Modeling and Prototyping Systems

Introduction

In previous sections we modeled a small number of classes to get familiarity with basic O-O concepts, documentation products, and CASE tools. In this section we will look at how to model and prototype a system of classes. We will apply the basic O-O concepts as listed below:

O-O Entities - system environment, system, class and object
O-O Connections - association, aggregation, message, generalization specialization
Models - object model, dynamic model, functional model
Documentation Products - diagrams, text, code
Tools - With Class, StateMaker, Windows Word Processor, Borland C++ Compiler

O-O modeling loosely follows the Analysis Model in Rumbaugh's OMT O-O methodology. O-O modeling emphasizes the early identification of the system environment and the identification of separate interface, control, and entity classes for highly portable systems. We will follow an iterative approach to model and prototype a system by describing each of the following:
1 - System Environment
2 - Classes - Object Model
3 - Classes - Dynamic Model
4 - Classes - Functional Model

Modeling the System Environment

Introduction

To rapidly model and prototype O-O systems, it is desirable to focus on a single system within a large system environment. With this focus we can model and prototype this single system with a dedicated team of developers, then integrate this single system with other systems. In the real world to construct a house, it is desirable to focus on a major component of a house, for example, the roof, with a dedicated team of workers. In S/W it is desirable to partition a large S/W system into manageable subsystems that can be linked or connected together. The purpose of this section is to present a "how to" focus on a single system and to model the system environment. In modeling the system environment, we will present the objective, key entities, key connections, and key products as introduced below.

Objective - The objective is to define the boundary and scope of the system within a larger environment in terms of interacting systems, event messages, response messages and problem domain objects. It is important to have a clearly defined boundary and scope to accomplish development requirements, cost, and schedule goals.

Entities - The key entities are the system being modeled, interacting systems (devices or programs), and problem domain objects.

Connections - The key connections are event messages and response messages.

Products (Diagrams and Text) - The key products are the requirements statement, drawing, block diagram, event flow diagram (messages), event list (messages), event scenario (messages), and data dictionary of problem domain objects.

Entities

There are several key entities in modeling the system environment: system to be modeled, interacting system (device or program), and problem domain objects. These are defined below.

System Environment - The system environment consists of a system and interacting systems (devices or programs). In S/W a GUI based text processor is an system environment with a system, for example, the text processor program and interacting system (device or program), for example, a GUI system and a file system.

System - The system is the system to be modeled and prototyped. It is our center of focus. It generally consists of 10 to 100 classes as a rough order of magnitude. It is generally programmed as a single .EXE or concurrent process. The text editor program is an example of a system. A large system may be partitioned into subsystems. A subsystem may consist of other subsystems or the subsystem may consist of classes.

Interacting System (Device Or Program) - An interacting system (device or program) is any entity that interacts with the system. An interacting system can be a HW device, S/W program, a GUI program, a data management program, a communications program, etc. An interacting system is sometimes called an actor, terminator, outside agent, or external system.

Problem Domain Object - A problem domain object is an object that

is passed as a parameter in an event message and or an response message. It exists in the system environment and is manipulated by the system. Examples are bank account, customer, registration, product, etc. A problem domain object is sometimes called a real world object, an application object, or domain object.

Connections

The key connections in modeling the system are event messages and response messages. These are defined below.

Event Message - An event message is a message (communication) from an interacting system (device or program) to the system. The system must respond. An event message is based on a user or physical phenomenon such as a user action of pressing a function key. Examples of event messages are turn on, turn off, record data, store data, print report, etc. Generally, a menu item or button in a GUI window represents an event message. The event messages represent the functional requirements of the system, i.e. what the system must do.

Response Message - A response message is a message from the system to an interacting system (device or program). It is in response to an event message such as a message to a printer. Examples of response messages to an interacting GUI system are open window, show dialog box, display text, etc. Examples of response messages to an interacting data base management system are get record, update record, delete record, find record, etc.

Strategy to Identify Major Entities in the System Environment

The following is a recommended strategy to identify major entities in the system environment.

1. Start with the customer provided information, e.g. requirements statement, drawing, block diagram, etc.

2. Identify the system to be modeled and interacting systems which interact (communicate) with the system. This is to set the scope and boundary of the system.

3. For a large system, identify smaller subsystems to developed individually.

4. Identify event messages. These are based upon events which occur

in the system environment.

5. Identify response messages. These are responses to event messages.

6. Identify problem domain objects that are passed into and from the system in messages. The problem domain objects will be basic objects used in the system.

Products to Describe the System Environment

There are many products to describe the system environment as described below.

Requirements Statement - States the requirements for the system.

Drawing or Physical Representation - Provides a physical, real world view of the system environment.

Block Diagram - Shows the system and interacting systems within the system environment.

Event Flow Diagram (Messages) - Shows event and response messages and problem domain object. It sets the boundary for the system.

Event List (Messages) displays the event messages to which the system must respond and response messages that the system initiates to interacting systems.

The Event Scenario (Messages) displays the time ordered sequence of event and response messages.

Data Dictionary of Problem Domain Objects displays problem domain objects that exist in the system environment. These are received from or sent to interacting systems (devices or programs) in event messages and response messages.

User Interface Drawing, Description, and GUI Prototype - Drawing, description, and GUI prototype of the user interface, e.g., menus, dialog boxes, windows

General Steps to Create Products for the System Environment

The following are the general steps to create products (diagrams, text,

and code) to describe the system environment.

1 - Collect written information and interview the client, users, and experts about the system environment that consists of the system and interacting systems (devices or programs).

2 - Create the system requirements statement in text form with the description of commands (event messages), responses (response messages), and input and output data (problem domain objects).

3 - Make a drawing or physical representation of the major components of the system environment, e.g., HW devices and SW programs.

4 - Make a block diagram of the system environment to show the system and connections to interacting systems (devices or programs).

5 - If required, collect domain analysis information about the problem to be solved, similar problems and similar systems.

