Introduction to UML 2.0

Terry Quatrani
UML Evangelist
Session Objectives

- After completing this session, you should be able to:
  - Identify the different UML diagrams
  - Describe the purpose of each diagram
  - State where diagrams can be used
Agenda

- The Importance of Modeling
- The Unified Modeling Language
- Model-Driven Architecture
- UML Diagrams
- Extending the UML
Agenda

- The Importance of Modeling
- The Unified Modeling Language
- Model-Driven Architecture
- UML Diagrams
- Extending the UML
The Importance of Modeling
Why do we model?

- To manage complexity
- To detect errors and omissions early in the lifecycle
- To communicate with stakeholders
- To understand requirements
- To drive implementation
- To understand the impact of change
- To ensure that resources are deployed efficiently
Agenda

- The Importance of Modeling
- The Unified Modeling Language
- Model-Driven Architecture
- UML Diagrams
- Extending the UML
The Unified Modeling Language

- The UML is the standard language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.
Blobs with writing in their hair
And small adornments in the air
And has-relations everywhere
I've looked at clouds that way.
But now I've purged them from my Sun
Eliminated every one
So many things I would have done
To drive the clouds away.
I've looked at clouds from both sides now
Both in and out, and still somehow
It's clouds' delusions I appall
I really can't stand clouds at all...
Balls for multiplicity
Black and white for clarity
And data flows arranged in trees
Were part of OMT.
But now I've had to let them go
We'll do it differently, you know
'Cause Grady said, they've got to go
We can't use OMT.
Approach to Evolving UML 2.0

- Evolutionary rather than revolutionary
- Improved precision of the infrastructure
- Small number of new features
- New feature selection criteria
  - Required for supporting large industrial-scale applications
  - Non-intrusive on UML 1.x users (and tool builders)
- Backward compatibility with 1.x
Formal RFP Requirements

- **Infrastructure – UML internals**
  - More precise conceptual base for better MDA support
  - MOF-UML alignment

- **Superstructure – User-level features**
  - New capabilities for large-scale software systems
  - Consolidation of existing features

- **OCL – Constraint language**
  - Full conceptual alignment with UML

- **Diagram interchange standard**
  - For exchanging graphic information (model diagrams)
Infrastructure Requirements

- Precise MOF alignment
  - Fully shared “common core” metamodel
- Refine the semantic foundations of UML (the UML metamodel)
  - Improve precision
  - Harmonize conceptual foundations and eliminate semantic overlaps
  - Provide clearer and more complete definition of instance semantics (static and dynamic)
OCL Requirements

- Define an OCL metamodel and align it with the UML metamodel
  - OCL navigates through class and object diagrams ⇒ must share a common definition of Class, Association, Multiplicity, etc.

- New modeling features available to general UML users
  - Beyond constraints
  - General-purpose query language
Diagram Interchange Requirements

- Ability to exchange graphical information between tools
  - Currently only non-graphical information is preserved during model interchange
  - Diagrams and contents (size and relative position of diagram elements, etc.)
Superstructure Requirements (1 of 2)

- More direct support for architectural modeling
  - Based on existing architectural description languages (UML-RT, ACME, SDL, etc.)
  - Reusable interaction specifications (UML-RT protocols)
- Behavior harmonization
  - Generalized notion of behavior and causality
  - Support choice of formalisms for specifying behavior
- Hierarchical interactions modeling
- Better support for component-based development
- More sophisticated activity graph modeling
  - To better support business process modeling
Superstructure Requirements (2 of 2)

- New statechart capabilities
  - Better modularity
- Clarification of semantics for key relationship types
  - Association, generalization, realization, etc.
- Remove unused and ill-defined modeling concepts
- Clearer mapping of notation to metamodel
- Backward compatibility
  - Support 1.x style of usage
  - New features only if required
Language Architecture

- A core language + a set of optional “language units”
  - Some language units have multiple “levels”

MOF | Profiles | OCL
---|---|---
State Machines | Structured Classes and Components | Activities
Level 1 | Level 2 | Level 3

Basic UML
(Classes, Basic behavior, Internal structure, Use cases...)

