The Unified Modeling Language v1.1 and Beyond: The Techniques of Object-Oriented Modeling

An AmbySoft Inc. White Paper

Scott W. Ambler
Object-Oriented Consultant
AmbySoft Inc.

Material for this White Paper has been modified from the books
Building Object Applications That Work
and
Process Patterns
by Scott W. Ambler

http://www.ambysoft.com/umlAndBeyond.pdf

Finalized: February 11, 2000

Table Of Contents

List of Figures

1. The Unified Modeling Language and Beyond

This white paper describes the object-oriented (OO) modeling techniques employed on large-scale, mission-critical applications. The modeling techniques described by the Unified Modeling Language (UML) v1.1 (Rational, 1997) are used as a basis for OO modeling. Because it is my experience that they are not sufficient (Ambler, 1998a) for completely modeling an OO application I have extended the UML with additional industry-standard modeling techniques. Heresy? Read on and decide for yourself.

2. Understanding Your Modeling Options

In this white paper I will briefly review the major modeling techniques and diagrams used on object-oriented development projects. The goal of this white paper is to describe each modeling approach, provide an example where appropriate, and indicate when the approach should be used. I am not going to go into enough detail to teach you how to actually model with them, there are many good books written about OO analysis and design, I instead direct you to the best sources for each modeling approach.

The modeling techniques that we will cover in this white paper include:
- Class responsibility collaborator (CRC) modeling
- Use cases and use-case scenarios
- Use-case diagrams
- Interface flow diagrams
- Class diagrams
- Process diagrams
- Data diagrams
- Sequence diagrams
- Component diagrams
- Deployment diagrams
- Statechart diagrams
- Collaboration diagrams

2.1 CRC Modeling

CRC modeling is an analysis technique in which users form most of the modeling team.

CRC (class responsibility collaborator) modeling (Beck & Cunningham, 1989; Wirfs-Brock, Wilkerson, and Wiener, 1990; Jacobson, Christerson, Jonsson, and Overgaard, 1992; Ambler, 2000a) provides a simple yet effective technique for working with your users to determine their needs. CRC modeling is a process in which a group of business domain experts (BDEs) analyze their own needs for a system. CRC modeling sessions typically start with brainstorming, a technique in which people suggest whatever ideas they come up with about the application. Brainstorming allows people to get loosened up, as well as to gain a better understanding of where the other BDEs are coming from. After the brainstorming is finished the group produces a CRC model together, which describes the requirements for the system.

A CRC model is a collection CRC cards, standard index cards (usually 5 x7) that have been divided into three sections as shown in Figure 1. A class is any person, place, thing, event, concept, screen or report. The responsibilities of a class are the things that it knows and does, its attributes and methods. The collaborators of a class are the other classes that it works with to fulfill its responsibilities. CRC cards are a simple, easy to explain, low-tech approach to working with users to define the requirements for an application.

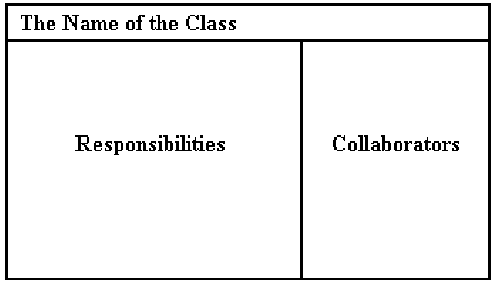
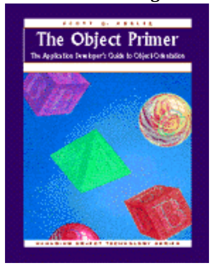


Figure 1. The layout of a CRC card.

Experience shows that CRC modeling works best with front-line employees, whereas techniques such as use cases, discussed in the next section, are more effective with upper management. CRC modeling originated as a training tool (Beck & Cunningham, 1989) to teach experienced developers OO concepts and is also used as a design brainstorming technique that leads directly to class diagramming (object modeling). Because CRC models approach requirements definition from a different direction than do use cases they are often used to validate the information gathered by use cases (and vice versa of course).



2.2 Defining Use Cases, Use-Case Scenarios, and Use-Case Diagrams

A use case (Jacobson, Christerson, Jonsson, Overgaard, 1992; Ambler, 2000a) is a description, typically written in structured English or point form, of a potential business situation that an application may or may not be able to handle. You can also say that a use case describes a way in which a real-world actor a person, organization, or external system interacts with the application. For example, the following would be considered use-cases for a university information system:

Use cases describe the basic business logic of an application.

- Enrol students in courses.
- Output seminar enrolment lists.
- Remove students from courses.
- Produce student transcripts.

A *use-case scenario* is a specific example of a use-case. Potential use-case scenarios for the use case Enrol students in courses are:

Use-case scenarios are

- A student wants to enrol in a course but they are missing a prerequisite.
- A students wants to enrol in a course but the course is over-booked for the term.
- A student wants to enrol in a course, they have the prerequisites and there is still room left.

specific examples of use cases.

To put our use cases into context, we will draw a use-case diagram (Rational, 1997; Jacobson, Christerson, Jonsson, Overgaard, 1992; Ambler, 1998a), an example of which is shown in Figure 2. Use-case diagrams are straightforward, showing the *actors*, the use cases they are involved with, and the system itself. An actor is any person, organization, or system that interacts with the application but is external to it. Actors are shown as stick figures, use cases are shown as ovals, and the system is shown as a box. The arrows indicate which actor is involved in which use cases, and the direction of the arrow indicates flow of information (in the UML, indicating the flow is optional, although I highly suggest it). In this example, students are enrolling in courses via the help of registrars. Professors input and read marks, and registrars authorize the sending out of transcripts (report cards) to students. Note how for some use cases there is more than one actor involved, and that sometimes the flow of information is in only one direction.

