Introduction to UML

A Practical Approach to Product Development

Description

- This seminar is designed to help you get started using the Unified Modeling Language (UML) as a visual design tool using a development process.
- A simple case study is presented to demonstrate domain modeling and design techniques as an integrated method.
- Familiarity with object technology will enhance the attendee’s understanding of the models, but is not required.
Speakers

♦ Jim Stafford
  ➢ DoD Contracting
  ➢ OO Mentoring
  ➢ UCA Framework System Engineering
  ➢ Distributed Computing Infrastructure
  ➢ RABA Technologies, Inc
  ➢ http://www.raba.com/~jcstaff
  ➢ JHU/APL OOP/C++ Instructor
  ➢ jcstaff@raba.com

♦ Michele Moore
  ➢ DoD Contracting
  ➢ OO Mentoring
  ➢ Software Development
  ➢ RABA Technologies, Inc
  ➢ JHU/SCS OOP, C++, Java instructor
  ➢ michele.moore@raba.com

Objectives

♦ Introduce UML in the context of a practical object-oriented work process
♦ Introduce a proven object-oriented process
♦ Understand how UML models fit into the process
♦ Discover how to create UML models
Agenda

♦ **(Day 1) Process and UML: Introduction**
  ➢ UML Background
  ➢ Getting Started with Object Modeling
  ➢ Modeling At A Glance with UML
  ➢ Process Overview

♦ **(Day 2) Process and UML: Insertion**
  ➢ Process and UML Applied
  ➢ Case Study: *Answering Machine*
  ➢ Other Resources

UML Background

♦ History of UML
♦ The “Methodology Wars”
♦ What UML is
♦ What UML is not
History of UML

- Design methods popularized in 70’s & 80’s
- Technical community inundated with models, methodologies, notations by early 90’s
- Standardization was needed, but no one was willing to champion the cause and make it successful; many were opposed to the idea
- OOPSLA ‘94 Grady Booch and James Rumbaugh announced the merging of their methods
- OOPSLA ‘95 revealed the first public description of the Unified Method, with Ivar Jacobson joining the duo
- During 1996, the “Three Amigos” worked on the new method, renaming it as the Unified Modeling Language (UML)
- In January ‘97, UML was proposed to OMG as a standard to facilitate the interchange of models; UML 1.1 adopted by OMG in Nov ‘97

Methodology Wars of the 90’s

- Rational
  Booch Method - Booch
- GE ACC
  OMT - Rumbaugh, Blaha, Premerlani, Eddy, Lorensen
- Objectory
  OOSE - Jacobson
- Object International
  Coad/Yourdon - Coad
- Others...
### Methodologist Terminology

<table>
<thead>
<tr>
<th>Methodology</th>
<th>UML</th>
<th>Booch</th>
<th>Coad</th>
<th>Jacobson</th>
<th>Odell</th>
<th>Rumbaugh</th>
<th>Shlaer/Mellor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class</td>
<td>Class</td>
<td>Class &amp; Object</td>
<td>Object</td>
<td>Object Type</td>
<td>Class</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Association</td>
<td>Uses</td>
<td>Instance Connection</td>
<td>Acquaintance Association</td>
<td>Relationship</td>
<td>Association</td>
<td>Relationship</td>
</tr>
<tr>
<td></td>
<td>Generalization</td>
<td>Inherits</td>
<td>Gen-Spec</td>
<td>Inherits</td>
<td>Subtype</td>
<td>Generalization</td>
<td>Subtype</td>
</tr>
<tr>
<td></td>
<td>Aggregation</td>
<td>Containing</td>
<td>Part-Whole</td>
<td>Consists of</td>
<td>Composition</td>
<td>Aggregation</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### 1995 Unified Method from Rational

- Rational
- Unified Method - Booch, Rumbaugh
- Objectory
- OOSE - Jacobson
- Object International
- Coad/Yourdon - Coad
- Others...
What UML Is

- UML is a modeling language, a notation used to express and document designs
- UML unifies the notation of Booch, Rumbaugh (OMT) and Jacobson, and augmented with other contributors once submitted to OMG
- UML proposes a standard for technical exchange of models and designs
- UML also defines a “meta-model”, a diagram that defines the syntax of the UML notation

