OBJECT-ORIENTED ANALYSIS AND MODELING PROCESS

I. PURPOSE:

This procedure describes the Object-Oriented Analysis and Modeling Process for the development of Takshila Systems Inc. software solutions. The main objectives are to use the capabilities of OO Analysis to significantly improve quality and to define the concepts, notation, and processes appropriate for object modeling. Prerequisites for this procedure include the Introduction to the Object Oriented Approach and the Introduction to the Unified Modeling Language.

II. SCOPE:

This procedure applies to software development projects and the technical staff who need to understand the pragmatics of the Object-Oriented Analysis and the Object Modeling Process. This includes System or Software Managers, System Architects, System Engineers, System Designers, Software Engineers, Software Designers, and Software Developers/Programmers.

III. GENERAL:

The Object Modeling Process

Where Does Object Modeling Fit In?

Domain Modeling is started first. This establishes a core definition of the key entities in the system. It is needed to establish a common understanding of the real world. Most reusable: Least sensitive to changes in the application.

Application Modeling is usually done together with Use Case models. This captures application objects and their requirements.

Start the Object view before the other views. Class Diagram is the core diagram for UML and the base for the other views. This is the most stable diagram as the system evolves. It organizes systems into manageable pieces. Object Modeling techniques similar in both domain and application modeling.
The Object Modeling Process:

Collect Existing Information:

- Before beginning the object modeling process:
  - Collect all existing information
  - Include domain experts in the process
  - Use real-world knowledge of the system
- Consider object modeling steps as non-sequential:
  - Many steps occur concurrently
  - We describe them sequentially for ease of presentation

Types of Existing Information:

- Problem statement
- Requirements specification
- Use Case information
- Concepts of Operations document
- Expert knowledge of the system
- Real-world knowledge
- Prototypes

It depends on what is available or supplied by the client.
Build Context Diagram:

- Helps to visualize system boundaries
- Choose a descriptive name for the system. The problem statement may already contain one (or more).
- Scan existing information for interfaces with entities (actors) external to the system
- Identify all communications being exchanged between the system and each of the actors
- Draw the diagram

Context Diagram:

- External actors are outside the system and communicate with the system
- Communications may be stimulus, responses, events, or data
- Other objects are within the system

Identify Candidate Objects:

- Underline all nouns in the existing information (or Use Case)
  - Begin with the nouns
  - Impose no restrictions on candidate objects
  - Want the broadest possible scope of objects
  - Use relevance to the problem as the main criterion for selection
  - Include both physical entities and concepts
- Do not try to organize using aggregation or generalization at this time
- Do not try to differentiate between objects and attributes at this time
Example Problem Statement:

- Create a payroll system for restaurants and hotels. Make certain that the usual deductions are taken into consideration. The payroll must accommodate both salaried and hourly employees. The waiters are salaried, but the busboys are hourly employees. The payroll system must print checks weekly. The system will produce a payroll register which will be turned over to auditors monthly. Income and tax reports including W-2s must be prepared according to legal requirements. Reports concerning voluntary deductions will be prepared for various agencies on a quarterly basis.

- The payroll must treat part-time employees as hourly employees. Full-time employees and salaried employees may take advantage of the various company benefits, part-time employees may not. Restaurant employees will be able to eat meals at their restaurant but will have the cost of the meals deducted from their paycheck. Hotel employees will have room costs deducted if they live in the hotel. There are voluntary deductions and mandatory government deductions that must be taken into account.