UML Infrastructure

Multiple levels of compliance
The UML 2.0 Specification

Agenda

- The Importance of Modeling
- The Unified Modeling Language
- Model-Driven Architecture
- UML Diagrams
- Extending the UML
Model-Driven Architecture (MDA)

- An OMG initiative to support model-driven development through a series of open standards

(1) ABSTRACTION       (2) AUTOMATION

MDA

(3) OPEN STANDARDS
- Modeling languages
- Interchange facilities
- Model transformations
- Software processes
- etc.
Set of modeling languages for specific purposes

The Languages of MDA

- MetaObject Facility (MOF)
  - MOF “core”
  - A modeling language for defining modeling languages

- General Standard UML
  - General-purpose modeling language

- Common Warehouse Metamodel (CWM)
  - For exchanging information about business and data

- Real-Time profile
- EAI profile
- Software process profile
- etc.

- UML “bootstrap”
MOF (Metamodel) Example

- Uses (mostly) class diagram concepts to define

```mermaid
graph TD
  Element[0..1 /owner
         * /ownedElement
  Relationship
  DirectedRelationship
  Element[1..* /source
           * /target
           1..*]
  Element
  Comment
```
Agenda

- The Importance of Modeling
- The Unified Modeling Language
- Model-Driven Architecture
- UML Diagrams
- Extending the UML
Meta-models and Notations
Metamodel Description of Objects
Metamodel Structure

- Different Behavior Formalisms
- Structure-Behavior Dependency
- Shared Behavior Semantics

Diagram:
- Classes
- CommonBehaviors
- Activities
- Interactions
- StateMachines
- UseCases
- Actions
Diagram Elements

- Each diagram has a frame, a content area and a heading
- The frame is a rectangle and is used to denote a border
  - Frame is optional
- The heading is a string contained in a name tag which is a rectangle with cut off corners in the upper left hand corner of the frame
  - Format \([\text{kind}] \ [\text{name}] \ [\text{parameters}]\)
  - \(\text{kind}\) can be activity, class, component, interaction, package, state machine, use case
Frame Example

Package P

Class 1

Class 2
UML 2.0 Diagrams
UML 2.0 Diagrams
Activities

- Significantly enriched in UML 2.0 (relative to UML 1.x activities)
  - More flexible semantics for greater modeling power (e.g., rich concurrency model based on Petri Nets)
  - Many new features
- Major influences for UML 2.0 activity semantics
  - Business Process Execution Language for Web Services (BPEL4WS) – a de facto standard supported by key industry players (Microsoft, IBM, etc.)
  - Functional modeling from the systems engineering community (INCOSE)
Activity Diagram

- **Activity diagrams** show flow of control and data flow
- Typically used to model
  - Business process workflow
  - Flow within a use case
  - Business rules logic
  - Functional processes
  - UI screen flows
Flight Selection

«post condition» Flight selected

Enter Departure Airport

Departure Airport

Lookup City

List of Alternatives

Select Flight

Cancel Request

Cancel Selection

[found > 1 flight]

[found 1 flight]

[found 0 flights]
### Partitioning capabilities

<table>
<thead>
<tr>
<th>Company</th>
<th>Accounting</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reno</td>
<td>Invoice</td>
<td>Make payment</td>
</tr>
<tr>
<td></td>
<td>Send invoice</td>
<td></td>
</tr>
</tbody>
</table>

- **Receive order**
- **Fill order**
- **Ship order**
- **Close order**
Partitioning capabilities

- **Receive order**
- **Fill order**
- **Ship order**
- **Close order**
- **Send invoice**
- **Accept payment**
- **Make payment**
- **Invoice**
Activities: Basic Notational Elements

- **Control/Data Flow**
- **Activity or Action**
- **Object Node (may include state)**
- **Pin (Object)**
- **Choice**
- **(Simple) Join**

- **Control Fork**
- **Control Join**
- **Initial Node**
- **Activity Final**
- **Flow Final**
Extended Concurrency Model

- Fully independent concurrent streams ("tokens")

“Tokens” represent individual execution threads (executions of activities)

NB: Not part of the notation

Valid output streams:
- ABCXYZ
- AXYBCZ
- ABXCYZ
- AXBCYZ
- ABXYCZ
- AXBCYZ
Activities: Token Queuing Capabilities

- Tokens can
  - queue up in “in/out” pins
  - backup in network
  - prevent upstream behaviors from taking new inputs

- …or, they can flow through continuously
  - taken as input while behavior is executing
  - given as output while behavior is executing
  - identified by a {stream} adornment on a pin or object node
Actions in UML

