Adaptive Software Engineering
G22.3033-007

Session 3 – Sub-Topic Presentation 1
Use Case Modeling

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Part I

Environmental Diagrams
What it is

• Environmental Diagram

[Diagram showing a customer renting a video from a video store, and a payroll clerk paying employees.]

What it is

• A picture containing all the important players (Actors)
• Includes players both inside and outside of the system
• Actors are a critical component
• External events are a second critical component
Creating the Diagram

- To create an environmental diagram
- 1. Identify all the initiating actors
- 2. Identify all the related external events associated with each actor

Why it is used

- A diagram is needed to show the context or scope of the proposed system
- At this time actors and external events are the critical components
- It is helpful to include all the participants as well
Creating the Diagram

3. Identify all the participating Actors

These actors may be inside (internal) or outside (external) to the system

Examples of an internal actor

- Clerk who enters the purchase into a Point of Sale terminal
- Clerk who places paper in the printer
- Accountant who audits report
Creating the Diagram

- Examples of an external actor
  - Accountant who audits report
  - A credit authorizing service
  - A DMV check for renting a car

Creating the Diagram

- 4. Draw a cloud
- 5. Then draw initiating actors on the left of the cloud
- 6. Then draw participating external actors outside the cloud
- 7. Then draw participating internal actors inside the cloud
- Recall actors are stick figures
Creating the Diagram

- 8. Lastly connect the initiation actors to the cloud
- 9. Label each connection with an external event name
- 10. It is not necessary to label connections to the participating external actors; just connect them

Creating the Diagram

- First draw a cloud
Creating the Diagram

• Label the system

POST Application

Creating the Diagram

• Insert and label the initiating actor

Customer

POST Application
Creating the Diagram

• Connect the actor with an external event

![Diagram showing a customer buying items](image)

Creating the Diagram

• Insert and label any internal participating actors

![Diagram showing a customer buying items](image)
Creating the Diagram

• Insert and label any external participating actors

Customer ✈️ Buy Items ➔ POST Application ➔ Cashier ➔ Manager

Summary

• The environmental diagram is a useful to depict a lot of useful information
• At a glance it shows all the critical entities (actors) that interact with the system
Part II

*Introduction to UML*

The Unified Modeling Language

“The Unified Modeling Language (UML) is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system.”

- Grady Booch
- James Rumbaugh (OMT)
- Ivar Jacobson (OOSE)
Building Blocks of UML

• Things

• Relationships

• Diagrams

Things

• Structural things
  – classes, interfaces, collaborations, use cases, active classes, components, nodes.

• Behavioral things
  – interactions, state machines.

• Grouping things
  – packages.

• Annotational things
  – notes.
Relationships

- Dependency
- Association
- Generalization
- Realization

Diagrams

- 1. Class diagram
- 2. Object diagram
- 3. Use case diagram
- 4. Sequence diagram
- 5. Collaboration diagram
- 6. Statechart diagram
- 7. Activity diagram
- 8. Component diagram
- 9. Deployment diagram
Structural Things

Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical.

Structural Things (cont’d)

• Class
  • A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.
  – Attribute
    • An attribute is a named property of a class that describes a range of values that instances of the property may hold.
  – Operation
    • An operation is the implementation of a service that can be requested from any object of the class to affect behavior.
Structural Things (cont’d)

• Use case
  • A use case specifies the behavior of a system or a part of a system and is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor.
  
  – Actor
  • An actor represents a coherent set of roles that users of use cases play when interacting with these use cases.

Structural Things (cont’d)

• Interface
  • An interface is a collection of operations that specify a service of a class or component.

• Collaboration
  • A collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements.
Structural Things (cont’d)

• Active class
  • An active class is a class whose objects own one or more processes or threads and therefore can initiate control activity.

• Component
  • A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.

• Node
  • A node is a physical element that exists at run time and represents a computational resource.

Behavioral Things

Behavioral things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space.
Behavioral Things (cont’d)

• Interaction
  • An interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose.

• State machine
  • A state machine is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its response to those events.

Grouping and Annotational Things

Grouping things are the organizational parts of UML models.

• Package
  • A package is a general purpose mechanism for organizing elements into groups.

Annotational things are the explanatory parts of UML models.

• Note
  • A note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.
Relationships

- **Dependency**
  - A dependency is a using relationship that states that a change in specification of one thing may affect another thing that uses it, but not necessarily the reverse.

- **Association**
  - An association is a structural relationship that specifies that objects of one thing are connected to objects of another.

Relationships (cont’d)

- **Aggregation**
  - An aggregation is a special form of association that specifies a whole-part relationship between the aggregate (the whole) and a component (the part).

- **Generalization**
  - A generalization is a relationship between a general thing and a more specific kind of that thing. Sometimes it is called an "is-a-kind-of" relationship.

- **Realization**
  - A realization is a semantic relationship between classifiers, wherein, one classifier specifies a contract that another classifier guarantees to carry out.
Diagrams

• Class diagram
  • A class diagram shows a set of classes, interfaces, and collaborations and their relationships.

