Chapter 4 – Domain Modeling

Table of Contents

Chapter Overview

The focus of this chapter is to identify and define the entities or problem domain classes. Traditional system development methodologies called these “things” data entities and used Entity-Relationship diagrams to model the data structures. Object-oriented techniques call these “things” problem domain classes, or just classes for short. A UML class diagram is used to model the data structure. Either type of model can be used and results in equivalent solutions. Students should be familiar with both Entity-Relationship diagrams and Class diagrams.

The first part of the chapter is a discussion to help students find, identify, and classify these “things.” The second part of the chapter instructs how to build an entity-relationship diagram, and the final topic in the chapter is how to build a class diagram domain model.

The objects identified during analysis can have a dynamic existence or life. In other words, they come into existence, they change from state to state, and they can be destroyed. This life span of an object is documented through the use of a State Machine Diagram.

Learning Objectives

After reading this chapter, the student should be able to:

- Explain how the concept of “things” in the problem domain also defines requirements
- Identify and analyze data entities and domain classes needed in the system
- Read, interpret, and create an entity-relationship diagram

- Read, interpret, and create a domain model class diagram
- Understand and interpret the domain model class diagram for RMO
- Read, interpret, and create a state machine diagram that models object behavior

Notes on Opening Case and EOC Cases

Opening Case

Waiters On Call Meal-Delivery System (Part 2): Two points can be emphasized in this case. First the analyst is identifying those "informational things" that the system needs to keep track of. Those things define the problem domain classes and include restaurants, menu items, customers, orders, and so forth. This short case does not attempt to distinguish between classes and attributes. The other point is that the analyst continues to ask probing and detailed questions. He does this both to ensure he has identified all the classes, but also so that he understands all the details (attributes) that will eventually need to be defined.

EOC Cases

The running cases go throughout the rest of the book, e.g. for each chapter.

Metropolitan Car Service Bureau: This end-of-chapter case describes a system to maintain information about car service records. The case describes classes, attributes, and relationships. The problems are to draw a UML class diagram and answer questions about the business relationships based on the class diagram. The problems help the student understand what kind of information is captured in a domain model.

Community Board of Realtors (running case)**:** Community Board of Realtors is a professional organization that supports real estate offices and agents. This case description gives information about domain classes and relationships. The students will have to make some assumptions about relationship multiplicity values. The problems require the student to draw UML class diagrams. Each problem enhances the model by adding generalization/specialization and more complex relationships.

Spring Breaks 'R' Us Travel Services (SBRU) (running case)**:** SBRU is an online travel services that books spring break trips to resorts for college students. The system will also contain a social networking subsystem to allow students to connect with each other during the spring break vacation. This case focuses on the social networking subsystem. The students are asked to identify classes and attributes, and then build a UML class diagram. Students will have to make several assumptions about classes, attributes and relationships.

On the Spot Courier Services (running case)**:** On the Spot is a small, but growing, courier service that needs to track customers, package pickups, package deliveries, and delivery routes. The case description for this chapter is quite elaborate and describes both processing and data that will occur for the system. Students are asked to use the noun technique and identify nouns that become classes and attributes, using both this narrative and the narrative in previous chapters. Students then build relationships and a final class diagram with class, attributes, primary keys, relationships, and multiplicity constraints. This case is quite rich.

Sandia Medical Devices (running case)**:** Sandia Medical Devices is a company that specializes in medical monitoring through remote, mobile telecommunication devices. This case provides a basic class diagram for the system. Additional requirements are described to enhance the class diagram with more classes and relationships. In particular association classes and generalization/specialization classes and abstract classes can be added. The case description is short, but the diagram is elaborate.

Instructor's Notes

"Things" in the Problem Domain

Key Terms

- **problem domain** – the specific area (or domain) of the user's business need (or problem) that is within the scope of the new system
- **brainstorming technique** – a technique to identify problem domain objects in which developers work with users in an open group setting
- **noun technique** – a technique to identify problem domain objects by finding and classifying the nouns in a dialog or description
- **attributes** – descriptive pieces of information about things or objects
- **identifier or key** – an attribute the value of which uniquely identifies an individual thing or object
- **compound attribute** – an attribute that consists of multiple pieces of information but is best treated in the aggregate
- **association** – a term, in UML, that describes a naturally occurring relationship between specific things, sometimes called a relationship
- **relationship** – a term that describes a naturally occurring association between specific things, sometimes called an association
- **cardinality** – a measure of the number of links between one object and another object in a relationship
- **multiplicity** – a measure, in UML, of the number of links between one object and another object in an association
- **multiplicity constraints** – the actual numeric count of the constraints on objects allowed in an association
- **binary associations** – associations between exactly two distinct types of things
- **unary association** – an association between two instances of the same type of thing
- **ternary association** – an association between exactly three distinct types of things
- **n-ary association** – an association between n distinct types of things

Lecture Notes

A basic problem with data modeling is what to use for the generic term of items in the real world that a system analyst is trying to discover. We have chosen to simply call them "things." Although "things" is a very ambiguous term that can be used to identify almost any real world item or abstract idea, that information is in fact what data modeling is trying to capture. The discussion on finding "things" is really quite comprehensive and should provide a good foundation for the students.