6 - Create the Event Flow Diagram (Messages) with the system in the middle surrounded by interacting systems (devices or programs). Show or list event messages, response messages and problem domain objects.

7 - Create the Event List (Messages). Describe the "10 most important" event messages. For each event message state time, priority, throughput, and other requirements. List the response messages to interacting systems

8 - Create the Event Scenario (Message Scenario). For each event message, state the response messages in a scenario (script). State the simplest scenario (script) first then add variations and errors later.

9 - Create the Data Dictionary of Problem Domain Objects. Describe the "10 most important" problem domain objects. An object of problem domain may be an atomic, collection, or composite object. Describe each object of problem domain in terms of its name, direction to or from the system, class, and other relevant information.

10 - Trace each event message from interacting systems (devices or programs) to the system and then to other interacting systems (devices or programs) as response messages.

Creating the Requirements Statement

Description - The requirements statement is a primary document to describe the requirements and functions of the system. The requirements statement is the problem statement and charter. The emphasis in the requirements statement is to describe the system and its interactions with interacting systems (devices or programs). For simple systems, the requirements statement consists of a few sentences stating what the system must do. For complex systems, the requirements statement may be thousands of pages of detailed requirements and specifications. In this tutorial only very simple requirements statements are presented. This section describes the requirements statement, its purpose, steps to create, and a sample requirements statement.

Purpose of the Requirements Statement - The purpose of the requirements statement is to state the requirements for the system environment with emphasis on the system.

Steps to Create the Requirements Statement

The following are the steps to create a requirements statement.

1. Identify the system environment with the system and interacting systems (devices or programs).

2. State the functional requirements of the system, i.e. what the system must do.

3. State the users, user types, user roles, or user categories of the system, e.g., customer, operator, supervisor, end user, etc. In "Object-Oriented Software Engineering", Jacobson et. al. uses the term **actor** to refer to a user role or user category. Each actor has one or more events and a message scenario (**use case**) of messages through the system.

4. Describe the system's commands (event messages), responses (response messages), and input and output data (problem domain objects).

5. If necessary, describe the interacting systems (device or program).

TV Controller Case Study - The following is the Requirements Statement for the TV Controller Case Study.

TV Controller System Requirements Statement

The TV Controller is a software program (system) that receives commands (event messages) from a TV Buttons Device (interacting system) and sends messages (response messages) to TV Speaker Device, TV Channel Device, and TV Storage (interacting system). The TV Controller stores the last used volume and channel selection. The commands (event messages) are turn on, turn off, increase volume, decrease volume, increase channel, and decrease channel. The TV Controller initiates the following response messages: get volume, set volume, store volume, get channel, set channel, and store channel. The following data (Problem Domain Objects) are passed in the response messages: volume setting and channel setting.

Creating the Drawing - A Physical Representation

Description - A drawing is a physical diagram of the system environment showing typical users, hardware devices, software programs, and objects passed into and from the system. This section describes the requirements statement including its purpose, steps to create, and a sample requirements statement.

Purpose - The purpose of the drawing is to provide a physical, real world view of the system environment including the user, H/W devices, S/W programs, etc.

The drawing is very important for the following reasons:
- helps identify H/W and S/W systems,
- helps identify the boundary of the system,
- helps define links (communications) between the system and H/W and S/W systems,
- helps define input and output data (problem domain objects) that is passed to and from the system in messages,
- helps maintain an object orientation versus a functional orientation,
- provides an excellent "starting point" because it is easy to understand and discuss with the client and users,

Steps to Create the Drawing - The following are the steps to create the drawing.

1. Show the system as a box in the middle of the drawing.

2. Show users and their user roles, e.g., user, operator, etc.

3. Show H/W devices that interact with the system.

4. Show S/W programs that interact with the system.

5. Show links between the H/W devices and S/W programs with the system.

6. Draw small symbols to represent input and output data (problem domain objects) that is passed to and from the system in messages.

TV Controller Case Study - The drawing for the TV Controller is shown in file **tvdraw.omt.**

Creating the Block Diagram

Description - The block diagram has boxes for each major entity in the system environment. It shows major hardware devices, software programs, and connections. This section describes the block diagram including its purpose, steps to create, and a sample block diagram.

The block diagram shows the system being modeled and interacting systems (devices or programs) within the system environment. The block diagram may show interacting systems that initiate commands (event messages). The block diagram may show interacting systems that receive responses (response messages). The block diagram leads to the event flow diagram (messages).

Purpose of the Block Diagram - The block diagram shows a high level view of the system environment with the system and interacting systems (devices or programs).

Steps to Create the Block Diagram - The following are the steps to create the block diagram.

1. Show the system being modeled and interacting systems (devices or programs) as boxes.

2. Place the system being modeled in the box in the middle of the diagram.

3. Place interacting systems that send commands (event messages) on the top or left of the diagram.

4. Place interacting systems that receive commands (response messages) from the system on the right or bottom of the diagram.

5. Show line between interacting systems and the system being modeled. These lines represent interaction or communication.

TV Controller Case Study - The block diagram for the TV Controller is shown in **tvblock.omt**.

Creating the Event Flow Diagram (Messages)

Description - The event flow diagram (messages) displays the system, event messages, response messages, and problem domain objects in graphic form. The event flow diagram (messages) is functionally equivalent to the structured analysis context diagram described by Ware and Mellor in "Structured Development for Real-Time Systems" Volume 1. Both the event flow diagram (messages) and structured analysis context diagram set the boundary of the system. This section describes the event flow diagram (messages) including its purpose, steps to create, and a sample diagram.

Purpose - The purpose of the event flow diagram (messages) is to show the scope and boundary of the system, event and response messages, and problem domain objects (incoming and outgoing objects - data).

Steps to Create the Event Flow Diagram (Messages) - The following are the steps to create the event flow diagram (messages).