What UML Is Not

- UML is not a method or methodology (Method = Notation (e.g., UML) + Process)
- UML does not dictate a particular process (although the “Rational Objectory Process” is being proposed by Booch, Rumbaugh and Jacobson)
- UML can be used to record the resulting domain and design models, independent of the process
- Choose an appropriate process for a particular project, independent of the modeling language
Why Use UML

- Standardized notation without sacrificing specialized model data
- Common language that can be used from product conception to delivery, from system to detailed design levels
- Reduced learning curve across projects
- Increased domain and design model reuse
- Increased customer involvement/understanding of problem translation to product solution

Where to Start?

- Identify key domain abstractions … **classes integrating:**
  > Attributes
  > Behavior (responsibilities, methods)
  > Messaging
    ♦ providing logical independence between client and object
  > Polymorphism
    ♦ providing physical independence between client and implementation
- Consider relationships … **integrating classes and objects to form higher levels of abstraction**
  > Association ("Uses, Needs")
  > Aggregation ("Has-A")
  > Inheritance ("Is-A")
- Begin constructing your models
Model Perspectives

♦ Conceptual
  ♦ Book [Title]
    ➢ objects, “things” from the domain
    ➢ conceptual map to implementation

♦ Specification
  ♦ BookIface { void setTitle(String value); }
    ➢ identifies how to obtain properties

♦ Implementation
  ♦ PersistentBook : BookIface { -> DB }
    ➢ identifies how interface will be implemented

Initial Modeling Results

♦ List of use cases, describing system requirements
♦ Domain model, capturing your understanding of the business process and key domain classes
♦ Design model, realizing both the information in the domain objects and the behavior described in the use cases
♦ Add classes in the design model that actually do the work and also provide a reusable architecture for future extensions
UML Notation Baseline

- Use Case Diagrams
- Class Diagrams
- Package Diagrams
- Activity Diagrams
- State-Transition Diagrams
- Event Trace (Interaction) Diagrams
  - Sequence Diagrams
  - Collaboration Diagrams
- Deployment Diagrams

Use Case Diagrams

- Show the external actors and their connection to the functionality (use cases) of the system
- Use cases provide the basis of communication between sponsors/customers and implementers in the planning of a project
  - Capture some user-visible function
  - May be small or large
  - Achieves a discrete goal for the user
Class Diagrams

- Show the static structure of the domain abstractions (classes) of the system
- Describe the types of objects in the system and the various kinds of static relationships that exist among them
  - Associations
  - Derivations
- Show the attributes and operations of a class and the constraints for the way objects collaborate

Package Diagrams

- Shows the breakdown of larger systems into a logical grouping of smaller subsystems (e.g., Coad/Yourdon’s Subject Layer)
- Shows groupings of classes and the dependencies among them
- A dependency exists between two elements if changes to the definition of one element may cause changes to the other
Activity Diagrams

- Show the sequential flow of activities
  - typically in an operation
  - also in a use case or event trace
- Complement the class diagram by showing the workflow of the business (aka “Flowchart”)
- Encourage discovery of parallel processes which helps eliminate unnecessary sequences in business processes

State-Transition Diagrams

- Show all the possible states that objects of the class can have and which events cause them to change
- Show how the object’s state changes as a result of events that are handled by the object
- Good to use when a class has complex lifecycle behavior
Interaction (Event Trace) Diagrams (1)

**Sequence Diagrams**
- Show the dynamic collaboration between objects for a sequence of messages sent between them in a sequence of time
- Time sequence is easier to see in the sequence diagram, read from top to bottom
- Choose sequence diagram when only the sequence of operations needs to be shown

![Sequence Diagram Example]

Interaction (Event Trace) Diagrams (2)

**Collaboration Diagrams**
- Show the actual objects and their links, the “network of objects” that are collaborating
- Time sequence is shown by numbering the message label of the links between objects
- Choose collaboration diagram when the objects and their links facilitate understanding the interaction, and sequence of time is not as important

