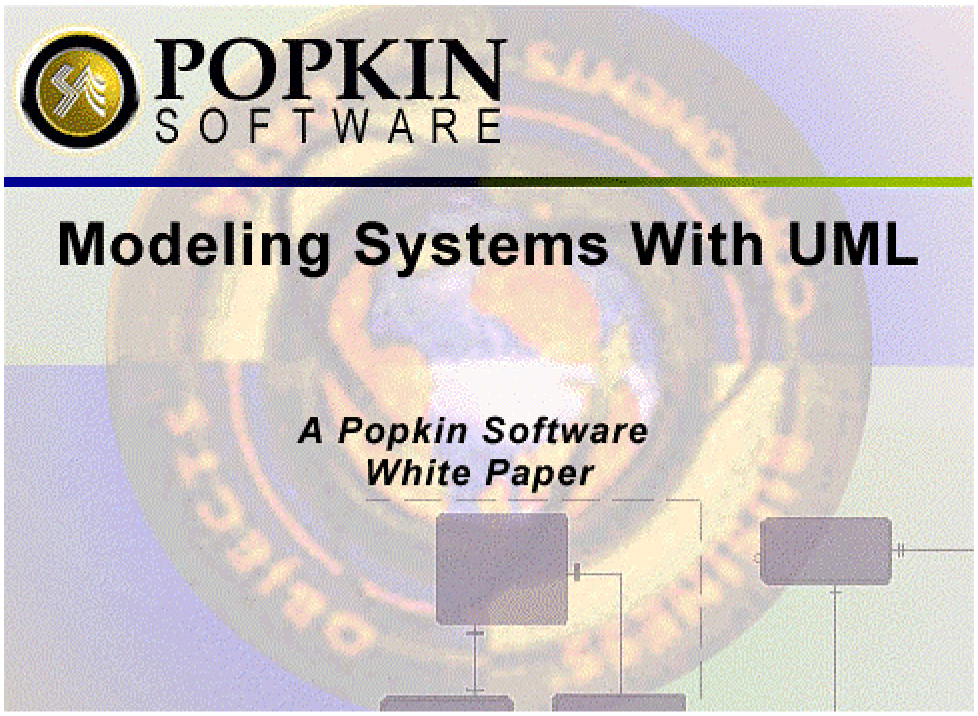

POPKIN
SOFTWARE
Modeling Systems With UML
A Popkin Software
White Paper

Contents

Introduction

This whitepaper introduces the Unified Modeling Language (UML), version 1.1. It reviews the diagrams that comprise UML, and offers a Use-Case-driven approach on how these diagrams are used to model systems. The paper also discusses UML's built-in extensibility mechanisms, which enable its notation and semantics to be extended. This paper also suggests extending UML by two non-built-in techniques: CRC cards for responsibility-driven analysis and entity relation diagrams for modeling of relational databases.

What Is UML?

The Unified Modeling Language prescribes a standard set of diagrams and notations for modeling object-oriented systems, and describes the underlying semantics of what these diagrams and symbols mean. Whereas there has been to this point many notations and methods used for object-oriented design, now there is a single notation for modelers to learn.

UML can be used to model different kinds of systems: software systems, hardware systems, and real-world organizations. UML offers nine diagrams in which to model systems:

- **Use Case diagram** for modeling the business processes
- **Sequence diagram** for modeling message passing between objects
- **Collaboration diagram** for modeling object interactions
- **State diagram** for modeling the behavior of objects in the system
- **Activity diagram** for modeling the behavior of Use Cases, objects, or operations
- **Class diagram** for modeling the static structure of classes in the system
- **Object diagram** for modeling the static structure of objects in the system
- **Component diagram** for modeling components
- **Deployment diagram** for modeling distribution of the system.

UML is a consolidation of many of the most used object-oriented notations and concepts. It began as a consolidation of the work of Grady Booch, James Rumbaugh, and Ivar Jacobson, creators of three of the most popular object-oriented methodologies.

In 1996, the Object Management Group (OMG), a standards body for the object-oriented community, issued a request for proposal for a standard object-oriented analysis notation and semantic metamodel. UML, version 1.0, was proposed as an answer to this submission in January of 1997. There were five other rival submissions. During the course of 1997, all six submitters united their work and presented to OMG a revised UML document, called UML version 1.1. This document was approved by the OMG in November 1997. The OMG calls this document OMG UML version 1.1. The OMG is currently in the process of performing a technical editing of this specification, scheduled for completion by April 1, 1999.

UML Provides Standard Notation and Semantics

UML prescribes a standard *notation* and underlying *semantics* for modeling an object-oriented system. Previously, an object-oriented design might have been modeled with any one of a half dozen popular methodologies, causing reviewers to have to learn the notational semantics of the methodology before trying to understand the design. Now with UML, different designers modeling different systems can readily understand each other's designs.

UML Is Not *a Method*

UML, however, does not prescribe a standard process or method for developing a system. There are a number of popular, published methodologies; a few of the most popular of which include the following:

Catalysis:	An object-oriented method that fuses much of the recent work on object-oriented methods, and in addition provides specific techniques for modeling distributed components.
Objectory:	A Use-Case driven method for development created by Ivar Jacobson.
Shlaer/Mellor:	"The" method for design of real-time systems, put forth by Sally Shlaer and Steven Mellor in two 1991 books, *Object Lifecycles, Modeling the World in States*, and *Object Lifecycles, Modeling the World in Data* (Prentice Hall). Shlaer/Mellor continue to

	periodically update their method (most recent update is The OOA96 Report), and recently published a white paper on how to use UML notation with Shlaer/Mellor.
Fusion:	Developed at Hewlett Packard in the mid-nineties as the first attempt at a standard object-oriented method. Combines OMT and Booch with CRC cards and formal methods. (www.hpl.hp.com/fusion/file/teameps.pdf)
OMT:	The Object Modeling Technique was developed by James Rumbaugh and others, and published in the seminal OO book, *Object-Oriented Modeling and Design* (Prentice Hall, 1991). A method preaching iterative analysis and design, heavy on the analysis side.
Booch:	Similar to OMT, and also very popular, Grady Booch's first and second editions of *Object-Oriented Design, With Applications (Benjamin Cummings, 1991 and 1994)* detail a method preaching iterative analysis and design, heavy on the design side.

In addition, many organizations have developed their own internal methodology, using different diagrams and techniques from various sources. Examples are the **Catalyst** methodology by Computer Sciences Corporation (CSC) or the Worldwide Solution Design and Delivery Method (**WSDDM**) by IBM. These methodologies vary, but generally combine workflow analysis, requirements capture, and business modeling with data modeling, with object modeling using various notations (OMT, Booch, etc), and sometimes include additional object-modeling techniques such as Use Cases and CRC cards. Most of these organizations are adopting and incorporating UML as the object-oriented notation of their methodology.

Some modelers will use a subset of UML to model what they're after, for example just the class diagram, or just the class and sequence diagrams with Use Cases. Others will use a fuller suite, including the state and activity diagrams to model real-time systems and the implementation diagram to model distributed systems. Still others will not be satisfied with the diagrams offered by UML, and will need to extend UML with other diagrams such as relational data models and CRC cards.

UML 1.1 Extensions

Built-in extensibility mechanisms enable UML to be a somewhat open specification that can cover aspects of modeling not specified in the 1.1 document. These mechanisms enable UML's notation and semantics to be expanded.

