An Introduction to

Unified Modeling Language

1. Importance of modeling

The primary goal of SDLC (Software Development Life Cycle) is to produce the software that meets the all requirements – modeling comes as the secondary goal. But in order to develop the software of lasting quality, the modeling works as a strong architectural foundation as well as a central part of activities that lead up to the development and deployment of a good software. It provides the clear understanding of desired structure and behavior of the system.

To build industry-strength software, the main idea lies in creating the right software and even writing fewer amounts of codes. It makes the quality software engineering an issue of architecture, process and tools. In addressing these issues, modeling is a proven and well-accepted engineering technique that simplifies the complex reality.

1.1. There are four aims of a model: -

1. Help us to visualize a system as it is or as we want it to be.
2. Permit us to specify the structure or behavior of a system.
3. Give us a template that guides us in constructing a system.
4. Document the decision we have made.

1.2. Principles of modeling

1. The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped
   - Database development point of view may come up with E-R diagram as a model and triggers and stored procedures will be taken as behavior of the system,
   - A structured analyst will propose algorithmic-centric models that are purely process driven and DFD will be considered instead of behaviors
   - An OO practitioner will take the system as architecture of classes and patterns of interaction among the classes.
2. Every model may be expressed at different levels of precision.
   - A small and quickly executable model might be enough for some software.
   - Sometimes the model requires the modeler to go down to the machine level and even to struggle with data transfer through networks (e.g. cross-platform system)
   - Depending upon the type of stakeholders and applications, different level of views and models are needed.
3. The best models are connected to reality.
   - All models should simplify the real system but it must not mask any realistic detail e.g. only a mathematical model for an aircraft is not connected to reality.
   - OO models pay an attention to connecting to reality in analysis, design and implementation phases as well as to the inter-connection between phases too.
4. No single model is sufficient. Every non trivial system is best approached through a small set of nearly independent models.
   - Nearly independent but still inter related models and views help understand the architecture of the system in different point of view

- In OO model, use case view exposes the requirements, design view captures the vocabulary of the problem and solution space, process view models the distribution of processes and threads, implementation view addresses the physical realization of the system, and the deployment view focuses on system engineering issues.
- Different models have different level of importance depending upon the type of problem domain.

2. UML: An overview

- Booch, Rumbaugh and Jacobson invented a standard language for software designs – called UML.
- UML is a language that addresses different views of a system's architecture as it evolves throughout the SDLC
- The vocabulary and rules of UML tell you how to create and read well-formed models.
- UML, in broader sense, is a language for visualizing, specifying, constructing and documenting the artifacts of an objective oriented software system.

2.1. UML as a language for Visualizing

- Some models are best represented with text (like use cases) and some with graphically.
- It provides a standard convention of graphics to visualize all models, which would be kept inside the programmer's mind instead.
- Keeps the graphical documentation available for everybody to interpret in the same was as the original developer.
- It is more than a group of graphical symbols – it also provides well defined semantics behind each symbol.

2.2. UML as a language for Specifying

- It specifies models precisely, unambiguously and completely.
- UML has specification for all artifacts like analysis, design and implementation decisions.

2.3. UML as a language for Constructing

- Though it is not programming language, all its models can be mapped to codes of many popular languages like Java, C++, VB etc. (Forward engineering)
- Reverse engineering is also possible with a certain amount of human intervention. Thus UML makes the round-trip engineering possible in OO software engineering.

2.4. UML as a language for Documenting

- It addresses the systems architecture and all of its details by keeping a well organized documentation along with executables.
- UML also has means to express requirements and to model project planning and release management activities.

3. Building blocks of UML

There are three kinds of building blocks: -

1. **Things** : first level abstractions in a model
2. **Relationships** : tie things together
3. **Diagram** : group collection of things

3.1. Things in UML

There are four types of things: -

1. Structural things
2. Behavioural things
3. Grouping things
4. Annotational things

3.1.1. Structural things

They are usually nouns of UML model and are static parts of a model. They represent elements either physical or logical things of a model. There are seven (7) different structural things.