![Collaboration Diagram Example]
Deployment Models

- Show the physical architecture of the hardware and software of the system
- Highlight the physical relationships among software and hardware components in the delivered system
- Components on the diagram typically represent physical modules of code and correspond exactly to the packages on a package diagram

Process Overview

- Conceptualization
- Requirements Definition
- Architecture Design
- Package Development
- Delivery
- Maintenance
Conceptualization

*entire system*

- Define Purpose and Prioritized Features
- “Sketch” Early Domain Class Model
- Identify Use Cases and Primary Paths
- Identify Schedule, Risks, Resources

Domain Class Model Example
Use Case Identification Using Activity Diagrams

- Take Caller Message
- Answer Phone Line
  - Play Greeting
    - Record Caller Message
    - Play to Output Device
      - [Call Stopped]
      - [Message]
      - [No Message]
    - Update Indicator

Use Case Identification Example

- Actors: External Caller, External Owner
- Use Case: Answer Caller
- Use Case: Review Caller Message
- Use Case: Record Greeting
- Use Case: Set Answer Model
- Use Case: Take Caller Message
- Use Case: Play Greeting
- Use Case: Delete Caller Message
Conceptualization Applied

♦ Consider a University Student Registration System
   ➢ Identify 5 key abstractions, attributes, responsibilities, relationships

(Eyes on your own paper! :)
Requirements Definition

*(per build OR entire system)*

- Define Domain Class Model
- Describe Use Cases, Primary & Alternate Paths
- Map Requirements to Classes & Use Cases/Use Case Paths

**Results**
- Documented in an Operational Concept Document (OCD) and Software Requirements Spec (SRS)
- Presented at Software Requirements Review (SRR)

---

Domain Class Model Example
Class Specification Example

Class: CallerMessage

Messages are recorded from the caller with associated information.
The caller messages are flagged as new until the owner reviews them and are only deleted upon specific request.

Attributes:
- Date Time - The call is recorded with the date/time at the start of the call.
- Caller - The caller is identified by the phone company.
- Reviewed - Any caller message that has been played is considered reviewed.
- Message - Caller message data must contain a message; empty messages are deleted

Behavior:
- Record - Caller messages are recorded from the phone line and are terminated when the caller hangs up, the maximum message length has been reached, or the owner terminates the recording. Empty messages will be ignored.
- Play - Caller messages are played to the speaker for the owner to listen to.
- Delete - The owner can delete specific messages.
- Stop - The play or recording of the caller messages can be stopped by the owner.

System Requirements Example

Caller Message

1. The Answering System shall provide the capability for an owner to review a caller message. (Req-2, Review Caller Messages)
2. The Answering System shall output the date and time of the caller message to the owner on playback. (Req-11, Review Caller Messages)
3. The Answering System shall output the identity of the caller of the caller message to the owner on playback. (Req-12, Review Caller Messages)
4. The Answering System shall provide the capability for an owner to review new caller messages only. (Req-14, Review Caller Messages)

...etc.
Requirements Definition Applied

♦ Identify 5 System Requirements for the University Student Registration System
♦ Describe them as specific use cases

Exercise Results
Use Case Diagram at a Glance

Use Case Specification Example

**Use Case: Answer Caller**
- This use case begins with a caller ringing the answering machine and completes after the call has been completed. This use case is meant to be extended by mode specific processing.
- **Actors:**
  - Caller
- **Pre-conditions:**
  - Answering machine is idle
- **Post-conditions:**
  - Greeting is played to caller
  - Mode-specific processing is performed

**Primary Path (Caller Listens to Greeting):**
1. The caller rings the phone line connected to the answering machine.
2. The system waits for the ring count to reach the answering ring count.
3. The system plays a greeting.
4. The system performs any mode-specific processing

**Alternate Path (Caller Hangs Up During Greeting):**
- The caller hangs up during the playing of the greeting. The system recognizes the hang-up, stops the greeting and prepares for the next caller.
Architecture Design

(whole system)

- Identify package boundaries and allocate primary responsibility of domain classes to a package
- Break complex use cases into uses relationships that fit within identified packages
- Identify interfaces for domain classes that cross package boundaries
- Define event traces for key interactions between packages
- Resolve Strategic Issues