• Object diagram
  • An object diagram shows a set of objects and their relationships.

• Use case diagram
  • A use case diagram shows a set of use cases and actors and their relationships.

Diagrams (cont’d)

• Sequence diagram
  • A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages.

• Collaboration diagram
  • A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.

• Statechart diagram
  • A statechart diagram shows a state machine, consisting of states, transitions, events, and activities.
Diagrams (cont’d)

• Activity diagram
  • An activity diagram is a special kind of a statechart diagram that shows the flow from activity to activity within a system.

• Component diagram
  • A component diagram shows the organization and dependencies among a set of components.

• Deployment diagram
  • A deployment diagram shows the configuration of runtime processing nodes and the components that live on them.

Example

• A University wants to computerize their registration system
  – The Registrar sets up the curriculum for a semester
    • One course may have multiple course offerings
  – Students select 4 primary courses and 2 alternate courses
  – Once a student registers for a semester, the billing system is notified so the student may be billed for the semester
  – Students may use the system to add/drop courses for a period of time after registration
  – Professors use the system to receive their course offering rosters
  – Users of the registration system are assigned passwords which are used at logon validation
Actors

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

![Actor Diagram]

Use Cases

- 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
  - Registrar -- maintain the curriculum
  - Professor -- request roster
  - Student -- maintain schedule
  - Billing System -- receive billing information from registration

![Use Case Diagram]
Use Case Diagram

- Student
  - Maintain Schedule
  - Request Course Roster
  - Maintain Curriculum

Sequence Diagram

1: Student
2: registration
3: registration
4: math
5: add (joe)
6: are you open?
7: add (joe)
Collaboration Diagram

Classes

RegistrationForm

ScheduleAlgorithm

RegistrationManager

Course

Student

Professor

CourseOffering
Classes

RegistrationForm

RegistrationManager
- addStudent(Course, StudentInfo)

Course
- name
- numberCredits
- open()
- addStudent(StudentInfo)

Student
- name
- major

Professor
- name
- tenureStatus

CourseOffering
- status
- open()
- addStudent(StudentInfo)

ScheduleAlgorithm

Relationships

RegistrationForm

RegistrationManager
- addStudent(Course, StudentInfo)

Course
- name
- numberCredits
- open()
- addStudent(StudentInfo)

Student
- name
- major

Professor
- name
- tenureStatus

CourseOffering
- status
- open()
- addStudent(StudentInfo)

ScheduleAlgorithm
State Transition Diagram

- **Initialization**
  - Event: Register student
  - Action: Increment count
- **Open**
  - Action: Initialize course
- **Closed**
  - Action: Finalize course
- **Canceled**
  - Action: Notify registered students

Component Diagram

- **Billing.exe**
  - System: Billing System
- **Register.exe**
- **People.dll**
  - User
- **Course.dll**
  - Course
  - Course Offering
- **Student.dll**
- **Professor.dll**
Part III

Discovering Objects
Discovering Objects

- Searching the Requirements
- Searching the Use Case
- Using a Concept Category List

Searching the Requirements

- Recall objects are nouns
- Most functions have a verb with a noun
- For example
  - Record the Sale (Object Sale)
  - Reduce Inventory Quantity (Object Inventory)
  - Provide a persistent storage mechanism (Object Data Base)
Searching the Requirements

- Watch out for adjectives
- Adjectives are attributes of objects
- For example
  - Calculate the current sale total (total is not an object)
  - Cashier must log in with ID (ID is not an object)
  - Display price of item (price is not an object)

Searching the Use Case

- Again recall objects are nouns
- Most sentences have a subject, verb, and object
- For example
  - Customer arrives at POST (Objects Customer & POST)
  - Customer gives a cash payment (Objects Customer & Payment)
Searching the Use Case

- Again watch out for adjectives
- Adjectives are attributes of objects
- For example
  - Customer submits the upc for each item (upc is not an object)
  - Customer selects the car and price (price is not an object)

Using a Concept Category List

- This is one of the best ways to identify objects
- Your author has a list that contains many redundancies
- Your author also has missed some categories
Using a Concept Category List

A shorter category list by Stephen Mellor:

• Tangible objects (Appliance)
• Intangible objects (Company)
• Specifications (Appliance Description)
• Roles of people (Customer)
• Events (Accident)
• Interaction (Sale)

Even Mellor missed some basic categories

• Peter Coad added the ‘has a’ and ‘is a’ objects
  – Has A (Container or Things in a Container)
  – Has A (Assembly or Part of)
  – Has A (Organization or Member)
Using a Concept Category List

- At this point if in doubt, add the concept to the list
- Later, objects that do not have behavior or attributes can be omitted

Summary

- Make sure concepts are nouns and not adjectives
- Objects are discovered using the:
  - Requirements
  - Use Cases
  - Concept Category Lists
- At this point in the SDLC, one includes a concept if they are in doubt
Part VII