1. ***Class***: description of set of objects that share the same attributes, operations, relationships, and semantics	2. ***Interface***: collection of operations that specify service of a class or component
3. ***Collaboration***: society of roles and other elements that work together. It defines an interaction between elements and provide some cooperative behaviour	4. ***Use Case***: description of set of sequence of actions. It results an observable result to an actor.
5. ***Active Class***: a class whose objects own one or more processes or threads and therefore can initiate control activity.	6. ***Component*** : physical and replaceable part of the system that conforms to and provides the realization of a set of interfaces
7. ***Node*** : a physical element that exist at run time and represents a computational resource at least some memory and processing capability.	

3.1.2. Behavioural things

They are dynamic parts of UML models – generally verbs of model, representing behaviour over time and space. There are primarily two kinds of behavioural things

1. ***Interaction*** : consists of set of messages exchanged among a set of objects	2. ***State Machine*** : specifies the sequences of states an object or an interaction goes through.

3.1.3. Grouping things

These are boxes into which a model can be decomposed. There is one primary kind of grouping things called packages.

Package: a general purpose mechanism for organizing elements into groups. A package may contain structural and behavioural things and even another grouping thing itself.

3.1.4. Annotational things

These are explanatory part of UML model. A primary kind of annotational thing is there – i.e. note.
Note: It is simply a symbol for keeping comments and constraints attached to an element.

3.2. Relationships in UML

Relationships join things together showing the kind of link a thing has with another thing.

1. ***Dependency***: semantic relationship between two things in which a change to one thing may affect the semantics of the other thing.	2. ***Association***: structural relationship that describes a set of links i.e. connection among objects. Aggregation is a special kind of association, which is a relationship of a whole to its parts.
3. ***Generalization***: specialization or generalization relationship in which objects of the specialized element (child) are substitutable for objects of the generalized element (parent).	4. ***Realization***: semantic relationship between classifiers, wherein one classifier species a contract that another classifier guarantees to carry out.

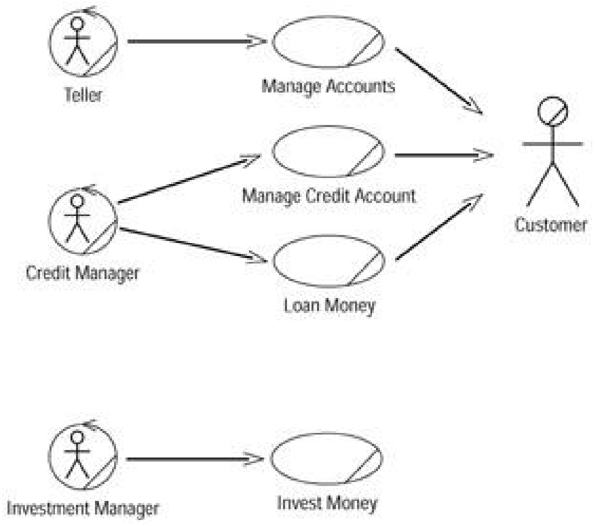
3.3. Diagrams in UML

A diagram is the graphical representation of a set of elements – more often a connected graph of things and relationships. There are nine (9) diagrams: -

1. ***Class diagram***: shows a set of classes, interfaces and collaborations and their relationships. It gives static design view of a system	2. ***Object diagram***: shows a set of objects and their relationships. As class diagram, it also gives static design view of a system but from the perspective of prototypical case.
3. ***Use case diagram***: shows a set of use cases and actors and their relationships. It addresses the static use case view of a system.	4. ***Sequence diagram***: shows an interaction, consisting of a set of objects and their relationships, including the messages. It addresses a dynamic view of a system.
5. ***Collaboration diagram***: shows an interaction diagram with the emphasis on structural organization of the objects that sends and receives messages.	6. ***Statechart diagram***: shows a state machine, consisting of states, transitions, events and activities. It addresses a dynamic view of a system.
7. ***Activity diagram***: special kind of statechart diagram that shows a flow from activity to activity within a system.	8. ***Component diagram***: shows the organizations and dependencies among a set of components. It addresses a static implementation view of a system.
9. ***Deployment diagram***: shows the configuration of run-time processing nodes and the components that live on them. It addresses static deployment of architecture.	