The term "**problem domain**" is an interesting word choice. Often when we think of problem, we think that something is broken. However, in this context, the word problem simply means a need that requires a business solution. In other words, system developers are developing a solution for a business need or business problem. Hence the term problem domain, or the context and arena in which the business problem exists.

The Brainstorming Technique

This technique is a joint effort between the analyst and the users. Figure 4-1 illustrates the many different types of things that should be considered. "Things" ranges from very tangible things such as books and vehicles to roles such as employees and customers to quite abstract things such as flights and orders. (You cannot observe a flight or an order with any of your five senses.)

Here are the steps to follow when using the brainstorming technique:

1. Identify a user and a set of use cases.
2. Brainstorm with the user to identify things involved when carrying out the use case—that is, things about which information should be captured by the system.
3. Use the types of things (categories) to systematically ask questions about potential things, such as the following: Are there any tangible things you store information about? Are there any locations involved? Are there roles played by people that you need to remember?
4. Continue to work with all types of users and stakeholders to expand the brainstorming list.
5. Merge the results, eliminate any duplicates, and compile an initial list.

Care should be taken to distinguish between classes and attributes. See the next sections.

The Noun Technique

The noun technique is a more mechanical approach to identifying classes, but it is also a powerful technique. The basic idea is to identify all the nouns, which are always some type of "thing," and build the list of all these nouns. Then refine the list to remove duplicates and identify which are classes and which are attributes of classes.

Here are the steps to follow when using the noun technique:

1. Using the use cases, actors, and other information about the system—including inputs and outputs—identify all nouns.
2. Using other information from existing systems, current procedures, and current reports or forms, add items or categories of information needed.

3. As this list of nouns builds, you will need to refine it. Ask these questions about each noun to help you decide whether you should include it:
    - Ask questions, such as is it important and inside the scope, to see if it should be included.
    - Ask questions to see if it really should be excluded, such as is it only a report or an input or is it an attribute.
    - Ask questions to see if it needs further research. In other words that you cannot answer whether it needs to be included or excluded.
4. Create a master list of all nouns identified and then note whether each one should be included, excluded, or researched further.
5. Review the list with users, stakeholders, and team members and then refine the list of things in the problem domain.

Attributes of Things

The noun technique involves listing all the nouns that come up in discussions or documents about the requirements. Many of these nouns are actually **attributes**. During the refinement of the nouns on the list of nouns, an evaluation of each noun will determine whether the noun is an independent item, e.g. a class, or whether it describes or characterizes another noun and hence is an attribute. One attribute may be used to identify a specific thing, such as a Social Security number for an employee or an order number for a purchase. The attribute that uniquely identifies the thing is called an **identifier or key**.

Associations among Things

An **association** is a naturally occurring relationship between specific things, such as *an order is placed by a customer* and *an employee works in a department*. *Is placed by* and *works in* are two associations that naturally occur between specific things. Information systems need to store information about employees and about departments, but equally important is storing information about the specific associations.

In database management, the term **relationship** is often used in place of association, which is the term used when modeling in UML. We will use *association* in this book because we emphasize UML diagrams and terms.

It is also important to understand the nature of each association in terms of the number of links for each thing. **Cardinality** can be one-to-one or one-to-many. The term **multiplicity** is used to refer to the number of links in UML and should be used when discussing UML models. Multiplicity is established for each direction of the association. It is important to describe not just the multiplicity but also the range of possible values of the multiplicity (the minimum and maximum multiplicity). These terms are referred to as **multiplicity constraints**. The next sections on the notation explain more about the various combinations of cardinality and multiplicity constraints.

Associations between two classes are called **binary associations**. Sometimes an association will be between different elements in the same class or set. These are called **unary associations**. Associations

can also exist between multiple classes, such as three classes – a **ternary association**, or some arbitrary number n – **n-ary associations.**

Quick Quiz

Q: What are the two techniques discussed in the chapter to identify data entities or domain classes? What is the difference between these two techniques?

> A: Brainstorming technique and the noun technique. Brainstorming is a cognitive technique to work with users to identify all those things that must be "remembered" by the system. Noun technique is initially more rote in that it lists all nouns. Cognitive analysis is performed on the noun list to identify which are really domain classes.

Q: What is the difference between a domain class and an attribute? Both are nouns.

> A: A domain class is an independent item that must be recorded individually. Attributes are descriptors for the domain classes and are recorded as part of the class.

Q: What is the difference between a relationship and an association?

> A: They are the same. Relationship is used with database terminology (and entity-relationship models as discussed next), and association is used with UML object-oriented modeling.

Q: What is the difference between cardinality constraint and a multiplicity constraint?

> A: Cardinality is used with entity-relationship modeling and multiplicity is used with UML modeling.

Q: What is a binary association?

> A: It is an association between exactly to separate classes.