Stereotypes

Stereotype is the most widely used built-in extensibility mechanism within UML. A stereotype represents a usage distinction. It can be applied to any modeling element, including classes, packages, inheritance relationships, etc. For example, a class with stereotype <<*actor*>> is a class used as an external agent in business modeling. A template class is modeled as a class with stereotype <<*parameterized*>>, meaning it contains parameters.

Business Modeling Extensions

A separate document within UML specification calls out specific class and association stereotypes that extend UML to cover business-modeling concepts. This includes stereotyping a class as an actor, a worker (both internal and case), or an entity, and stereotyping an association as a simple communication, or a subscription between a source and a target.

Object Constraint Language (OCL)

A picture can only describe so many words. Similarly, a graphical model can only describe a certain amount of behavior, after which it is necessary to fill in additional details with words. Describing something with words, however, almost always results in ambiguities; i.e., "what did he mean when he wrote that?" The Object

Constraint Language (OCL) is incorporated into UML as a standard for specifying additional details, or precise constraints on the structure of the models.

Developed within the IBM Insurance Division as a business modeling language, the OCL is a formal language designed to be easy to read and write. OCL is more formal than natural language, but not as precise as a programming language – it cannot be used to write program logic or flow control. Since OCL is a language for pure expression, its statements are guaranteed to be without side effect – they simply deliver a value and can never change the state of the system.

Further Extensions

Two specific areas that UML does not address currently, even with its specified extensions, are responsibility-driven analysis and modeling of relational databases. This paper introduces these techniques as current, real-world extensions to UML that should be considered.

Responsibility-Driven Analysis with CRC Cards

A widely used technique for getting into the 'mindset' of object-oriented thinking is responsibility-driven analysis with Class Responsibility and Collaborator (CRC) cards. With this technique, classes discovered during analysis can be filtered to determine which classes are truly necessary for the system.

Relational Data Modeling

Although object-oriented databases are becoming more popular, in today's development environment, the relational database remains the predominant method for data storage. The UML class diagram can be used to model the relational database the system is based on, however, traditional data modeling diagrams capture more information about the relational database and are better suited to model it. This paper discusses using Entity Relationship (ER) diagrams as an important UML extension for relational database modeling.

An Overview of UML

A Use Case Driven Tour

Once again, UML is a notation, not a method. It does not prescribe a process for modeling a system. However, because UML includes the Use Case diagram, it is considered to lend itself to a problem-centric, Use Case driven approach to design. The Use Case diagram provides the entry point into analyzing the requirements of the system, and the problem that needs to be solved. *Figure 1* provides a general flow of how diagrams of UML, with extensions, interact in a Use Case-driven approach to design.

Use Cases and Interaction Diagrams

A Use Case is modeled for all processes the system must perform. Processes are described within the Use Case by a textual description or a sequence of steps performed. Activity diagrams can also be used to graphically model the scenarios. Once the system's behavior is captured in this way, the Use Cases are examined and amplified to show what objects interrelate to make this behavior happen. Sequence and Collaboration diagrams are used to show object interrelationships.

Class and Implementation Diagrams

As the objects are found, they can be grouped by type and classified in a Class diagram. It is the Class diagram that becomes the central analysis diagram of the object-oriented design, and one that shows the static structure of the system. The class diagram can be divided into business, application, and data layers, which show the classes involved with the user-interface, the software logic of the application, and the data storage, respectively. Component diagrams are used to group classes into components or modules. Overall hardware distribution of the system is modeled using the Deployment diagram.

CRC Cards – An Informal UML Extension

As an informal extension to UML, the CRC Card technique can be used to drive the system through responsibility-driven analysis. Classes are examined, filtered, and refined based on their responsibilities to the system, and the classes they need to collaborate with in order to fulfill their responsibilities.

State Diagrams

Real-time behavior of each class that has significant dynamic behavior is modeled using a State diagram. The Activity diagram can again be used at this point, this time as an extension to the State diagram, to show the details of actions performed by objects in response to internal events. The Activity diagram can also be used to graphically represent the actions of class methods.

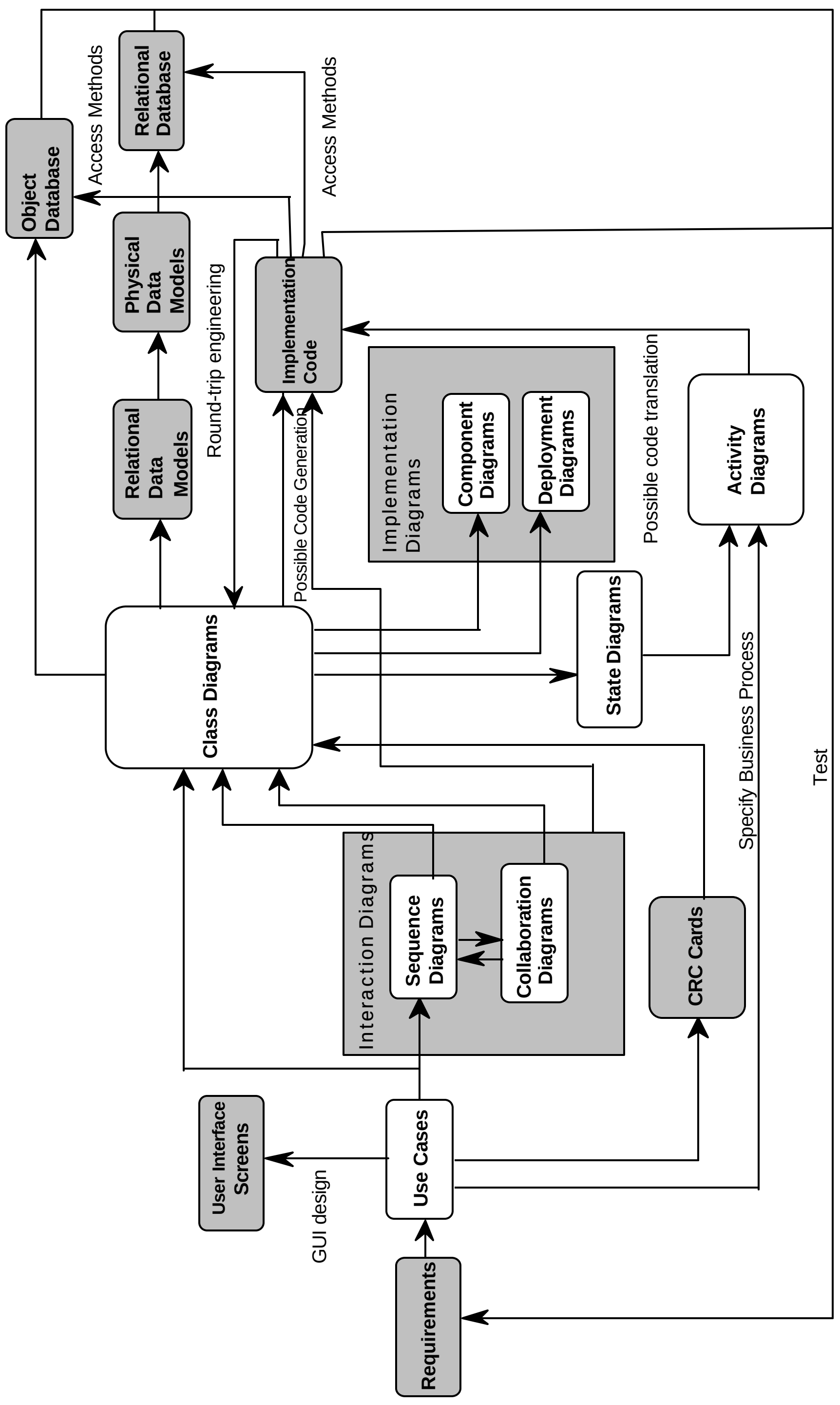


Figure 1 Use Case Driven Approach to OO Development with UML, Including CRC Card and Data Modeling Extensions

Implementing the Design

Implementation of the system concerns translating information from multiple UML models into code and database structure. When modeling a large system, it is useful to break the system down into its business layer (including user-interface objects), its application layer (including implementation objects), and its data layer (including database structure and access objects).

