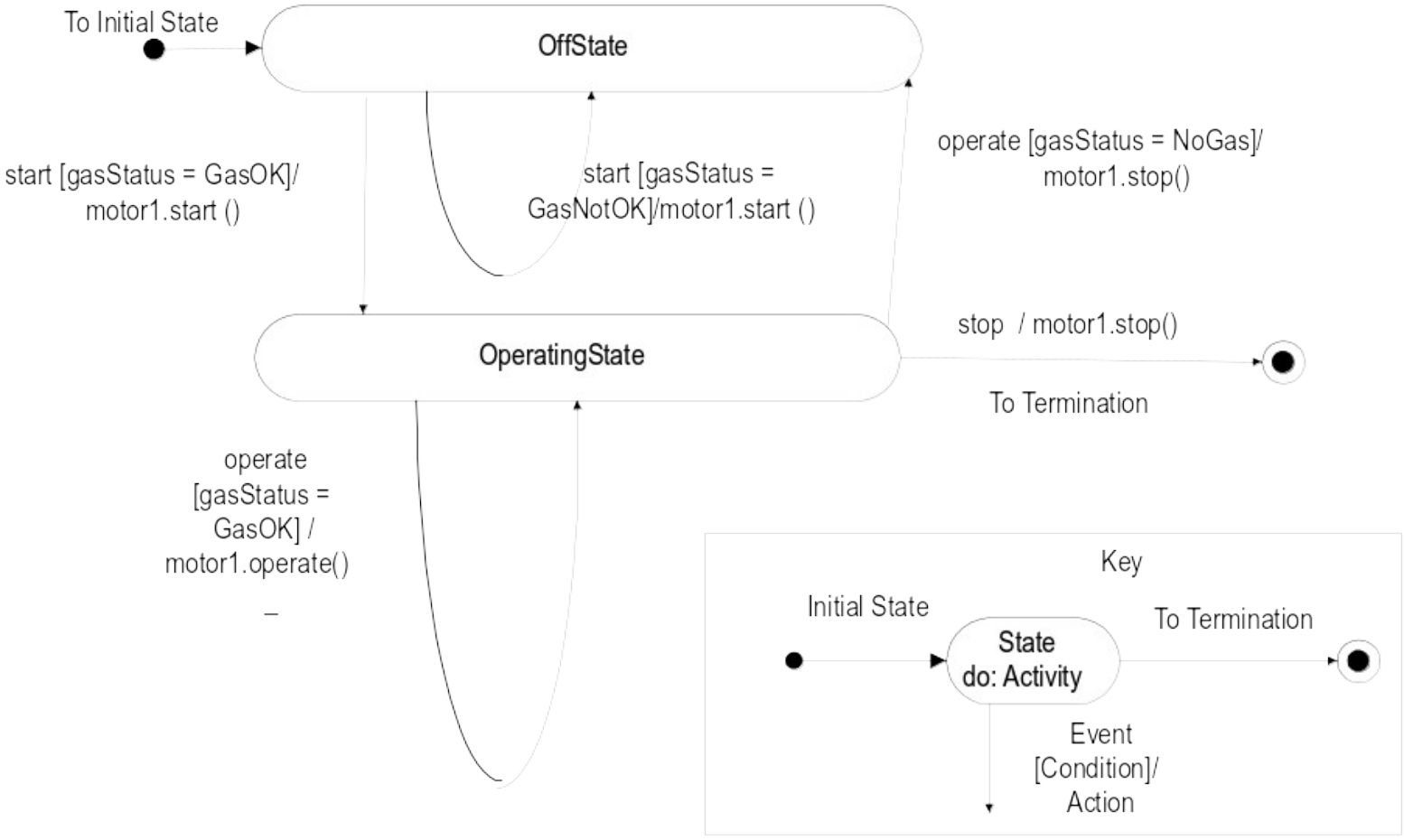
Step 6 - Identify the conditions that affect the stimulus response logic. A

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".

Step 7 - Identify the transitions. A **transition** is a unique pattern of an event, conditions, actions, and a destination state. For each state identify applicable events. Then for each event identify the applicable conditions, actions, and the destination state.

Step 8 - Walk-through the state diagram by sending each event to ensure the correct transitions.

Step 9 - Prototype, test, and iterate.



The state diagram may be created with a Windows drawing tool or StateMaker. The above diagram was created with a Windows drawing tool. The steps to

create the state diagram using a Windows drawing tool or StateMaker are listed below.

>> Create a directory, e.g. c:\md car
>> Run the Windows drawing tool or StateMaker from Windows
>> Select "File - SaveAs" c:\car\carstate.sm
>> Select the oval or State Icon
>> Enter the State Name, e.g. OffState
>> Double Click OK
>> Select the oval or State Icon
>> Enter the State Name, e.g. OperatingState
>> Double Click OK
>> Select the arrow or Transition Icon
>> Enter the Event, e.g. start ()
>> Enter the Action, e.g. start ()
>> Double Click OK
>> Connect two states together with the transition
>> If using StateMaker, select "Compile - To C++" to generate C++ code
>> Enter the file name for the C++ source code, e.g. c:\car\carstate.cpp
>> Select "File - Save" to save the diagram

3.7. Updated Data Dictionary

In the dynamic model update the data dictionary to list and describe new terms, such as object names and state names.