Using UML


http://www.holub.com/class/oo_design/uml.html

Development Life Cycle Model

![Diagram of Development Life Cycle Model]

Project Management Support Processes:
- Risk Reduction
- Training
- Planning
- Configuration Management
- Estimating
- Metrics
- Quality Assurance
UML …

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

UML is not …

• … a method or methodology (Method = Notation (e.g., UML) + Process)
• … a proponent of a particular process (although the “Rational Objectory Process” is being proposed by Booch, Rumbaugh and Jacobson)
Starting Point

• 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”)

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
Model Perspective Hints

- Works as a map of the system
- Different subsystems become UML packages
- Keep dependencies simple and domain-related
- Define relationships and interactions between packages
- Address both functional and non-functional requirements
- Take time to factor in reuse

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

<table>
<thead>
<tr>
<th>Diagram Name</th>
<th>Type</th>
<th>Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case</td>
<td>Static*</td>
<td>Analysis</td>
</tr>
<tr>
<td>Class</td>
<td>Static</td>
<td>Analysis</td>
</tr>
<tr>
<td>Activity</td>
<td>Dynamic&quot;</td>
<td>Analysis</td>
</tr>
<tr>
<td>State-Transition</td>
<td>Dynamic</td>
<td>Analysis</td>
</tr>
<tr>
<td>Event Trace (Interaction)</td>
<td>Dynamic</td>
<td>Design</td>
</tr>
<tr>
<td>Sequence</td>
<td>Static</td>
<td>Design</td>
</tr>
<tr>
<td>Collaboration</td>
<td>Dynamic</td>
<td>Design</td>
</tr>
<tr>
<td>Package</td>
<td>Static</td>
<td>Delivery</td>
</tr>
<tr>
<td>Deployment</td>
<td>Dynamic</td>
<td>Delivery</td>
</tr>
</tbody>
</table>

*Static describes structural system properties
"Dynamic describes behavioral system properties.

Static Notation - Classes

Box contains three compartments:

1. The name compartment (required) contains the class name and other documentation-related information: E.g.:
   Some_class «abstract»
   Guillemets identify stereotypes. E.g.: «utility», «abstract» «JavaBean». Can use graphic instead of word.
   Access privileges (see below) can precede name
   Inner (nested) classes identify outer class as prefix of class name: (Outer.Inner or Outer::Inner)

2. The attributes compartment (optional):
   During Analysis: identify the attributes (i.e. defining characteristics) of the object.
   During Design: identify a relationship to a stock class.

3. The operations compartment (optional) contains method definitions:
   message_name( arguments ): return_type

   If attributes and operations are both omitted, a more-complete definition is assumed to be on another sheet.
**Static Notation - Associations**

Associated classes are connected by lines.
The relationship is identified, if necessary, with a < or > to indicate direction (or use solid arrowheads).
The role that a class plays in the relationship is identified on that class's side of the line.
Stereotypes (like «friend») are appropriate.
Unidirectional message flow can be indicated by an arrow (but is implicit in situations where there is only one role)
Cardinality:
1 - (usually omitted if 1:1)
N - (unknown at compile time, but bound)
0..1 - (1..2 1..n)
1..* - (1 or more)
* - (0 or more)

**Static Notation - Implementation**

**Inheritance (Generalize/Specialize)**

Identifies derivation with the derived class as the base class and with additional (or modified) properties. Derived (sub) class is a specialization of (extends) the base (super) class.
Variations include:
Static Notation - Interface Inheritance
(Specifies/Refines)

A contract that specifies a set of methods that must be implemented by the derived class. In C++, an interface is a class containing nothing but pure virtual methods. Java supports them directly. (c.f. "abstract class," which can contain method and field definitions in addition to the abstract declarations.)

Interfaces contain no attributes, so the "attributes" compartment is always empty.

The "inheritance" relationship line is dashed if the base class is an interface.

User uses Resource, but Resource is not a member of the User class. If Resource is modified, some method of User might need to be modified. Resource is typically a local variable or argument of some method in User.
Static Notation - Aggregation

Aggregation (comprises) relationship. Destroying the "whole" does not destroy the parts.

Composition (has) relationship. The parts are destroyed along with the "whole." Doesn't really exist in Java. In C++:

```cpp
class Container
{
    Item item1; // both of these are
    Item *item2; // "composition"

public:
    Container() { item2 = new Item; }
    ~Container() { delete item2; }
}
```
Static Notation - Navigability

Messages flow in direction of arrow (only). Implicit when no role present: if an object doesn't have a role in some relationship, then there's no way to send messages to it.

Static Notation - Constraint

A constrained relationship requires some rule to be applied. (e.g. \{ordered\}) Often combined with aggregation, composition, etc.
Static Notation -

In the case of the or, only one of the indicated relationships will exist at any given moment (a C++ union).

Subset does the obvious.

In official UML, put arbitrary constraints that affect more than one relationship in a "comment" box, as shown.