3.3.1. Business Use Case Diagrams

Business Use Case diagrams are used to represent the functionality provided by an organization as a whole. They answer the questions "What does the business do?" and "Why are we building the system?" They are used extensively during business modeling activities to set the context for the system and to form a foundation for creating the use cases.



Business Use Case diagram for a financial institution

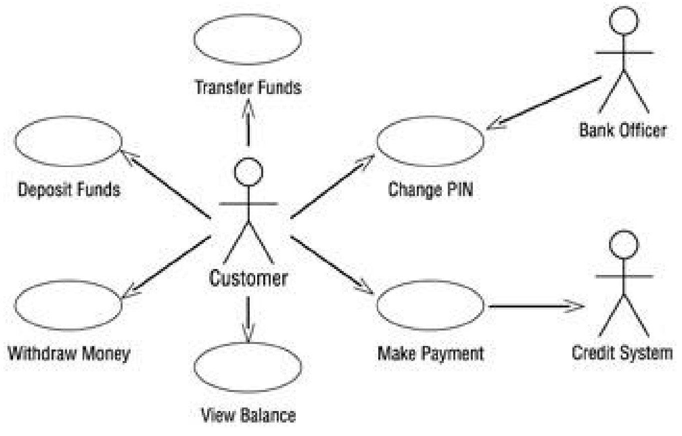
Business Use Case diagrams are drawn from the organizational perspective. They do not differentiate between manual and automated processes. (Use Case diagrams, which will be discussed next, focus on the automated processes.) Business Use Case diagrams show the interactions between business use cases and business actors. Business use cases represent the processes that a business performs, and business actors represent roles with which the business interacts, such as customers or vendors. In other words, business actors represent anyone or anything outside the business that interacts with the business; they do not represent roles or workers within a business. Workers within a business are represented by business workers.

3.3.2. Use Case Diagrams

Use Case diagrams show the interactions between use cases and actors. Use cases represent system functionality, the requirements of the system from the user's perspective. Actors represent the people or systems that provide or receive information from the system; they are among the stakeholders of a system. Use Case diagrams, therefore, show which actors initiate use cases; they also illustrate that an actor receives information from a use case. In essence, a Use Case diagram can illustrate the requirements of the system.

While Business Use Case diagrams are not concerned with what is automated, Use Case diagrams focus on just the automated processes. There is not a one-to-one relationship between business use cases and use cases. A single business use case may require 30 use cases, for example, to implement the process.

This Use Case diagram shows the interactions between the use cases and actors of an ATM system. In this example, the bank's customer initiates a number of use cases: Withdraw Money, Deposit Funds, Transfer Funds, Make Payment, View Balance, and Change PIN. A few of the relationships are worthy of further mention. The bank officer can also initiate the Change PIN use case. The Make Payment use case shows an arrow going to the credit system. External systems may be actors and, in this case, the credit system is shown as an actor because it is external to



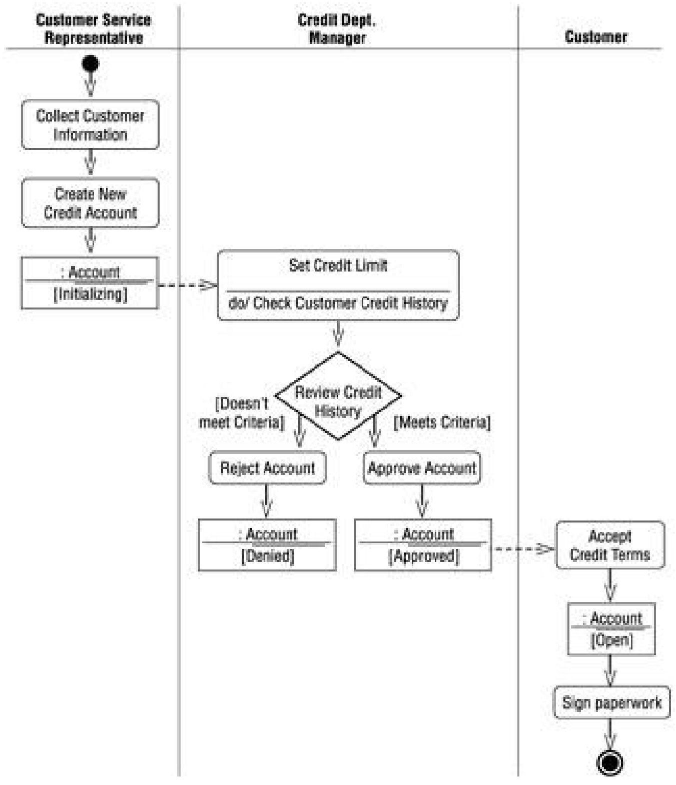
Use Case diagram for an Automated Teller Machine (ATM) system