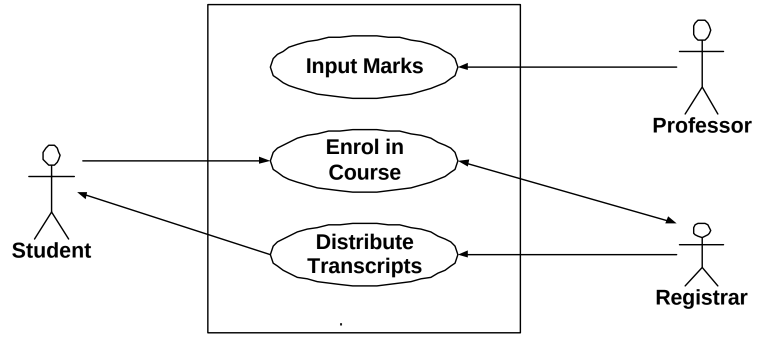


Figure 2. An example of a use-case diagram.

2.3 Interface-Flow Diagrams

To your users the user interface is the system. It is as simple as that. Does it not make sense that you should have some sort of mechanism to help you design a user interface? Prototypes are one means of describing your user interface, although with prototypes you can often get bogged down in the details of how the interface will actually work. As a result you often miss high-level relationships and interactions between the interface objects (usually screens) of your application. *Interface-flow diagrams* (Page-Jones, 1995; Ambler, 1998a) allow you to model these high-level relationships.

Interface flow diagrams show the relationships between the user interface components, screens and reports, that make up your application.

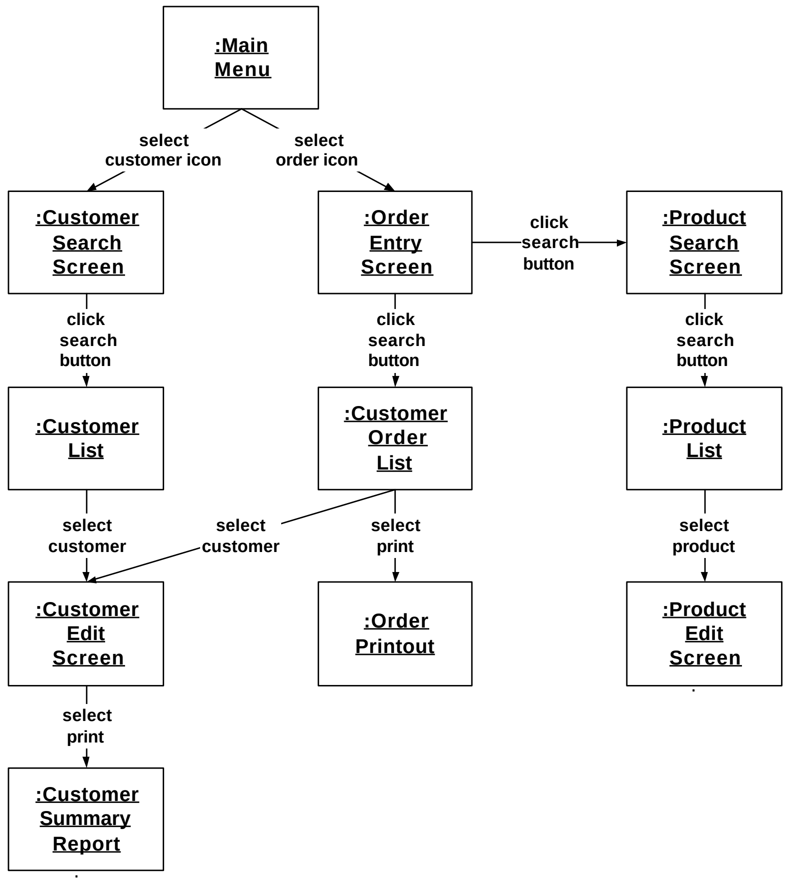


Figure 3. An interface flow diagram for an order-entry system.

In Figure 3 we see an example of an interface-flow diagram for an order-entry system. The boxes represent interface objects (screens, reports, or forms) and the arrows represent the possible flow between screens. The labels on the arrows represent the action that the user has to take to traverse from one interface object to another. For example, when you are on the main menu screen you can go to either the customer search screen or to the order-entry screen by clicking on the appropriate icons. Once you are on the order-entry screen you can go to the product search screen or to the customer order list by clicking on the corresponding buttons. Interface-flow diagrams allow you to easily gain a high-level overview of the interface for your application.

Because interface-flow diagrams offer a high-level view of the interface of a system you can quickly gain an understanding of how the system is expected to work. It puts you into a position where you can easily do some reality checking. For example, does the screen flow make sense? I am not so sure. Why cannot I get from the customer edit screen to the customer order list, which is a list of all the orders that a customer has

ever made. Furthermore, why cannot I get the same sort of list from the point of view of a product? In some cases it might be interesting to find out which orders include a certain product, especially when the product is back-ordered or no longer available.

The boxes are often documented by the appropriate screen, report, or form designs as well as a description of their purpose. Although the UML doesn t directly include interface-flow diagrams in a pinch you can substitute collaboration diagrams, discussed elsewhere in this paper, where the instances are screen objects.

2.4 Class Diagrams

Class diagrams (object models) are the mainstay of OO modeling.

Class diagrams (Rational, 1997; Ambler, 1998a; Booch 1994; Rumbaugh, Blaha, Premerlani, Eddy, & Lorenson, 1991; Shlaer & Mellor 1992), formerly called *object models*, show the classes of the system and their intrarelationships (including inheritance, aggregation, and associations). Figure4 shows an example class diagram, using the UML notation, which models the Contact-Point analysis pattern (Ambler, 1998a). Class diagrams are the mainstay of OO modeling and are used to show both what the system will be able to do (analysis) and how it will be built (design).