Results
- Package interactions generated
- Define risk-based build schedule
- Documented in Software Design Description (SDD); presented at System PDR

Class Packages

July 1998 Introduction to UML Slide 43

July 1998 Introduction to UML Slide 44
Key Event Trace Example (1)

Answer Caller (Caller Listens to Greeting)

1. The caller rings the phone line connected to the phone line.
2. The phone line reports the event of a new call to the current answer mode.
3-4. The answer mode waits for the owner-specified.
5. The phone line is told to answer the caller.
6-8. The answer mode plays the greeting.
9. The specific answer mode will perform any mode-specific processing.
10. The phone line is hung up.

Use Case/Event Trace Text Comparison

**Use Case Script**
1. The caller rings the phone line connected to the answering machine.
2. The system waits for the ring count to reach the answering ring count.
3. The system plays a greeting.
4. The system performs any mode-specific processing.

**Event Trace Script**
1. The caller rings the phone line connected to the phone line.
2. The phone line reports the event of a new call to the current answer mode.
3-4. The answer mode waits for the owner-specified.
5. The phone line is told to answer the caller.
6-8. The answer mode plays the greeting.
9. The specific answer mode will perform any mode-specific processing.
Key Event Trace Example (2)

Take Caller Message (Caller Leaves Message)

- External Caller
- PhoneLine
- MessageMode
- Message
- Call
- Message
- Speaker
- Message
- Indicator

1. Answer Complete
2. CallerId
3. Create (caller, timeOfCall)
4. Record (InputDevice)
5. Listen
6. Play
7. Disconnect
8. Caller Message Complete
9. Record (message)
10. New Message

Package Requirements Example (1)

1. The Answering System shall provide the capability for an owner to review a caller message. (Req-2, Review Caller Messages)
   > The IODevices Package shall provide the capability to play audio to a speaker. (Pkg-4)
   > The Messages Package shall provide the capability to play caller messages to an output device. (Pkg-10)

2. The Answering System shall output the date and time of the caller message to the owner on playback. (Req-11, Review Caller Messages)
   > The Foundation Package shall provide the capability to obtain the time of day. (Pkg-1)
   > The Messages Package shall timestamp caller messages with the time of the call. (Pkg-13)
   > The Messages Package shall play the time of day when a caller message is played. (Pkg-19)
Package Requirements Example (2)

3. The Answering System shall output the identity of the caller of the caller message to the owner on playback. (Req-12, Review Caller Messages)
   - The IODEvices Package shall provide the capability to obtain the identity of the caller. (Pkg-8)
   - The Messages Package shall record the identity of the caller. (Pkg-14)
   - The Messages Package shall output the identity of the caller when a caller message is played. (Pkg-20)

4. The Answering System shall provide the capability for an owner to review new caller messages only. (Req-14, Review Caller Messages)
   - The Messages Package shall provide the capability for an owner to review new caller messages only. (Pkg-17)

Package Development

- Class Design (per customer build)
  - Identify internal design classes and their interfaces
  - Define Method Traces
  - Results
    - Documented in SDD
    - Presented at CDR
- Class Implementation (per internal build)
  - Implement classes
  - Unit & Integration Testing
- Package Integration
  - Software Integration Testing (SWIT)
Class Design Example

**InputDevice**
- Listen()
- Initialize()
- Audio()
- Terminate()

**OutputDevice**
- Listen()
- Initialize()
- Audio()
- Terminate()

**Observer**
(from Foundation)
- Notify()

**AudioSample**
- Play()
- Initialize()
- Terminate()

**PhoneLine**
- RingCount
- Hangup()
- Answer()
- Listen()
- Play()
- Ring()
- connect()
- callerId()
- initialize()

**MicrophoneImpl**

**SpeakerImpl**

**PhoneLineImpl**

July 1998
Introduction to UML
Slide 51

Method Trace Example

**Take Caller Message**

1: Ring
2: notify (int, void*)
3: newCall()
4: callerId()
5: CallerMessage (callerId, dateTime)
6: record (InputDevice&)
7: listen()
8: record (CallerMessage&)