Implementing the Application

The class diagram is used to generate a skeletal structure of the code in the chosen language. Information from the Interaction, State, and Activity diagrams can provide details of the procedural part of the implementation code.

Implementing the Database Design

The data layer of the class diagram can be used to directly implement an object-oriented database design, or, as a UML extension, be mapped to an Entity Relation diagram for further analysis of entity relationships. It is in the ER diagram that relationships between entities can be modeled based on keyed attributes. The logical ER diagram provides a basis from which to build a Physical diagram representing the actual tables and relationships of the relational database.

Testing Against Requirements

Use Cases are also used to test the system to see if it satisfies initial requirements. The steps of the Use Cases are walked through to determine if the system is satisfying user requirements.

An In-Depth Look at UML

The following sections present a more detailed look at modeling with UML. A very simple Airline Reservation System is used to illustrate UML modeling techniques and diagrams. The following topics are covered:

- Organizing your system with packages
- Modeling with Use Cases, and using them to capture system requirements
- Modeling with Sequence and Collaboration diagrams
- Analyzing and designing with the Class diagram, and extending UML with the CRC card technique
- Modeling behavior with State and Activity diagrams
- Modeling software components, distribution, and implementation
- Extending the UML with relational database design.

Organizing Your System with Packages

One of the key tasks to modeling a large software system is to break it down into manageable areas first. Whether these areas are called domains, categories, or subsystems, the idea is the same: break the system into areas that have similar subject matter.

UML introduces the notion of a package as the universal item to group elements, enabling modelers to subdivide and categorize systems. Packages can be used on every level, from the highest level, where they are used to subdivide the system into domains, to the lowest level, where they are used to group individual Use Cases, classes, or components.

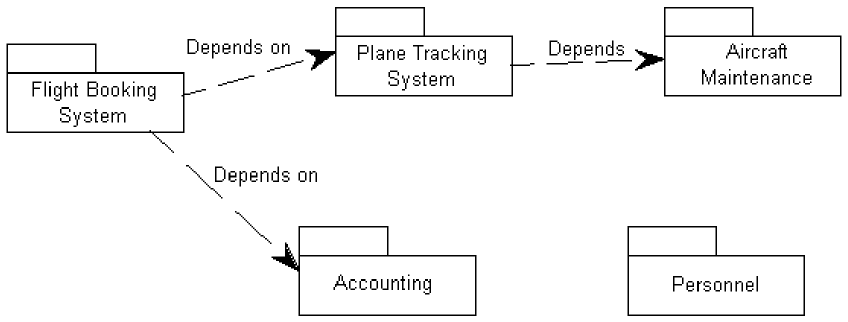
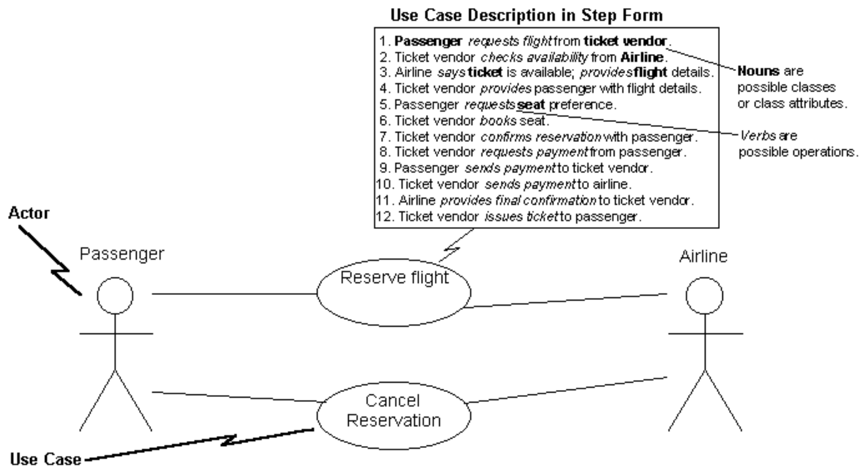


Figure 2 Organizing System Using Packages

Use Case Modeling

Use Case modeling is the simplest and most effective technique for modeling system requirements from a user's perspective. Use Cases are used to model how a system or business currently works, or how the users wish it to work. It is not really an object-oriented approach; it is really a form of process modeling. It is, however, an excellent way to lead into object-oriented analysis of systems. Use cases are generally the starting point of object-oriented analysis with UML.

The Use Case model consists of actors and use cases. Actors represent users and other systems that interact with the system. They are drawn as stick figures. They actually represent a type of user, not an instance of a user. Use cases represent the behavior of the system, scenarios that the system goes through in response to stimuli from an actor. They are drawn as ellipses.


Use Case Description in Step Form

1. **Passenger** *requests flight* from **ticket vendor**.
2. Ticket vendor *checks availability* from **Airline**.
3. Airline *says* **ticket** is available; *provides* **flight** details.
4. Ticket vendor *provides* passenger with flight details.
5. Passenger *requests* **seat** preference.
6. Ticket vendor *books* seat.
7. Ticket vendor *confirms reservation* with passenger.
8. Ticket vendor *requests payment* from passenger.
9. Passenger *sends payment* to ticket vendor.
10. Ticket vendor *sends payment* to airline.
11. Airline *provides final confirmation* to ticket vendor.
12. Ticket vendor *issues ticket* to passenger.

Nouns are possible classes or class attributes.

Verbs are possible operations.

Figure 3 Use Case Modeling

Each Use Case is documented by a description of the scenario. The description can be written in textual form or in a step-by-step format. Each Use Case can also be defined by other properties, such as the pre- and post-conditions of the scenario – conditions that exist before the scenario begins, and conditions that exist after the scenario completes. Activity Diagrams provide a graphical tool to model the process of a Use Case. These are described in a later section of this document.

Capture and/or Verify Requirements

The final objective of any software design is to satisfy the user requirements for the system. These requirements can be software requirements, product requirements, or testing requirements. The goal of capturing and verifying user requirements is to ensure that all requirements are fulfilled by the design, and that the design conforms to the defined requirements.

Oftentimes system requirements exist already in the form of requirements documents. Use Cases are used to correlate every scenario to the requirements it fulfills. If the requirements do not exist, modeling the system through Use Cases enables discovery of requirements.

Organization of Use Case Diagrams

During business analysis of the system, you can develop one Use Case model for the system, and build packages to represent the various business domains of the system. You may decompose each package with a Use Case diagram that contains the Use Cases of the domain, with actor interactions.

A Use Case for Every Scenario

The goal is to build a Use Case diagram for each significantly different kind of scenario in the system. Each scenario shows a different sequence of interactions between actors and the system, with no 'or' conditions.