Example Context Diagram from the Problem Statement:
Example Candidate Objects from the Problem Statement:

<table>
<thead>
<tr>
<th>SYSTEM</th>
<th>HOURLY EMPLOYEE</th>
<th>INCOME REPORT</th>
<th>QUARTERLY BASIS</th>
<th>MEALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAYROLL</td>
<td>WAITERS</td>
<td>TAX REPORT</td>
<td>TIMECARD</td>
<td>COST OF MEALS</td>
</tr>
<tr>
<td>RESTAURANT</td>
<td>BUSBOYS</td>
<td>W-2</td>
<td>VOL. DEDUCT. REPORTS</td>
<td>PAYCHECK</td>
</tr>
<tr>
<td>HOTEL</td>
<td>PAYROLL SYSTEM</td>
<td>REQUIREMENT</td>
<td>PART-TIME EMPLOYEES</td>
<td>HOTEL EMPLOYEES</td>
</tr>
<tr>
<td>DEDUCTION</td>
<td>CHECKS</td>
<td>REPORTS</td>
<td>FULL-TIME EMPLOYEES</td>
<td>ROOM COST</td>
</tr>
<tr>
<td>CONSIDERATION</td>
<td>PAYROLL REGISTER</td>
<td>VOLUNTARY DEDUCTION</td>
<td>COMPANY BENEFITS</td>
<td>MANDATORY DEDUCTION</td>
</tr>
<tr>
<td>SALARIED EMPLOYEE</td>
<td>AUDITOR</td>
<td>AGENCIES</td>
<td>RESTAURANT EMPLOYEES</td>
<td>ACCOUNT</td>
</tr>
</tbody>
</table>

Modify Initial List of Objects:

- Use the Context Diagram as a guide to:
  - Put aside
    - Objects representing the aggregate system
    - Actors not intimately involved with the system
    - These objects will generally only appear on use case, sequence, and context diagrams
  - Add
    - Actors not previously found in the system
    - Interface objects that flow into and out of the system

- Update the list of candidate objects:
Select Classes:

- Apply the following filters to sort and identify real classes
  - Group objects into classes
  - Remove candidate classes that are:
    - Redundant, irrelevant, attributes, operations, roles, events, implementation constructs, or vague
- Give each class a good, intuitive name
  - Make names descriptive and unambiguous
  - Establish style standards (e.g. plural vs. singular nouns)
- Start the Model Dictionary

Remove Redundant Classes:

- Two classes that show the same information are redundant
- Keep the most descriptive name

Example – checks and paychecks refer to the same object, but paycheck is a more descriptive name … therefore keep paycheck

Remove Irrelevant Classes:

- Remove classes with little or nothing to do with the problem
- An important class in one context may be irrelevant in another
- Judgment will be involved

Example – The cost of meals at a restaurant may have nothing to do with the payroll system unless employees are charged for meals by docking their pay.

Remove Classes that are Attributes:

- Attributes ~ Names that describe objects are attributes not classes

Example – Suppose the problem statement contained the following: “The employee name, address, and pay number shall be printed on the individual employee’s paycheck.”

The following nouns would have been listed in our original set of objects: employee name, employee address, and employee pay number.

These objects, during this step, might be identified as attributes of employee rather than as separate domain objects.
Remove Classes that are Operations:

- **Operations** ~ If a name describes an operation that is applied to a class then it is not a domain class in its own right

**Example** – Suppose the problem statement contained the following: “Submissions for meal and room costs shall be entered on employee timecards.”

The following nouns would have been listed in our original set of objects: submissions, meal costs, room costs, and timecard.

Submissions here, while expressed as a noun, is actually an operation performed on timecards, i.e. timecards are submitted.

Remove Classes that are Roles:

- Remove roles
  - A name should show the nature of the object
  - A name should not show the role it plays in an association

**Example** – Waiter is a role that a particular employee plays. The employee may also be a busboy or a dining room host. For a payroll system, the term employee is best.

Remove Classes that are Events:

- Often nouns are used to describe things from the dynamic view (e.g. events, times, frequency, conditions, states)
- Put them aside until considering the dynamics

**Example** – Quarterly basis indicates the frequency of a report. Other events might have been found in our problem statement, e.g. “Print checks weekly” could have been “Paychecks are printed once a week”, and “Trainees do not get paid for their vacation until their first anniversary.”
Remove Implementation Constructs

- Remove items that are not part of the real world
  - These items may be part of the implementation but are not appropriate in an analysis level class diagram
  - Data structures such as tables and trees are almost always implementation details

Example – CPU, subroutines, and algorithms. CPU may be an appropriate class if we are building a computer but not for modeling a payroll system.