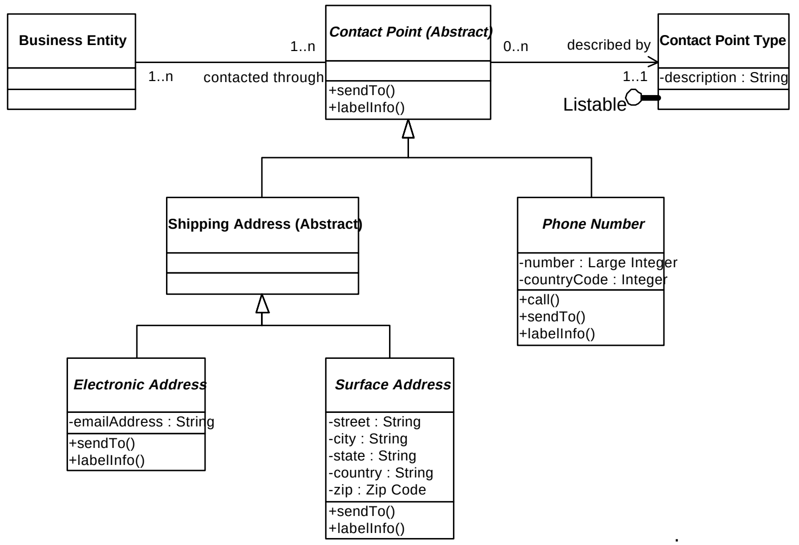


Figure 4. A class diagram representing the Contact-Point analysis pattern.

Class diagrams are typically drawn by a team of people lead by an experienced OO modeler. Depending on what is being modeled the team will be composed of subject matter experts who supply the business knowledge captured by the model, and/or other developers who provide input into how the application should be designed. The information contained in a class diagram directly maps to the source code that will be written to implement the application and therefore a class diagram must always be drawn for an OO application. Notice how the class **Contact Point Type** implements the **Listable** interface (interfaces are Java programming constructs).

Classes are documented with a description of what they do, methods are documented with a description of their logic, and attributes are documented with a description of what they contain, their type, and an indication of a range of values if applicable. Statechart diagrams, see below, are used to describe complex classes. Relationships between classes are documented with a description of their purpose and an indication of their cardinality (how many objects are involved in the relationship) and their optionality (whether or not object must be involved in the relationship).

2.5 Activity Diagrams

Activity diagrams are used to model the logic of a business process or method.

Activity diagrams (Rational, 1997) are used to document the logic of a single operation/method or the flow of logic of a business process. In many ways activity diagrams are the object-oriented equivalent of flow charts and data-flow diagrams (DFDs) from structured development (Gane & Sarson, 1978).

The activity diagram of Figure 5 shows the business logic for using a credit-card operated pump for filling your car with gasoline/petrol. The rounded rectangles represent processes to perform, the diamonds represent decision points, the arrows represent transitions between processes, the thick bars represent the start and end of potentially parallel processes, the filled circle represents the starting point of the activity, and the filled circle with a border represents the ending point.

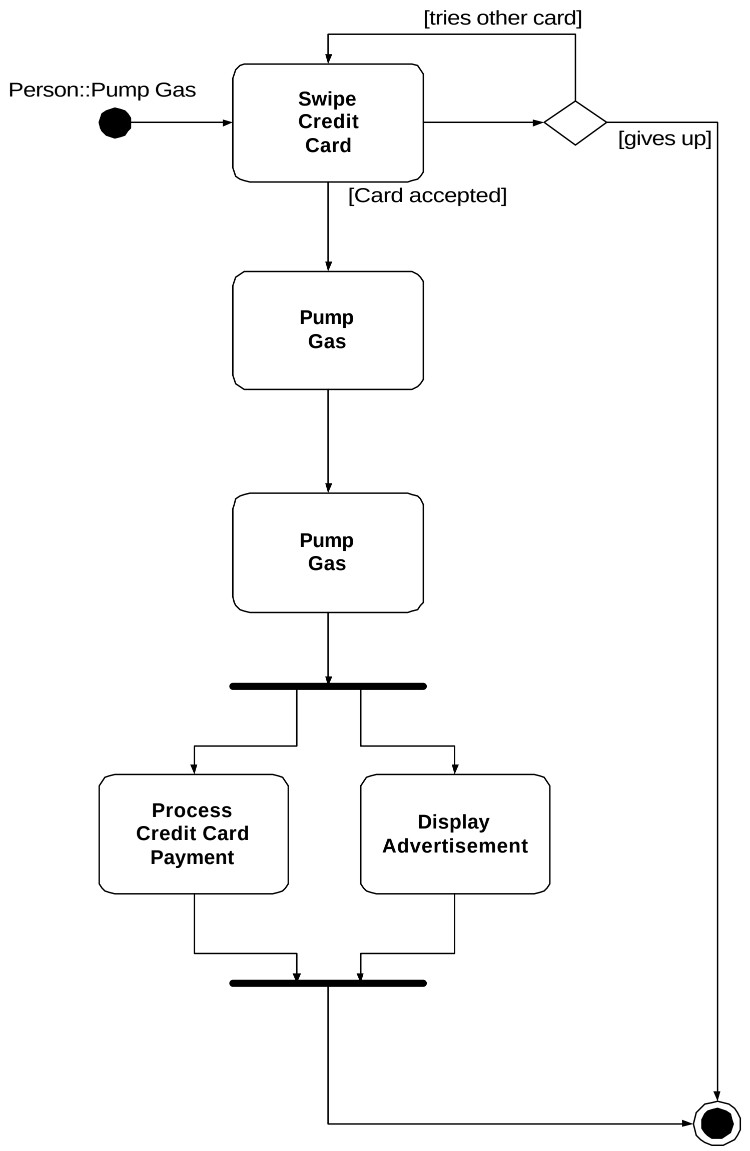


Figure 5. An activity diagram for using an automated gasoline/petrol pump.

Activity diagrams are usually documented, if at all, with a brief description of the activity and an indication of any actions taken during a process. In fact, processes can be described with more detailed activity diagrams or with a brief description. In many ways activity diagrams are simply a variation of statechart diagrams, described in section 2.10 below.

2.6 Data Diagrams

RDBs are commonly used to store objects. Therefore we need a diagram that describes how we will use them.

Relational databases are often used as the primary storage mechanism to make your objects persistent. Because relational databases do not completely support OO concepts the physical design of your database is often different than the design of your class diagram. *Data diagrams* (Hay, 1996; Ambler, 1998a) are used to communicate the physical design of a relational database.