Model Alternate Sequences through "Extends" Relationship

Typically, one models each Use Case with a normal sequence of actions. The user then considers "what if" conditions for each step, and develops Use Cases based on these alternate sequences of events. The alternate sequences are modeled in separate Use Cases, which are related to the original Use Case by an "Extends" relationship. The Extends relationship can be thought of as a Use Case equivalent to inheritance, in that the Extending Use Case inherits and overrides behavior of the original Use Case.

Eliminate Redundant Modeling through "Uses" Relationship

To eliminate redundant modeling of a chunk of behavior that appears in multiple Use Cases, the chunk of behavior can be modeled in a separate Use Case that is related to the other Use Cases by the Uses relationship. The Uses relationship can be thought of as a Use Case equivalent of aggregation.

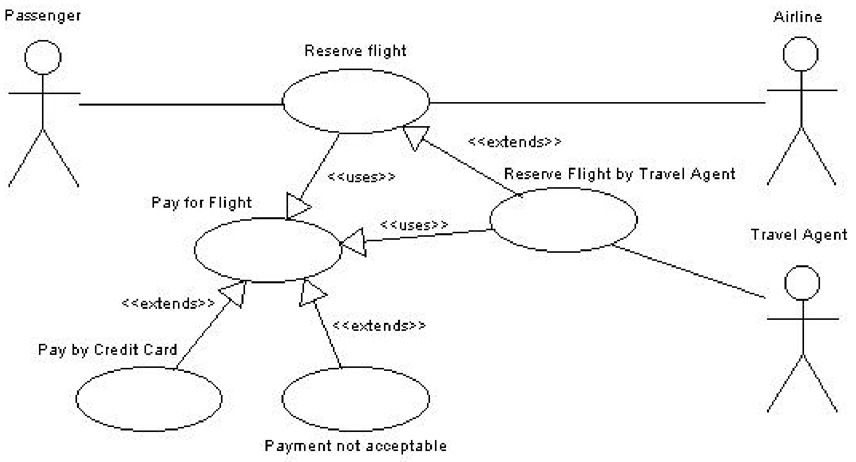


Figure 4   Use Case Extends Versus Uses Relationships

Use Cases Aid in Testing System against Requirements

Use Cases are also used to build test scripts that are used to verify that the application satisfies the business and system requirements.

When you have arrived at the lowest Use Case level, you may create a Sequence diagram for the Use Case. With the Sequence and Collaboration diagrams, you can model the implementation of the scenario.

Sequence Diagrams

The Sequence diagram is one of the most effective diagrams to model object interactions in a system. A Sequence diagram is modeled for every Use Case. Whereas the Use Case diagram enables modeling of a business view of the scenario, the Sequence diagram contains implementation details of the scenario, including the objects and classes that are used to implement the scenario, and messages passed between the objects.

Typically one examines the description of the Use Case to determine what objects are necessary to implement the scenario. If you have modeled the description of the Use Case as a sequence of steps, then you can 'walk through' the steps to discover what objects are necessary for the steps to occur.

A Sequence diagram shows objects involved in the scenario by vertical dashed lines, and messages passed between the objects as horizontal vectors. The messages are drawn chronologically from the top of the diagram to the bottom; the horizontal spacing of objects is arbitrary.

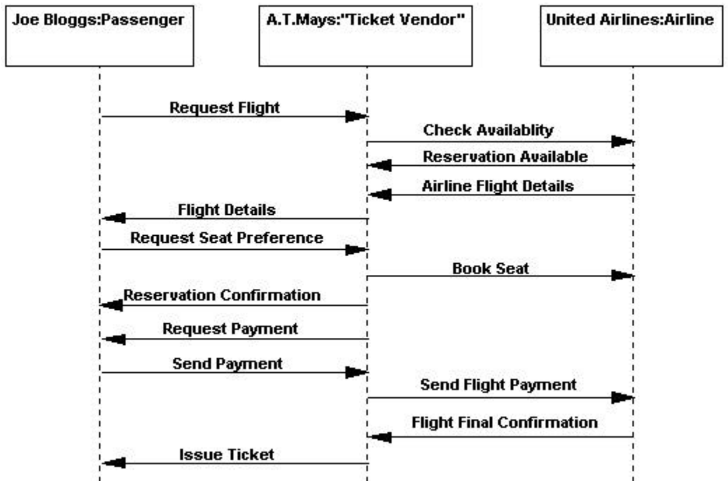


Figure 5 Sequence Diagram for a Scenario

During initial analysis, the modeler typically places the business name of a message on the message line. Later, during design, the business name is replaced with the name of the method being called by one object on the other. The method called, or invoked, belongs to the definition of the class instantiated by the object on the receiving end of the message.

Collaboration Diagrams

The Collaboration Diagram presents an alternate to the Sequence Diagram for modeling interactions between objects in the system. Whereas in the Sequence Diagram the focus is on the chronological sequence of the scenario being modeled, in the Collaboration Diagram the focus is on understanding all of the effects on a given object during a scenario. Objects are connected by links, each link representing an instance of an association between the respective classes involved. The link shows messages sent between the objects, the type of message passed (synchronous, asynchronous, simple, balking, and time-out), and the visibility of objects to each other.

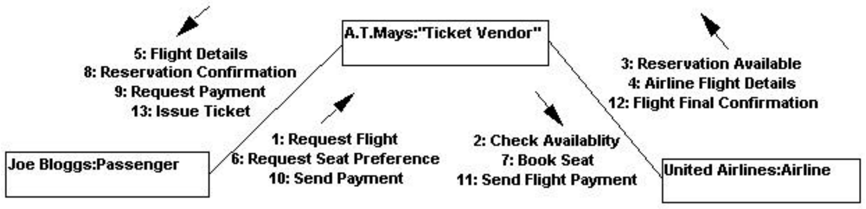


Figure 6 Collaboration Diagram for a Group of Objects

Analysis and Design with the Class Diagram

The class diagram is the main static analysis and design diagram for a system. In it, the class structure of the system is specified, with relationships between classes and inheritance structures. During analysis of the system, the diagram is developed with an eye for an ideal solution. During design, the same diagram is used, and modified to conform to implementation details.

Development of Class Diagram During Analysis

Use Case Driven Approach

In a Use Case-driven approach to OO analysis, the Class diagram is developed through information garnered in the Use Cases, Sequence diagrams, and Collaboration diagrams. The objects found during analysis are modeled in terms of the classes they instantiate, and the object interactions are mapped to relationships between the instantiated classes.