1. Start with the system requirements statement, drawing, and block diagram.

2. Place the system being modeled in the box in the middle.

3. Place interacting systems that send commands (event messages) on the top or left of the diagram.

4. Place interacting systems that receive commands (response messages) from the system on the right or bottom of the diagram. However, many interacting systems both send and receive messages.

5. Show event messages as arrows from interacting systems to the system being modeled.

6. Show response messages as arrows from the system being modeled to the interacting systems.

7. If desired, show problem domain objects as small arrows. The small arrow should point in the direction that the problem domain object is

passed.

8. Walk-through each event (stimulus) message from the interacting systems to the interacting systems.

TV Controller Case Study - The event flow diagram (messages) of the TV Controller is shown in **tvevents.omt**.

Creating the Event List (Messages)

Description and Purpose - The event list (messages) lists and describes all the event (stimulus) messages to which the system must respond and all the resulting response messages from the system. Its purpose is to provide a list of event and response messages. This section describes the event list (messages) and event scenario (messages) with its purpose, steps to create, and a sample event list (messages).

The event list (messages) is useful for the following reasons:
- to understand system requirements,
- to generate initial test cases,
- to generate menus and dialog boxes,
- to provide a test scenario for walk-throughs

Steps to Create the Event List (Messages) - The following are the steps to create the event list (messages).

1. Identify event (stimulus) messages from interacting systems to which the system must respond. Question is "What occurs in the external environment to which the system must respond?

2. Group event messages by user type, user role, or user category, e.g., user, customer, operator, accountant, etc.

3. Describe each event message. For each event message state timing, priority, throughput, and other requirements.

4. Identify response messages from the system to interacting systems. Question: For each event (stimulus) message what are the response messages?

5. Using the event flow diagram (messages), walk-through all messages to check completeness.

TV Controller Case Study - The following is the Event List

(Messages) for the TV Controller Case Study.

Event List (Messages)

Event Messages (Interacting System --> TVController)

Event Message	Input Parameter Object/Class	Return Parameter Object/Class	Interacting System(Sender)
TurnOn	None	None	TVButtons Device
IncreaseVolume	None	None	"
DecreaseVolume	None	None	"
IncreaseChannel	None	None	"
DecreaseChannel	None	None	"
TurnOff	None	None	"

Response Messages (TVController --> Interacting System)

Response Message	Input Parameter Object/Class	Return Parameter Object/Class	Interacting System(Receiver)
GetVolume	None	VolumeSetting /Integer	Storage Device
StoreVolume	VolumeSetting /Integer	None	Storage Device
GetChannel	None	ChannelSetting /Integer	Storage Device
SetChannel	ChannelSetting /Integer	None	Channel Device
SetVolume	VolumeSetting /Integer	None	Volume Device
SetChannel	ChannelSetting /Integer	None	Channel Device

Creating the Event Scenario (Messages)
Description and Purpose - The event scenario (messages) shows the time ordered sequence of event and response messages. This section describes the event scenario (messages) with its purpose, steps to create, and a sample event list (messages).

Steps to Create the Event Scenario (Messages) - The following are the steps to create the event scenario (messages).

1. Start with the event list (messages).

2. Create the event scenario (messages). For each event (stimulus) message state the resulting response messages in a scenario (script). State the simplest scenario (script) first then add variations and errors later. The recommended table includes the following headings: sequence from 1 to ?, interacting system that sends the message (sender), the system that receives the message (receiver), the message name, and remarks.

3. If desired create an event trace diagram (message scenario) as described by Rumbaugh et. al. in "Object -oriented Modeling and Design". A sample event trace diagram (message scenario) is shown below.

4. Using the event flow diagram (messages), walk-through all messages to check completeness.

TV Controller Case Study - The following is the event scenario (messages) for the TV Controller Case Study.

Event Scenario (Message)

Sequence	**Sender**	**Receiver**	**Message**	**Remarks**
1 -	TVButtonsDevice	TVController	TurnOn	Event Msg
2 -	TVController	StorageDevice	GetVolume	Response Msg
3 -	TVController	StorageDevice	GetChannel	Response Msg
---	---	---	---	---
1 -	TVButtonsDevice	TVController	IncreaseVolume	

Event Msg
2 - TVController SpeakerDevice SetVolume Response Msg
---

1 - TVButtonsDevice TVController DecreaseVolume Event Msg
2 - TVController SpeakerDevice SetVolume Response Msg
---

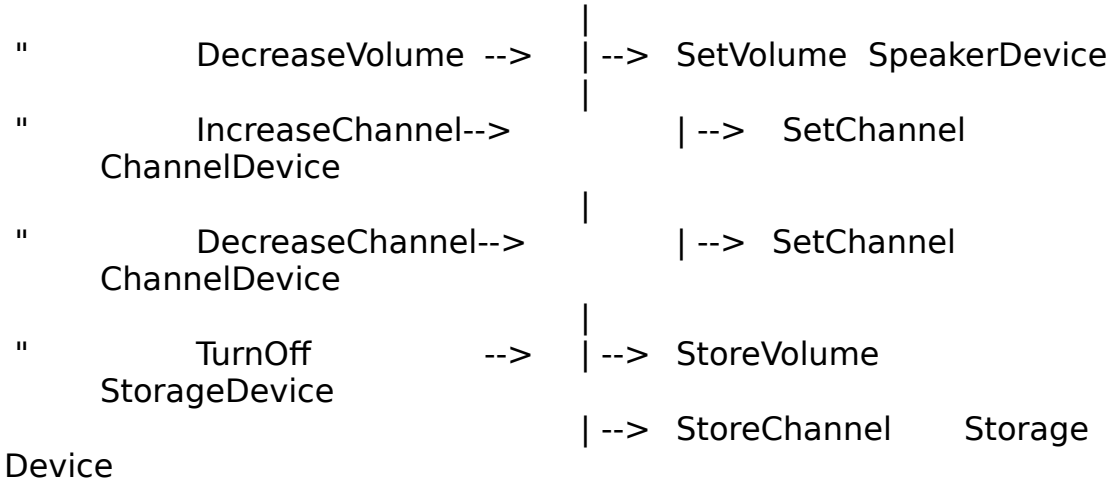
1 - TVButtonsDevice TVController IncreaseChannel Event Msg
2 - TVController ChannelDevice SetChannel Response Msg
---