The Entity-Relationship Diagram

Key Terms

- **data entities –** the term used in an ER diagram to describe sets of things or individual things
- **entity-relationship diagram (ERD) –** a diagram consisting of data entities (i.e., sets of things) and their relationships
- **semantic net –** a graphical representation of an individual data entity and its relationship with other individual data entities

Lecture Notes

A data model commonly used by traditional analysts and database analysts is called the **entity-relationship diagram (ERD)**. The ERD is not a UML diagram, but it is very commonly used and is quite similar to the UML domain model class diagram.

Examples of ERD Notation

On the entity-relationship diagram, rectangles represent **data entities**, and the lines connecting the rectangles show the relationships among data entities. Cardinality constraints are expressed by the notations on the lines. Remember that the cardinality applies to each end of the relationship and contains and minimum and maximum value. Refer Figure 4-7 to note the bars, circles, and crows-feet that designate minimum and maximum cardinality values. A bar represents one, a circle represents zero, and crows-feet represent many.

Sometimes it is obvious what the relationships between data entities should be. Often it is also clear what the cardinality constraints should be. However, when it is not clear, creating a semantic net can be helpful. A **semantic net** is simply a drawing of the individual data entities showing what relationships can exist. For example in Figure 4-9, orders are always only connected to one customer, but a customer will have multiple orders. It is even possible to have a customer, such as Mary, that has not made any orders yet. A semantic net can be useful to help define cardinality constraints.

Quick Quiz

Q: What are the three symbols to denote cardinality constraints? What do they mean?

A: A bar represents one, a circle represents zero, and crows-feet represent many.

Q: What is a semantic net and what is it used for?

A: A semantic net is a drawing of individual data entities with the individual links between them. It is useful to help to define accurately the cardinality constraints.

Q: How is a primary key denoted on an ERD?

A: With a suffix of PK after the attribute.

The Domain Model Class Diagram

Key Terms

- **class –** a category or classification of a set of objects or things
- **domain classes –** classes that describe objects from the problem domain
- **class diagram –** a diagram consisting of classes (i.e., sets of objects) and associations among the classes
- **domain model class diagram –** a class diagram that only includes classes from the problem domain
- **camelback notation or camelcase notation –** when words are concatenated to form a single word and the first letter of each embedded word is capitalized
- **association class –** an association that is also treated as a class; often required in order to capture attributes for the association

- **generalization/specialization relationship –** a type of hierarchical relationship in which subordinate classes are subsets of objects of the superior classes; an inheritance hierarchy
- **superclass –** the superior or more general class in a generalization/specialization relationship
- **subclass –** the subordinate or more specialized class in a generalization/specialization relationship
- **inheritance –** the concept that specialization classes inherit the attributes of the generalization class
- **abstract class –** a class that describes a category or set of objects but that never includes individual objects or instances
- **concrete class –** a class that allows individual objects or instances to exist
- **whole-part relationship –** a relationship between classes in which one class is a part or a component portion of another class
- **aggregation –** a type of whole-part relationship in which the component parts also exist as individual objects apart from the aggregate
- **composition –** a type of whole-part relationship in which the component parts cannot exist as individual objects apart from the total composition

Lecture Notes

Current approaches to system development, primarily object-oriented approaches, use the term *class* rather than data entity and use concepts and notations based on UML to model the things in the problem domain. A **class** is a category or classification used to describe a set of objects. So a class is a category, but it is also a set of objects. Classes that describe things in the problem domain are called **domain class**es. Domain classes have attributes and associations. Multiplicity (called cardinality in an ERD) applies among classes.

On a **class diagram**, rectangles represent classes, and the lines connecting the rectangles show the associations among classes. The domain class symbol is a rectangle with two sections. The top section contains the name of the class, and the bottom section lists the attributes of the class. Class names are capitalized and attributes begin with lower case. Both use **camelcase** notation. Later, you will learn that the design class symbol includes a third section at the bottom for listing methods of the class; methods do not apply to problem domain classes.

Domain Model Class Diagram Notation

This section contains many examples of UML class diagram notation as explained above. **Multiplicity** is denoted by numbers placed next to the class at either end of a binary association line. Minimum and maximum multiplicity is denoted by 0..1 (zero or one) and 1..* (one/many), and so forth. Figure 4-13 summarizes multiplicity notation.

Many-to-many associations allow data objects from one class to be connected to many objects of the associated class, and this association occurs in both directions. Sometimes the association itself will need to have attributes associated with it. An example in the book (Figures 4-15 and 4-16) where a

CourseSection will have many students and a Student will be registered in many course sections. The student also will receive a grade for that course section. The grade cannot be associated solely with the student, nor with the course section. The grade belongs to the association. In other words we need to treat the CourseSection—Student association as a class. As shown in Figure 4-16 that is done by a dashed line to a class called an **association class**. (Note the troubleshooting tips for this topic. Students often have problems identifying association classes.)

More Complex Issues about Classes of Objects

Generalization/specialization relationships are based on the idea that people classify things in terms of similarities and differences. A generalization/specialization relationship is used to structure these things from the more general to the more special. Each class of things in the hierarchy might have a more general class above it, called a **superclass**. At the same time, a class might have a more specialized class below it, called a **subclass**. UML class diagram notation uses a triangle that points to the superclass to show a generalization/specialization hierarchy.