- Action = fundamental unit of behavior
  - for modeling fine-grained behavior
  - Level of traditional programming languages

- UML defines:
  - A set of action types
  - A semantics for those actions
    - i.e. what happens when the actions are executed
    - Pre- and post-condition specifications (using OCL)
  - No concrete syntax for individual kinds of actions (notation)
    - Flexibility: can be realized using different concrete languages

- In UML 2, the metamodel of actions was consolidated
  - Shared semantics between actions and activities (Basic Actions)
Object Behavior Basics

- Support for multiple computational paradigms
Categories of Actions

- Communication actions (send, call, receive,…)
- Primitive function action
- Object actions (create, destroy, reclassify,start,…)
- Structural feature actions (read, write, clear,…)
- Link actions (create, destroy, read, write,…)
- Variable actions (read, write, clear,…)
- Exception action (raise)
UML 2.0 Diagrams

- Use Case Diagrams
- Interaction Diagrams
- Activity Diagrams
- Class Diagrams
- Deployment Diagrams
- Composite Structure Diagrams
- Component Diagrams
- State Machine Diagrams

Unified Modeling Language (UML)
Use Case Diagram

- **Use case diagrams** are created to visualize the relationships between actors and use cases.
- Use cases are a visualization of the functional requirements of a system.

![Use Case Diagram](image)
Actors

- An **actor** is someone or some thing that must interact with the system under development

[Diagram showing two stick figures labeled 'Passenger' and 'Bank']
Use Cases

- A **use case** is a pattern of behavior the system exhibits
  - Each use case is a sequence of related transactions performed by an actor and the system in a dialogue
- Actors are examined to determine their needs
  - Passenger – Search for Flights, Make Reservation, Pay for Flight
  - Bank -- receive payment information from Pay for Flight

Circle diagrams:
- Search for Flights
- Make Reservation
- Pay for Flight
Documenting Use Cases

- A **use case specification** document is created for each use cases
  - Written from an actor point of view
- Details what the system must provide to the actor when the use cases is executed
- Typical contents
  - How the use case starts and ends
  - Normal flow of events
  - Alternate flow of events
  - Exceptional flow of events
Use Case Diagram

Flight Reservation System

- Search for flights
- Make reservation
  - Extension points
    - Seat Selection
  - Condition: {seat Selection available on flight}
  - Extension point: Seat Selection
- Pay for Flight
- Select seat

Passenger

Bank
UML 2.0 Changes

- No notational changes for use case diagrams in UML 2.0
UML 2.0 Diagrams

- Use Case Diagrams
- Interaction Diagrams
- Class Diagrams
- Composite Structure Diagrams
- Component Diagrams
- State Machine Diagrams
- Activity Diagrams
- Deployment Diagrams
Interaction Diagrams

- **Interaction diagrams** show the communication behavior between parts of the system

- Four types of diagrams
  - Sequence diagram
    - Emphasis on the sequence of communications between parts
  - Communication diagram
    - Emphasis on structure and the communication paths between parts
  - Timing diagram
    - Emphasis on change in state over time
  - Interaction overview diagram
    - Emphasis on flow of control between interactions
Sequence Diagram

Make reservation → Enter PIN → verify PIN(number) → valid → Ask for destination → Destination

reservation system → reservation manager

destination

Show flights (destination)
Framed Sequence Diagram

SD Make Reservation

- Make reservation
- Enter PIN
- Ask for destination
- Destination

: Customer

reservation system

reservation manager

destination

verify PIN(number)

valid

Show flights(destination)
Composite Diagrams

SD Make Reservation

- Make reservation
- Enter PIN
- Ask for destination
- Destination
- Show flights (destination)

Check PIN

- Enter PIN
- verify PIN(number)
- valid

SD Check PIN

- Customer
- reservation system
- reservation manager
- destination

- Customer
- reservation system
- reservation manager
- destination
- valid
Combined Fragment Types

- **Alternatives (alt)**
  - choice of behaviors – at most one will execute
  - depends on the value of the guard ("else" guard supported)

- **Option (opt)**
  - Special case of alternative

- **Loop (loop)**
  - Optional guard: [<min>, <max>, <Boolean-expression>]
  - No guard means no specified limit

- **Break (break)**
  - Represents an alternative that is executed instead of the remainder of the fragment (like a break in a loop)
Combined Fragment Types