Static Notation - Qualified Association

Qualified Association (hash-table, associative array, "dictionary").

class User
{
    // A Hashtable is an associative array,
    // indexed by some key and containing
    // some value.

    private Hashtable bag = new HashTable();

    private void add(String key, Item value)
    {
        bag.put(key, value);
    }
}
Static Notation - Association

Class

Use when a class is required to define a relationship.

Somewhere, an additional relationship is required to show ownership. (The one between Person and Ticket in the current example.)

Static Notation - Summary

- Dependency
- Aggregation
- Composition
- Navigability
- Constraint
- Or
- Subset
Dynamic Notation - Objects and Messages

Vertical lines represent objects, not classes. May optionally add a "class" to the box if it makes the diagram more readable.

→ represents synchronous message. (message handler doesn't return until done).
represents return. (Label arrow with name/type of returned object.) Return arrows are essential in UML style, otherwise control flow is ambiguous.

Sending object's class must have: A association of some sort with receiving-object's class.
The receiver-side class's "role" must be the same as the name of the receiving object.

Dynamic Notation - Object Creation

Name box appears at point of creation.
«creates» form for automatic creation, e.g., in C++:
class Creator1
{  Object1 the_object; // not a reference
}

(There is no equivalent operation in Java)

If message shown instead of «creates», then the message handler creates the object. Think of new Fred() as Fred.new(). Method does not have to be new().
Dynamic Notation - Conditions, Loops, Grouping

Message sent only if condition in brackets is true.

An asterisk next to the condition signifies iteration.

An asterisk without a condition means "every," when receiver is an aggregate.

The heavy line used to group together the messages that are affected by some condition is my own extension to UML.

The box at the bottom of the diagram is UML's Grouping notation (awkward when you try to group all indirect messages). A * indicates a loop, omit for an "if."

Dynamic Notation - Asynchronous Messages

Half arrowhead means "asynchronous." (see table, below).

Widening of line means "activated" (thread is running). (In diagram to left, sender thread sends message() and then suspends itself waiting for the reply().)

Break in box (on "receiver" in diagram to left) means other activity is happening on the thread that isn't relevant in the current scenario. (E.g. Sender could wait() after sending message. reply() message executes notify().)

Large X means object deleted (in this case, it's self deleting). An external kill is represented as:
Dynamic Notation - State Transition

- 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

Activity Diagrams - Starting and Stopping
Activity Diagrams -
Synchronization (Fork/Join)

When several activities can go on in parallel, indicates when all activities must be finished in order to continue. The heavy bar at the top is a fork. After the fork, all activities can (but are not required to) go on in parallel. Progress cannot continue past the bar on the bottom (a join) until all the activities that feed into the join complete. A join is an AND operation.

Activity Diagrams - Guards (tests)

This path is used only if the text in the brackets is true.
Activity Diagrams - Decision (Branch/Merge)

A decision activity, the guard labels the decision that was made. The diamond with outgoing arrows (the branch) specifies an OR operation, with a condition imposed by the guard. The diamond with incoming arrows (a merge) simply provides an end to the OR operation. A merge can occur without an associated branch if the diagram has multiple start states.

Activity Diagrams - Swim Lanes

Activities are arranged into vertical zones delimited with dashed lines. Each zone represents a broad area of responsibilities, typically implemented by a set of classes or objects. For example, the swim lane labeled accounting could represent objects of several classes (Bookkeeper, Clerk, MailRoom, Accountant) working in concert to perform the single "cut paycheck" activity.
**Development Life Cycle Model**

<table>
<thead>
<tr>
<th>Process Steps</th>
<th>Process Gates</th>
<th>Prototypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQUIREMENTS DEFINITION</td>
<td>REVIEW</td>
<td>PROTOTYPE 1</td>
</tr>
<tr>
<td>HIGH LEVEL DESIGN</td>
<td>REVIEW</td>
<td>PROTOTYPE 2</td>
</tr>
<tr>
<td>DETAIL DESIGN</td>
<td>REVIEW</td>
<td>PROTOTYPE 3</td>
</tr>
<tr>
<td>SYSTEM CONSTRUCTION</td>
<td>REVIEW</td>
<td></td>
</tr>
<tr>
<td>VERIFICATION &amp; VALIDATION</td>
<td>REVIEW</td>
<td></td>
</tr>
<tr>
<td>SYSTEM DELIVERY</td>
<td>REVIEW</td>
<td>POST IMPLEMENTATION REVIEW</td>
</tr>
</tbody>
</table>

**Project Management Support Processes**

- Risk Reduction
- Training
- Planning
- Configuration Management
- Estimating
- Metrics
- Quality Assurance

**Use Case**

A use case is a relatively large end-to-end process description that typically includes many steps or transactions; it is not normally an individual step or activity in a process.
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 among 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

