Object-oriented methodologies

- Object-oriented methodologies are part of a paradigm shift that has been percolating in the software engineering community for many years.
  - The long-term nature of software development and the extensive experience required to successfully develop software makes it difficult for any paradigm shift to occur suddenly.
- Object design can support atomic actions and better recovery.
- A software product can be characterized as a model of the real world, a world that is constantly changing [Schach 93].
  - A strong case can be made that effective object-oriented modeling and implementation can greatly facilitate software maintenance.
  - This is an enabling technology that allows software changes to keep pace with system changes.
Distributed computing is an area that offers significantly new and powerful capabilities at the cost of additional complexity.

Object decomposition and modeling seem naturally suited for distributed computer systems.

In the previous chapter, the importance of interfaces in a distributed system was discussed.

The object-oriented paradigm fully supports the design of common interfaces for heterogeneous objects.

Message passing and other interobject communication methods are fully supported.

In this lecture we will provide an overview of the object-oriented paradigm, common methodologies, how it is an outgrowth of discrete-event simulation, and then its applicability to distributed systems.
Modeling Real-World Systems

• The term “paradigm shift” is a consultant’s best friend. The hype associated with object-oriented methods has exceeded the hype of structured techniques.

• The “software crisis,” the inability of software development to keep pace with demand, has been well documented.
  – Software products continue to be late, over budget, and deficient in meeting stated requirements.
  – Modeling systems has been difficult because real-world systems do not directly map to current functional decomposition strategies.
  – In a distributed system this is even more bothersome since functional decompositions do not easily lend themselves to implementation on a distributed system.
  – Even in a distributed problem domain such as a neural net, a functional decomposition is unlikely to produce an easy mapping to a distributed design.
Software engineering is a discipline which overlaps and is intertwined with simulation.

Implementing a simulation model on a computer is a specialized application of software engineering.

The figure at right shows a simple relationship between simulation and software engineering.

The iterative nature of the process is noted by the bi-directional arrows.
Why we consider OOA here

• System modeling is the key to simulation design.
  – Modeling is directly related to the software engineering design phase.
  – Hence, object-oriented analysis and design techniques are of interest to both the simulation designer and the software engineer.

• Compared to the more traditional functional decomposition methods, object-oriented methodologies make it easier to decompose a problem in which the design and the implementation more closely resemble the real-world problem domain.
  – An implementation in which the computer-oriented structures are closely tied to real-world-oriented structures is inherently easier to maintain.
  – Any worthwhile system will be changed – both during development and after deployment.
  – Conveniently, not only does the object-oriented paradigm support real world modeling, it also supports partitioning and allocation across multiple processors.
The term “object-oriented” is used in many contexts. System life-cycle activities of interest are:

- object-oriented analysis (OOA)
- object-oriented design (OOD)
- object-oriented programming (OOP)

Increasingly, OOA and OOD are seamlessly integrated and are often referred to as object-oriented analysis and design (OOAD).

There exists an entire alphabet soup associated with the object-oriented paradigm.

- Object-oriented environments, object-oriented graphical user interfaces, object-oriented databases, and a seemingly never ending list of applications and techniques which can have “object-oriented” added as a prefix.
- Some OO labels have been applied more correctly than others [Lawlis 92].
- We will confine ourselves to defining the software life-cycle and development aspects of the object-oriented paradigm.
• **Class** – description of a collection of like “things” (objects), including their structure and behavior; description of a set of objects that share a common structure and a common behavior [Booch 91]. Classes provide a template which includes a set of attributes and a set of services (*methods* or *operations*).

• There are two general types of classes: classes of instances and classes of classes (sometimes called *meta-classes*). Meta-classes have members which also must be classes. A class of computer workstations could be composed of subclasses of workstations such as DEC Alpha, RS6000, or Sun SPARC types (class of classes). A class of Sun SPARCs would have instances of individual workstations named Luke, Yoda, and Lando (class of instances).
Instance

- **Instance** – description of a particular member of a class (object). The term object is sometimes used with the same meaning, but some authors extend the definition of object.
- We prefer using *instance* because it is less ambiguous.
  - An instance assumes all characteristics of the class unless specifically overridden and has a “state” defined by the information known about it (specific values of its attributes).
  - Instance example: A Macintosh IIcx is not an instance of a class of computers (it’s a subclass).
  - A specific Mac named Deathstar is an instance of a Macintosh IIcx model computer.
  - Each instance has a unique identity. An instance (object) has identity, state (determined by the values of its attributes), and behavior.
  - The structure and behavior of similar objects are defined in their common class. Any number of Mac IIcxs may be instantiated.
  - In such a case each instance will have its own unique state information. It is possible for a class or subclass to have only one instance.
• **Attributes** – definition of the structure of the data in a class or instance; some data or state information for which each instance of a class has its own value.