- Parallel (par)
  - Concurrent (interleaved) sub-scenarios
- Negative (neg)
  - Identifies sequences that must not occur
- Assertion (assert)
  - This must happen
- Critical Region (region)
  - Traces cannot be interleaved with events on any of the participating lifelines
Combined Fragments Diagram

SD Make Reservation

client: atm: dbase:

insertCard

ref

CheckPin

alt

[chk= OK]

ref

DoTransaction

[else]

error(badPIN)
**Communication Diagram Example**

```
sd Reserve Flights

:Web Client

1 approve flights
1.1 approve flights
1.2 confirm flights
1.3 confirm flights

2 pay
2.1 receipt

:Web Server

↓ 1.1 approve flights

:Flight DB
```
Timing Diagram Example

- sd Reader
  - Reading
  - Idle
  - Uninitialized

- r : Reader
  - Initialize
  - Read
  - ReadDone

- t1

- {d..d+0.5}
- {t1..t1+0.1}
Interaction Overview Example

- Like flow charts
  - Use activity graph notation for control constructs
- Better overview of complex interactions
  - Alternatives, options etc.
- Multiple diagram types could be included/referenced
UML 2.0 Changes

- Interaction occurrences and combined fragments added
- Communication diagrams created to replace collaboration diagrams used to show interactions
- New diagrams: Timing Diagram and Interaction Overview Diagram
UML 2.0 Diagrams

- Activity Diagrams
- Use Case Diagrams
- Deployment Diagrams
- Component Diagrams
- Interaction Diagrams
- State Machine Diagrams
- Class Diagrams
- Composite Structure Diagrams
Class Diagram

- **Class diagrams** show static structure
  - This is the diagram that is used to generate code
Classes

- A **class** defines a set of objects with common structure, common behavior, common relationships and common semantics.
- Classes can be “discovered” by examining the objects in sequence and collaboration diagram.
- A class is drawn as a rectangle with three compartments.
- Classes should be named using the vocabulary of the domain.
  - Naming standards should be created.
  - e.g., all classes are singular nouns starting with a capital letter.
Classes

- Mailer
- «interface» I Person
  - PassengerInterface
  - Passenger
  - CorporatePassenger
- ReservationManager
- Reservation
- Seat
- Flight
- Flight
Operations

- The behavior of a class is represented by its **operations**
- Operations may be found by examining interaction diagrams

```
:ReservationManager

:Reservation

Assign seat row and position

Reservation

assignSeat(row, position)
```
Attributes

- The structure of a class is represented by its attributes.
- Attributes may be found by examining class definitions, the problem requirements, business rules and by applying domain knowledge.

The name, address and phone number for each person is needed before a reservation can be made.

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>address</td>
</tr>
<tr>
<td>phoneNumber</td>
</tr>
</tbody>
</table>
Classes with Operations and Attributes

- **Mailer**
  - `getName()`
  - `getAddress()`

- **ReservationManager**

- **IPerson**
  - `getName()`
  - `getAddress()`

- **Passenger**
  - `name`
  - `address`

- **Reservation**
  - `addItem()`
  - `assignSeat()`
  - `deleteSeat()`

- **CorporatePassenger**
  - `discount`

- **Flight**
  - `airline`
  - `flightNumber`
  - `update()`

- **Seat**
  - `row`
  - `position`
Relationships

- Relationships provide a pathway for communication between objects.
- Sequence and/or communication diagrams are examined to determine what links between objects need to exist to accomplish the behavior -- if two objects need to “talk” there must be a link between them.
- Relationship types
  - Association
  - Aggregation
  - Composition
  - Dependency
Relationships

- An **association** is a bi-directional connection between classes.

- An **aggregation** is a stronger form of association where the relationship is between a whole and its parts.

- A **composition** is a stronger form of aggregation where the part is contained in at most one whole and the whole is responsible for the creation of its parts.