External Caller

Audio

CallerMessage

MessageMode

AnsweringMachine

Mailbox

July 1998
Introduction to UML
Slide 52
Delivery

*(per build)*

- Integrate Changes
- System Testing
- Develop/Update User Documentation
- Field System
- Track Error Reports
- Deployment Diagrams

Maintenance

- Collect Change/Enhancement Requests
  - Prioritize
  - Group
- Enter the Process Spiral Again
Overall Process Mapping

Data Requirement

Use Case

Functional Requirement

Domain Class

Use Case Path

Concrete Use Case Scenario

Class Design

Event Trace

Concrete Analysis Scenario

Code

Method Trace

Concrete Design Scenario

Process Roles

- Requirements
  - Use Cases
  - Use Case Paths
  - Domain Classes
  - Event Traces
  - Class Designs
  - Message Traces
  - Code
  - Unit Test
  - Integration Test
  - Acceptance Testing

- Customer
  - Customer Representative
  - System Engineer

- Software Architect
  - Subsystem Lead

- Developer
  - Subsystem Lead
  - System Engineer
Session 2: Process Insertion

♦ Process Goals / Results
  ➢ Conceptualization
  ➢ Requirements Definition
  ➢ Architecture Design
  ➢ Package Development
  ➢ Delivery
  ➢ Maintenance

♦ UML In Action
  ➢ Case Study: Answering Machine System

Conceptualization/Requirements Definition Models

♦ Domain Class Model
♦ Use Case Models
♦ Activity Diagrams
♦ Requirements Management
Class Models

Conceptual Level

Class Model Overview

- Shows static properties of a Class
- Depicted graphically in Class Diagrams and textually in Class Specifications
- May be organized into Packages
- Class Diagrams
  > Show an aspect of a set of Classes
  > Need not contain entire Package
  > Need not be restricted to Package boundaries
**Packages**

- Grouping of model elements (including non-class models)
- Can be nested
- Lists classes (optional) contained within package
  - + Public Visibility
  - -/# Non-public Visibility

**Class Symbols**

- Solid rectangle
- Three compartments
  - Class Name and general properties (optional)
  - Package (Package::)
  - Stereotypes (<< >>)
  - Constraints ({} )
  - Meta-model attributes
  - Attributes (optional)
  - Operations (optional)
  - Additional compartments allowed
Class Descriptions

Class: CallerMessage

Caller Messages are recorded from the caller with associated information. The caller messages are flagged as new until the owner reviews them and are only deleted upon specific request.

Adding Class Properties

<table>
<thead>
<tr>
<th>CallerMessage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateTime</td>
</tr>
<tr>
<td>Caller</td>
</tr>
<tr>
<td>Reviewed</td>
</tr>
<tr>
<td>Message</td>
</tr>
<tr>
<td>Record()</td>
</tr>
<tr>
<td>Play()</td>
</tr>
<tr>
<td>Delete()</td>
</tr>
<tr>
<td>Stop()</td>
</tr>
</tbody>
</table>

♦ Attributes
➢ Describing data properties of a class

♦ Operations (Behaviors)
➢ Describing localized behavior of a class
Property Descriptions

Class: CallerMessage

♦ Attributes
  ► DateTime - The caller message is recorded with the date/time at the start of the call.

♦ Behaviors (Operations)
  ► Record() - Caller messages are recorded from the phone line and are terminated when the caller hangs up, the maximum message length has been reached, or the owner terminates the recording. Empty messages will be ignored.