  - This is similar in nature to the attributes of a database entity. Usually fields of a record define the structure of the class (a data structure definition). Attribute example:

```pascal
type printer is
  record
    Speed : pages_per_minute;
    Resolution : dots_per_inch;
    Memory : megabytes;
    Paper : boolean;
  end record;
```
Behavior

- **Behavior** – description of the actions and reactions of classes and instances. Behavior is how an object acts and reacts, in terms of its state changes and message passing.

- Behaviors are often implemented by procedure or function calls. An object sends a message to a service (its own service or that of another object) to trigger an action. The service which responds to the message may take one or more of the following actions:
  - Examine the state of the object to which it belongs (one or more attributes).
  - Change the state of the object to which it belongs (one or more attributes).
  - Trigger another message to another service.
Behavior Example

• Behavior example:
  – A procedure call is made to the service, which prepares a file for printing and dispatches that file to the printer.
  – That is, a message is sent from some other object and received by the printer object.
  – A check is made to ensure that the printer has paper.
  – The state of the object is examined via a call to a service of the object (call to a boolean function.)
  – The file is printed. The printer’s paper count is decremented, changing the state of the object.
Relationships – description of organization or structure among classes and instances (including inheritance).

- Relationships describe how classes and objects relate to one another.
- They demonstrate that an object is an instance of a particular class.
- Relationships can show that a class or object depends on another class or object in order to accomplish its behavior.
- Two relationships of particular importance are the whole-part relationship and the generalization-specialization relationship.
Abstraction

- *Abstraction* is the ability to separate objects and operations on those objects.
- Abstraction is a key concept in any modeling or design effort.
  - The system designer concentrates on the essential aspects of an entity and is not concerned with unimportant properties.
  - A high-fidelity simulation of automobile traffic is not likely to model an automobile’s cigarette lighter because it is unlikely to have a major impact upon traffic patterns.
  - Hence the model automobile is an abstraction that includes only the characteristics necessary to ensure fidelity with the real-world system.
Encapsulation

- **Encapsulation** is the process of hiding an object’s internal implementation details while providing an external interface which is accessible to other objects.
- One way to look at encapsulation is in terms of *coupling* and *cohesion*.
  - Cohesion refers to the degree of interaction within an object.
  - Coupling is the degree of interaction between objects.
- Encapsulation:
  - keeps related content together;
  - it minimizes traffic between different parts of the work;
  - and it separates certain specified requirements from other parts of the specification that may use those requirements [Coad 91a].
Inheritance

- Inheritance allows new abstractions to acquire properties from existing abstractions.
- Inheritance is a generalization-specialization relationship.
- Inheritance is a relationship among classes wherein one class shares the structure and/or behavior defined in one (single inheritance) or more (multiple inheritance) other classes.
- In an inheritance relationship, the specialized classes have many of the same attributes and services as their superclasses (parent classes), but they can also extend these attributes and/or extend the services.
Inheritance Examples

Multiple Inheritance

Single Inheritance
• Whole-part relationships occur when a set of classes or objects are a part of another class or object.
### Modified CRC Cards – Object Definition

<table>
<thead>
<tr>
<th>Name of Object</th>
<th>Inherits From</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes/Logical Properties</th>
<th>Traces</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Provided Services</th>
<th>Traces</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Contracted Services</th>
<th>Objects</th>
<th>Traces</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Card Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

- Object modeling cards are adapted CRC cards with expanded information content. The following actions lead to the development of the object modeling cards. (see above):
  - Determine different types of objects.
  - Accumulate attributes.
  - Accumulate provided services.
  - Accumulate contracted services.
Polymorphism