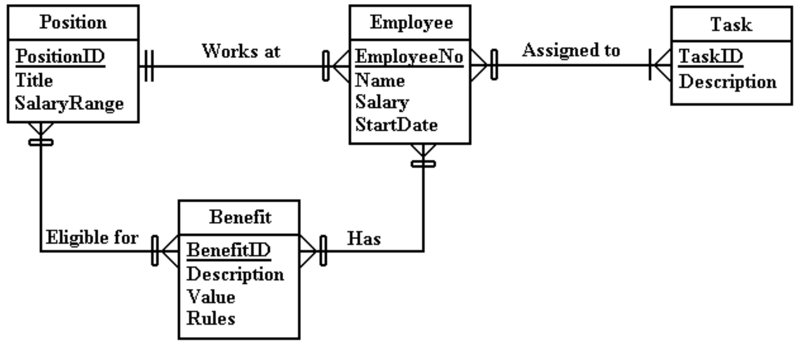


Figure 6. A data diagram for a simple human resources database.

In Figure 6 we see an example of a data diagram for the design of a simple human resources system. In the diagram we have four data entities **Position**, **Employee**, **Task**, and **Benefit** which in many ways are simply classes that have data but no functionality. The entities are connected by relationships. Relationships in a data diagram are identical in concept to associations in a class diagram. One interesting thing to note is the concept of a key: A key is one or more attributes that uniquely identify an entity. On data diagrams keys are indicated by underlining the attribute(s) that define them.

The strength of data diagrams is that data entities are conceptually the same as the tables of a relational database and that attributes are the same as table columns, providing a one-to-one mapping. Although often tempted to use data diagrams to drive the development of class diagrams, I tend to shy away from this approach. It is my experience that to successfully use relational technology for object-oriented applications you should let your class diagram drive the design of your data diagram because the class diagram models the full picture, data and behavior, needed by your OO application. In other words create the class diagram that is right for your application and then use it to derive the data base design for that application.

Data entities are described by a paragraph and their attributes, like those of classes, are documented with a description of what they contain, their type, and an indication of a range of values if applicable. Relationships between entities are documented with a description of their purpose and an indication of their cardinality (how many objects are involved in the relationship) and their optionality (whether or not object must be involved in the relationship).

2.7 Sequence Diagrams

Sequence diagrams are used to rigorously document and verify the logic contained within use cases.

A *sequence diagram* (Rational 1997; Jacobson, Christerson, Jonsson, Overgaard, 1992; Ambler, 1998a) is often used to rigorously define the logic for a use-case scenario. Because sequence diagrams look at the use case from a different direction from which it was originally developed, it is common to use sequence diagrams to validate your use cases. Figure 7 shows an example, using the UML notation, of a sequence diagram. Sequence diagrams are a design construct that are typically drawn by a group of developers, often the programmers responsible for implementing the scenario, lead by the designer or architect for the project.

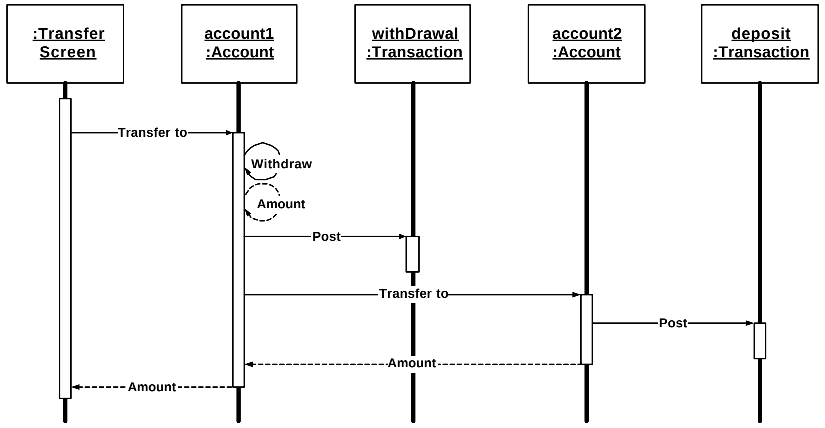


Figure 7. A sequence diagram for transferring funds from one account to another.

Traditional sequence diagrams show the types of objects[1] involved in the use case, the messages that they send each other, and any return values associated with the messages. For large applications it is quite common to show the components and use cases in addition to objects across the top of the diagram. The basic idea is that a sequence diagram shows the flow of logic of a use case in a visual manner, allowing you to both document and reality check your application design at the same time. The boxes on the vertical lines are called *method-invocation boxes* and they represent the running of a method in that object.

Sequence diagrams are a great way to review your work as they force you to walk through the logic to fulfill a use-case scenario. Second, they document your design, at least from the point of view of use cases. Third, by looking at what messages are being sent to an object/component/use case, and by looking at

[1] Objects (instances) in the UML are shown underlined to distinguish them from classes. In Figure 7 we have named and unnamed objects: :Transfer Screen is an instance of the **Transfer Screen** class and both account1 and account2 are instances of **Account**. We didn t name the instance of **Transfer Screen** because there is only one and we really do not care which one, whereas there are two instances of **Account** for this example therefore they needed to be named. If we had cared about which instance of **Transfer Screen**, perhaps it has to be screen 1701 for some reason (probably for testing), then we would have named it as well.

roughly how long it takes to run the invoked method, you quickly get an understanding of potential bottlenecks, allowing you to rework your design to avoid them.

When documenting a sequence diagram it is important to maintain traceability to the appropriate methods in your class diagram(s). The methods should already have their internal logic described as well as their return values (if they do not, time to document them).

2.8 Component Diagrams

Component diagrams show software components, their interfaces, and their interrelationships.

Component diagrams (Rational, 1997; Booch 1994) show the software components that make up a larger peice of software, their interfaces, and their interrelationships. For the sake of our discussion, a component may be any large-grain item such as a common subsystem, an exectuable binary file, a commercial off-the-shelf (COTS) system, an OO application, or a wrapped legacy application that is used in the day-to-day operations of your business. In many ways a component diagram is simply a class diagram at a larger, albeit less-detailed, scale.

Figure 8 shows an example of a component diagram being used to model the architectural business view of a telecommunications company. The boxes represent components, in this case either applications or internal subsystems, and the dotted lines represent dependencies between components. One of the main goals of architectural modeling is to partition a system into cohesive components that have stable interfaces, creating a core that need not change in reponse to subsystem-level changes (Mowbray, 1997). Component diagrams are ideal for this purpose.