Adding Relationships

Shown as an attribute

AnsweringMachine
Answer Modes [2]

Shown as an Association

AnsweringMachine
1

Answer Modes
1

AnswerMode
1

Greeting

July 1998
Introduction to UML
Slide 65

July 1998
Introduction to UML
Slide 66
**Associations**

- **Name (optional)**: further describes association between members
- **Roles**: role played by associated member
  - named
  - unnamed
- **Multiplicity**: quantity of associated member
  - unspecified
  - 1 - mandatory
  - 0..1 - optional
  - 0..* (or *) - many
  - # - specified value or range

---

**Reading Associations**

- **An Answer Mode is associated with “1” Answering Machine**
- **The related Answer Modes play the role of the Answering Machine’s “Answer Modes”**
- **The Greeting plays the role of a “Greeting”**

---

July 1998 Rev 1.0  Introduction to UML
Adding Properties to Associations

Use named/unnamed associations where there are no properties unique to the association between two classes

Use Attribute Class to add properties unique to the association between two classes

Adding Constraints on Associations

The caller will be identified as either a person or an organization
Adding Aggregation to Associations

Aggregation (open diamond)
- shows side of association that represents the aggregate

Composite (filled in diamond)
- A refinement of aggregation showing the aggregate having ownership of the member

Adding Uses Dependency Association

Uses Relationship
- dependency shown with <<uses>> stereotype
  - shows a usage relationship between associated classes
  - the association is temporary (created and forgotten during the completion of a behavior)
Association Descriptions

Class: CallerMessage
- Associations:
  - Mailbox [1] - Caller messages are assigned to a particular mailbox.

Class: Mailbox
- Associations:
  - CallerMessage [0..*] - A mailbox contains all caller messages until they are deleted.
- Uses:
  - OutputDevice - Caller Messages are played to the specified output device.

Inheritance
- Indicates type/sub-type
  - derived classes (sub-type) comply with base class (type) interface definitions
- Indicates class/sub-class
  - derived classes (sub-class) reuse or extend base class (class) implementation
**Inheritance Descriptions**

**Class: OutputDevice**
- **Behaviors:**
  - Play() - Output devices can play audio.

**Class: PhoneLine**
- **Inheritance:**
  - OutputDevice, InputDevice
- **Behaviors:**
  - Play() - Audio can be played to the phone line until the caller or system hangs up.
Example: Answering Machine Packages

AnsweringSystem
+ AnsweringMachine
+ AnswerMode
+ AnnounceMode
+ MessageMode

IODevices
+ InputDevice
+ OutputDevice
+ PhoneLine
+ Microphone
+ Speaker

Messages
+ CallerMessage
+ Mailbox
+ Greeting

Example: IODevices Package

InputDevice
Listen()

OutputDevice
Play()

Microphone
Listen()

PhoneLine

Speaker
RingCount
Hangup()
Answer()
Listen()
Play()
Play()}
Use Cases

Specifications and Use Case Diagrams

Use Case Overview

♦ Represents the functionality of the system (or class) as seen by the external user
Use Case Diagram Elements

- **Actor**
  - *shown by a stick figure with name*
  - role of an object or objects external to the system
  - one object may play many roles

- **Use Case**
  - *shown by oval with name inside or below oval - may also contain compartment for extension points*
  - unit of useful functionality provided by system
  - sequence of actions

Use Case Specification

**Use Case: Answer Caller**
This use case begins with a caller ringing the answering machine and completes after the call has been completed. This use case is meant to be extended by mode specific processing.

- **Pre-conditions:**
  - Answering machine is idle
- **Post-conditions:**
  - Greeting is played to caller
  - Mode-specific processing is performed
- **Primary Path (Caller Listens to Greeting):**
  - 1. The caller rings the phone line connected to the answering machine.
  - 2. The system waits for the ring count to reach the answering ring count.
  - 3. The system plays a greeting.
  - 4. The system performs any mode-specific processing
- **Alternate Path (Caller Hangs Up During Greeting):**
  - The caller hangs up during the playing of the greeting. The system recognizes the hang-up, stops the greeting and prepares for the next caller.
Adding Use Case Relationships

Communicates

Uses

• participation of an actor in a use case
• the only relationship between an actor and a use case

Extends

• the extended use case may contain behavior specified by extending use case

Uses

• specifies that the using use case will also include behavior of the use use case
Communicates Relationship

- Participation of an actor in a use case
- This is the only relationship between actors and use cases

**Participates in**
- Use Case: Answer Caller
- Actor: External Caller
- External Caller
- Answer Caller

Uses Relationship

- Specifies that the using use case will also include behavior of the used use case

**Using Use Case: Answer Caller**
- includes the same behavior specified in

**Used Use Case: Play Greeting**
- External Caller
- Play Greeting
- Answer Caller

<<uses>>
Extends Use Case

- The extended use case may contain behavior specified by extending use case.