Term	**Description**
car1	Object of Car class
motor1	Object of the Motor class
passenger1	Object of the Passenger class
cellularPhone1	Object of the CellularPhone class
OffState	State of Car class
OperatingState	State of Car class

3.8. Summary

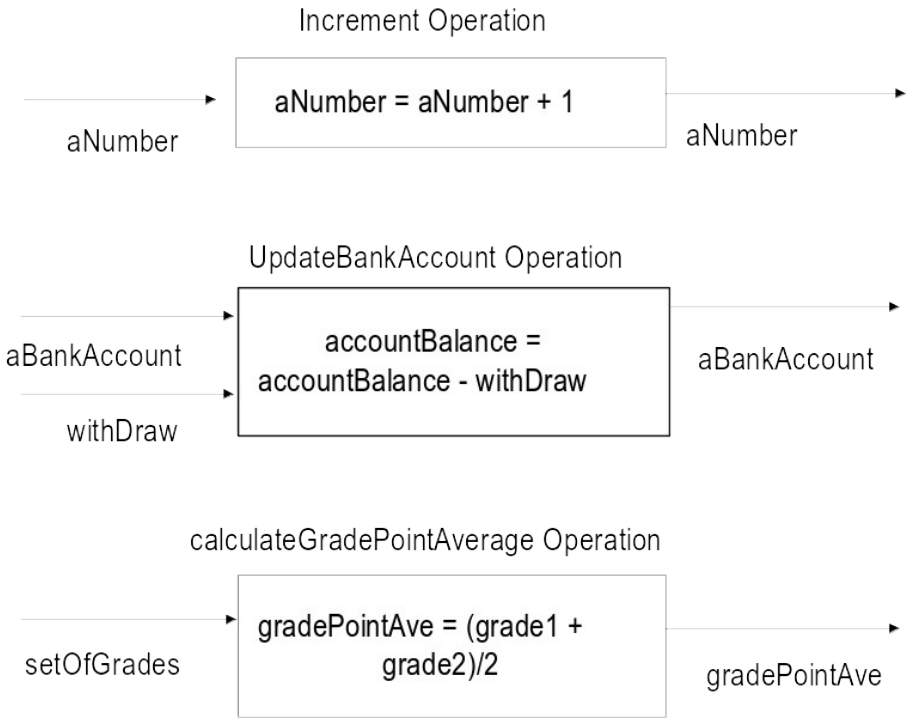
The dynamic model is the description of the stimulus response behavior of the system and objects in the system. Stimulus response behavior is sum of stimuli that a system or object detects and the responses to those stimuli. For the system as a whole, we describe stimulus response behavior in terms of input events leading to output events. For objects in the system, we describe stimulus response behavior in terms of the time ordered sequence of messages between objects. Some systems and objects have modes or states which are described in state diagrams. The dynamic model consists of documentation products, e.g. system event flow diagram, object interaction diagrams (messages), object interaction scenarios (messages), state diagrams, data

dictionary, and code.

4. Modeling Operations (Transformations) in the Functional Model

4.1. Description of the Functional Model

The functional model describes transformations of object values in operations from an input to an output. Several simple operations which transform object values are shown below.



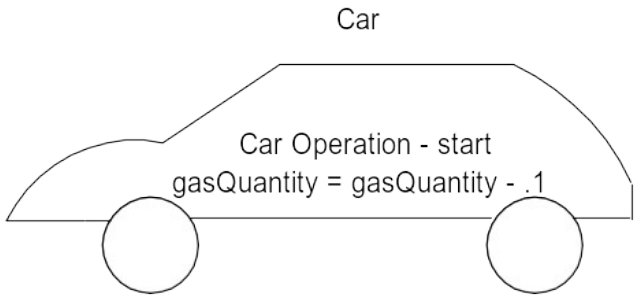
A transformation could be described as follows:
- a table showing the mapping from one object value to another,
- a formula or equations showing the relationship between an input object value and an output object value,
- a text description of the transformation.

For example, one of the simplest transformations is the increment transformation shown above. A table describing the increment transformation would have the entries: 1, 2; 2, 3; 3, 4; etc. The table shows the input and output values. An equation describing the increment transformation would be “aNumber = aNumber + 1”. A text description of the increment transformation would be “add 1 to the input value”.

A transformation may change the value of an atomic object (1, 4.2, ‘c’), a structured object (student1), or a collection object (aListOfStudents). A transformation may change the value of atomic object, the value of an attribute of a structured object, or the value of a member of a collection object. The term value may refer to a literal value, e.g. 1, 4.2, or ‘c’ or to a reference value, e.g. an object ID such as student1.