- A **dependency** is a weaker form of relationship showing a relationship between a client and a supplier where the client does not have semantic knowledge of the supplier.
Finding Relationships

- Relationships are discovered by examining interaction diagrams
  - If two objects must “talk” there must be a pathway for communication

:ReservationManager :Reservation

Assign seat row and position
Multiplicity and Navigation

- **Multiplicity** defines how many objects participate in a relationship
  - Multiplicity is the number of instances of one class related to ONE instance of the other class
  - For each association and aggregation, there are two multiplicity decisions to make: one for each end of the relationship

- Although associations and aggregations are bi-directional by default, it is often desirable to restrict navigation to one direction
  - If navigation is restricted, an arrowhead is added to indicate the direction of the navigation
Multiplicity
Navigation

```
-Mailer

 «interface»
 IPerson

 getName ( )
 getAddress ( )

-Passenger

 name
 address

 PassengerInterface

 getName ( )
 getAddress ( )

 ReservationManager

 1

 Reservation

 0..*

 addItem ( )
 assignSeat ( )
 deleteSeat ( )

 PassengerInterface

 0..*

ReservationManager

 1

 Seat

 1

 Flight

 update( )

 CorporatePassenger

 discount

 Seat

 row
 position

 CorporatePassenger

 discount

 Flight

 airline
 flightNumber

 CorporatePassenger

 discount

 Flight

 airline
 flightNumber

 CorporatePassenger

 discount

 Flight

 airline
 flightNumber
```
Inheritance

- **Inheritance** is a relationships between a superclass and its subclasses
- There are two ways to find inheritance:
  - Generalization
  - Specialization
- Common attributes, operations, and/or relationships are shown at the highest applicable level in the hierarchy
Inheritance

- Mailer
- Passenger
  - name
  - address
- CorporatePassenger
  - discount
- PassengerInterface
  - getName ( )
  - getAddress ( )
- ReservationManager
- Reservation
  - addItem ( )
  - assignSeat ( )
  - deleteSeat ( )
- Seat
  - row
  - position
- Flight
  - airline
  - flightNumber
  - update ( )
- «interface» IPerson
  - getName ( )
  - getAddress ( )
Realization

- **Realization** is a relationship between a specification and its implementation
Realization
PassengerInterface is the *implementation* of the IPerson interface. Mailer *uses* the IPerson interface.
Realization

- Mailer
- PassengerInterface
  - getName()
  - getAddress()
- IPerson
- ReservationManager
- Reservation
  - addItem()
  - assignSeat()
  - deleteSeat()
- Passenger
  - name
  - address
- CorporatePassenger
  - discount
- Seat
  - row
  - position
- Flight
  - airline
  - flightNumber
  - update()
UML 2.0 Changes

- Notation for a required interfaces (___C___) added
UML 2.0 Diagrams
Composite Structure Diagrams

- **Composite Structure diagrams** show the internal structure of a classifier and its interaction points to other parts of the system.
  - Instance view of the world
- They show how the contained parts work together to supply the behavior of the container.

![Composite Structure Diagram](image)
Aren’t Class Diagrams Sufficient?

- No!
  - Because they abstract out certain specifics, class diagrams are not suitable for performance analysis
- Need to model structure at the instance/role level

Same class diagram describes both systems!
Collaborations

- In UML 2.0 a collaboration is a purely structural concept
  - More general than an instance model
Roles and Instances

- Specific object instances playing specific the roles in a collaboration

```
Roles:
- c1/client : Client
- a/arbiter : Arb
- d1/primary : Dbase
- d2/backup1 : Dbase
- d2/backup2 : Dbase
```
Structured Class

- A complex class comprised of internal “parts”
- Desired structure is asserted rather than constructed
  - Class constructor automatically creates desired structures
  - Class destructor automatically cleans up
Ports

- Interaction points
- Each port is dedicated to a specific purpose and presents the interface appropriate to that purpose
Port Semantics

- A port can support multiple interface specifications
  - Provided interfaces (what the object can do) - incoming
  - Required interfaces (what the object needs to do its job) - outgoing
Ports: Alternative Notation

- Shorthand “lollipop” notation with 1.x backward compatibility

- Diagram showing a diagram of a system with the following ports:
  - **Provided Interface**
    - buyTicket
  - **Required Interface**
    - sellTicket

- The system is labeled **WebClient**.
Assembling Communicating Objects

- Ports can be joined by connectors to model communication channels
  - At runtime, the WebClient is linked to the TicketServer

A connector is constrained by a protocol
Static typing rules apply (compatible protocols)
Protocols: Reusable Interaction Sequences

- Communication sequences that
  - Conform to a pre-defined dynamic order
  - Are defined generically in terms of role players
  - E.g., ticket purchase protocol