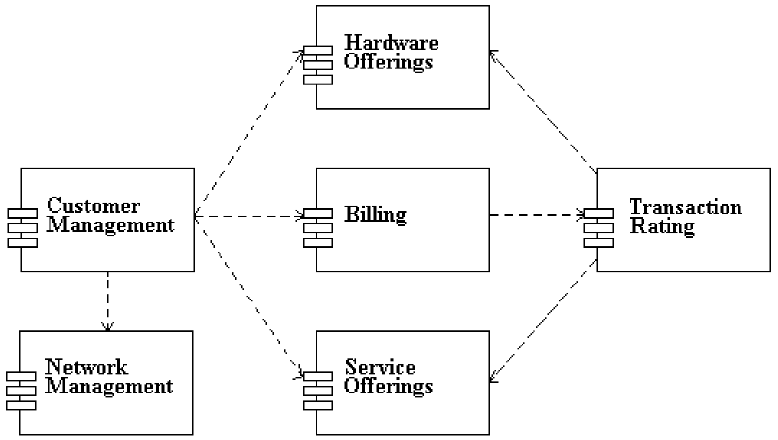


Figure 8. A component diagram for the architectural business view of a telecommunications company.

Each component within the diagram will be documented either by a more detailed component diagram, a use-case diagram, or by a class diagram. In the example presented in Figure 8 it is likely that you would want to develop a set of detailed models for the component **Customer Management** because it is a reasonably well-defined subset. At the same time you would draw a more detailed component diagram for **Network Management** because it is a large and complex domain that needs to be broken down further.

Components can be implemented using a wide range of technologies, include CORBA, Microsoft's COM+, Java, and Enterprise JavaBeans (EJB).

2.9 Deployment Diagrams

Deployment diagrams show the run-time configuration of hardware and software components.

Deployment diagrams (Rational, 1997) show the configuration of run-time processing components and the software that runs on them. Figure 9 shows an example of a deployment diagram, using the UML notation, which models the configuration of a customer service application that takes a three-tier client server approach. Deployment diagrams are reasonably simple models that are used to show how the hardware and software components will be configured and deployed for an application.

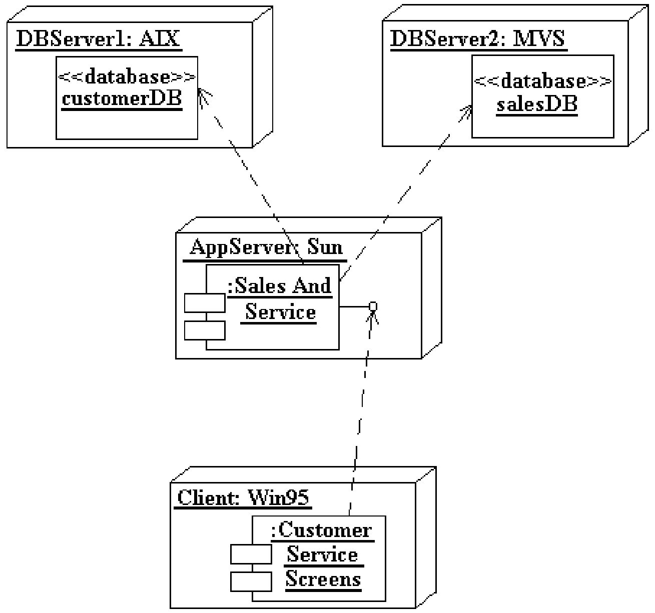


Figure 9. A deployment diagram for a three-tier client/server application.

Deployment diagrams reflect decisions that have been made by the technical architecture group. In Figure 9 a three-tier client/server architecture has been chosen, as well as those within the detailed models developed for the application. The message flow between components is often analyzed to determine which software components should be deployed to which hardware devices. The idea is that you want to utilize the hardware at your disposal in the best way possible to meet the requirements for your application.

For each component of a deployment diagram you will want to document the applicable technical issues, such as the required transaction volume, the expected network traffic, and the required response time. Furthermore, each component will be documented by a set of appropriate models. For example the databases will be described with data models, the application server will be described with a component diagram and/or class diagram, and the customer service screens would at least be documented by an interface-flow diagram and a prototype.

2.10 State Diagrams

Objects have both behavior and state, in other words they do things and they know things. Some objects do and know more things, or at least more complicated things, than other objects. Some objects are incredibly complex, so to better understand them we often draw a *statechart diagram* (Rational, 1997; Ambler, 1998a; Booch 1994; Rumbaugh, Blaha, Premerlani, Eddy, & Lorenson, 1991; Shlaer & Mellor 1992) to describe how they work.

Statechart diagrams show the various states, and the transitions between those states, of an object.

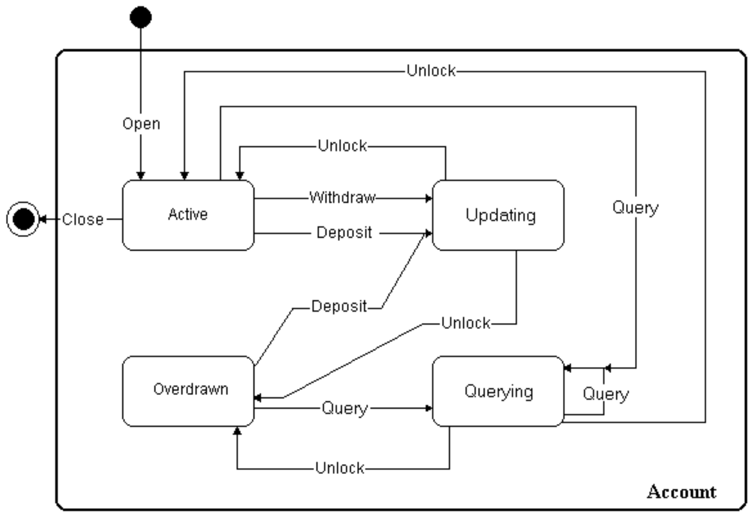


Figure 10. A statechart diagram for a bank account.