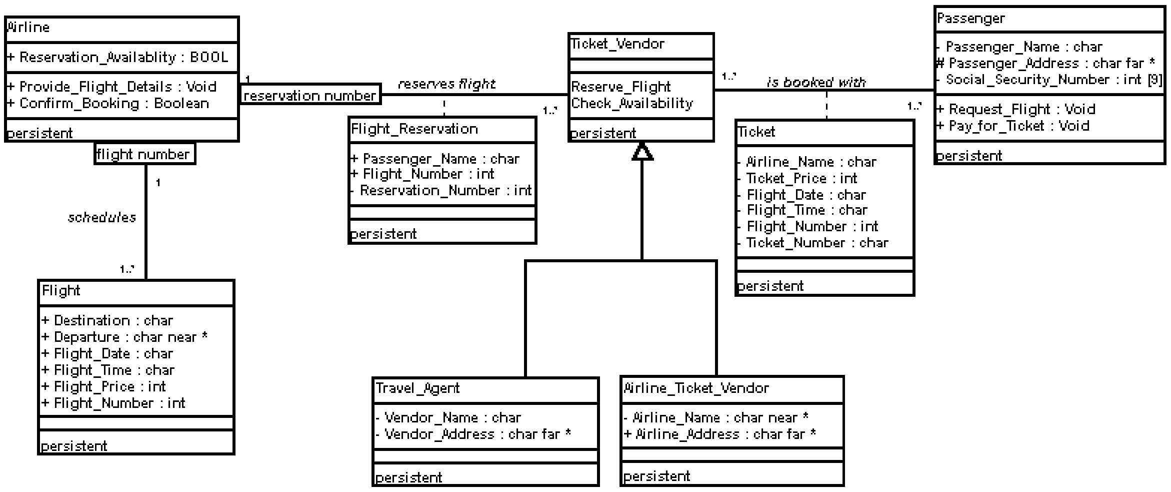
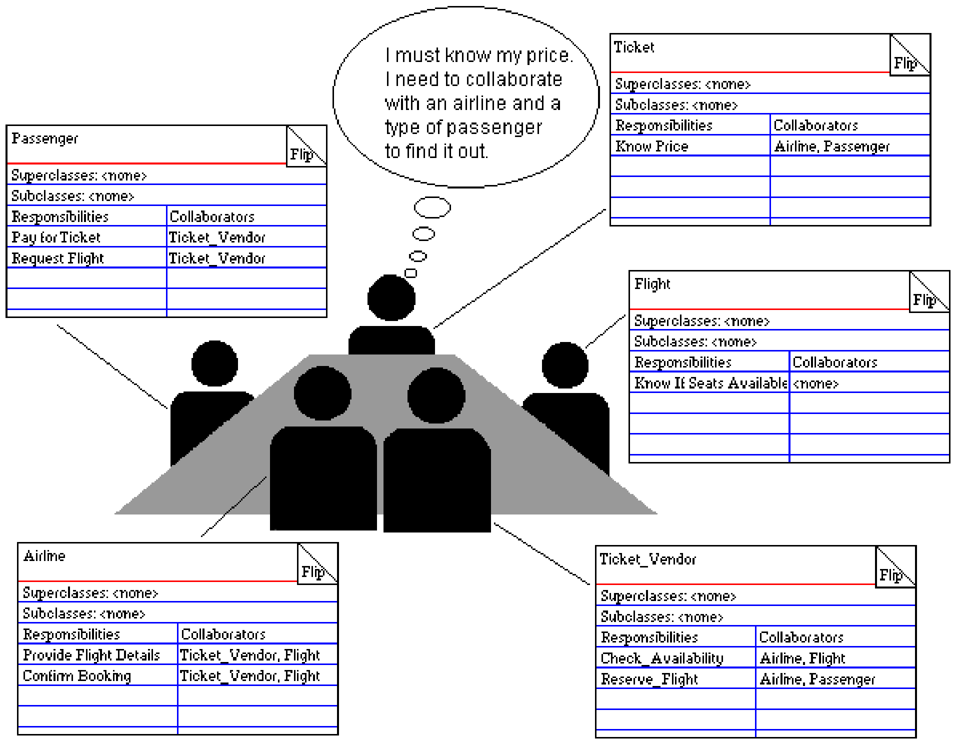


Figure 7 Class Diagram During Analysis Stage

Responsibility-Driven Extension

The CRC card technique is sometimes used as an extension to UML for responsibility-driven analysis. Class definitions are refined based on the class's responsibilities and other classes it collaborates with to fulfill responsibilities.

Each class is represented on an index card, and designers play-act the roles of classes in the system to determine their job, and who they need to collaborate with to fulfill their responsibilities. This information translates directly into a class diagram; responsibilities correspond to class methods, collaborations translate to associations between classes.


I must know my price. I need to collaborate with an airline and a type of passenger to find it out.
Flip

Passenger

Superclasses: <none>

Subclasses: <none>

Responsibilities	Collaborators
Pay for Ticket	Ticket_Vendor
Request Flight	Ticket_Vendor

Ticket

Superclasses: <none>

Subclasses: <none>

Responsibilities	Collaborators
Know Price	Airline, Passenger

Flight

Superclasses: <none>

Subclasses: <none>

Responsibilities	Collaborators
Know If Seats Available	<none>

Airline

Superclasses: <none>

Subclasses: <none>

Responsibilities	Collaborators
Provide Flight Details	Ticket_Vendor, Flight
Confirm Booking	Ticket_Vendor, Flight

Ticket_Vendor

Superclasses: <none>

Subclasses: <none>

Responsibilities	Collaborators
Check_Availability	Airline, Flight
Reserve_Flight	Airline, Passenger

Figure 8 Informal UML Extension -- CRC Cards for Responsibility Driven Analysis

Design of System with Class Diagram

During design, the class diagram is elaborated to take into account the concrete details of implementing the system.

Multi-Tiered Architectures

One concern during design is to establish the architecture of the system. This includes establishing whether it will be a simple system designed to run on a single machine, a two-tiered system consisting of a client and a server, or a multi-tiered system with user-interface objects separate from business application objects separate from the database, each potentially running on different platforms.

One approach to managing the class diagram for a complex system is to separate the diagram into sections that show the application logic, the user interface design, and the classes involved with the storage of data. This can be physically done by segmenting the class diagram, using separate diagrams for each section, or simply by adding a property to each class that tracks which tier it belongs to.

Component Design

A component is a group of interacting objects or smaller components that combine to provide a service. A component is similar to a black box, in which the services of the component are specified by its interface or interfaces, without providing knowledge of the component's internal design and implementation.
Component-based development is the process of assembling the right combination of components in the right configuration to achieve desired functionality for a system.
Components are represented in the UML class diagram by specifying the interface of a class or package. There are two notations to show an interface – one is to show the interface as a regular class symbol with the stereotype <<interface>>, with a list of operations supported by the interface listed in the operation department. The alternate, shortcut notation is to display the interface as a small circle attached to the class by a solid line, with the name of the interface by the circle.

The example in Figure 9 shows that the class Passenger provides a ***move(x coord, y coord)*** operation for it's appearance on a GUI, through it's ***Displayable*** interface. Both UML interface notations are shown in the figure. In addition, the Passenger class also provides a ***save(store at)*** operation through its ***Persistent*** interface. A database connectivity class or component might use this interface.