A generalization/specialization relationship is also a set/subset relationship. All of the objects in the specialized class are also within the generalized class. Thus all of the attributes in the generalized class are also attributes of the specialized class. This is called **inheritance**, since the subclass or subset inherits the attributes of the superclass or superset. Figure 4-19 is an example. Sometimes the superclass is defined only to allow inheritance of attributes, but has no actual real-life objects in it. In that situation, we call the superclass an **abstract class**, and is denoted by an italicized name. A class that does have objects is a **concrete class**.

Whole-part relationships are used to show an association between one class and other classes that are parts of that class. There are two types of whole-part relationships: **aggregation** and **composition**. Aggregation refers to a type of whole-part relationship between the aggregate (whole) and its components (parts), where the parts can exist separately and is represented by an open diamond. Composition refers to whole-part relationships that are even stronger, where the parts, once associated, can no longer exist separately, and is represented by a solid dark diamond. Whole-part relationships—aggregation and composition—mainly allow the analyst to express subtle distinctions about associations among classes. As with any association, multiplicity can apply.

Quick Quiz

Q: What are the symbols shown in a class diagram?

> A: A rectangle with three sections inside for class, a line with a triangle for generalization/specialization (inheritance), a line with a diamond for aggregation (whole part), and a line with multiplicity written on it for association relationships.

Q: What is an association class? Where can it exist?

> A: It is an association, e.g. a relationship that is also a class. It can have attribute as any class. Only associations that have many-to-many multiplicity can have an association class.

Q: What is the difference between aggregation and composition?

> A: Aggregation is an “aggregate” of objects that are combined together. In other words, the

objects also exist independently. Composition is a "composite" object and the subparts cannot exist without being a member of the composition.

Q: What is the difference between generalization/specialization and inheritance.

A: Generalization/specialization refers to the relationship between classes, and inheritance refers to the concept that specialized classes inherit from the generalized class. They always occur together, but they are slightly different concepts.

Q: What is an abstract class?

A: It is a class with no objects solely contained within it.

The Ridgeline Mountain Outfitters Domain Model Class Diagram

Lecture Notes

This section presents the class diagrams for the Ridgeline Mountain Outfitters. The two subsystems are presented individually, and then are combined together into a comprehensive class diagram. The two subsystems are the Sales Subsystem and the Customer Account Subsystem.

Quick Quiz

Various quiz questions can be used to ensure that the students can read the class diagrams correctly. For example using the Sales Subsystem class diagram:

Q: Which are the abstract classes? What does that mean?

A: Sale and OnlineCart. It means that all objects exist in the specialization classes and not the generalization class.

Q: Can a ProductComment apply to more than one ProductItem? How many comments can a ProductItem have?

A: No. Many.

Q: Can this model indicate which customer is associated with a SaleTransaction?

A: Yes, because each SaleTrans is associated with only one Sale and each Sale is associated with one Customer.

From the Customer Account Subsystem:

Q: What does the FriendLink class mean? How many can exist for a given Customer?

A: A FriendLink class is an association class from one customer to another customer. There can be many in either direction.

Q: Why are there two associations between Customer and Message? What does it mean?

A: Each message has two associations to customers, the From customer and the To customer. A message can only come from one customer, but it may go to many customers. And a customer may send or receive many messages.

Q: How many addresses can a customer have? Can there be an address without any customer?

A: A customer can have many addresses. Yes there can exist an address without a customer.

The State Machine Diagram – Identifying Object Behavior

Key Terms

- **state** – a condition during an object's life when it satisfies some criterion, performs some action, or waits for an event
- **transition** – the movement of an object from one state to another state
- **state machine diagram –** a diagram showing the life of an object in states and transitions
- **pseudostate –** the starting point of a state machine diagram, indicated by a black dot
- **destination state –** for a particular transition, the state to which an object moves after the completion of a transition
- **origin state –** for a particular transition, the original state of an object from which the transition occurs
- **action-expressions –** descriptions of the activities performed as part of a transition
- **guard-condition –** a true/false test to see whether a transition can fire
- **concurrency** or **concurrent states –** the condition of being in more than one state at a time
- **path –** a sequential set of connected states and transitions
- **concurrent paths –** when one or more states in a path are parallel to one or more states in another path

Lecture Notes

Foundation

Sometimes, it is important for a computer system to maintain information about the status of problem domain objects. The status condition for a real-world object is often referred to as the state of the object. a **state** of an object is a condition that occurs during its life when it satisfies some criterion, performs some action, or waits for an event. For real-world objects, we equate the state of an object with its status condition.

A state might have a name of a simple condition, such as On or In repair. Other states are more active, with names consisting of gerunds or verb phrases, such as *Being shipped* or *Working*. The name of a state shouldn't be an object (or noun); it should be something that describes the object (or noun).