In Figure 10 we see the statechart diagram for a bank account. The rectangles represent *states* which are stages in the behavior of an object. States are represented by the attribute values of an object. The arrows represent *transitions*, progressions from one state to another that are represented by the invocation of a method on an object/class. Transitions are often a reflection of our business rules. There are also two kinds of psuedo states, an initial state in which an object is first created and a final state that an object doesn t leave once it enter it. Initial states are shown as closed circles and final states shown as an open circle enclosing a solid circle. In Figure 10 we see that when an account is active we see that we can withdraw from it, deposit to it, query it, and close it.

States are documented by a paragraph describing them and a indication of the range of values that applicable attributes take in the state, for example when an account is overdrawn the balance is negative. It is also appropriate to document any actions that are taken when an object enters a state, for example when an account becomes overdrawn a twenty-five dollar fine is charged to it. Transitions are documented with an indication of the event that triggers them. Where statechart diagrams are used to document the internal complexities of a class, collaboration diagrams are used to document the external interactions between objects.

2.11 Collaboration Diagrams

Unlike some notations (Coad & Yourdon, 1991; Ambler, 2000a) that show both state and behavior on class diagrams, the UML separates out behavior into *collaboration diagrams* (Rational, 1997; Ambler, 1998a). The basic difference between the two approaches is that UML class diagrams do not include messages, which makes sense because messages tend to clutter your class diagram and make them difficult to read. Because UML class diagrams do not show the message flow between classes a separate diagram, the collaboration

Collaboration diagrams show the collaborations (messages), but not their order, between objects.

diagram, was created to do so. Collaboration diagrams show the message flow between objects in an OO application and imply the basic associations (relationships) between classes.

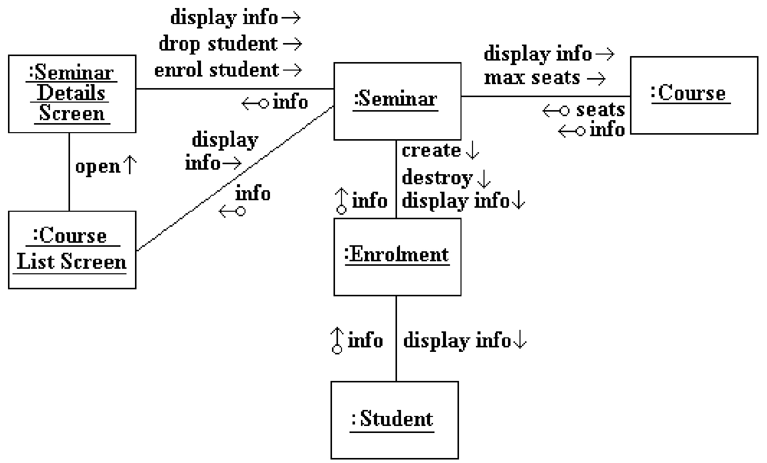


Figure 11. A collaboration diagram for a simple university.

Figure11 presents a simplified collaboration diagram for a university application. The rectangles represent the various objects[2] that make up the application, and the lines between the classes represent the relationships/associations between them. Messages are shown as a label followed by an arrow indicating the flow of the message and return values are shown as labels with arrow-circles beside them. In the figure there are instances of the **Seminar** and **Enrolment** classes, **open** and **display info** are both messages, and **seats** is a return value (presumably the result of sending the message **max seats** to **Course**).

Collaboration diagrams are usually drawn in parallel with class diagrams and sequence diagrams. Class diagrams provide input into the basic relationships between objects, and sequence diagrams provide an indication of the message flow between objects. The basic idea is that you identify the objects, the associations between the objects, and the messages that are passed between the objects. Collaboration diagrams are used to get a big picture outlook for the system, incorporating the message flow of many use case scenarios. Although you can indicate the order of message flow on a Collaboration Diagram, by numbering the messages, this typically is not done as sequence diagrams are much better at showing message ordering.

3. How the Modeling Techniques Fit Together

Detailed modeling   also called component modeling, application modeling, or subsystem modeling concentrates on the modeling of one application or subsystem. Where the architectural models define the components needed to support your organization, a detailed model defines the inner workings of a single component.

[2] The UML also allows you to indicate the roles that objects take on, a common occurrence in both collaboration diagrams and sequence diagrams.

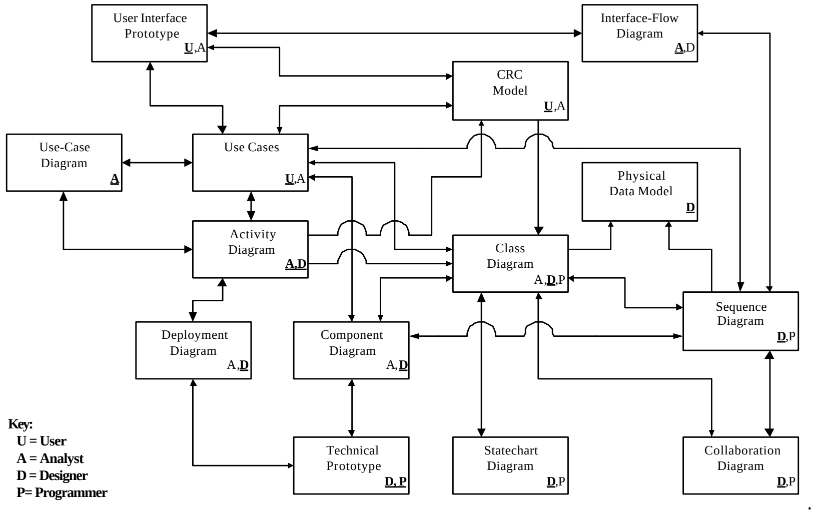


Figure 12. The Detailed Modeling process pattern.

Figure 12 depicts the Detailed Modeling process pattern (Ambler, 1998b) in which the boxes represent the main techniques/diagrams of OO modeling and the arrows show the relationships between them, with the arrow heads indicating an input into relationship. For example, we see that an activity diagram is an input into a class diagram. In the bottom right-hand corner of each box are letters which indicate who is typically involved in working on that technique/diagram. The key is straightforward: U=User, A=Analyst, D=Designer, and P=Programmer. The letter that is underlined indicates the group that performs the majority of the work for that diagram. For example, we see that users form the majority of the people involved in developing a CRC model and designers form the majority of those creating statechart diagrams.