Finding Concepts

<table>
<thead>
<tr>
<th>Concept Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical or tangible object</td>
<td></td>
</tr>
<tr>
<td>Specifications, designs, or descriptions</td>
<td></td>
</tr>
<tr>
<td>Places</td>
<td></td>
</tr>
<tr>
<td>Transactions</td>
<td></td>
</tr>
<tr>
<td>Transaction line items</td>
<td></td>
</tr>
<tr>
<td>Roles of people</td>
<td></td>
</tr>
<tr>
<td>Containers of other things</td>
<td></td>
</tr>
<tr>
<td>Things in a container</td>
<td></td>
</tr>
<tr>
<td>Other computer or electro-mechanical systems external to our system</td>
<td></td>
</tr>
<tr>
<td>Abstract noun types</td>
<td></td>
</tr>
<tr>
<td>Organizations</td>
<td></td>
</tr>
<tr>
<td>Events</td>
<td></td>
</tr>
<tr>
<td>Processes (often not represented as a concept)</td>
<td></td>
</tr>
<tr>
<td>Rules and policies</td>
<td></td>
</tr>
<tr>
<td>Catalogs</td>
<td></td>
</tr>
<tr>
<td>Records of finance, work, contracts, legal matters</td>
<td></td>
</tr>
<tr>
<td>Financial instruments and services</td>
<td></td>
</tr>
<tr>
<td>Manuals, books</td>
<td></td>
</tr>
</tbody>
</table>
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

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
  - Confirm Availability for each chosen course
  - Attend Class
Dynamic Notation - State Transition

- 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

Development Life Cycle Model

<table>
<thead>
<tr>
<th>Process Steps</th>
<th>Process Gates</th>
<th>Prototypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQUIREMENTS DEFINITION</td>
<td>REVIEW</td>
<td>PROTOTYPE 1</td>
</tr>
<tr>
<td>HIGH LEVEL DESIGN</td>
<td>REVIEW</td>
<td>PROTOTYPE 2</td>
</tr>
<tr>
<td>DETAIL DESIGN</td>
<td>REVIEW</td>
<td>PROTOTYPE 3</td>
</tr>
<tr>
<td>SYSTEM CONSTRUCTION</td>
<td>REVIEW</td>
<td></td>
</tr>
<tr>
<td>VERIFICATION &amp; VALIDATION</td>
<td>REVIEW</td>
<td></td>
</tr>
<tr>
<td>SYSTEM DELIVERY</td>
<td>REVIEW</td>
<td></td>
</tr>
</tbody>
</table>

Project Management Support Processes
Risk Reduction  Training  Planning  Configuration Management  Estimating  Metrics  Quality Assurance
Problem Conceptualization

1. Define Purpose and Prioritized Features
2. “Sketch” Early Domain Class Model
3. Identify Use Cases and Primary Paths
4. 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

- In your teams, identify the following for LadarVision:
  - Abstractions
  - Attributes
  - Responsibilities
  - Relationships

Requirements Definition

- 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 Applied

- In your teams, identify 5 system requirements for LadarVision
Development Life Cycle Model

- Process Steps
  - REQUIREMENTS DEFINITION
  - HIGH LEVEL DESIGN
  - DETAIL DESIGN
  - SYSTEM CONSTRUCTION
  - VERIFICATION & VALIDATION
  - SYSTEM DELIVERY

- Process Gates
  - REVIEW

- Prototypes
  - PROTOTYPE 1
  - PROTOTYPE 2
  - PROTOTYPE 3

Project Management Support Processes
- Risk Reduction
- Training
- Planning
- Configuration Management
- Estimating
- Metrics
- Quality Assurance

The Lift Problem – Classics in Software Engineering
The Elevator Problem

A product is to be installed to control elevators in a building with 3 floors. The problem concerns the logic required to move elevators between floors according to the following constraints:

1. Each elevator has a set of 3 buttons, one for each floor. These illuminate when pressed and cause the elevator to visit the corresponding floor. The illumination is canceled when the elevator visits the corresponding floor.

2. Each floor, except the first floor and top floor has two buttons, one to request an up-elevator and one to request a down-elevator. These buttons illuminate when pressed. The illumination is canceled when an elevator visits the floor and then moves in the desired direction.

3. When an elevator has no requests, it remains at its current floor with its doors closed.

3 Views of the Problem Domain
Solution Space

Problem Context
Level 0 DFD

Entity Relationship Diagram
Behavior – What it does

Unified Modeling Language (UML)

UML is a modeling language that specifies semantics and notation without process is currently defined. These are the basic diagrams to use:

– OOA Models
  • Use Case Diagram
  • Class Diagram
  • State Diagram

– OOD Models
  • Detailed Class Diagram
  • Collaboration Diagram
  • Sequence Diagram
Object-Oriented Analysis Phase

- **OO counterpart to the "structured" portions of the classical specification phase: ER diagrams, DFDs, FSMs, etc**

- **OOA:** Semi-formal specification technique, builds on the above.
  - With OO, data and action treated as equal partners
  - A well designed object has high cohesion, low coupling,
  - models all aspects of one physical entity

- Initially, many different methods emerged (Booch, OMT, Shlaer-Mellor, Coad-Yourdon) — all essentially equivalent

- UML (unified modeling language): a common notation for representing OOA, emerging as defacto standard