States are described as semi-permanent conditions because external events can interrupt a state and cause the object to go to a new state. An object remains in a state until some event causes it to move, or

transition, to another state. A **transition**, then, is the movement of an object from one state to another state. Transitioning is the mechanism that causes an object to leave a state and change to a new state. States are semi-permanent because transitions interrupt them and cause them to end. Generally, transitions are short in duration—compared with states—and they can't be interrupted.

A **state machine diagram** is composed of ovals representing the states of an object and arrows representing the transitions. After a transition begins, it runs to completion by taking the object to the new state, called the **destination state**. A transition begins with an arrow from an **origin state**—the state prior to the transition—to a destination state, and it is labeled with a string to describe the components of the transition.

The transition label consists of the following syntax with three components:

transition-name (parameters, ...) [guard-condition] / action-expression

any of which may be empty and which have the following meanings:

- **transition-name –** the trigger or event that causes the transition to fire
- **parameters –** any parameters that need to be passed to the object from the triggering object
- **guard-condition –** this "guards" the transition, or prohibits it from executing, even if the trigger fires, unless the guard condition is true
- **action-expression –** some actions that must be completed as part of the transition. The action expression may require the input parameters.

Concurrency and Concurrent States

In the real world, it is very common for an object to be in multiple states at the same time. The condition of being in more than one state at a time is called concurrency, or **concurrent state**s. One way to show this is with a synchronization bar and concurrent paths, as in activity diagrams. A sequential set of connected states and transitions become a **path**. An object travels along the path by transitioning from state to state. Not only can an object be in multiple separate states at the same time, but there can exist **concurrent paths**, each of which has multiple states.

Rules for Developing State Machine Diagrams

Usually, the primary challenge in building a state machine diagram is to identify the right states for the object. It is often helpful to have students do object think exercises and pretend to be the object itself, such as "I am an order," "how do I come into existence," and "what are my status conditions that need to be recorded."

The other major area of difficulty for new analysts is to identify and handle composite states with nested threads. Usually, the primary cause of this difficulty is a lack of experience in thinking about concurrent behavior. The best solution is to remember that developing state machine diagrams is an iterative behavior—more so than developing any other type of diagram. Analysts seldom get a state machine diagram right the first time. They always draw it and then refine it again and again.

Finally, don't forget to ask about an exception condition—especially when there are the words verify or check. Usually, there will be two transitions out of states that verify something: one for acceptance and

one for rejection.

Here is a list of steps that will help in developing state machine diagrams:

1. Review the class diagram and select the classes that might require state machine diagrams.
2. For each selected class in the group, make a list of all the status conditions that can be identified
3. Begin building state machine diagram fragments by identifying the transitions that cause an object to leave the identified state
4. Sequence these state-transition combinations in the correct order
5. Review the paths and look for independent, concurrent paths
6. Look for additional transitions
7. Expand each transition with the appropriate message event, guard-condition, and action-expression
8. Review and test each state machine diagram
    - Make sure states apply to that – the correct – object class
    - Trace the entire life cycle of the object to make sure you have all states
    - Handle all exception conditions as well as normal paths
    - Double check for concurrent paths

Developing RMO State Machine Diagrams

This section contains two examples of how to develop a state machine. Each state machine diagram is for a single class. For these examples the *SaleItem* and the *InventoryItem* classes are chosen. First a state machine diagram for *SaleItem* is developed and then a state machine diagram for *InventoryItem* is developed.

The example to create the *SaleItem* state machine diagram follows the eight step program as identified above. Figure 4-28 shows down through step three with the states and exit transitions identified. Figure 4-29 completes step four with a preliminary partial state machine that includes the states and transitions in the correct order. Notice that is it is not complete. There are not start and end points. Additional transitions, such as for error conditions or backward transitions, have not yet been identified. Figure 4-30 is the completed state transition machine. The beginning and ending transitions have been added. Transitions to handle back ordered situations have been added, along with an action expression to make a purchase order. This state transition machine is straightforward with no need to have concurrent states or complex paths.

The state machine diagram for the *InventoryItem* is slightly more complex. In this situation, it is noted that an *InventoryItem* can be in stock or not in stock. It can also be on order or not. As we consider these different states we conclude that these set of states are independent. Figure 4-31 identifies these states and exiting transitions. Figure 4-32 connects them together. Note that there are two independent paths. The solution is simply to keep these two paths separate by placing a synchronization bar at the beginning and ending to indicate that they are in fact concurrent parallel paths. Figure 4-33 illustrates

this final solution.

The benefit of developing a state machine diagram for a problem domain object is that it helps you capture and clarify business rules. As always, the benefits of careful model building help us gain a true understanding of the system requirements.

Quick Quiz

Q: What is a state? What is the difference between a state, a concurrent state, and a composite state?

> A: A state is a semi-permanent condition of an object. A state is simply a statement of an object's condition. A concurrent state is where an object is in multiple states at the same time. Each of those states are concurrent states. A composite state, is a high-level state such that other states and conditions can be occurring simultaneous to the object remaining in the composite state.

Q: What is the difference between a state and a transition?

> A: A state is the semi-permanent condition of an object. A transition is a movement from one state to another state, and usually occurs rapidly. A transition moves to completion so that the object will move to another semi-permanent condition.