An interesting feature of Figure 12 is that it illustrates that the object-oriented modeling process is both serial in the large and iterative in the small. The serial nature is exemplified when you look from the top-left corner to the bottom right corner: the techniques move from requirements gathering to analysis to design. You see the iterative nature of OO modeling from the fact that each technique drives, and is driven by, other techniques. In other words you iterate back and forth between models.

From a serial perspective, Figure 13 depicts the Deliverables Drive Deliverables approach process pattern (Ambler, 1998b), indicating the general order in which you will work on deliverables during the Construct Phase. It is important to point out that the views in Figures 12 and 13 are complementary, not contradictory. In Figure 12 we see that we generally start modeling with techniques, such as use cases and CRC models, that focus on user requirements, moving into analysis-oriented techniques such as sequence and component diagrams, then into design techniques and finally to code. The arrows in Figure 13 represent a documents relationship. For example a use-case diagram is documented by use cases, which in turn are documented by sequence diagrams. Component diagrams are interesting in that a component within a component diagram is often documented by either another component diagram, a class diagram, and/or a use-case diagram.

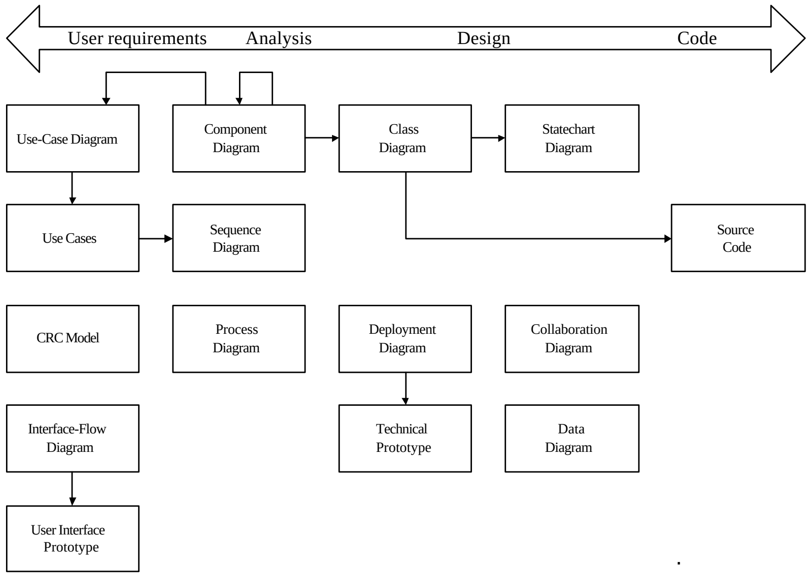


Figure 13. The Deliverables Document Deliverables process pattern.

As an aside, in the UML (Rational, 1997) the *traces* stereotype is used to connect related pieces of information in separate models to maintain traceability throughout your work. Traceability is an important concept for testing.

4. Summary

It is my experience that the UML is a very good start at describing the models that are needed to develop a model representing an OO application, but that it is not sufficient. In this white paper we explored several techniques for modeling an OO application, many of which are included in the UML, and saw how they fit together.

5. References

Ambler, S.W. (1998a). *Building Object Applications That Work Your Step-by-Step Handbook for Developing Robust Systems With Object Technology*. New York: Cambridge University Press.

Ambler, S.W. (1998b). *Process Patterns Building Large Scale Systems Using Object Technology* . New York: Cambridge University Press.

Ambler, S.W. (1999). *More Process Patterns: Delivering Large-Scale Systems Using Object Technology*. New York: Cambridge University Press.

Ambler, S.W. (2000a). *The Object Primer2nd Edition The Application Developer s Guide to Object-Orientation*. New York: Cambridge University Press.

Ambler, S.W. (2000b). *The Unified Process Inception Phase*. Gilroy, CA: R&D Books.

Ambler, S.W. (2000c). *The Unified Process Elaboration Phase*. Gilroy, CA: R&D Books.

Ambler, S.W. (2000d). *The Unified Process Construction Phase*. Gilroy, CA: R&D Books.

Beck, K. & Cunningham, W. (1989). *A Laboratory for Teaching Object-Oriented Thinking.* Proceedings of OOPSLA 89, pp. 1-6.

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications, 2nd Edition*. Redwood City, California: The Benjamin/Cummings Publishing Company, Inc.

Coad, P. (1992) *Object-Oriented Patterns*. Communications of the ACM, 35(9) pp. 152-159.

Coad, P., North, D., & Mayfield, M. (1995). *Object Models Strategies, Patterns, & Applications*. Englewood Cliffs, NJ: Yourdon Press.

Coad, P., Yourdon, E. (1991). *Object-Oriented Analysis, 2nd Edition*. Englewood Cliffs, New Jersey: Yourdon Press.

Gane, C., Sarson, T. (1978). *Structured Systems Analysis: Tools and Techniques*. Englewood Cliffs, New Jersey: Prentice Hall, Inc.

Hay, D.C. (1996). *Data Model Patterns: Conventions of Thought*. New York: Dorset House Publishing.

Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G. (1992). *Object-Oriented Software Engineering A Use Case Driven Approach*. ACM Press.

Mowbray, T. (1997). *Architectures: The Seven Deadly Sins of OO Architecture*. New York: SIGS Publishing, Object Magazine April, 1997, 7(1), pp. 22-24.

Page-Jones, M. (1995). *What Every Programmer Should Know About Object-Oriented Design*. New York: Dorset-House Publishing.

Rational (1997). *The Unified Modeling Language v1.1 Documentation Set*. Rational Software Corporation, Monterey California.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, Inc.

Shlaer, S., Mellor, S. (1992). *Object Life Cycles Modeling the World in States*. Englewood Cliffs, New Jersey: Yourdon Press.

Wirfs-Brock, R., Wilkerson, B., & Wiener, L. (1990). *Designing Object-Oriented Software*. New Jersey: Prentice Hall, Inc.