1 - TVButtonsDevice TVController DecreaseChannel Event Msg
2 - TVController ChannelDevice SetChannel Response Msg
---

1 - TVButtonsDevice TVController TurnOff Event Msg
2 - TVController StorageDevice StoreVolume Response Msg
3 - TVController StorageDevice StoreVolume Response Msg

Event Trace Diagram (Message Scenario)
(Sender Interacting System --> TVController --> Receiver Interacting System)

Interacting System (Sender)	**Event Message**		**TVController System**	**Response Message**	**Interacting System (Receiver)**
TV Buttons Device	Turn On	-->	| -->	Get Volume	StorageDevice
			| -->	Get Channel	StorageDevice
"	IncreaseVolume	-->	| -->	SetVolume	SpeakerDevice



Creating the Data Dictionary for Problem Domain Objects

Description and Purpose - The Data Dictionary for Problem Domain Objects shows and describes each incoming and outgoing problem domain object. A problem domain object exists in the system environment, e.g. account, customer, transaction, etc. They are basic classes in the system. This section describes the data dictionary of problem domain objects with its purpose, steps to create, and a sample data dictionary of problem domain objects.

Steps to Create the Data Dictionary for Problem Domain Objects - The following are the steps to create the data dictionary of problem domain objects.

1. Identify problem domain objects that are passed into or from the system in messages. Question: What data and information (problem domain objects) are passed into and from the system?

2. Describe each problem domain object in terms of its name, class, direction passed to or from the system, and descriptive information.

3. List and describe each problem domain object either in a narrative form or in the following table form.

Problem Domain Object	**Class**	**Description**

TV Controller Case Study - The following is the Data Dictionary of Problem Domain Objects for the TV Controller Case Study in narrative form.

Data Dictionary for Problem Domain Objects

(Objects Passed To and From the TV Controller)

VolumeSetting is an object of the integer class. It is passed to and from the TV Controller and the Storage Device. It is passed from the TV Controller to the Speaker Device. It indicates the magnitude of the desired or current TV audio volume setting.

ChannelSetting is an object of the integer class. It is passed to and from the TV Controller and the Storage Device. It is passed from the TV Controller to the Channel Device. It indicates the desired or current TV channel setting.

User Interface Description and Drawing

Description and Purpose - The user interface description documents user interface requirements in drawings and text descriptions for button panels, function keys, menus, windows, input forms, output reports, etc. It documents H/W and S/W interfaces to be simulated in prototype with interface objects.

Steps to Draw the User Interface - The following are the steps to draw the user interface.

1. Select the form of the interface, e.g., Text or GUI

2. For a text interface drawing, show the following:

- Show menu and menu items for each event message.

- Show text prompts for data input for each incoming problem domain object.

- Show text statement for data output for each outgoing problem domain object.

3. For a GUI interface drawing, show the following:

- Show a menu, menu item, button or similar GUI object for each event message.

- Show an input text box for each data input for each incoming problem domain object.

- Show an output text box or similar GUI object for each data output for each outgoing problem domain object.

4. Prototype the user interface using a screen painter or interface builder to get user feedback. See file **tvgui.omt**.

Evaluating the System Environment Products

The following are various criteria and questions to ask in evaluating the adequacy of the products describing the system environment.

- Have we collected enough information to identify and describe the system and interfaces to interacting systems (devices or programs)?

- Have we grouped interacting systems together that have a common function?

- Have we organized the interacting systems for loose message coupling?

- Have we described the system in terms of commands, functions, and data inputs and outputs?

- Have we identified users and their user roles, e.g., operator, supervisor?

- Have we clearly stated the scope (boundary) of the system in terms of the "10 most important" event messages, response messages, and problem domain objects?

- Have we grouped smaller external data items into larger composite problem domain objects with strong cohesion.

- Does the customer (requester of the system) agree that we have clearly stated the scope (boundary) and the functionality of the system?

Modeling Classes - The Object Model

Introduction

To rapidly model and prototype O-O systems, it is desirable to "plug and play" as we now do with stereo components. In S/W terms it is desirable to be able to quickly "change out" and replace an interacting system within a system environment. In the real world to construct a house, it is desirable to be able to replace the roof without major disruption to the rest of the house. The purpose of this section is to

present a "how to" to model and prototype interface, control, and entity classes. We will present the objective, key entities, key connections, and key products as introduced below.

Objective - The objective is to identify, describe, organize, and prototype interface, control, and entity classes and objects.

Entities - Key entities are the initializer class, interface classes, control classes, and entity classes

Connections - Key connections are association, aggregation, and generalization specialization. Defer messages until the dynamic model.

Products (Diagrams, Text, Code) - Key products in the object model are as follows: class diagram, data dictionary listing classes, class specification, and prototype;

Key Entities - Basic Classes in the System

The system can be described as being composed of interface, control, and entity classes. It is useful to have separate interface, control, and entity classes for modifiability. Ideally, to replace an old interacting system with a new interacting system, one would change the interface class not entity classes. The terms used in this section, e.g., interface object, control object, and entity object were introduced in "Object-Oriented Software Engineering" by Jacobson et. al. These classes are:

- Initializer class - initializes the system
- Interface classes - interact with interacting system (device or program)
- Control classes - manage time based and state based event behavior
- Entity classes - manage application information.

Initializer Class - An initializer class is a high level class. It starts and initializes the system. It is the "main" in C++.

It is useful for several reasons:
- provides a starting point to trace messages,
- as a top level composite, it provides a means to construct a hierarchical system,
- is required to construct programs in C++ and other languages

A guideline is to create a initializer class for the system to start and initialize the system. For example, we have a single initializer class in

the TV Controller System named TVMain.