Q: How many objects does a state machine diagram apply to?

> A: All of the objects in a single object class.

Q: What is in the label for a transition? What does each part mean?

> A: Name = trigger, Parameter = input data, guard-condition = true/false test to see if transition can execute, action-expression = executable function that must complete before transition is considered complete.

Classroom Activities

1. Patient Record and Scheduling System Exercise

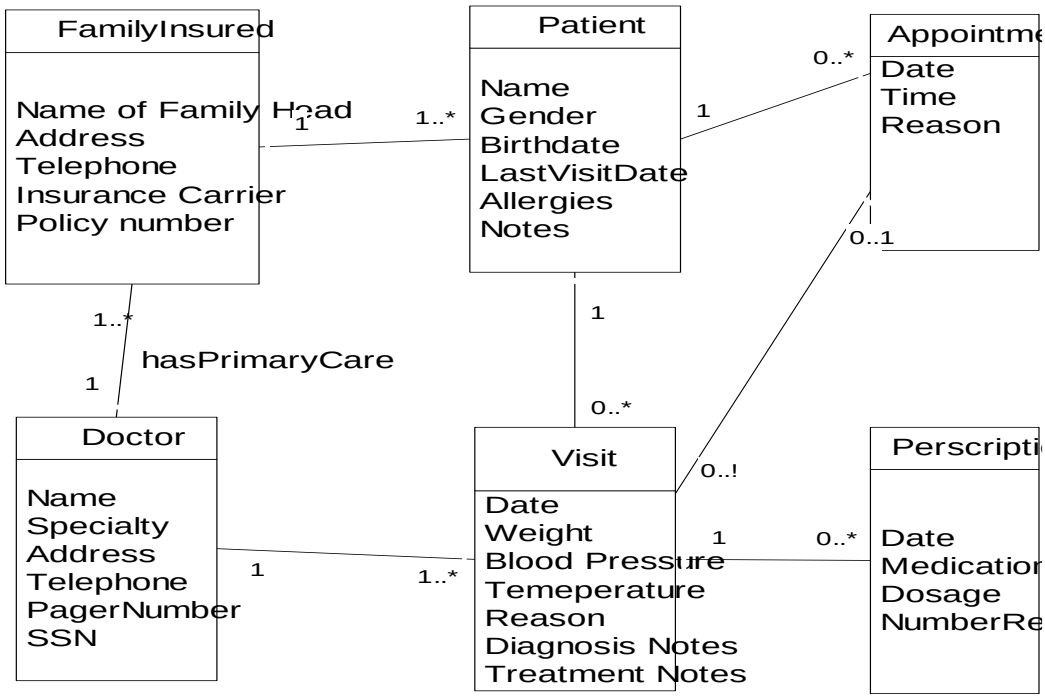
In the Instructor's Manual for Chapter 3 we provided a short Doctor-Patient case to use for an in class exercise to identify use cases. You can use this same case as an in class example of identifying classes and creating a class diagram. We repeat the description here and provide an example class diagram.

The following short case makes a good classroom example without having to use the chapter cases.

Patient Record and Scheduling System

A patient record and scheduling system in a doctor's office is used by the receptionists, nurses, and doctors. The receptionists use the system to enter new patient information when first-time patients visit the doctor. They also schedule all appointments. The nurses use the system to keep track of the results of each visit including diagnosis and medications. For each visit, free form text fields are used captures information on diagnosis and treatment. Multiple medications may be prescribed during each visit. The nurses can also access the information to print out a history of patient visits. The doctors primarily use the system to view patient history. The doctors may enter some patient

treatment information and prescriptions occasionally, but most frequently they let the nurses enter this information. -- Each patient is assigned to a family. The head of family is responsible for the person with the primary medical coverage. Information about doctors is maintained since a family has a primary care physician, but different doctors may be the ones seeing the patient during the visit.



2. Object-Think Exercise

An "Object Think" Exercise

Sometimes students have difficulty understanding the object-oriented approach, particularly if they have experience with traditional structured development. It is important to demonstrate how the object-oriented approach actually works. The following exercise, loosely based on the "Object Think" approach of Peter Coad, should help convey the key points of the object-oriented approach.

One of the problems that students will have is distinguishing between a transient object and a persistent object. Students who have had programming classes have learned about transient objects – things like windows, buttons, text boxes, and other user interface objects. They may not have learned about persistent objects – things like customers, orders, and payments. Obviously, the difference is that persistent objects, often referred to as problem domain objects, are usually maintained in a database. The other difficult concept that needs to be taught is the difference between a programming object, i.e. one living in memory, and a database record. Later chapters address these ideas in more depth, but an introduction to problem domain objects should be done in this chapter.

The following exercise is a fun class activity which illustrates what programming objects are, and how user interface objects, e.g. transient objects, and problem domain objects, e.g. persistent objects are different. We present this activity in substantial detail.

You can present this exercise to the class earlier during your discussion of Chapter 2 if your course emphases the object-oriented approach. During the exercise, people pretend to be objects that are capable of knowing information and knowing how to do activities when told. They respond to messages by acting as though they are objects. Data files or database tables do not behave in this way—they are acted upon by programs. But objects know what to do when told.