A transformation may occur in the following forms of operations:
- A **system operation** is an operation for the system as a whole. A system operation receives an input object and changes an output object. A system operation may access or modify a system attribute. A sample system operation for a bank ATM system is “withdrawCash”.
- A **class or object operation** is an operation defined in a class that accesses or modifies an attribute value. An operation may receive an input parameter and may send an output parameter. A sample operation in a bank ATM Account class is “updateAccountBalance”. This operation modifies the attribute “accountBalance”.

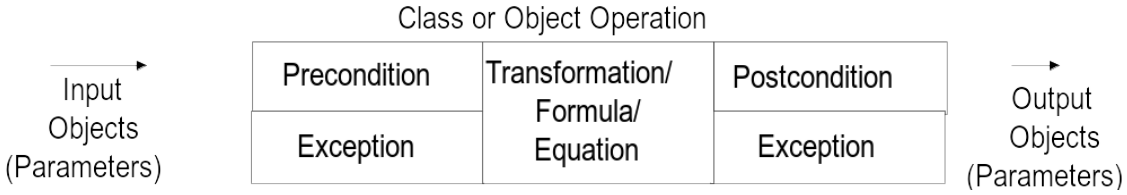
In the Car class, there is transformation in the start operation. The transformation is represented by the formula “ gasQuantity = gasQuantity - .1”. In this example, when the start operation is invoked, then there is a transformation of the attribute gasQuantity. Specifically, the value of the gasQuantity is changed. When the start operation is invoked the value of the gasQuantity is reduced. There are no input or output parameters for this operation. In the functional model we identify, examine, and describe operations in which there is a transformation of objects.



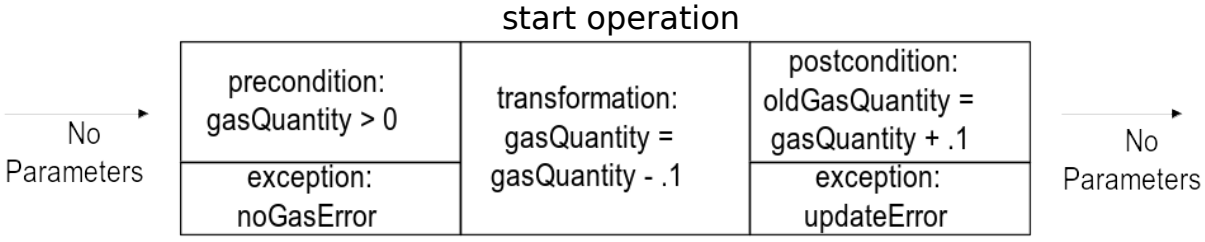
The functional model is the set of transformations and correctness assertions, e.g. preconditions, postconditions, and invariants. A **transformation** is a description of how an object value may be correctly changed in a formula, expression, table, etc. An **assertion** is a rule or expression for correctness, e.g. a 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. listFullError or noGasError 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 gasQuantity must be greater than zero. The postcondition of the start operation is the verification that the correct value of gasQuantity has been computed.. The invariant is that gasQuantity must be equal to or greater than zero and equal to or less than the maximumGasQuantity. In the functional model, we update class specifications with transformations and correctness assertions.

Key aspects of an operation are shown below. Only input and output parameters are shown in the diagram below. An operation may access or update attribute values or global objects.



The key aspects of the start operation are graphically shown below. It shows that there are no object inputs or outputs (parameters), the precondition of "gasQuantity > 0", the postcondition of "oldGasQuantity = gasQuantity + 1", the transformation formula, and the exceptions noGasError and updateError.



The functional model is important for many reasons. First, transformations (operations) are important because they accomplish work required by the system requirements. Second, identifying transformations (operations) leads to identifying the system input objects, system output objects, and system operations. Third, identifying operation preconditions, postconditions, and exceptions help create highly reliable operations. Fourth, operations are

implemented by coding the transformation equations, preconditions, postconditions, and exceptions.

4.2. Steps to Create the Functional Model

In the functional model we specify transformations of objects in operations. In the functional model, we specify how objects are transformed in a system. Follow these steps.

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

Step 2 - If required update the **data dictionary** to list and define new terms.

Step 3 - Check and update consistency with other models, e.g. the object model and dynamic model. Check all diagrams and text specifications for the consistent spelling, capitalization, and meaning of all names.

Step 4 - Update the **C++ source code** to reflect the transformations and correctness assertions. Compile and execute the program.

4.3. Specifying Class and Object Operations

To describe class and object operations, specify transformations and correctness assertions. Using With Class, update each class specification. A sample operation specification is shown in a previous section.

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
>> Double click on an operation
>> Enter transformations and correctness assertions in the operation specification form

4.4. Summary

The functional model describes transformations of object values in operations from input to output. For the system as a whole, we describe input objects, output objects, and system operations (transformations). Typically, a system operation transforms (changes) an input object into an output object. For classes, we describe class and object operations. For each operation we describe the operation transformation, preconditions, postconditions, and exceptions.