the ATM system. The arrow going from a use case to an actor illustrates that the use case produces some information that an actor uses. In this case, the Make Payment use case provides credit card payment information to the credit system.

Much information can be gleaned from viewing Use Case diagrams. This one diagram shows the overall functionality of the system. Users, project managers, analysts, developers, quality assurance engineers, and anyone else interested in the system as a whole can view these diagrams and understand what the system is supposed to accomplish.

3.3.3. Activity Diagrams

Activity diagrams illustrate the flow of functionality in a system. They may be used in business modeling to show the business workflow. They may be used in requirements gathering to illustrate the flow of events through a use case. These diagrams define where the workflow starts, where it ends, what activities occur during the workflow, and in what order the activities occur. An activity is a task that is performed during the workflow.



Activity diagram for opening an account

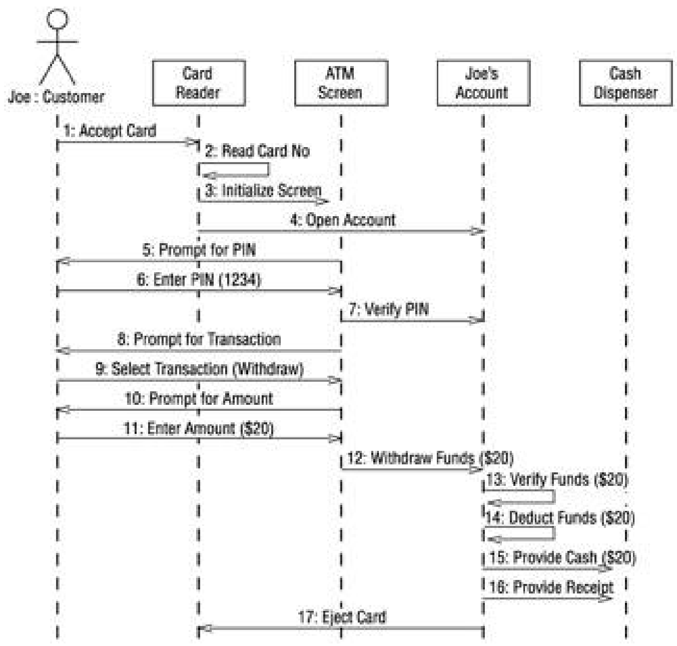
The structure of an activity diagram is similar to a Statechart diagram. The activities in the diagram are represented by rounded rectangles. These are the steps that occur as you progress through the workflow. Objects that are affected by the workflow are represented by squares. There is a start state, which represents the beginning of the workflow, and an end state, which represents the end. Decision points are represented by diamonds.

You can see the object flow through the diagram by examining the dashed lines. The object flow shows you which objects are used or created by an activity and how the object changes state as it progresses through the workflow. The solid lines, known as transitions, show how one activity leads to another in the process. If needed, you can place greater detail on the transitions, describing the circumstances under which the transition may or may not occur and what actions will be taken during the transition.

The activity diagram may be divided into vertical swimlanes. Each swimlane represents a different role within the workflow. By looking at the activities within a given swimlane, you can find out the responsibility of that role. By looking at the transitions between activities in different swimlanes, you can find out who needs to communicate with whom. All of this is very valuable information when trying to model or understand the business process.
Activity diagrams do not need to be created for every workflow, but they are powerful communication tools, especially with large and complex workflows.