Interface Classes - An interface class defines objects that communicate directly with interacting systems (devices or programs). An interface class has the following characteristics:
- encapsulates protocol details of an interacting system (device or program),
- interacts with a single interacting system (device or program) or group of interacting systems,
- has an operation for each event message handled,
- has an operation for each response message sent.

Typically, an event message is initially handled by an interface object, then sent to a control object, then sent to an entity object, then sent to another interface object. It is important to identify interface classes to encapsulate protocol details of interacting systems (devices or programs) for portability. Interface classes facilitate portability to rapidly transfer a system from one environment to another.

There are two major choices to identify and group interface classes.
1. First choice, identify an interface class for each individual interacting system (device or program). This is appropriate under the following conditions:
- there is a relatively small number of interacting systems (devices or programs), i.e. less than ten;
- the interacting systems (devices or programs) are very well defined, e.g., a GUI program and a data base program;
- to group interacting systems (devices or programs) together as a logical unit is not appropriate. e.g., to group a data base program and a communications program together is not appropriate.

2. Second choice, identify an interface class for a group of logically related interacting systems. This is appropriate if a group of several interacting systems (devices or programs) work together as a unit, e.g., several HW devices could be grouped together
- if they have a common purpose such as user interface, or
- if they may be changed as a unit.

A guideline is to create an interface class for each interacting system (device or program) (or group of systems) that send event messages or receive response messages. For example, we have an interface class for each interacting system to the TVController.

Control Classes - A control class defines objects that have time based

and state based behavior for events. It has the time based sequencing and logic for events. It has the state based logic for events, i.e. finite state machine logic. It has decision logic for stimulus response behavior.

A guideline is to create a control class for each user role, i.e. group of related events. The control class has an operation for each event that it handles. For example, we have a single control class in the TVController System named TVController.

Entity Classes - An entity class defines objects that manage, computes, and store application information without interface details. The following are several ways to identify entity classes:
- Identify an entity class for each problem domain object.
- Identify an entity class for any application entity that you can draw.
- Identify an entity class for any application entity you need to process an event message.
- Identify an entity class for any application entity you enclose in an input form, GUI dialog box, or text box.
- Identify an entity class for any application entity that has a search key or lookup key.
- Identify an entity class for any application entity in a data model, e.g., semantic data model, entity relationship data model, data tables, etc.
- Identify an entity class for any application entity that has a Class - Responsibility - Collaboration - Card (CRC Card). See "Designing Object-Oriented Software" by Wirfs-Brock, Wilkerson, and Werner.

A guideline is to create an entity class for each problem domain object and other important application entities. Typical operations in entity classes are get, set, and calculate attribute values. In the TVController System, we have an entity class for each problem domain object, e.g. Volume and Channel.

Connections

The key connections are association, aggregation, and generalization specialization as presented earlier. Defer messages until the dynamic model.

Strategy to Identify and to Organize Classes

The following is a recommended strategy to identify and organize interface, control, and entity classes.

1. Start with the description of the system environment with the

following identified: system, interacting system (device or program), event messages, response messages, problem domain objects.

2. Identify an initializer class to be the composite class for the entire system. This class starts and initializes the system.

3. Identify interface classes to interact with interacting systems (devices or programs). There are two primary choices. First choice, identify an interface class for each individual interacting system (device or program). Second choice, identify an interface class for a group of logically related interacting systems.

4. Identify a control class for each user role, e.g., group of related event messages.

5. Identify connections for each class, e.g., association, aggregation, and generalization specialization.

6. Organize the initializer class, interface classes, control classes, and entity classes.

7. Identify attributes and operations for each class. The following are guidelines:

- Initializer class - operation "initialize" or "start",

- Interface classes - operation for each event message and an operation for each response message,

- Control classes - operation for each event message.

- Entity classes - query, update, and computation operations.

8. Test the class diagram with the check of connections.

- Check associations and aggregation. Start with the initializer class and trace the "has a" or "part of" connections.

- Check generalization specialization. Start with highest superclass.

Products (Diagrams, Text, and Code) to Describe Classes

There are four major documentation products in the object model as described below.

Class Diagram - Displays classes and objects with their attributes, operations, and connections.

Data Dictionary Listing Classes - Lists classes with their attributes

Class Specifications - Displays all relevant information for each class. Provides design documentation to build the prototype.

Prototype - C++ source code and demonstrations

General Steps to Create Products Describing Classes
The following are the general steps to create products (diagrams, text, and code) to describe the interface classes.

1 - Create the Class Diagram. Initially identify an initializer class, interface classes, and control classes. Then add entity classes.

2 - Create the Data Dictionary Listing Classes. Identify and describe all classes.

3 - Develop a Class Specification for each class to document the attributes, operations, and other details, e.g., description, superclasses, persistence, etc.

4 - Program the prototype and demonstrate to get user feedback. This is to ensure that you understand the fundamental requirements.

Creating the Class Diagram

Description and Purpose - The class diagram graphically show the classes in a system with their attributes, operations, and connections. There are various forms and graphic notation of class diagrams that have been published by Coad/Yourdon, Rumbaugh, Booch, and numerous others.

Steps to Create the Class Diagram - The following are the steps to create the class diagram.

1. Start with the descriptions of the System Environment.

2. Identify an initializer class to be the composite class for the entire system.

3. Identify an interface classes to interact with each interacting system (device or program) or group of interacting systems.

4. Identify a control classes for each user role, e.g., group of related events.

5. Identify an entity classes for each problem domain object and other application entities.

6. Identify connections for each class, e.g., association, aggregation, and generalization specialization.

7. Organize the initializer class, interface classes, and entity classes.

8. Identify attributes and operations for each class. The following are guidelines for operations:

- Initializer class - operation "initialize" or "start",
- Interface classes - operation for each event message and an operation for each response message,
- Control classes - operation for each event message.
- Entity classes - query, update, and compute operations.