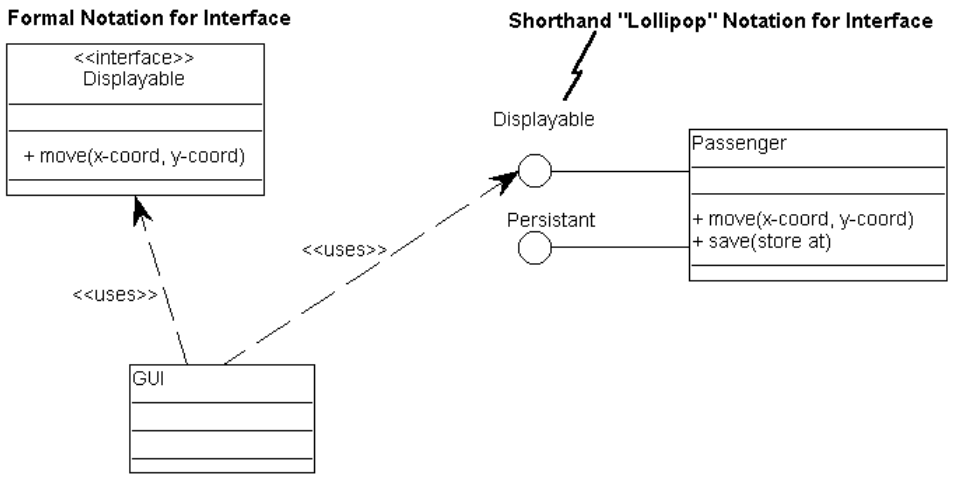


Figure 9 Component Design: Specifying Interface of Class

Iterative Analysis and Design

The class diagram can be developed in an iterative fashion, through a repeated cycle of analysis, design, and implementation, and then back to analysis, to begin the cycle again. This process is often referred to as ‘round-trip engineering’. Modeling tools such as System Architect 2001 can facilitate this process by enabling you to implement the design in a language such as C++ or Java, and then reverse the code back into the existing class diagram, automatically updating the information stored on the diagram and in the underlying repository.

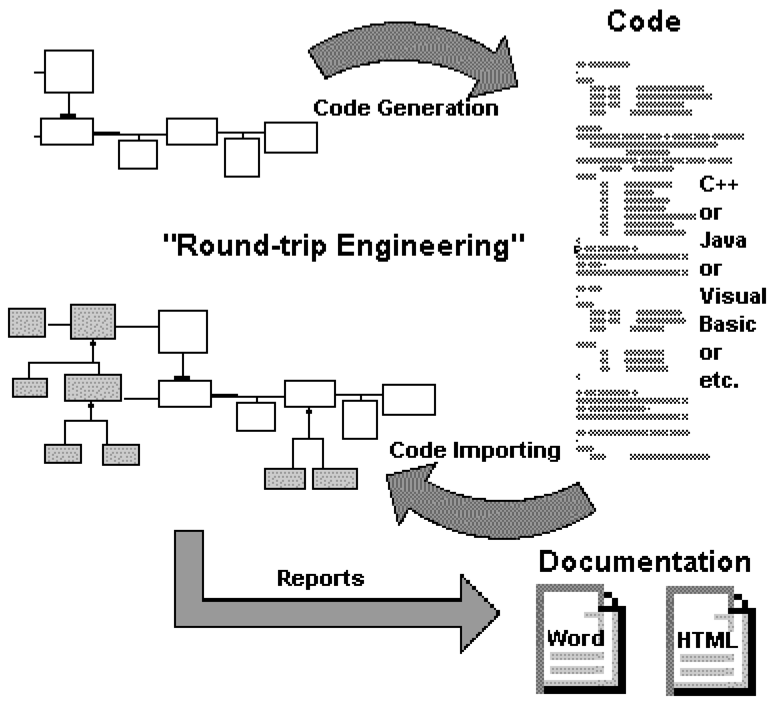


Figure 10 Iterative Analysis and Design and Documentation with Class Diagram

Modeling Class Behavior with State Diagram

While interaction and collaboration diagrams model dynamic sequences of action between groups of objects in a system, the state diagram is used to model the dynamic behavior of a particular object, or class of objects.

A state diagram is modeled for all classes deemed to have significant dynamic behavior. In it, you model the sequence of states that an object of the class goes through during its life in response to received stimuli, together with its own responses and actions.

For example, an object's behavior is modeled in terms of what *state* it is in initially, and what state it *transitions* to when a particular *event* is received. You also model what actions an object performs while in a certain state.

States represent the conditions of objects at certain points in time. Events represent incidents that cause objects to move from one state to another. Transition lines depict the movement from one state to another. Each transition line is labeled with the event that causes the transition. Actions occur when an object arrives in a state.

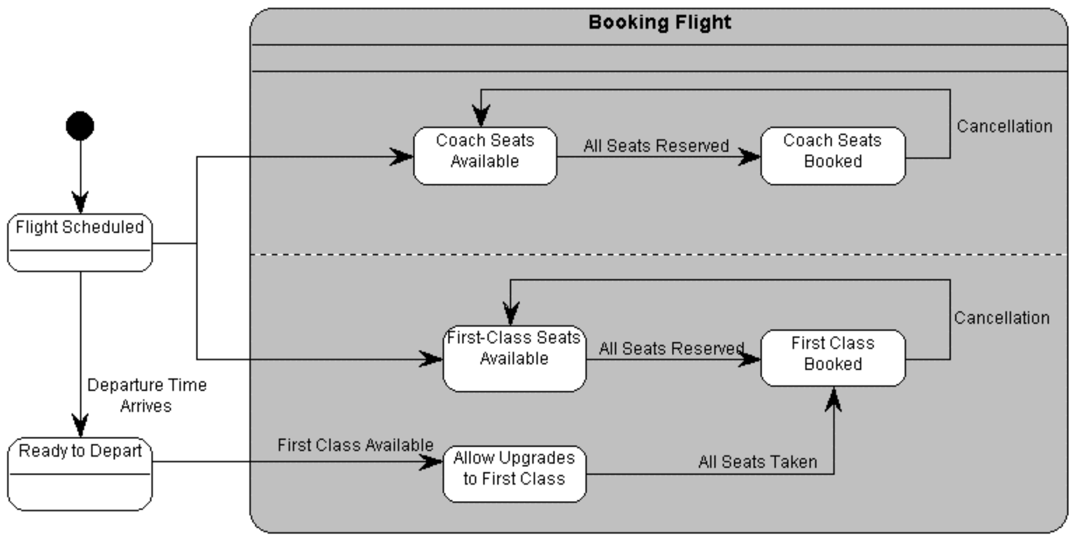


Figure 11 Modeling Dynamic Behavior of Flight Object with State Diagram

Activity Diagrams

The Activity Diagram is a multi-purpose process flow diagram that is used to model behavior of the system. Activity diagrams can be used to model a Use Case, or a class, or a complicated method.

An Activity Diagram is similar to a flow chart; the one key difference is that activity diagrams can show parallel processing. This is important when using activity diagrams to model business processes, some of which can be performed in parallel, and for modeling multiple threads in concurrent programs.

Using Activity Diagrams to Model Use Cases

Activity Diagrams provide a graphical tool to model the process of a Use Case. They can be used in addition to, or in place of, a textual description of the Use Case, or a listing of the steps of the Use Case. A textual description, code, or another activity diagram can detail the activity further.

Using Activity Diagrams to Model Classes

When modeling the behavior of a class, a UML State Diagram is normally used to model situations where asynchronous events occur. The Activity Diagram is used when all or most of the events represent the completion of internally generated actions. You should assign activities to classes before you are done with the activity diagram.