Clarify or Remove Vague Classes:

- Rename classes that are not specific
- Look for classes with very broad scope
- Look for classes with poorly defined boundaries

Example – Company benefits needs more definition.

Refine the List of Candidate Objects:
Describe Classes in the Model Dictionary:

- The **Model Dictionary** or **Glossary** should include all modeling entities
- Establish it as soon as classes are identified
- Define a project-standard template for each entry type
- For each class, describe:
  - The scope of the class in the problem space
  - Assumptions or restrictions on membership or use of the class
  - Associations
  - Attributes
  - Important operations
  - Roles
- Keep definitions current as the Class Diagram develops

Build the Class Diagram:

- Establish the Class Diagram **early** in the process
- Begin with classes found during the “Select Classes” step
- Use the Class Diagram to stimulate input from experts and users
- Make the Class Diagram the repository for all collected information
- The Class Diagram will change continually throughout the process
- As each object modeling step continues, add the latest information to the Class Diagram
Identify Relationships:

- Identify associations among the classes
- Annotate the associations to clarify the kind of relationship that exists between the classes
  - **Aggregate** associations imply a whole/part relationship
  - **Dependency** associations imply that one class depends on the services of a second class
  - **Inheritance** associations imply that one class inherits the structure and behavior of another class
  - **Reflexive** associations imply that one instance of a class is related to another instance of the same class

- Remove associations
  - With eliminated classes
  - That are implementation-specific
  - That are irrelevant
  - That are redundant

- Estimate multiplicity for each association

- Refine associations
  - Correct misnamed associations
  - Clarify ambiguous associations
  - Reduce multiplicity of associations

- Associations and aggregations are stated as verbs or verb phrases in the problem statement
  - Physical location (next to, part of, contained in)
    - Example – Payroll register contains payroll information
  - Directed actions (drives)
    - Example – Taxes are deducted from paychecks
  - Communication (talks to)
    - Example – Employee specifies number of dependents
  - Ownership (has, part of)
    - Example – Company owns hotels and restaurants
  - Satisfaction of some condition (works for, manages)
    - Example – Employee works for the company

Explicit Relationships:

- Can be extracted from the problem statement, e.g.:
  - Payroll system accommodates salaried and hourly employees
  - System prints checks
  - System produces payroll register
  - Reports prepared for agencies
  - Full-time and salaried employees receive company benefits
  - Restaurant employees can eat meals
  - Cost of meals is deducted from paycheck
Implicit Relationships

- Associations may not be explicitly stated within the problem statement
- Implicit associations may be discovered during analysis
- Add any missing associations
  - Real-world knowledge of the application domain should be used
  - Include a domain expert in the process

**Example: Employee receives a paycheck each time period**

![Diagram](image)

- Examples of Implicit Relationships:
  - Employees receive paychecks
  - Employees submit timecard
  - Hours worked add to the paycheck
  - Voluntary and mandatory deductions are taken from paychecks
  - Payroll register reflects the information printed in the paychecks
  - Company consists of hotels and restaurants
  - Company employees work in restaurants and/or hotels

**Example Payroll System Associations:**

![Diagram](image)
Determine Attributes and Operations:

- Identify attributes
  - Nouns describing characteristics of classes
  - Association classes or qualifiers
- Identify operations
  - Verbs denoting actions or activities
  - Transformations or algorithms

Refine Model for Inheritance:

- Organize classes by refining the inheritance associations
  - Inheritance can generalize common parts of existing classes into superclasses
    - A bottom-up process
  - Inheritance can refine existing classes into specialized subclasses
    - A top-down process
- Determine if the generalization associations are disjoint or overlapping
- Annotate the incomplete generalizations

Example Hierarchy of Inheritance:
Test the Model:

- Trace access paths through the Class Diagram
  - When a unique result is expected, see if the model will yield that result
  - Think of questions that might be asked and see if the solution models them
  - For any questions that cannot be answered, alter the model
- Trace the answers to ad hoc queries through the model
- After you have done all that, you can externally review the model

Have a review cycle or walk through with peers and end-users.