9. Test the class diagram by checking connections.

- Check associations and aggregation. Start with the initializer class and trace the "has a" or "part of" connections.

- Check generalization specialization. Start with highest superclass.

TV Controller Case Study - The class diagram for the TV Controller Case Study. are shown in **tvclass1.omt** showing interface and control classes and **tvclass2.omt** showing all classes including entity classes.

Creating the Data Dictionary Listing Classes

Description and Purpose - The data dictionary listing classes textually lists and describes the class in a system. with a brief description including purpose, attributes, operations, and connections. The data dictionary listing classes is abbreviated documentation for each class. The data dictionary listing classes is the object oriented equivalent of the traditional data dictionary.

Steps to Create the Data Dictionary Listing Classes - The following are the steps to create the data dictionary listing classes.

1. Start with the Class Diagram showing all the classes in the system.

2. List classes starting with the initializer class, then interface classes, then control classes, and finally entity classes.

3. Briefly describe each class, e.g., purpose, attributes, operations, connections, and other information.

TV Controller Case Study - The following is the Data Dictionary Listing Classes for the TV Controller Case Study.

Data Dictionary Listing Classes

Class	Description
TVControllerMain	Initializer Class
TVButtonsInterface	Interface Class
StorageDeviceInterface	Interface Class
SpeakerDeviceInterface	Interface Class
ChannelDeviceInterface	Control Class
TVItem	Entity Class
Volume	Entity Class
Channel	Entity Class

Creating the Class Specification

Description - The class specification 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 the Booch class specification presented in "Object Oriented Design with Applications" by Grady Booch. He presents the key aspects of each class, e.g., description, enclosing system, superclasses, visibility, cardinality, concurrency, persistence, space, applicable documents, and remarks. The class specification is described in more detail in earlier section.

Steps to Create the Class Specification - The following are the steps to create the class specification.

1. Start with the class diagram or data dictionary listing classes.

2. Document each class with the following information: description, enclosing system, superclasses, visibility, cardinality, concurrency, persistence, space, applicable documents, and remarks.

Evaluating the Products in the Object Model

The following are various criteria and questions to ask in evaluating the adequacy of the products in the object model.

- Have we identified an initializer class?

- Have we identified interface classes with operations to handle all event messages?

- Have we identified interface classes with operations to send response messages?

- Have we identified a control class for each user role with an operation for each event message?

- Have we identified entity classes for each problem domain object and other important application entities with get, set, and compute operations?

- Does the prototype show the basic functionality of the system?

Modeling Classes - The Dynamic Model

Introduction

To rapidly model and prototype O-O systems, we must understand the stimulus response behavior of the system over time. We must model the time oriented sequence of messages from event messages to response messages. The dynamic model defines the documentation products to model the stimulus response behavior of a system and classes. The purpose of this section is to present a "how to" to model stimulus response behavior in terms of messages and states. We will present the objective, key entities, key connections, and key products as introduced below.

Objective - The objective is to identify messages, the sequence of messages, the rules which govern messages including state based rules.

Entities - Key entities are the initializer class, interface classes, control classes, and entity classes.

Connections - The key connection is messages. Association, aggregation, and generalization specialization connections are identified in the object model.

Products (Diagrams, Text, Code) - Key products in the dynamic model are as follows: message diagram (event flow diagram), event scenario (messages), state diagram, and prototype.

Creating the Message Diagram (Event Flow Diagram)

Description and Purpose - The message diagram (event flow diagram) graphically shows the classes in a system with message connections. This diagram is briefly described in "Object-oriented Design and Modeling" by Rumbaugh et. al.

The steps to create the message diagram (event flow diagram) are listed below.

1. Start with the description of the system environment and the object model. In particular examine the system message diagram (messages), the event list (messages), event scenario (messages), and the class diagram.

2. For each event message, identify the sequence of message connections from class to class through the system. Message connections may be shown between classes but the actual execution of message occurs between objects.

3. On the message diagram (class diagram showing messages), show a message connection whenever an object of a class sends a message to an object of another class. Generally there will be a message connection whenever there is an association or aggregation connection.

4. For each event message to an interface class, trace resulting messages through the system. Generally an event message has a response message in an interface class.

5. Initially create an message diagram (event flow diagram) for interface and control classes. Then add entity classes.

TV Controller Case Study - The message diagrams (event flow diagrams) for the TV Controller Case Study is shown in With Class files are **tvmsg1.omt** showing interface and control classes and **tvmsg2.omt** showing interface, control, and entity classes.

Creating the Message Scenario

Description and Purpose - The message scenario lists messages from an event message to response messages. A message scenario lists messages in a time ordered sequence for the first message to last message through the system. The message scenario is referred to as a "Use Case" by Jacobson et. al. in "Object Oriented Software Engineering".

Steps to Create the Message Scenario - The following are the steps to create the message scenario.

1. Start with the system event list (messages) and event scenario (messages).

2. For each event message, identify the sequence of messages through the system.

3. For each message, identify the sender object, receiver object, invoked operation, input parameters, and output parameters.

4. Check the message diagram (event flow diagram) and trace messages through the system.

5. Initially identify the simplest sequence of messages, later identify more complex sequences with error and unusual conditions.

TV Controller Case Study - The following is the Message Scenario for the TV Controller Case Study.

Message Scenario for TurnOn Event
(Uses C++ Message Form)

Sequence	**Sender Object (Class & Object)**	**Receiver Object. Invoked Operation (Input Parameter Class & Object)**	**Output Parameter**
1	TVButtonsDevice	aTVButtonsInterface.TurnOn()	--
2	aTVButtonsInterface	aTVController.TurnOn();	--
3	aTVController	aVolume.GetVolume ();	--
4	aVolume	aStorageDeviceInterface. GetVolume ();	int aVolumeSetting

5	aVolume	aSpeakerInterface. SetVolume (int aVolumeSetting);	--
6	aTVController	aChannel.GetVolume ();	--
7	aChannel	aStorageDeviceInterface.;	int aChannelSetting
		GetChannel ()	
8	aChannel	aChannelDeviceInterface. SetChannel (int aChannelSetting);	--

Creating the State Diagram

Description - The State Diagram (State Transition Diagram) specifies stimulus response logic for a system or class. It specifies the pattern of event messages, conditions, actions, and states. Definitions and steps are presented in the previous section on "Creating the State Diagram".