![Diagram showing loop, request, propose, confirm, and confirm interactions between WebClient and TicketServer]
Modeling Protocols with UML 2.0

- A collaboration structure with interactions

**Ticket Purchase Protocol**

- «interface» buyTicket
  - request() Confirm()

- «interface» sellTicket
  - Propose() Confirm()

- «interface» buyTicket
  - request() Confirm()

- «interface» sellTicket
  - propose confirm

- Loop
  - request propose confirm

- WebClient

- TicketServer

- customer

- agency
Structured Classes: Putting Them Together

OnlineAgency

sd
ReserveFlights

sd
SelectFlights

Customer: WebClient

Agency: TicketServer
Structured Classes: Putting Them Together

```
OnlineAgency
  sd ReserveFlights
  sd SelectFlights

Customer: WebClient
  Agency: TicketServer

AirlineTicketReservation
  sd
  sd
```

UML 2.0 Changes

- Composite structure diagrams, structured classes, ports and connectors are new
UML 2.0 Diagrams

- Use Case Diagrams
- Interaction Diagrams
- Class Diagrams
- Composite Structure Diagrams
- Activity Diagrams
- Deployment Diagrams
- Component Diagrams
- State Machine Diagrams
State Machine Diagram

- A **state machine diagram** shows
  - The life history of a given class
  - The events that cause a transition from one state to another
  - The actions that result from a state change
- State machine diagrams are created for objects with significant dynamic behavior
State Diagram

state machine Make Payment

Enter credit card data

OK

Do transaction: Payment Transaction

Aborted

Display Confirmation Page

Done

Display Cancellation Page
Specialization

- Redefinition as part of standard class specialization

ATM
- acceptCard()
- outOfService()
- amount()

FlexibleATM
- otherAmount()
- rejectTransaction()

Statemachine

<<Redefine>>
Example: State Machine Redefinition

ATM

- VerifyCard \{final\}
  - acceptCard
  - VerifyTransaction \{final\}
    - releaseCard
  - ReadAmount
    - selectAmount
      - amount
      - outOfService \{final\}
        - outOfService
        - selectAmount
UML 2.0 Changes

- Protocol state machines added
UML 2.0 Diagrams

- Use Case Diagrams
- Interaction Diagrams
- Class Diagrams
- Composite Structure Diagrams
- Deployment Diagrams
- Component Diagrams
- State Machine Diagrams
- Activity Diagrams
Component Diagram

- A **component** is a modular unit with well defined interfaces that is replaceable within its environment.
- Components can be logical or physical.
- Logical components
  - Business components, process components etc.
- Physical components
  - EJB components, .NET components, COM components etc.
Component Diagram

<<component>>

:Ticket

<<component>>

:Reservation

IPerson

<<component>>

:Passenger
UML 2.0 Changes

- Notation for a component changed
- Component may have ports
UML 2.0 Diagrams

- Use Case Diagrams
- Interaction Diagrams
- Class Diagrams
- Composite Structure Diagrams
- Component Diagrams
- State Machine Diagrams
- Activity Diagrams
- Deployment Diagrams
Deployment Diagram

- **Deployment diagrams** show the execution architecture of systems
- Nodes are connected by communication paths to create network systems

![Deployment Diagram](diagram)
Artifacts

- An **artifact** represents a physical entity
- Examples
  - Model files, source files, scripts, binary executable files

```
<<artifact>>

Ticket.jar

<<manifest>>

<<component>>

:Ticket
```
UML 2.0 Changes

- Artifacts and deployment specifications are new
- New Node types introduced
  - Execution environment (virtual machine, app server, and other “middleware”)
Agenda

- The Importance of Modeling
- The Unified Modeling Language
- UML Diagrams
- Extending the UML
Extending the UML

- In order to model something effectively, the language that you use to model it must be rich and expressive enough to do so.
- “Out of the box” UML is sufficient for modeling object-oriented software systems.
- BUT… there are many more models that are useful in the software development process.
- UML can easily be extended to add more semantics to cover other modeling situations.
  - Database models
  - Business process
  - Web pages
  - On and on….
Extension Mechanisms

- Stereotypes
- Tag definitions and tagged values
- Constraints
Stereotypes

- A more refined semantic interpretation for a model element
Stereotype Examples
Deployment Diagram
Tag Definitions and Tagged Values