OOA Consists of Three Basic Steps:

1. **Use-case modeling** (mostly action-oriented)
   - Focus: how results computed by product (w/o RT sequencing)
   - utilizes use cases and scenarios (parallels to DFDs)

2. **Class modeling** ("object modeling") (purely data oriented)
   - Determine classes, attributes, and
   - relationships between objects (parallels to ER diagrams)
   - deduce classes from:
     - use cases, noun extraction, CRC cards

3. **Dynamic modeling** (purely action-oriented)
   - Determine the actions performed by or to each class
   - utilize state diagrams (parallels to FSMs)

Note: OOA is iterative, above steps are repeatedly revisited
Elevator Problem: OOA

• Step I: Use-Case Modeling
  – Use case: generic description of overall functionality
  – Scenario: instance of a use case
    • List typical scenarios of actions performed by each class
    • Get comprehensive insight into behavior of product

The Elevator Problem

A product is to be installed to control elevators in a building with 3 floors. The problem concerns the logic required to move elevators between floors according to the following constraints:

1. Each elevator has a set of 3 buttons, one for each floor. These illuminate when pressed and cause the elevator to visit the corresponding floor. The illumination is canceled when the elevator visits the corresponding floor.

2. Each floor, except the first floor and top floor has two buttons, one to request an up-elevator and one to request a down-elevator. These buttons illuminate when pressed. The illumination is canceled when an elevator visits the floor and then moves in the desired direction.

3. When an elevator has no requests, it remains at its current floor with its doors closed.
"Normal" Scenario

1. User A presses Up floor button at floor 3 to request elevator. User A wishes to go to floor 7.
2. Up floor button is turned on.
3. An elevator arrives at floor 3. It contains User B who has entered the elevator at floor 1 and pressed the elevator button for floor 9.
4. Up floor button is turned off.
7. Floor 7 elevator button is turned on.
8. Elevator doors close.
9. Elevator travels to floor 7.
10. Floor 7 elevator button is turned off.
11. Elevator doors open to allow User A to exit elevator.
13. Elevator doors close after timeout.
14. Elevator proceeds to floor 9 with User B.

"Abnormal" Scenario

1. User A presses Up floor button at floor 3 to request elevator. User A wishes to go to floor 1.
2. Up floor button is turned on.
3. An elevator arrives at floor 3. It contains User B who has entered the elevator at floor 1 and pressed the elevator button for floor 9.
4. Up floor button is turned off.
6. User A presses elevator button for floor 1.
7. Floor 1 elevator button is turned on.
8. Elevator doors close after timeout.
10. Floor 9 elevator button is turned off.
11. Elevator doors open to allow User B to exit elevator.
13. Elevator doors close after timeout.
14. Elevator proceeds to floor 1 with User A.
Use Case Diagram

- A generalized description of how a system will be used.
- Provides an overview of the intended functionality of the system.
- Understandable by laymen as well as professionals.

Elevator basic scenario that can be extracted from Use Case Diagram:

1. Passenger pressed floor button
2. Elevator system detects floor button pressed
3. Elevator moves to the floor
4. Elevator doors open
5. Passenger gets in and presses elevator button
6. Elevator doors closes
7. Elevator moves to required floor
8. Elevator doors open
9. Passenger gets out
10. Elevator doors closes
Step II of OOA: Class Modeling

- Goal: Extract classes and their attributes, represent relationships (including inheritance) between classes using an ER diagram
- Approaches:
  - Deduce classes from use cases and their scenarios
  - often many scenarios
  - danger: inferring too many candidate classes
- Noun extraction
  - 'always' works (gives you something to start with)

Walkthroughs

- technique used to uncover application's desired behavior
- pretend you already have a working application, walk through the various uses of the system
- walkthroughs help to uncover all intended uses of system
- When to stop scenario walkthroughs:
  - continue until you handle every aspect of expected scenarios
  - questions that arise may need to be resolved by the client
  - stop when you feel your components are "good enough"
  - let customer decide what special cases need to be implemented
Noun Extraction Approach to Class Modeling

For developers without domain experience
Three stages from highly abstract to less abstract
- Stage 1 of Noun Extraction: Concise Definition
- Stage 2 of Noun Extraction: Informal Strategy
- Stage 3 of Noun Extraction: Formalize the Strategy

Stage 1 of Noun Extraction: Concise Definition
- Define product as concisely in a single sentence if possible!