3.3.4. Sequence Diagrams

Sequence diagrams are used to show the flow of functionality through a use case. For example, the Withdraw Money use case has several possible sequences, such as withdrawing money, attempting to withdraw without available funds, attempting to withdraw with the wrong PIN, and several others. The normal scenario of withdrawing $20 (without any problems such as entering the wrong PIN or insufficient funds in the account) is shown here.



Sequence diagram for Joe withdrawing $20

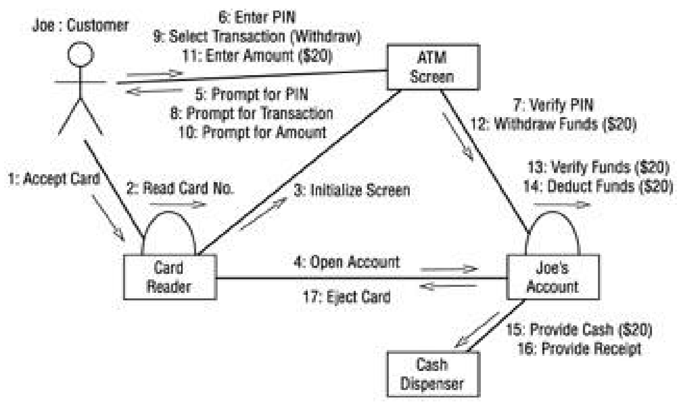
This Sequence diagram shows the flow of processing through the Withdraw Money use case. Any actors involved are shown at the top of the diagram; the customer actor is shown in the above example. The objects that the system needs in order to perform the Withdraw Money use case are also shown at the top of the diagram. Each arrow represents a message passed between actor and object or object and object to perform the needed functionality. One other note about Sequence diagrams—they display objects, not classes. Classes represent types of objects, they are specific; instead of just customer, the Sequence diagram shows Joe.

The use case starts with the customer inserting his card into the card reader, an object indicated by the rectangle at the top of the diagram. Then, the card reader reads the card number, opens Joe's account object, and initializes the ATM screen. The screen prompts Joe for his PIN. He enters 1234. The screen verifies the PIN with the account object and they match. The screen presents Joe with his options, and he chooses withdraw. The screen then prompts Joe for the amount to withdraw. He chooses $20. Then, the screen withdraws the funds from the account. This initiates a series of processes that the account object performs. First, Joe's account verifies that the account contains at least $20. Then, it deducts the funds from the account. Next, it instructs the cash dispenser to provide $20 in cash. Joe's account also instructs the dispenser to provide a receipt. Lastly, it instructs the card reader to eject the card.

This Sequence diagram illustrated the entire flow of processing for the Withdraw Money use case by showing a specific example of Joe withdrawing $20 from his account. Users can look at these diagrams to see the specifics of their business processing. Analysts see the flow of processing in the Sequence diagrams. Developers see objects that need to be developed and operations for those objects. Quality assurance engineers can see the details of the process and develop test cases based on the processing. Sequence diagrams are therefore useful for all stakeholders in the project.

3.3.5. Collaboration Diagrams

Collaboration diagrams show exactly the same information as the Sequence diagrams. However, Collaboration diagrams show this information in a different way and with a different purpose.



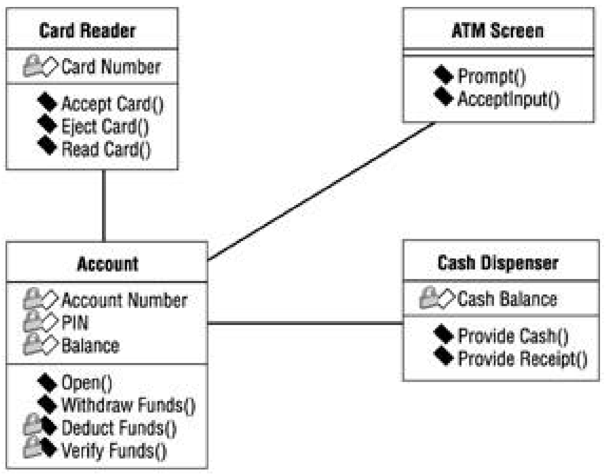
Collaboration diagram for Joe withdrawing $20