• **Polymorphism** means to have many forms.
• In terms of object-oriented programming languages, it refers to the ability for a language construct to assume differing types or to interface with objects assuming different types.
  – *Parametric polymorphism* uses types as parameters in generic type definitions [Khosh 90].
  – *Overloading* is another form of polymorphism. An overloaded operator can be applied to a different type.
    • For example, an overloaded “+” operator may be applied to integers and then applied to floating point numbers.
Object-Oriented Babylon

• Booch
  – Ada-based
  – Micro-development Process
  – CRC (class, responsibility, collaboration) Cards

• Coad and Yourdon
  – Differentiated OOA and OOD strongly
  – Class & Object notation

• Rumbaugh, Blaha, Premerlani, Eddy and Lorensen
  – OMT
  – Object, Dynamic, Functional Models

• Rubin and Goldberg
  – Object Behavior Analysis
## Methodology Summary

<table>
<thead>
<tr>
<th></th>
<th>Booch</th>
<th>Coad &amp; Yourdon</th>
<th>Rumbaugh et al</th>
<th>Rubin &amp; Goldberg</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OOA Process</strong></td>
<td>XXXXX</td>
<td>XXXXX</td>
<td>XXXXX</td>
<td>XXXXX</td>
</tr>
<tr>
<td><strong>OOD Process</strong></td>
<td>XXXXX</td>
<td>XXXXX</td>
<td>XXXXX</td>
<td>XXXXX</td>
</tr>
<tr>
<td><strong>Representations</strong></td>
<td>XXXXX</td>
<td>XXXXX</td>
<td>XXXXX</td>
<td>XXXXX</td>
</tr>
<tr>
<td><strong>Complexity Management</strong></td>
<td>some</td>
<td>some</td>
<td>some</td>
<td></td>
</tr>
</tbody>
</table>

See Monarchi & Puhr, CACM Sept ’92 for details
Unified Modeling Language (UML)

- Unification of:
  - Booch (Rational)
  - Rumbaugh, et al/OMT (General Electric)
  - Jacobson/Use-Cases (Objectory)

- OOPSLA ’95 – Unified Method version 0.8

- Standardized by the Object Management Group (OMG)


- UML 2.0 RFP
UML Overview

• UML standardizes four types of structural diagrams:
  – Class diagram
  – Object diagram
  – Component diagram
  – Deployment diagram

• Also five types of behavioral diagrams:
  – Use Case diagram
  – Sequence diagram
  – Collaboration diagram
  – Statechart diagram
  – Activity diagram

• And three types of model management diagrams:
  – Package diagram
  – Model diagram
  – Subsystem diagram
**Use Cases (Jacobson)**

- **A Use-Case Model of an ATM System**
- **Actors are the Environment of the System**
- **Use-Cases Specify the System**

_A use case specifies a sequence of actions, including variants, that the system can perform and that yields an observable result of value to a particular actor._
Dynamic model notation

- Based on Harel’s state charts [Harel 87].
- A special form of state transition diagrams with substates.

Rumbaugh et al give the following example:
- An automated teller machine may want to split control and then re-synchronize control.
- After dispensing cash, the machine does not want to reset until the customer has retrieved both his cash and his card.
- The order in which the customer accomplishes these actions is unpredictable.
- In complex applications, control may be split into multiple threads of control. Each thread of control may have more than one substate.
Processes are usually necessary to run executable computer programs. These processes are created by the operating system and usually run in a dedicated memory space with priority usage of the processor(s).

“Threads" are often required by many applications, particularly for batch or asynchronous processing.

Material from: http://www.therationaledge.com/content/may_01/t_activity2_bl.html
Process View Extensions

• <<interrupt>> Stereotype. Batch processing must allow for interruptions of the processing flow for input of information and control of the system.
  – When this stereotype is applied to the standard activity element, it indicates that the activity will generate a system interrupt to the current processing. These elements are colored red to draw particular attention to this type of critical system interaction.
  – Interrupts can occur on both processes and threads if allowed by the deployment operating system.
• <<idle>> Stereotype. All batch processing activities, user interface activities, and asynchronous communications require the ability of the process or thread to go into an idle state. While in this state, the process/thread requires limited processor time and can permit other system activities to proceed.
• The <<idle>> stereotype applied to activities that will enter into this "wait" state.
More Process View Extensions

• <<thread>> Stereotype. Many processes, particularly those involved in complicated batch processing, will spawn lightweight processes that run within the parent processing memory space.
  - These are referred to as process threads.
  - The <<thread>> stereotype is used to indicate that an activity is related to the creation, suspension, resumption, or destruction of a system thread.