Buttons in elevators and on floors are to be used to control motion of n elevators in building with m floors
Stage 2 of Noun Extraction: **Informal Strategy**

- Incorporate constraints into Stage 1
- express result (preferably) in a single paragraph

_Buttons in elevators and on floors control movement of n elevators in building with m floors. Buttons illuminate when pressed to request elevator to stop at specific floor; illumination is canceled when request has been satisfied. If elevator has no requests, it remains at its current floor with its doors closed._

Stage 3 of Noun Extraction: **Formalize the Strategy**

- Identify nouns in informal strategy.
- Use nouns as candidate classes
  - Nouns
    - button, elevator, floor, movement, building, illumination, illumination, door
    - floor, building, door are **outside problem boundary**
    - movement, illumination, illumination are abstract nouns — **exclude but may become attributes**
Class and Subclass Candidates

- Candidate classes: Elevator and Button
- Subclasses: Elevator Button and Floor Button
- Strive for all relationships either 1-to-1 or 1-to-n
  - Makes design, implementation easier

Class Description for Elevator Controller

**Class**
Elevator Controller

**Responsibility**
1. Turn on elevator button
2. Turn off elevator button
3. Turn on floor button
4. Turn off floor button
5. Open elevator doors
6. Close elevator doors
7. Move elevator one floor up
8. Move elevator one floor down

**Collaboration**
1. Class Elevator Button
2. Class Floor Button
3. Class Elevator
Class diagrams show the static structure of the object, their internal structure, and their relationships.

Step III of OOA: Dynamic Modeling

- **Aim** - Produce UML state diagram
- **State diagram is less formal than an FSM**
  - State, event, predicate distributed over UML state diagram
  - UML "guards" (predicates) are in brackets
- **State diagram shows which state is entered given a starting state, and predicates (state transitions) that hold.**
- **State diagrams help determine actions performed by or to each class. An action-oriented step of OOA**
- **Note: UML reserved word do indicates an action that is to be carried out**
Controller class's 1st responsibility

Class: Elevator Controller
Responsibility

1. Turn on elevator button
2. Turn off elevator button
3. Turn on floor button
4. Turn off floor button
5. Open elevator doors
6. Close elevator doors
7. Move elevator one floor up
8. Move elevator one floor down

Collaboration

1. Class Elevator Button
2. Class Floor Button
3. Class Elevator

Elevator Controller Class's 1st Responsibility

• Turn on elevator button
  – Poor from an OOA standpoint
  – Elevator Button objects should be responsible for turning themselves on/off
  – Elevator Controller should not have knowledge of the internals of Elevator Button Responsibility

• Turn on elevator button should be Send message to Elevator Button to turn on button

Another problem with above CRC cards: a class has been overlooked.
Concept of *state* is Important

- If an item's state changes during implementation's execution, then item likely should be a class.
  - Elevator doors have a *state* that changes during execution
  - Add class Elevator Doors
- Safety/Security considerations may also add classes

State diagram

- A state diagram shows the sequences of states an object goes through during it's life cycle in response to stimuli, together with its responses and actions.
- Looks like our Finite State Machine model from Structured Analysis.
State Diagram

• A generalized description of how a system will be used.
• Provides an overview of the intended functionality of the system.
• Understandable by laymen as well as professionals.

Use Case Diagram

- Detect if (E,F) button is pressed
- Button illumination is turned on/off
- Move/stop Elevator
- Open/close Doors
Elevator basic scenario that can be extracted from Use Case Diagram:

1. Passenger pressed floor button
2. Elevator system detects floor button pressed
3. Elevator moves to the floor
4. Elevator doors open
5. Passenger gets in and presses elevator button
6. Elevator doors closes
7. Elevator moves to required floor
8. Elevator doors open
9. Passenger gets out
10. Elevator doors closes

Class Diagram

Class diagrams show the static structure of the object, their internal structure, and their relationships.
Transitioning to the OOD Phase

Once requirements are approved the design team gets:

- specification document,
- use case scenarios,
- CRC cards and
- UML diagrams
A Sequence diagram shows the explicit sequence of messages suitable for modeling a real-time system.
Collaboration Diagram

A collaboration diagram shows the relationships between objects.
Detailed Class Diagram

At the end, we get to Z!
Part IV

Software Architecture and the UML

Dimensions of software complexity

Higher technical complexity
- Embedded, real-time, distributed, fault-tolerant
- Custom, unprecedented, architecture reengineering
- High performance

Lower technical complexity
- Mostly 4GL, or component-based
- Application reengineering
- Interactive performance

Higher management complexity
- Large scale
- Contractual
- Many stakeholders
- “Projects”

Lower management complexity
- Small scale
- Informal
- Single stakeholder
- “Products”
Forces in Software

Our enemy is complexity, and it's our goal to kill it.

Jan Baan

The challenge over the next 20 years will not be speed or cost or performance; it will be a question of complexity.

Bill Raduchel, Chief Strategy Officer, Sun Microsystems

Our enemy is complexity, and it's our goal to kill it.

Jan Baan

Architectural style

• An architecture style defines a family of systems in terms of a pattern of structural organization.

• An architectural style defines
  – a vocabulary of components and connector types
  – a set of constraints on how they can be combined
  – one or more semantic models that specify how a system’s overall properties can be determined from the properties of its parts.
Many stakeholders, many views

- Architecture is many things to many different interested parties
  - end-user
  - customer
  - project manager
  - system engineer
  - developer
  - architect
  - maintainer
  - other developers
- Multidimensional reality
- Multiple stakeholders
  ➔ multiple views, multiple blueprints

How many views?