In this Collaboration diagram, the objects are represented as rectangles and the actors are stick figures, as before. Whereas the Sequence diagram illustrates the objects and actor interactions over time, the Collaboration diagram shows the objects and actor interactions without reference to time. For example, in this diagram, we see that the card reader instructs Joe's account to open and Joe's account instructs the card reader to eject the card. Also, objects that directly communicate with each other are shown with lines drawn between them. If the ATM screen and cash dispenser directly communicated with one another, a line would be drawn between them. The absence of a line means that no communication occurs directly between those two objects.

Collaboration diagrams, therefore, show the same information as Sequence diagrams, but people look at Collaboration diagrams for different reasons. Quality assurance engineers and system architects look at these to see the distribution of processing between objects. Suppose that the Collaboration diagram was shaped like a star, with several objects communicating with a central object. A system architect may conclude that the system is too dependent on the central object and redesign the objects to distribute the processing power more evenly. This type of interaction would have been difficult to see in a Sequence diagram.

3.3.6. Class Diagrams

Class diagrams show the interactions between classes in the system. Classes can be seen as the blueprint for objects. Joe's account, for example, is an object. An account is a blueprint for Joe's checking account; an account is a class. Classes contain information and behavior that acts on that information. The Account class contains the customer's PIN and behavior to check the PIN. A class on a Class diagram is created for each type of object in a Sequence or Collaboration diagram.



Class diagram for the ATM system's Withdraw Money use case

The Class diagram shows the relationships between the classes that implement the Withdraw Money use case. This is done with four classes: Card Reader, Account, ATM Screen, and Cash Dispenser. Each class on a Class diagram is represented by a rectangle divided into three sections. The first section shows the class name. The second section shows the attributes the class contains. An attribute is a piece of information that is associated with a class. For example, the Account class contains three attributes: Account Number, PIN, and Balance. The last section contains the operations of the class. An operation is some behavior that the class will provide. The Account class contains four operations: Open, Withdraw Funds, Deduct Funds, and Verify Funds.

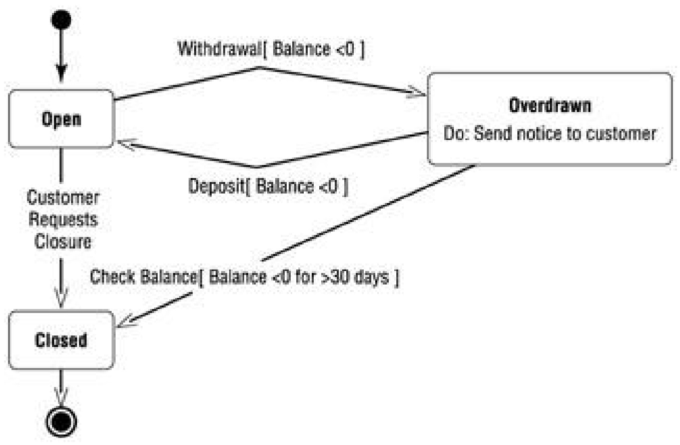
The lines connecting classes show the communication relationships between the classes. For instance, the Account class is connected with the ATM Screen class because the two directly communicate with each other. The Card Reader is not connected to the Cash Dispenser because the two do not communicate. Another point of interest is that some attributes and operations have small padlocks to the left of them. The padlock indicates a private attribute or operation. Private attributes and operations can only be accessed from within the class that contains them. The Account Number, PIN, and Balance are all private attributes of the Account class. In addition, the Deduct Funds and Verify Funds operations are private to the Account class.

Developers use Class diagrams to actually develop the classes. Tools such as Rose generate skeletal code for classes, then developers flesh out the details in the language of their choice. Analysts use Class diagrams to show the details of the system. Architects also look at Class diagrams to see the design of the system. If one class contains too much functionality, an architect can see this in the Class diagram and split out the functionality into multiple classes. Should no relationship exist between classes that communicate with each other, an architect or developer can see this too. Class diagrams should be created to show the classes that work together in each use case, and comprehensive diagrams containing whole systems or subsystems can be created as well.