This exercise includes user interface objects that the students are probably familiar with, as well as problem domain objects that they may find more difficult to imagine as having behavior. The exercise simulates an object-oriented system.

The following is a simplified system. Students at a university must choose to enroll in a college, such as business or arts and sciences. The class diagram for this system would show College attributes (college name, dean, and office location) and Student attributes (student ID and name). This system responds to two events: *Student wants to enroll in a college* and *Time to produce enrollment reports for administration*. The system implements the activities/use cases *Enroll student in college* and *Produce college enrollment reports*. The user and the computer work together to carry out an activity when the event occurs.

The user interface of the system includes a command button that enrolls a student in a college, a command button that prints a list of the students enrolled in a college, a list box that lists the colleges, a text box for typing in the name of a student, and a printer that writes information on the board. Note that there is one command button for each system activity (each based on an event).

Appoint nine students to play the following roles: three students, one college, the print button, the enroll student button, the list box, the text box, and the printer. The instructor plays the role of the user of the computer system.

Prepare a script for each participant based on the object that he or she plays. The participant should read his or her script part out loud and make notes as necessary.

The User Interface Objects

Print Button:
I am a button.
I know my caption says, "Print College Enrollment."
I know how to be clicked.
When I am clicked, I:

> Ask the list box what college it has selected.
> "Hey, List Box, what college is selected?"
>
> Ask that college to print its enrolled students.
> "College _____, print your enrolled students."

Enroll Button:
I am a button.
I know my caption says, "Enroll Student."
I know how to be clicked.
When I am clicked, I:

> Ask the list box what college it has selected.
> "Hey, List Box, what college is selected?"
>
> Ask that college to enroll a new student.
> "College _____, enroll a new student."

Text Box:
I am a text box.
The text typed into me is _____ (blank at first).
I know how to be typed into.
I know how to provide what is typed into me to a requester.
When asked to provide what's typed into me, I:

> Tell the requester what is typed into me.

List Box:
I am a list box.
I know my caption says, "Colleges."
I know the items in my list include: Business and Arts/Sciences.
I know the selected item is _____ (blank at first).
I know how to have an item selected.
I know how to tell a requester what item is selected.
When asked to provide what item is selected, I:

Tell the requester the item selected is _____.

Printer:
I am a printer.
I know my font size is _____ (small medium or large).
I know my color is _____ (black, yellow, red, and so on).
I know how to set my font size when asked.
I know how to set my color when asked.
I know how to print something.
When asked to print something, I:

Write what I'm told on the board using my font size and color.

The Problem Domain Objects

Student:
I am a student.
I know my name is _____.
I know my hometown is _____.
I know I am enrolled in the college _____ (blank at first).
I know how to enroll myself in a college when asked.
To enroll myself in a college, I:

Add the college name I am given to the space above.

I know how to provide my name and hometown when asked.
To provide my name and hometown, I:

Tell the requester my name is _____ and my hometown is _____.

College:
I am a college.
I know my name is _____.
I know my dean is _____.

I know my location is _____.
Enrolled students include _____, _____, _____ (blank at first).
I know how to enroll a new student when asked.

To enroll a new student, I:

> *Get the name of the student from the text box.*
>
> "Text Box, what is the value typed into you?"
>
> Tell the student object with that name to enroll itself in me.
> "Student _____, enroll yourself in the college named _____."
>
> Add the student's name to my list of enrolled students above.

I know how to print a list of enrolled students when asked.
To print enrolled students, I:

> Get the first name of an enrolled student from my list above.
>
> Ask the student object to provide its name and hometown.
> "Student _____, what is your name and hometown?"
>
> Ask the printer to print the student's name and hometown.
> "Printer, print student name _____ and hometown _____."
>
> Repeat for each student in the enrolled students list.

Procedure

1. Give one sheet of paper to each volunteer. The college has the most work to do. Have the person playing the college of business fill in the appropriate information on the sheet. Initially, there will be only one college, although you can add a second college if you desire to expand the simulation. Have each student fill in his or her name and hometown on the sheet. The instructor also needs to note who the students are.

2. Line up the objects at the front of the room, with the buttons, list box, and text box on one side, students and the college in the middle, and the printer on the other side (near the board). Give the printer a few board markers (various colors).

3. Have each object read off what it knows and what it knows how to do from the script. The idea is that each object has attributes and methods and knows what to do when asked. We assume the student objects already exist; they have been added to the system but are not yet enrolled in a specific college.

4. The instructor is the user. The user interacts with interface objects only. The objects, if they follow their instructions, should carry out the processing required. Note that there are no programs here, just a collection of objects that interact with each other.

   When everyone is ready, the user says:

   "Printer, set your font size to medium."
   "Printer, set your color to _____ (whatever)."
   "List Box, I am selecting "College of Business."
   "Text box, I am typing _____ (name of one of the students) into you."
   "Enroll Button, I am clicking you."