Use Case: Answer Caller

- **Actor:**
  - External Caller

- **Uses:**
  - Play Greeting

- **Extension Points:**
  - Mode-specific process will be performed after greeting is complete.

Specifying Use Case Relationships
Example: Answering Machine Use Diagram

Activity Diagrams
Activity Diagram Overview

- Represents the state of a procedure
  - Each state represents a step in the procedure
  - The exit event from the state is the completion of the procedure
- Shows a workflow of
  - Use Cases (Requirements Definition)
  - Operations (Package Specification)
- Can show concurrency

Answering System Example

Take Caller Message

Answer Phone Line

Play Greeting

Record Caller Message

Play to Output Device

[Call Stopped] [Message] [No Message] Update Indicator
Activity Diagram Symbols

- Initial State
  - Answer Phone Line
  - Activity

- State Transition
  - Play Greeting

- Synchronization Bar
  - Record Caller Message
  - Play to Output Device

- Synchronization Condition
  - [Call Stopped]

- Decision Activity
  - Guard
    - [Message]
    - [No Message]
  - End State

Multiple Triggers

- Review Messages

- * Multiple Trigger

- Play Message
Object Flows

Answer Phone Line

Play Greeting

Record Caller Message

[Call Stopped]

[Message]

[No Message]

Update Indicator

Object

Message [new]

Swim Lanes

Answering System

Messages

IO Devices

Answer Phone Line

Play Greeting

Record Caller Message

[Call Stopped]

[Message]

[No Message]

Update Indicator

Play to Output Device

Record Caller Message

Play to Output Device

Update Indicator

[Message]

[No Message]
Architecture Design: Results

- Class Models
- Method Traces
- Event Traces
- State Models
- Component Models
- Deployment Models

Class Design Models

Specification and Implementation Levels
Class Design Model Overview

- Specification
  - Define interfaces to software implementing the domain class model

- Implementation
  - Shows where and how implementations are performed

Packages

- Begin to identify classes that define the interface and the public/non-public classes that implement those interfaces
Class Symbols

**Observer**

<<interface>>

| notify() : void |

**Message**

{Abstract}

| message : AudioCut |
| play(device: OutputDevice) : void |

- **Indicate**
  - Types (interface)
    - defines the signatures of how to communicate with all sub-types
  - Base Classes (abstract)
    - also defines a type
    - defines property implementations to be inherited by sub-class class

- **Parameterized (templates) Classes**
  - Defines type-independent template structure and algorithms

- **Instantiated Classes**
  - Binds template to a data type

---

July 1998
Introduction to UML
Slide 103
Adding Class Properties

- get()/set() operations added for public attribute properties
- public (+), protected (#), and private (-) accesses designated for all properties
- specify scope
  - object (no underline)
  - class (underlined)

Attribute/Operation Descriptions

Class: PhoneLineImpl

- Attributes
  - deviceAddress_: long
    Contains the i/o port number for the modem

- Operations:
  - virtual play(audio : Audio) throw(NotReady, Hangup)
    Plays the vector of audio samples to the modem. Will throw a
    NotReady exception if the modem is not ready or a Hangup
    exception if the line is hung-up during the play.
Refining Relationships

- Implementations will be put together at run-time.
- Using implementations will be coupled to a subset (interface) of the used implementation’s capabilities.
- UML offers a shorthand interface notation.