3.3.7. Statechart Diagrams

Statechart diagrams provide a way to model the various states in which an object can exist. While the Class diagrams show a static picture of the classes and their relationships, Statechart diagrams are used to model the more dynamic behavior of a system. These types of diagrams are extensively used in building real-time systems. Rose can even generate the full code for a real-time system from the Statechart diagrams.

A Statechart diagram shows the behavior of an object. For example, a bank account can exist in several different states. It can be open, closed, or overdrawn. An account may behave differently when it is in each of these states. Statechart diagrams are used to show this information.


Withdrawal[Balance <0]
Open
Overdrawn
Do: Send notice to customer
Deposit[Balance <0]
Customer Requests Closure
Check Balance[Balance <0 for >30 days]
Closed

Statechart diagram for the Account class

In this diagram, we can see the states in which an account can exist. We can also see how an account moves from one state to another. For example, when an account is open and the customer requests the account's closure, the account moves to the closed state. The customer's request is called the event and the event is what causes a transition from one state to another.

If the account is open and the customer makes a withdrawal, the account may move to the overdrawn state. This will only happen if the balance of the account is less than zero. We show this by placing [Balance < 0] on the diagram. A condition enclosed in square brackets is called a guard condition, and controls when a transition can or cannot occur.

There are two special states—the start state and the stop state. The start state is represented by a black dot on the diagram, and indicates what state the object is in when it is first created. The stop state is represented by a bull's-eye, and shows what state the object is in just before it is destroyed. On a Statechart diagram, there is one and only one start state. On the other hand, you can have no stop state, or there can be as many stop states as you need.

Certain things may happen when the object is inside a particular state. In our example, when an account is overdrawn, a notice is sent to the customer. Processes that occur while an object is in a certain state are called actions.

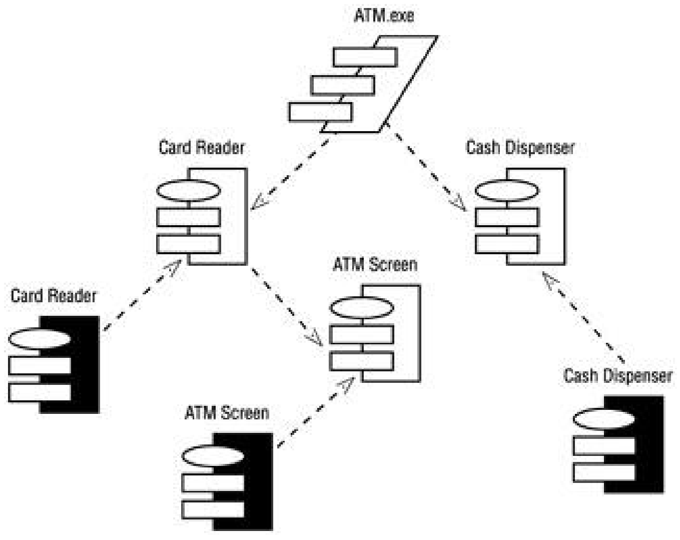
Statechart diagrams aren't created for every class; they are used only for very complex classes. If an object of the class can exist in several states, and behaves very differently in each of the states, you may want to create a Statechart diagram for it. Many projects won't need these diagrams at all. If they are created, developers will use them when developing the classes.

Statechart diagrams are created for documentation only. When you generate code from your Rose model, no code will be generated from the information on the Statechart diagrams. However, Rose add- ins are available for real- time systems that can generate executable code based on Statechart diagrams.

3.3.8. Component Diagrams

Component diagrams show you a physical view of your model, as well as the software components in your system and the relationships between them. There are two types of components on the diagram: executable components and code libraries.

Each of the classes in the model is mapped to a source code component. Once the components have been created, they are added to the Component diagram. Dependencies are then drawn between the components. Component dependencies show the compile- time and run- time dependencies between the components.