- **A tag definition** the ability to associate extra information with a modeling element
  - Defines the name and the type of information
- **Example**
  - A database table could be modeled as a stereotyped class
  - Columns could be modeled as stereotyped attributes
  - Tag definition for a column could be “NullsAllowed”
- **A tagged value** is the actual instance of a tag definition with a value that is associated to the modeling element
- **Example**
  - Create a table Person
  - Add a column, you MUST supply a value for “NullsAllowed” (true or false)
Constraints

- A **constraint** is a rule that is applied to a modeling element
  - Represented by curly braces in UML
- Used to evaluate if a modeling element is “well-formed”
- Example
  - The name of a column cannot exceed the maximum length for column names for the associated database
- The language of constraints is Object Constraint Language (OCL)
OCL

- OCL is another OMG specification
- Defines an object-oriented language that is similar to Smalltalk, Java and C++
- Formal way to express model "well formedness" rules

{let x = self.model().OwnedProfile->any(Name='Data Modeler').TaggedValueSet->any(Name='Preferences').TagDefinition->any(Name='maximumidentifierlength').DefaultValue.oclAsType(UML::IntegerTaggedValue).Value in self.Name.size() <= x}
Profiles

- A **profile** is a collection of stereotypes, tag definitions and constraints that work together to define new semantics for a model.

- Example
  - Data modeling profile
  - Business modeling profile
  - EJB profile
EJB Profile
Specializing UML

- Lightweight extensions
  - Extend semantics of existing UML concepts by specialization
  - Conform to standard UML (tool compatibility)
  - Profiles, stereotypes

- Heavyweight (MOF) extensions
  - Add new non-conformant concepts or
  - Incompatible change to existing UML semantics/concepts
The Profile-Based Approach to DSLs

- Profile = a compatible specialization of an existing modeling language by
  - Adding constraints, characteristics, new semantics to existing language constructs
  - Hiding unused language constructs

- Advantages:
  - Supported by the same tools that support the base language
  - Reuse of base language knowledge, experience, artifacts

- Example: ITU-T standard language SDL (Z.100)
  - Modeling language used in telecom applications
  - Now defined as a UML profile (Z.109)
UML Profile Example

- Defining a custom «clock» stereotype

Semantics: changes value synchronously with the progress of physical time

{has exactly one ClockValue Attribute}

{type = Integer, value >= 0}
Profiles: Notation

- E.g., specializing the standard Component concept

```
«profile» TimingDevices

«metaclass»
Class

Extension

«stereotype»
Clock

«stereotype»
Timer

«stereotype»
ToDclock
```
Templates

- More precise model than UML 1.x
- Limited to Classifiers, Packages, and Operations
Collaboration Templates

- Useful for capturing design patterns
  
  ObserverPattern

  subject : sType

  observer : oType

  oType, sType

  Collaboration template

  DeviceObserver

  ObserverPattern <oType->DevicePoller, sType>Device>

  «bind»
Package Templates

- Based on simple string substitution

```plaintext
CustomerAccountTemplate

$<customer>$

owner
1..*

$<kind>Acct$
0..*

SavingsBank

Person

owner
1..*

Personal

PersonalAcct
0..*

<bind> customer -> Person,
kind -> Personal

customer : StringExpression
kind : StringExpression

Name Expression

Package Templates

Based on simple string substitution

CustomerAccountTemplate

$<customer>$

owner
1..*

$<kind>Acct$
0..*

SavingsBank

Person

owner
1..*

Personal

PersonalAcct
0..*

<bind> customer -> Person,
kind -> Personal

customer : StringExpression
kind : StringExpression

Name Expression
```
Summary: UML 2.0 Highlights

- Greatly increased level of precision to better support MDD
  - More precise definition of concepts and their relationships
  - Extended and refined definition of semantics
- Improved language organization
  - Modularized structure
  - Simplified compliance model for easier interworking
- Improved support for modeling large-scale software systems
  - Modeling of complex software structures (architectural description language)
  - Modeling of complex end-to-end behavior
  - Modeling of distributed, concurrent process flows (e.g., business processes, complex signal processing flows)
- Improved support for defining domain-specific languages (DSLs)
- Consolidation and rationalization of existing concepts
Session Summary

- Now that you have completed this session, you should be able to:
  - Identify the different UML diagrams
  - Describe the purpose of each diagram
  - Use the different diagram and model elements
Questions
Thank You