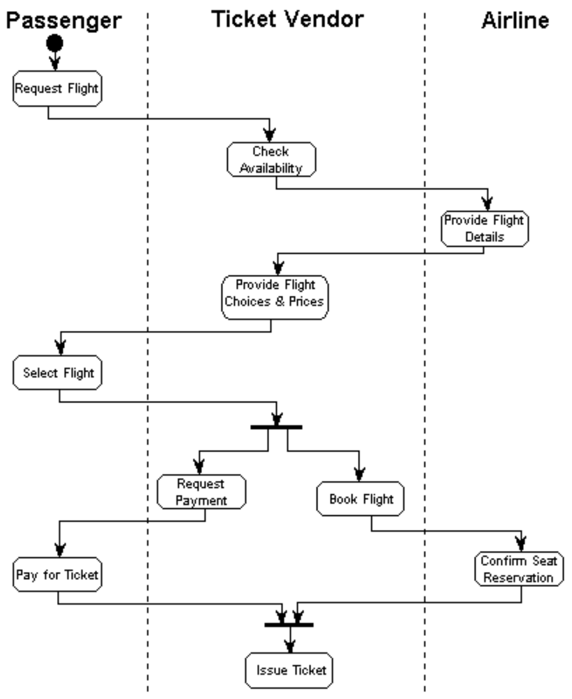


Figure 12 Activity Diagram

Modeling Software Components

The component diagram is used to model the structure of the software, including dependencies among software components, binary code components, and executable components.
In the component diagram you model system components, sometimes grouped by package, and the dependencies that exist between components (and packages of components).

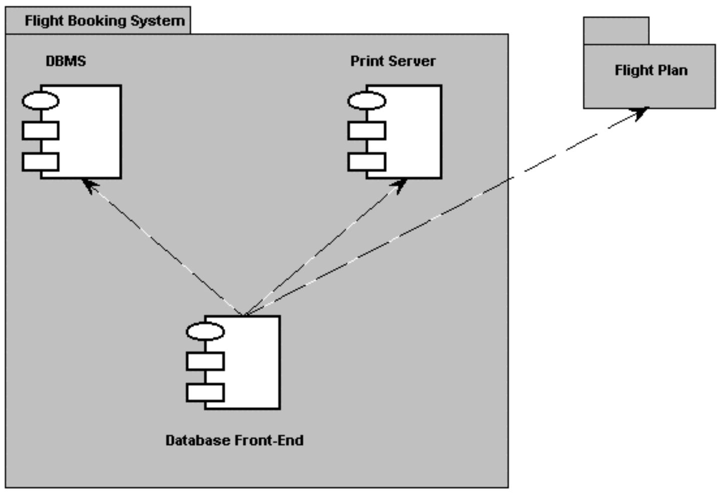


Figure 13 Modeling Components with Component Diagram

Modeling Distribution and Implementation

Deployment diagrams are used to model the configuration of run-time processing elements and the software components, processes, and objects that live on them. In the deployment diagram, you start by modeling the physical nodes and the communication associations that exist between them. For each node, you can indicate what component instances live or run on the node. You can also model the objects that are contained within the component.

Deployment diagrams are used to model only components that exist as ***run-time*** entities; they are ***not used*** to model compile-time only or link-time only components. You can also model components that migrate from node to node or objects that migrate from component to component using a dependency relationship with the ***becomes*** stereotype.

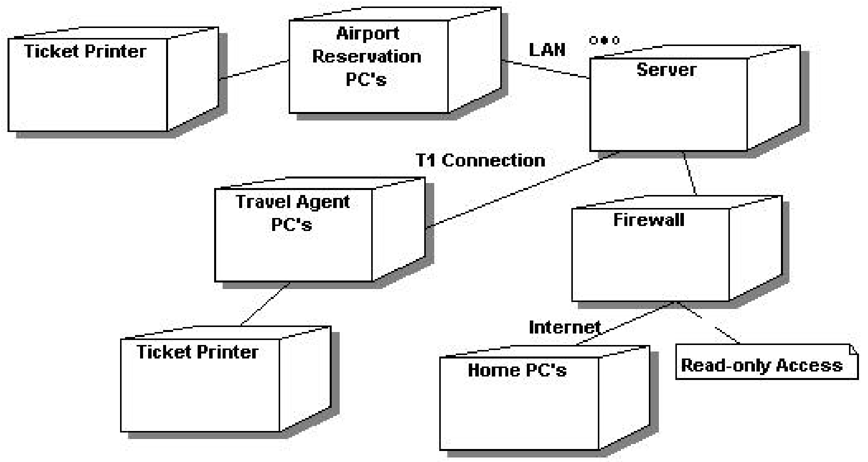


Figure 14 Modeling Distribution of System with Deployment Diagram

Relational Database Design – A UML Extension

The class diagram presents an implementation neutral mechanism to model the data storage aspects of the system. Persistent classes, their attributes, and their relationships can be implemented directly in an object oriented database. However, in today's development environment, the relational database remains the predominant method for data storage. It is in modeling this area that UML falls short. The UML Class diagram can be used to model some aspects of the relational database design, but it fails to cover all the semantics involved in relational modeling, most notably the notion of keyed attributes that relate tables to one another. To capture this information, an Entity Relation (ER) diagram is recommended as an extension to UML.

The class diagram can be used to model the logical framework of the database, independent of it being object-oriented or relational, with classes representing tables, and class attributes representing columns. If a relational database is the chosen implementation medium, then the class diagram can be mapped to a logical ER diagram. Persistent classes and their attributes map directly to logical entities and their attributes; the modeler is faced with choices on how to map associations over to relationships between entities. Inheritance relationships are mapped directly to super-sub relationships between entities in an ER diagram.

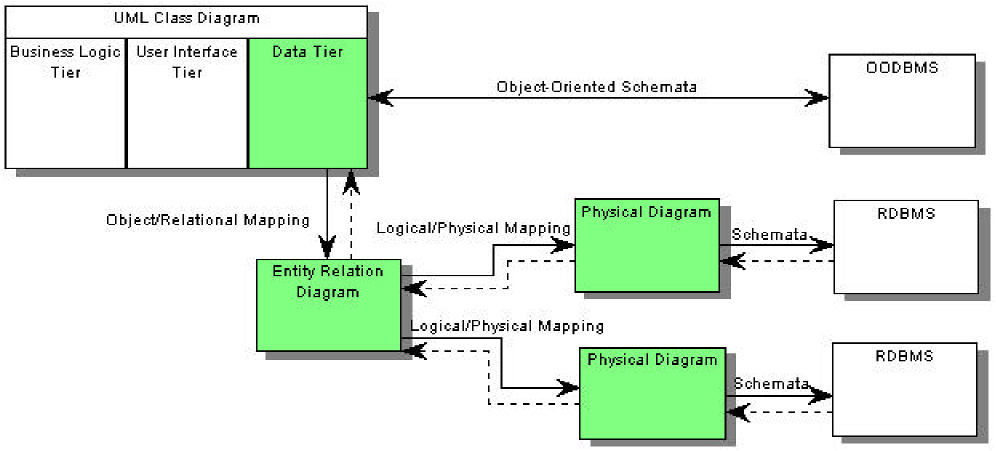


Figure 15 UML Extension -- Relational Database Design with ER Diagram

Once in the ER diagram, the modeler can begin the process of determining how the relational model fits together; and which attributes are primary keys, secondary keys, and foreign keys based on relationships with other entities. The idea is to build a logical model that conforms to the rules of data normalization.

When implementing the relational design, it is an advised strategy to map the logical ER diagram to a physical diagram representing the target RDBMS. The physical diagram can be denormalized to achieve a database design that has efficient data access times. Super-sub relationships between entities are resolved by actual table structures. In addition, the physical diagram is used to model vendor-specific properties for the RDBMS. Multiple physical diagrams are created if there are multiple RDBMS's being deployed; each physical diagram representing one target RDBMS.