   What should happen is this:

   Enroll Button asks List Box for the College selected.
   Enroll Button asks College of Business to enroll a Student.
   College of Business asks Text Box for the name of the Student.
   College tells that Student to enroll itself in College of Business.
   Student writes College of Business as its College.
   College adds the name of the Student to its list of enrolled Students.

"Text box, I am typing _____ (name of second student) into you."
"Enroll Button, I am clicking you."
   As before, note that List Box still has college of business selected.
"Text box, I am typing _____ (name of third student) into you."
   Same as before.
"Enroll Button, I am clicking you."
   Same as before.
"Print Button, I am clicking you." What should happen is this:
   Print Button asks List Box for the College selected.
   Print Button asks that College to print enrolled Students.
   College asks first Student in its enrollment list for name and hometown.
   College asks Printer to print name and hometown.
   Printer writes name and hometown on board using current font and color
   (last three steps repeat for each Student).

Conclusion

This exercise should provide a tangible experience that demonstrates how the object-oriented approach works. Objects interact in the system. Each object plays its part when asked. Some are user interface objects, and some are problem domain objects. During the analysis phase, the problem domain objects are modeled using the class diagram, and the interaction is modeled using the sequence diagram (see Chapter 7). During the design phase, interface objects are added to provide an interface for the user to

use to interact with the problem domain objects. These examples have some design flaws, but try not to worry about that too much at this point. If you want to emphasize design with this example, you could ask students to redesign the interaction.

Troubleshooting Tips

Sometimes students have a hard time understanding object-oriented concepts. Although those students who have been doing OO programming have less problem. The above Object-think exercise will help students understand objects and the difference between transient object (like user interface objects), and persistent objects (problem domain objects).

Sometimes students also struggle with the basic idea of a class, or classification. It is often helpful to approach the concept of classes both from the abstract idea of a classification, and from the concept of a set of objects. A class is always a set of objects. Generalization/specialization is a set/subset relationship. An abstract class is a set with no objects allowed other than the subset objects, e.g. it is only a classification.

Students also often have trouble identifying association classes. An association class only exists if the association between two classes is many-to-many, AND a candidate primary key of the association class must always be the concatenation of the primary keys of the two associated classes. For example, in the Patient Record and Scheduling System, students will sometimes define a relationship between the doctor and the patient called *Patient visits Doctor.* Since a doctor sees many patients, and a patient can see many doctors they define it as many-to-many. However, this is an invalid association because it cannot have an association class. The key of Doctor class (DrID) and Patient class (PatientID) concatenated together DrID-PatientID do NOT yield a unique identifier for *Patient visits Doctor* because the same patient may visit the same doctor today and again tomorrow. That key does not uniquely identify a particular visit. The correct way to model it is as shown. An alternative way to model it would be with a ternary relationship of Doctor-Patient-Date&Time. Those concatenated keys would produce a unique association class key.

State machine diagram

This model can also be somewhat difficult for students. The important point that they should grasp is that a state machine diagram describes the life of a single object in an object class. (Of course all objects in that class will have the same behavior.)

Probably the key to creating a state machine diagram is to be able to identify the correct states. One good way is to have the students do the object think exercise. Another way is to think about all of the status conditions for an object, particularly those status conditions that are of interest to the outside world. That is a good place to start to identify states.

The other major difficulty is to identify concurrent states, where an object is doing two things at the same time. Modeling concurrent states can get rather complex, either as parallel paths or as composite states. Some students will prefer parallel paths, while others will prefer composite states. Either will work. There does not seem to be an easy way for students to be able to do this successfully, other than practicing.

Discussion Questions

1. Identifying "Things" in the problem domain.

This chapter provided two techniques to identify classes from the problem domain. There are many discussion questions that could help students understand this process.

1. Which technique is better? When might one be better than the other? When might it be appropriate to use both techniques?
2. How do you know if you have identified all the domain classes? How do you know if you have identified the right classes? What can you do to validate the classes you have identified?
3. How can you distinguish between classes and attributes? Sometimes nouns can be either a class or an attribute (for example Date). How can you know whether to make it a class or an attribute?
4. If you have one solution with a problem domain model and a colleague has a different one, can they both be correct? How can you tell if it is a correct model? How would you choose which might be a better solution?

2. Choosing which relationships to include in the domain model.

Often there are many relationships that can be identified between object classes. However, it is not always correct to model all relationships. The following questions might be interesting.

1. Is it possible to have two (or more) relationships between the same two object classes? When would you need to do that? How would you distinguish between the two separate relationships? Can you think of an example? (For example, *Person is owner of Vehicle* and *Person is primary driver of Vehicle*. This would be for a car insurance system.)
2. Is it necessary, or even correct, to include all relationships between classes? What criteria would you use to decide whether to include a particular relationship? What is an example of a relationship that you would not need to include? (For example, transient relationships that are not persistent, i.e., do not need to be stored, should not be included in the domain model. An example might be *Man is married to Woman* may be important. But *Man telephones Woman* is probably not important unless the system is for Verizon or AT&T.)
3. Finally, how does one ensure that the correct relationships have been captured? How do you know if you have all the important ones? How do you know if you have identified relationships that are unimportant or redundant?