Interaction Diagrams

Sequence and Collaboration Diagrams
Interaction Diagram Overview

- Shows interaction between objects
- Varying levels of detail
  - Events (Event Trace*)
  - Methods (Method Trace*)  
    * non-UML terms
- Sequence Diagram
  - Visually arranged in time-order of messages
  - Shows lifeline of objects
- Collaboration Diagram
  - Visually arranged around objects
  - Shows object relationships

Sequence Diagram

(time)

(objects)

1: Initialize ()
2: Play ()
3: Audio ()
4: Terminate ()

Message

Return Message (optional)

Activation (focus of control)

Label

Device Player

OutputDevice

External Listener

July 1998

Introduction to UML

Slide 109
Adding Guard Conditions

- Places a restriction on the send of a message
- Each branch is labeled with a [guard]

Adding Concurrency

- Message Mode
- PhoneLine

---

May 1998
Introduction to UML
Slide 111

May 1998
Introduction to UML
Slide 112
Adding Object Lifeline

Example: Answer Caller
Example: Take Caller Message

Collaboration Diagram
Adding Other Elements

Multi-object

Object Lifetime ({new}, {deleted}, {transient})

Visibility Specification

<<global>> - global (global to related object)
<<field>> - attribute of related object
<<parameter>> - passed as temporary to related object
<<local>> - created within method of related object

Example: Answer and Take Caller Message
State Diagrams

State Diagram Overview

- Shows sequence of states an object goes through in response to a stimuli
- Shows actions and activities performed and events issued
- Allows identification of missing
  - Attributes - to remember states
  - Behaviors - to receive events
**Caller Message Example**

- **Record** event:
  - entry: set date/time
  - entry: set caller id
  - do: `^InputDevice.Listen`

- **Hangup** (Message empty):
  - Deleted

- **Hangup** (Message not empty):
  - New
  - Play / set reviewed = true
  - do: `^InputDevice.Play`

**State Diagram Elements**

- **Event**:
  - Start
  - Record
  - Hangup (Message empty)
  - Recording
  - Entry: set date/time
  - Entry: set caller id
  - do: `^InputDevice.Listen`
  - Hangup (Message not empty)

- **State**:
  - Transition
  - Guard Condition

- **Action**:
  - Activity
Key Diagram Elements

♦ Transition
  - Event arg-list [guard condition] / action expression

♦ Action - atomic, non-interruptible
  - entry / expression
  - exit / expression
  - Send Action
    - \(^{\text{Target Object.Message(arguments)}}\)

♦ Activity - on-going
  - do / expression

Transition/Entry Actions

Self-Transition

An Action taken on Play from New State

An Action taken no matter how state entered
### Composite States

- **Hangup**: Message empty
- **Deleted**
- **Delete**
- **Reviewed**
- **Play / set reviewed = true**
- **Idle**
- **Complete**
- **Playing**
  - `do: "OutputDevice.Play"`

### Concurrent States

- **Taking Message**
- **Recording**
- **Playing to Output Device**
Updating Class Model

Recording
entry: set date/time
entry: set caller id
do: "InputDevice.Listen"

Hangup[ Message not empty ]

New
Play / set reviewed = true

Caller/Message
Caller
Reviewed
DateTime
Record( )
Play( )
Delete( )

Hangup[ Message empty ]

Deleted

Deleted

Deleted

Recording

Record

Play / set reviewed = true

Recorded

Updated

Recording

Complete

Play

Playback

do: "OutputDevice.Play"

Play

Idle

Reviewed

Updating Class Model

Deployment Diagrams
Deployment Diagram Overview

- Shows mapping of software components to hardware
  - node - a computer
  - connection - a link between nodes
  - components - physical modules of code
  - dependencies - show communication between components

Deployment Diagram

- Node
- Connection
- Dependency
- Components
- Phone Company
- Workstation
- Messages
- Answering System
- I/O Devices
Refining Dependencies into Interfaces

UML and Design Patterns

- Use patterns to describe the key ideas in the system
- Document the designs that have been discarded; sometimes it is just as important to know why certain patterns/ideas are not being used
- Use UML to capture the pattern’s context for solving this domain problem
Modeling and UML

♦ What makes a good model?
♦ When to apply UML
♦ How to apply UML

Summary

♦ UML is a modeling language that can be used totally independent of the development process
♦ UML has been adopted by the OMG and is quickly becoming the notation of choice for visual modeling
♦ UML Resources
  > http://uml.systemhouse.mci.com (OMG UML info)
  > http://www.awl.com/cp/awweb.htm (survey of analysis and design methods)
  > http://www.rational.com/uml/documentation.html (Rational doc)
  > http://www.rational.com/products/rose/seed (Rational CASE tools)