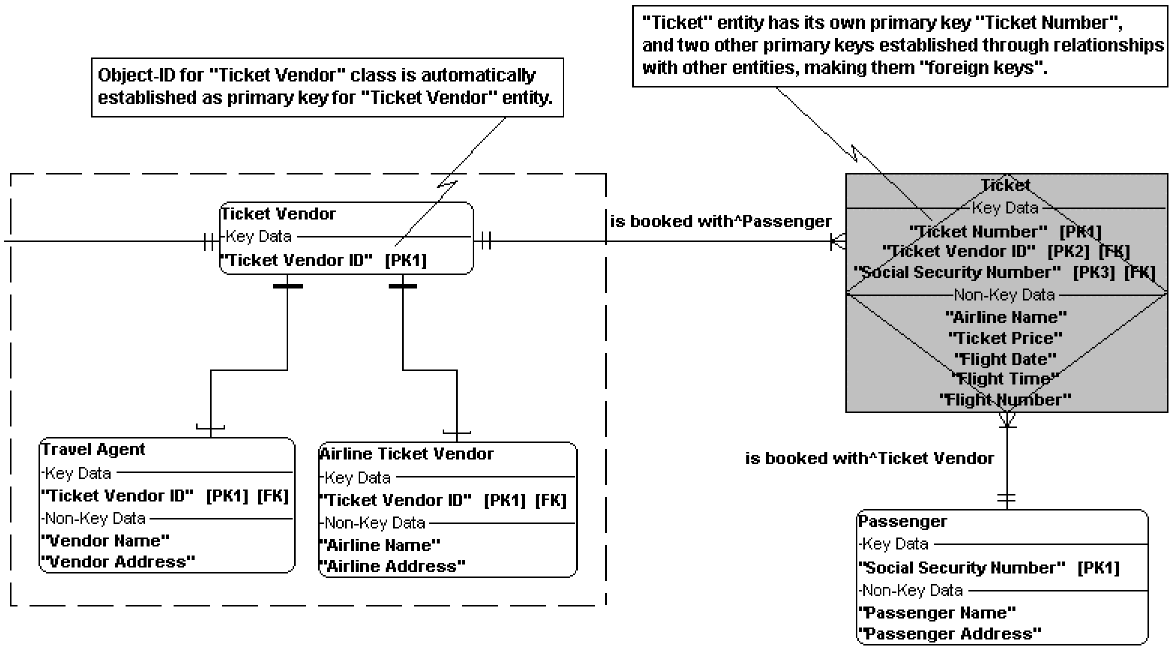

Object-ID for "Ticket Vendor" class is automatically established as primary key for "Ticket Vendor" entity.
"Ticket" entity has its own primary key "Ticket Number", and two other primary keys established through relationships with other entities, making them "foreign keys".
Ticket Vendor
Key Data
"Ticket Vendor ID" [PK1]
is booked with^Passenger
Ticket
Key Data
"Ticket Number" [PK1]
"Ticket Vendor ID" [PK2] [FK]
"Social Security Number" [PK3] [FK]
Non-Key Data
"Airline Name"
"Ticket Price"
"Flight Date"
"Flight Time"
"Flight Number"
is booked with^Ticket Vendor
Travel Agent
Key Data
"Ticket Vendor ID" [PK1] [FK]
Non-Key Data
"Vendor Name"
"Vendor Address"
Airline Ticket Vendor
Key Data
"Ticket Vendor ID" [PK1] [FK]
Non-Key Data
"Airline Name"
"Airline Address"
Passenger
Key Data
"Social Security Number" [PK1]
Non-Key Data
"Passenger Name"
"Passenger Address"

Figure 16 Keyed Relationships Between Entities in an ER Diagram

Use of a Modeling Tool

Exchange of design information and ideas using UML notation will be done in the mediums that have always been popular: blackboards, notebooks, and paper napkins to name a few. But UML is best served by a modeling tool, which can be used to capture, store, reuse, automatically integrate, and document design information.

As a side benefit to modelers, UML also makes it easier to choose a modeling tool. In the past, the modeler first had to select a methodology notation, and then is limited to selecting a tool that supported it. Now with UML as a standard, the choice of notation has already been made for the modeler. And with all major modeling tools supporting UML, the modeler can select the tool based on the key areas of functionality it supports that enable problems to be solved and solutions documented.

Like a good toolchest, a good modeling tool should provide all the tools necessary to get a variety of jobs done efficiently, never leaving you without the right tool. Within the systems design framework described in this paper, this includes the following:

- Support for the full UML notation and semantics.
- Support for a wide breadth of modeling techniques and diagrams to compliment UML – including CRC Cards, data modeling, flow charts, and user screen design. Ability to reuse information captured in other techniques still in play, such as traditional process modeling.
- Facilitate capture of information in an underlying repository — enabling reuse across diagrams.
- Ability to customize the definition properties underlying UML model elements.
- Enable multiple teams of analysts to work on same data at same time.
- Ability to capture requirements, associate them with model elements which satisfy them, and trace how requirements have been satisfied at all stages of development.
- Ability to create customized reports and documentation on your designs, and output these reports in multiple formats, including HTML for distribution on your corporate intranet or internet.
- Ability to generate and reverse code (i.e., C++, Java, etc) to facilitate iterative analysis and design, for reuse of existing code or class libraries, and for documentation of code.

System Architect 2001

Popkin Software provides extensive support for modeling systems with UML in System Architect 2001. It provides all of the features described above to enable efficient modeling of systems. For more information on the Popkin Software product line, browse www.popkin.com.

Summary

UML 1.1 is a good beginning – it provides system architects with a standard notation for modeling systems. Now, like architects reading blueprints, an object modeler can pick up any design and understand what is being captured. UML is also good for the modeling community – instead of spending time arguing about how to express information being captured, modelers can get to solving the problem at hand, which is designing the system.

However, although UML lends itself to a Use-Case driven approach, UML does not answer the question of how to build a system. This choice of what methodology, or process, to use is still left open for the modeler to decide, or figure out.

From a notational standpoint, UML 1.1 is not the complete solution yet; but it is expected that UML will continue to evolve with time. In the meantime, modelers will use UML as a base, extending it by a combination of other techniques, such as responsibility-driven analysis and relational data modeling, to model the real world.

References

Understanding UML: The Developer's Guide, With a Web-Based Application in Java, by Paul Harmon and Mark Watson; Morgan Kauffman Publishers, Inc., 1998 (www.mkp.com/books_catalog/1-55860-465-0.asp).

What's Missing from the UML?, an article by Scott Ambler, *Object Magazine*, October 1997, SIGS Publications (www.sigs.com/omo/articles/ambler.html).

UML 1.1 Specification (UML Resource Center at www.popkin.com).

Objects, Components, and Frameworks with UML, The Catalysis Approach, by Desmond F. D'Souza and Alan C. Wills, Addison Wesley Longman, 1998.

Picture Credit

All diagrams in this paper were drawn using System Architect 2001.

For more information on analysis and design tools that can make your UML project easier, contact Popkin Software and Systems, Inc., makers of System Architect 2001.

POPKIN SOFTWARE

Popkin Software & Systems, Inc.
11 Park Place
New York, NY 10007
212-571-3434

Popkin Software & Systems, LTD
2nd Floor, St. Albans House
Portland Street
Lemington Spa
Warwickshire CV32 5EZ England
44-1926-450858