Yourdon, E. (1997). *Death March: The Complete Software Developer s Guide to Surviving Mission Impossible Projects*. Upper Saddle River, NJ: Prentice-Hall , Inc.

6. Glossary

Activity diagram A UML diagram which can be used to model a high-level business process or the transitions between states of a class (in this respect activity diagrams are effectively specializations of statechart diagrams).

Actor Any person, organization, or system that interacts with an application but is external to it.

Analysis An approach to modeling where the goal is understanding the problem domain.

Analysis paralysis A derogatory term used by system professionals to describe the actions of a development team that spends too much time modeling, trying to document every minute detail.

Analysis pattern -- A pattern that describes a solution to a business/analysis problem.

Architectural modeling High-level modeling, either of the problem or technical domain, whose goal is to provide a common, overall vision of the problem domain. Architectural models provide a base from which detailed modeling can begin.

Business-domain expert (BDE) -- Someone with intimate knowledge of all or a portion of a problem domain. Often referred to as a subject matter expert (SME).

CASE Computer aided system engineering.

Class diagram -- Class diagrams show the classes of a system and their intrarelationships. Class diagrams are often mistakenly referred to as object models.

Collaboration diagram Collaboration diagrams show instances of classes, their interrelationships, and the message flow between them. The order of the messaging is not indicated.

Component diagram A diagram that shows the software components, their interrelationships, interactions, and their public interfaces that comprise an application, system, or enterprise.

CRC (Class Responsibility Collaborator) card A standard index card divided into three sections that show the name of the class, the responsibilities of the class, and the collaborators of the class.

CRC model A collection of CRC cards that describe the classes that make up a system or a component of a system.

Data diagram A diagram used to communicate the design of a (typically relational) database. Data diagrams are often referred to as entity-relationship (ER) diagrams.

Data model A data diagram and its corresponding documentation.

Diagram A visual representation of a problem or solution to a problem.

Domain architecture A collection of high-level models that describe the problem domain. Domain architectures are typically documented by high-level use cases, use-case diagrams, and class models that describe the various sub domains and the relationships between them.

Design A style of modeling with the goal of describing how a system will be built based on the defined requirements.

Design pattern A pattern that describes a solution to a design problem.

Enterprise modeling The act of modeling an organization and its external environment from a business, not and information system, viewpoint.

Feature creep The addition as development proceeds of new features to an application that are above and beyond what the original specification called for. This is also called scope creep.

Interface-flow diagram A diagram that models the interface objects of your system and the relationships between them.

Joint application design (JAD) A structured, facilitated meeting in which modeling is performed by both users and developers together. JADs are often held for gathering user requirements.

Message-invocation box The long, thin vertical boxes that appear on sequence diagrams that represent a method invocation in an object.

Middleware The technology that allows computer hardware to communicate with one another. This includes the network itself, its operating system, and anything needed to connect computers to the network.

Model An abstraction describing a problem domain and/or a solution to a problem domain. Traditionally models are thought of as diagrams plus their corresponding documentation although non-diagrams such as interview results, requirement documents, and collections of CRC cards are also considered to be models.

Modeling The act of creating or updating one or more models.

Notation The set of symbols that are used in the drawing of diagrams. The Unified Modeling Language (UML) defines a defacto industry-standard modeling notation.

Pattern A model of several classes that work together to solve a common problem in your problem or technical domain.

Persistence mechanism-- The permanent-storage facility used to store objects. Relational databases, flat files, and objectbases are all potential persistence mechanisms.

Process diagram A diagram that shows the movement of data within a system. Similar in concept to a DFD but not as rigid and documentation heavy.

Process pattern A pattern which describes a proven, successful approach and/or series of actions for developing software.

Prototyping An iterative analysis technique in which users are actively involved in the mocking up of the user interface for an application.

Requirements document A document which describes the user, technical, and environmental requirements for an application. This document potentially contains the major use cases, detailed use-case scenarios, and traditional requirements for the application as well.

Sequence diagram A diagram that shows the types of objects involved in a use-case scenario, including the messages they send to one another and the values that they return.

Statechart diagram A diagram that describes the states that an object may be in, as well as the transitions between states.

Technical architecture A set of models and documents that describes the technical components of an application, including but not limited to the hardware, software, middleware, persistence mechanisms, and operating systems to be deployed.

Unified Modeling Language (UML) The industry standard OO modeling notation proposed by Rational Corporation of Santa Clara California. At the time of this writing the UML is being considered by the Object Management Group (OMG) to make it the OMG standard.

Use case A description of a high-level user requirement that an application may or may not be expected to handle.

Use-case diagram A diagram that shows the use cases and actors for the application that we are developing.

Use-case scenario A description of a specific, detailed user requirement that an application may or may not be expected to handle. A use-case scenario is a detailed example of a use case.

7. About the Author

Scott W. Ambler is a Software Process Mentor living in Newmarket, Ontario, 45 km north of Toronto, Canada and is President of Ronin International (www.ronin-intl.com) a consulting firm specializing in object-oriented architecture, software process, and Enterprise JavaBeans (EJB) development. He has worked with OO technology since 1990 in various roles: Business Architect, System Analyst, System Designer, Process Mentor, Lead Modeler, Smalltalk Programmer, Java Programmer, and C++ Programmer. He has also been active in education and training as both a formal trainer and as an object mentor.

Scott has a Master of Information Science and a Bachelor of Computer Science from the University of Toronto. He is the author of the best-selling books *The Object Primer*, *Building Object Applications That Work*, *Process Patterns*, and *More Process Patterns* and co-author of *The Elements of Java Style*, all of which are published by Cambridge University Press (www.cup.org). Scott is also editor of *The Unified Process Series* from R&D Books (www.rdbooks.com) to be published in 2000. Scott is a contributing editor and columnist with *Software Development* (http://www.sdmagazine.com) and writes columns for *Computing Canada* (http://www.plesman.com).

He can be reached via e-mail at:
scott@ambysoft.com
scott.ambler@ronin-intl.com

Visit his personal web site:
http://www.AmbySoft.com

Visit his corporate web site:
http://www.ronin-intl.com

Index