TV Controller Case Study - The following is the State Diagram for the TV Controller Case Study. The applicable StateMaker diagram is tvstate.sm.

Evaluating the Products in the Dynamic Model

The following are various criteria and questions to ask in evaluating the adequacy of the products in the object model.

- Have we identified the sequence of messages for each event message possibly leading to a response message to an interacting system?

- Are the products in the dynamic model consistent with the products describing the system environment. For example have we modeled all event messages shown on the system event list (messages)?

- Have we modeled state based stimulus response behavior with a state diagram?

- Does the prototype correctly implement the state based stimulus response behavior?

Modeling Classes - The Functional Model

Introduction

To rapidly model and prototype O-O systems, it is important to specify

the rules and formulas for the correct transformation of data. In the functional model we identify those rules and formulas in terms of tranformation rules and correctness assertions. We update the class specification and update the prototype. The purpose of this section is to present a "how to" identify transformation rules and correctness assertions for classes in a system. The key documentation product is the class specification and prototype.

Updating the Class Specifications with Transformations and Correctness Assertions

Description - The class specification provides text information about the class. Class information should include transformations and correctness assertions and other information necessary for programming or documentation. The major question are "What are the transformations in the system?" "How do we ensure correct transformations?". We are interested in formulas, mathematical equations, rules, and correctness assertions that define or limit how we correctly change data values. A correctness assertion is any rule or expression, e.g., operation precondition, operation postcondition, and invariant.

Steps to Create Class Specifications with Transformations and Correctness Assertions

1. Start with the class diagram, data dictionary listing classes, and class specifications.

2. For every operation that changes a data value in an attribute or associated object, ask the questions "What is the rule, expression, formula or equation to correctly change the data value?" "What are the preconditions to ensure the correct transformation of the data value?" What are the postconditons to check that the correct transformation of the data value occurred?" "What are the invariants that must be satisfied at all times?" "What are the exceptions that are raised if an operation cannot correctly transform a data value?"

3. Update each class specification with transformation and correctness assertions.

4. Update the prototype with transformation rules and correctness assertions.

TV Controller Case Study - The following is a portion of the Volume and Channel Class Specifications with transformation and correctness

assertion information.

Volume Class
void increment ()
transformation - volumeSetting = volumeSetting + 1

void decrement ()
transformation - volumeSetting = volumeSetting - 1

Channel Class
void increment ()
transformation - channelSetting = channelSetting + 1
void decrement ()
transformation - channelSetting = channelSetting - 1

Evaluating the Products in the Functional Model

The following are various criteria and questions to ask in evaluating the adequacy of the products in the functional model.

- Have we specified all transactions with rules, expressions, equations, etc?

- Have we specified adequate correctness assertions, e.g. preconditions, postconditions, and invariants to ensure the correct transformation of data?

- Does the prototype implement all transformations and correctness assertions?

Developing the S/W Prototype

Description and Purpose - Our objective is to quickly create a software prototype to get user feedback on requirements, to test the interface elements (menu, dialog boxes, etc), and to test the architecture of classes, message passing, and major decision logic.

The initial prototype is an interface prototype using a screen painter or interface builder (Borland C++ Resource Workshop). The interface prototype tests the interface classes, objects and messages with no internal application logic or calculations. The following are several guidelines to create the interface prototype.
- Create a menu with a menu selection or radio button for each event message,
- Create a dialog box or text IO for problem domain objects,

- Demonstrate so a user can make menu selections and input and output information.

A C++ prototype may be created using With Class, StateMaker, and Borland C++. This prototype should test all classes, objects, and messages with major decision logic from decision tables, state transition diagrams, and if..then rules without extensive algorithmic coding. This prototype may be iteratively updated with additional messages, transformations, and correctness assertions.

Steps to Create the S/W Prototype - The following are steps to create the S/W prototype.

1. Use a screen painter or interface builder (Borland C++ Resource Workshop) to create an interface prototype to verify events represented as menu selections and buttons and to verify problem domain objects represented as text boxes and forms.

2. Generate C++ source code from the class diagram using With Class.

3. Generate C++ source code from the state diagrams using StateMaker.

4. Update the C++ source code as follows:
- integrate C++ source code from the state diagrams,
- add messages from the message scenarios and class diagrams with messages,
- add transformations and correctness assertions from class specifications.

5. In Borland C++, create the project and compile, link, and run.

Summary

In this tutorial we have provided a brief introduction to using With Class and StateMaker CASE tools. We have used a hybrid O-O methodology based upon Rumbaugh's OMT. We have provided a brief "system requirements statement to code" case study "The TV Controller" to show example diagrams and to list the steps to create diagrams and reports. We have listed the steps to automatically generate C++ source code.

References

Booch, Grady "Object Oriented Design with Applications" The Benjamin/Cummings Publishing Co

Coad and Yourdon "Object Oriented Analysis", "Object Oriented Design", and "Object Oriented Programming"

Felsinger, Richard "Object Oriented Analysis and Design" Seminar Course Notes

Harel, David "Science of Computer Programming"

Jacobson, Christerson, Jonsson, Overgaard "Object-Oriented Software Engineering A Use Case Driven Approach"

Shlaer and Mellor "Object Lifecycles Modeling the World in States"

"StateMaker User Diagram" MicroGold Software, Inc
Rumbaugh, Blaha, Premerlani, Eddy, Lorrensen "Object Oriented Modeling and Design" Prentice Hall

"With Class User's Guide" MicroGold Software, Inc.