Iterate and Refine:

- The Class Diagram is usually refined after it is created
- Development involves frequent iteration
- Often parts of the model are at different stages of development
- Refinements will be made after the dynamic and functional perspectives of the application system are modeled

Documentation:

- Consider documenting the Class Diagram in a narrative
  - Guide reader through the model
  - Justify the model’s structure
  - Clarify meanings of names
  - Explain the reasons for each class and association
- The Model Dictionary or the Use Case descriptions may serve this purpose

Summary of the Object Modeling Process:

- The steps to constructing the Class Diagram are:
  - Collect Existing Information
  - Define Context Diagram
  - Identify Candidate Objects
  - Select Classes
  - Identify and Annotate Relationships
  - Determine Attributes and Operations
  - Refine Model for Inheritance
  - Test the Model
IV. CONCLUSION:

➢ Object-Oriented approach analyzes problems based on real-world concepts
➢ The fundamental unit of OO technology is the object
  o An object has structure
  o An object has behavior
➢ OO helps with a number of tasks:
  o Understand problems
  o Communicate with users
  o Model the system
  o Design the database and system
  o Document the system

Object-Oriented Approach:

➢ OO methodology is more than just programming
➢ Make a model of the problem domain before starting design
➢ The object view is the frame on which everything else hangs
➢ Design by adding detail and optimizing the analysis model
➢ OO designs can be implemented in OO languages, non-OO languages, and databases
➢ Development is iterative
➢ Good programming style is still important

Object-Oriented Technology:

➢ Can be used in conceptualization, analysis, design, and implementation
➢ Differs from other well known analysis techniques
➢ Introduces modeling as an abstraction of the real world
➢ Uses the object, dynamic, and functional views
➢ **Analysis** is the phase in which a real-world problem is examined to understand its requirements without planning the implementation
Basic Object-Oriented Concepts:

- **Objects** ~ Each object is separate even if all attribute values are the same
- **Classification** ~ Objects that have the same data structure and operations are in the same class
- **Operation** ~ A function or transformation that may be applied to objects in a class
- **Method** ~ Implementation of an operation for a specific class
- **Polymorphism** ~ Same operations are implemented differently on different classes of objects, related to a common superclass or interface
- **Inheritance** ~ Sharing of attributes and operations among classes based on a generalization hierarchy
- **Aggregation** ~ A special association between a whole and its parts

Object Modeling:

- Choose the objects and classes
- Develop the model dictionary
- Select the associations and aggregations
- Determine the attributes of objects and links
- Organize into a hierarchy of inheritance
- Capture the model in class diagrams
- Test the model and refine it

Object Modeling Tips:

- Object view content is driven by relevance to an application
- Strive to keep the diagrams simple
- Choose names carefully
- During analysis, avoid implementation detail
- Do not try to perfect multiplicity too early
- Avoid deeply nested generalizations
- Challenge one-to-one associations
- Review the model with customers and users
- Document model components in the model dictionary
Use Object-Oriented Methodology:

- **In Analysis**
  - Start from problem statement
  - Work with the user
  - Concisely model *what* the system must do, not *how* to do it
- **In System Design**
  - Make decisions about the overall architecture
  - Organize the system into subsystems
  - Allocate resources
- **In Object Design**
  - Add implementation details
  - Add data structures and algorithms to each class
- **In Implementation**
  - Translate classes and associations into code

**Summary of Object-Oriented Benefits:**

- Object-oriented analysis reduces risk by assuring an understanding of requirements and interfaces
- Models improve communication with management and the customer
- OO methodology produces documentation as the system develops
  - No need for an extensive documentation phase after development is completed (XML-based systems can auto-generate using XSL)
- Applications are more adaptable to changing business needs

V. REFERENCES:

- **The Object-Oriented Thought Process**
  Matt Weisfeld
  SAMS Publishing

- **Designing Flexible Object-Oriented Systems with UML**
  Charles Richter
  Macmillan Technical Publishing