- Simplified models to fit the context
- Not all systems require all views:
  - Single processor: drop deployment view
  - Single process: drop process view
  - Very Small program: drop implementation view
- Adding views:
  - Data view, security view
The Value of the UML

- Is an open standard
- Supports the entire software development lifecycle
- Supports diverse applications areas
- Is based on experience and needs of the user community
- Supported by many tools

Creating the UML
UML Partners

- Rational Software Corporation
- Hewlett-Packard
- I-Logix
- IBM
- ICON Computing
- Intellicorp
- MCI Systemhouse
- Microsoft
- ObjeqTime
- Oracle
- Platinum Technology
- Taskon
- Texas Instruments/Sterling Software
- Unisys

Contributions to the UML

- Meyer
  - Before and after conditions
- Harel
  - Statecharts
- Gamma, et al
  - Frameworks and patterns
- Booch
  - Booch method
- Rumbaugh
  - OMT
- Jacobson
  - OOSE
- Wirfs-Brock
  - Responsibilities
- Embley
  - Singleton classes and high-level view
- HP Fusion
  - Operation descriptions and message numbering
Overview of the UML

• The UML is a language for
  – visualizing
  – specifying
  – constructing
  – documenting
the artifacts of a software-intensive system

Overview of the UML

• Modeling elements
• Relationships
• Extensibility Mechanisms
• Diagrams
Modeling Elements

- Structural elements
  - class, interface, collaboration, use case, active class, component, node
- Behavioral elements
  - interaction, state machine
- Grouping elements
  - package, subsysten
- Other elements
  - note

Relationships

- Dependency
- Association
- Generalization
- Realization
A model is a complete description of a system from a particular perspective.

A diagram is a view into a model
- Presented from the aspect of a particular stakeholder
- Provides a partial representation of the system
- Is semantically consistent with other views

In the UML, there are nine standard diagrams
- Static views: use case, class, object, component, deployment
- Dynamic views: sequence, collaboration, statechart, activity
Use Case Diagram

• Captures system functionality as seen by users

• Built in early stages of development

• Purpose
  – Specify the context of a system
  – Capture the requirements of a system
  – Validate a system’s architecture
  – Drive implementation and generate test cases

• Developed by analysts and domain experts
Class Diagram

• Captures the vocabulary of a system

• Built and refined throughout development

• Purpose
  – Name and model concepts in the system
  – Specify collaborations
  – Specify logical database schemas

• Developed by analysts, designers, and implementers
Object Diagram

- Captures instances and links

Object Diagram
- Shows instances and links
- Built during analysis and design
- Purpose
  - Illustrate data/object structures
  - Specify snapshots
- Developed by analysts, designers, and implementers
Component Diagram

- Captures the physical structure of the implementation
- Built as part of architectural specification
- Purpose
  - Organize source code
  - Construct an executable release
  - Specify a physical database
- Developed by architects and programmers
Deployment Diagram

- Captures the topology of a system’s hardware
- Built as part of architectural specification
- Purpose
  - Specify the distribution of components
  - Identify performance bottlenecks
- Developed by architects, networking engineers, and system engineers
Sequence Diagram

- Captures dynamic behavior (time-oriented)

Purpose
- Model flow of control
- Illustrate typical scenarios
Collaboration Diagram

- Captures dynamic behavior (message-oriented)

Purpose
- Model flow of control
- Illustrate coordination of object structure and control
Statechart Diagram

- Captures dynamic behavior (event-oriented)

State Machine

Purpose
- Model object lifecycle
- Model reactive objects (user interfaces, devices, etc.)
Activity Diagram

- Captures dynamic behavior (activity-oriented)

- Purpose
  - Model business workflows
  - Model operations
Software engineering process

A set of partially ordered steps intended to reach a goal. In software engineering the goal is to build a software product or to enhance an existing one.

• Architectural process
  – Sequence of activities that lead to the production of architectural artifacts:
    • A software architecture description
    • An architectural prototype

Key concepts

• Phase, Iterations  
  • Process Workflows
    – Activity, steps

• Artifacts
  – models
  – reports, documents

• Worker: Architect
Lifecycle Phases

- Inception: Define the scope of the project and develop business case
- Elaboration: Plan project, specify features, and baseline the architecture
- Construction: Build the product
- Transition: Transition the product to its users

Major Milestones

- Vision
- Baseline Architecture
- Initial Capability
- Product Release
**Phases and Iterations**

<table>
<thead>
<tr>
<th>Inception</th>
<th>Elaboration</th>
<th>Construction</th>
<th>Transition</th>
</tr>
</thead>
</table>

An iteration is a sequence of activities with an established plan and evaluation criteria, resulting in an executable release.

**Architecture-Centric**

- Models are vehicles for visualizing, specifying, constructing, and documenting architecture.
- The Unified Process prescribes the successive refinement of an executable architecture.
Architecture is making decisions

The life of a software architect is a long (and sometimes painful) succession of suboptimal decisions made partly in the dark.