Glossary of Key Terms

Action - any response to an event message. Actions may be updating an attribute, sending a message, or similar action.

Aggregation (Strong Association) - "Part of" or "bill of material" connection between an assembly and parts that has special semantics for "part of" (transitivity, antisymmetric, and propagation) and for the creation, copy, and deletion of an assembly and parts

Assertion - a rule or expression for correctness, e.g., a data value must always be greater than zero.

Association - A link, connection, or mapping between two or more objects, e.g., "has a", "knows about", "part of", or "bill of materials".

Atomic Class - An atomic class is a primitive data type whose objects are not logically decomposable, e.g., character, boolean, integer, and floating point types.

Attribute - A characteristic or property of an object. An attribute has a name (ID), class or type, and a value.

Class - A definition for one or more objects that have common

attributes, common behavior, common relationships, and common semantics. In S/W a module that encapsulates attributes, operations, exceptions, and relationships.
Cohesion - the degree of internal relatedness of elements within a larger, more complex entity/

Collection Class - A collection class defines an object that holds other objects called elements. Elements may be added or removed from the collection object. Elements are stored in the collection object with an index. Sample collection classes are array, sets, bags, lists, stacks, queues, rings, trees, etc.

Composite Class - A composite class defines an object that has attributes and associated/part objects.

Condition - a guard or boolean that may 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".

Connection or Relationship - A link or connection between classes or objects i.e. association "has a" or "part of", message "calls", and generalization specialization "is a" or "type of".

Control Class - defines time oriented or state based stimulus response behavior including rules and logic to respond to events.

Coupling - the degree of interconnectivity, interdependence, joining and linking of entities.

Dynamic Binding - Late binding (association) of an object name (ID) with an object and its class at run time. The object name may later be associated with a different object and its class. Also, the run time look-up of the correct version of a polymorphic operation.

Entity Class - manages and computes application information. An entity class is independent of protocol details of other interacting systems.

Event - an occurrence or physical phenomenon in the external environment that occurs ar a point in time such as a user pressing a button to which one or more systems must respond. For finite state machines, an event is any stimulus to an object that results in some action and that may result in a transition to a new state.

Event Message - a message from a H/W or S/W system to the system being modeled. An event message requests some action be taken.

Exception - An abnormal, unusual error condition that may result in an operation performing incorrectly. An exception check, e.g., a precondition or postcondition check, is an expression that detects the presence of an exception and invokes an exception handler. An exception handler takes some action, e.g., attempts correct the abnormal condition or notify a user.

Finite State Machine - an entity that has state based stimulus response behavior in which there may be different actions from the same event depending upon the state of the entity.

Generalization Specialization - An "is a" or "type of" connection between superclasses and subclasses.

Generic Parameterized Class (Template) - a class that can be modified with parameters to contain or operate on objects of the parameter class, e.g., a parameterized Stack Class.

Inheritance - The capability of a subclass receive for use attributes, operations, and exceptions defined in a superclass.

Initializer Class - a top level class that initializes the system. It becomes the "main" in a C++ program.

Interface Class - defines communications with another interacting system. An interface class handles event messages and/or response messages. An interface class isolates protocol details to communicate with other interacting systems.

Invariant - a general rule or expression that must be satisfied at all times by all applicable operations.

Message - A call to an object of a class to invoke one of its operations. A one way message is in one direction only from a requester to a server. A two way message is a peer to peer message objects both send and receive messages from each other, i.e. each can initiate a message to each other.

Object Oriented Modeling - A term referring to the modeling phases of object oriented S/W development including analysis, design, and prototyping. It does not include implementation, productization,

testing, etc..

Object Oriented Design - A software development methodology (set of steps) to build systems consisting of classes and objects.

Object - A thing; an instance of a class. An entity that has state (retained information), has behavior (responds to messages), sends and/or receives messages from other objects, and has relationships with other objects. In S/W a variable defined by a class.

Object Oriented Programming - A method to develop software using inheritance, dynamic binding, and polymorphism with object oriented languages such as Smalltalk, Actor, C++, Eiffel, Object Pascal, etc.

Operation precondition - a rule or expression that must be satisfied before the execution of an operation for correct results.

Operation postcondition - a rule or expression that is satisfied upon the correct execution of an operation.

Operation - An action, service, procedure, function that performs some action in response to a message.

Pattern - Two or more entities with a well defined purpose, behavior, connections, and structure, e.g., a tree pattern.

Polymorphism (one name many forms) - An object name may refer to objects of different classes. An operation name may refer to different implementations.

Problem Domain Object - an object that exists in the system environment that is passed in a message to or from the system.

Response Message - a message from the system to other H/W and S/W systems. A response message implements some action requested in an event message.

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.

Static Binding - The association of an object name (ID) with an object and its class at compile time. The object name (ID) is permanently bound to the object and its class for the life of the program.

Subclass - A refined, more specific class of a superclass. It defines more specific, attributes, operations, and exceptions.

Subsystem - a component of a larger system environment. A subsystem has components, e.g., smaller subsystems or classes that are connected together.

Subtype class - defines a specialized class. An object of the subtype may be substituted for an object of the supertype. Objects of supertype and subtype respond to the same messages.

Superclass - A general class that defines the most general attributes, operations, and exceptions which may be inherited by subclasses.

Supertype class - defines a general class that has the same operations (same protocol) as the specialized subtype classes.

System Environment - a complex system that has other systems (subsystems) as components.

System - a general term for a complex entity that can be treated as a unit and that has simpler components that work together to perform a function. The system is the the S/W system to be developed. The system which is the center of focus and which becomes a single program (.EXE file). The system generally consists of 10 to 100 classes as a very rough order of magnitude.

Transformation - a description of how a data value may be correctly changed in a formula, expression, table, etc.

Transition - 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.

Weak Association - "Has a" or "knows about" connection between associated objects that does not have aggregation semantics.