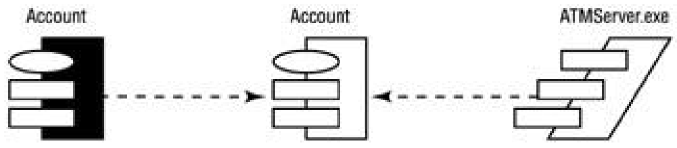


Component diagram for the ATM client

This Component diagram shows the client components in the ATM system. In this case, the team decided to build the system using C++. Each class has its own header and body file, so each class is mapped to its own components in the diagram. For example, the ATM Screen class is mapped to the ATM Screen component. The ATM Screen class is also mapped to a second ATM Screen component. These two components represent the header and body of the ATM Screen class. The shaded component is called a package body. It represents the body file (.cpp) of the ATM Screen class in C++. The unshaded component is called a package specification. The package specification represents the header (.h) file of the C++ class. The component called ATM.exe is a task specification and represents a thread of processing. In this case, the thread of processing is the executable program.

Components are connected by dashed lines showing the dependency relationships between them. For example, the Card Reader class is dependent upon the ATM Screen class. This means that the ATM Screen class must be available in order for the Card Reader class to compile. Once all of the classes have been compiled, then the executable called ATMClient.exe can be created.

The ATM example has two threads of processing and therefore two executables. One executable comprises the ATM client, including the Cash Dispenser, Card Reader, and ATM Screen. The second executable comprises the ATM server, including the Account component.



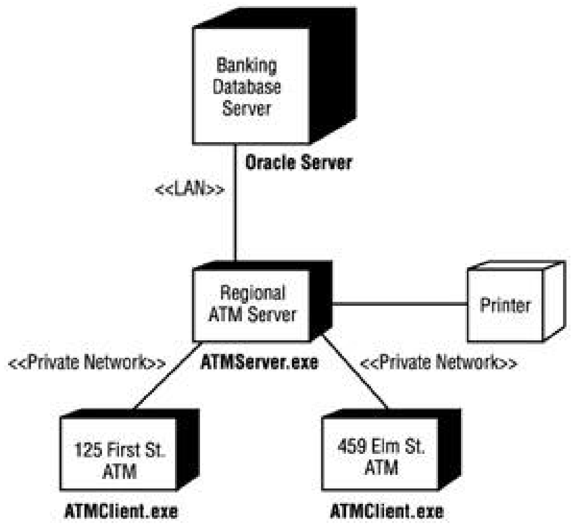
Component diagram for the ATM server

As this example shows, there can be multiple Component diagrams for a system, depending on the number of subsystems or executables. Each subsystem is a package of components. In general, packages are collections of objects. In this case, packages are collections of components. The ATM example includes two packages: the ATM client and the ATM server.

Component diagrams are used by whoever is responsible for compiling the system. The diagrams will tell this individual in what order the components need to be compiled. The diagrams will also show what run-time components will be created as a result of the compilation. Component diagrams show the mapping of classes to implementation components. These diagrams are also where code generation is initiated.

3.3.9. Deployment Diagrams

Deployment diagrams are the last type of diagram we will discuss. The Deployment diagram shows the physical layout of the network and where the various components will reside. In our ATM example, the ATM system comprises many subsystems running on separate physical devices, or nodes.



Deployment diagram for the ATM system

This Deployment diagram tells us much about the layout of the system. The ATM client executable will run on multiple ATMs located at different sites. The ATM client will communicate over a private network with the regional ATM server. The ATM server executable will run on the regional ATM server. The regional ATM server will, in turn, communicate over the local area network (LAN) with the banking database server running Oracle. Lastly, a printer is connected to the regional ATM server.

So, this one diagram shows us the physical setup for the system. Our ATM system will be following a three-tier architecture with one tier each for the database, regional server, and client.

The Deployment diagram is used by the project manager, users, architect, and deployment staff to understand the physical layout of the system and where the various subsystems will reside. This diagram helps the project manager communicate what the system will be like to the users. It also helps the staff responsible for deployment to plan their deployment efforts.

Sources:

- Booch, Rumbaugh and Jacobson, The Unified Modeling Language User Guide
- Boggs & Boggs, UML with Rational Rose 2002