• <<process>> Stereotype. All computer software is composed of processes that are loaded into and run on a hardware CPU.
  - The <<process>> stereotype is used to indicate an activity that will result in the creation, suspension, resumption, or destruction of a system process.
Multiple Threads in Swimlanes

Distributed Simulation – (6) Object-Oriented Methodology
Applying O-O designs to dist. systems

- A distributed system can be modeled in terms of objects and threads [Martin 94]. In this context we view an object as a set of defined states, a current state, and a set of operations.

- Two approaches to concurrency in object-based languages are defined in [Kramer 94]:
  - The **thread model of concurrency** (multiple threads of execution within an object).
  - The **active object model of concurrency** (the objects themselves are considered active but sequential).

- The **active object model** of concurrency restricts the grain of concurrency to coincide with an object.

- The **thread model of concurrency** permits objects to create new threads of execution by invoking commands such as cobegin .. coend.
  - Kramer, Magee, Sloman, and Dulay note that “Each thread can in turn send messages to invoke methods on other server objects, which could themselves introduce further concurrency – thereby forming a ‘tree’ of execution threads through objects.”
Modularity

- **Modularity** is a fundamental principle of software engineering.
- The object-oriented paradigm provides major support for modular design.
- In order to define object interfaces, it is important that object behavior is defined as services required and services provided.
- In a distributed system, object interfaces can be defined as the set of typed ports representing the methods that the object provides and the set of typed remote port references representing the methods required from other objects.
Threads of Control

• In Martin, Pedersen, and Bedford-Roberts’ taxonomy [Martin 94], threads of control are defined to impose a partial ordering of events in the distributed computing system.

• Objects in and of themselves have no inherent notion of time.

• The top level decomposition is made in three parts: threads, object properties, and separation.

• A thread of control is an execution series predefined by the programmer.

• When more than one thread is executing at the same time, we call these *concurrent* threads.
Object Properties for Objects in Isolation

- **Persistent** – A persistent object does not depend on a thread for its existence.
- **Restorable** – A restorable object can have prior states restored under programmer control.
- **Replicated** – A replicated object has multiple copies in the system that a client object can identify as a single object.
- **Partitioned** – An object is partitioned if the object’s state and methods are on different nodes in the system.
- **Protected** – Services that allow owners to selectively protect their objects from other users.
- **Existent** – Creation and destruction of objects.
- **Separation** is the trade-off between optimal integration of interacting objects and the flexibility achieved through non-integration. Separation encompasses identification, communication, partial failures, migration, and heterogeneity.
Open Systems

- Few vendors will admit to selling anything but an “open” system.
- However, the phrase “open system” is almost as abused and overused as “object-oriented.”
- An accepted definition for an open system is a system with behavior that can be easily modified and extended.
- Openness can be achieved dynamically through reactivity or statically through modularity [Wegner 92].
  - A reactive (interactive) system can react to stimuli by modifying its state and emitting a response.
  - A modular (encapsulated) system is one in which the number of components and/or functionality can be statically extended by an external agent.
- Most flexible open systems are both reactive and modular. Since object-oriented systems are both reactive and modular, they are considered strongly open.
Summary

- The object-oriented paradigm is not really new.
- Many of these concepts were being discussed by the Simula community nearly thirty years ago.
- Object-oriented modeling concepts make it possible for software developers to communicate better with the customer.
- Timothy Budd of Oregon State University observes that in an object-oriented program, the various entities in the universe are described, as well as how they will interact with one another, and then set in motion.
- This is similar to the methodology long used in discrete-event simulation.
Budd writes, [Budd 91]

“In brief, in a discrete event simulation, the user creates computer models of the various elements of the simulation, describes how they will interact with one another, and sets them moving. This is almost identical to the average object-oriented program, in which the user describes what the various entities in the universe for the program are, and how they will interact with one another and finally sets them in motion. Thus in object-oriented programming we have the view that computation is simulation.”