**SE203b: OO Design for Software Engineers**

**W8: OO Design Approach**

*Statechart, Activity, Components & Node Diagrams*

*Architectural frameworks*

---

**The Road Map**

- Introduction to Software Design
- Software Design Approaches
- Introduction to OO Paradigm
  - Software Design with OO Paradigm
  - Patterns in Design and Architecture
OO Software Requirements & Design Process

- Requirement capture using Use Cases
  - Use Case Diagrams
  - Use Case Description
- Analysis & Design Model
  - Class Diagrams
  - Data Dictionary
  - Interaction Diagrams
    - Statechart Diagrams
    - Activity Diagrams
  - Implementation Model
  - Deployment Model

Workflows and Models

- Requirements
- Analysis Design
- Implementation
- Test
**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

**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
Sequence Diagram

- Captures dynamic behavior (time-oriented)
- Purpose
  - Model flow of control
  - Illustrate typical scenarios

Some Reusable OO Design Patterns

- Façade
- Adapter
- Composite
- Proxy
- Observer
- Abstract Factory
Statechart Diagram

- Captures entity behavior (event-oriented)
  - Purpose
    - Model object lifecycle
    - Model reactive objects (user interfaces, devices, etc.)
States & Events

- At any given point in time, the system is in one state
  - It will remain in this state until an event occurs
    - that causes it to change state

- Events invoke changes in state
  - Final state due to a given event depends upon the initial state
  - An event separates two states

- A transition represents a change of state in response to an event
  - It is considered to occur instantaneously
  - The label on each transition is the event that causes the change of state

- Special states:
  - A black circle \( \bullet \) represents the start state
  - A circle with a ring around it \( \bigcirc \) represents an end state

Actions & Activities

- Events trigger operations in the state diagram
  - Two types of operations
    - An action is an instantaneous operation and not interruptible
      - i.e., its internal structure is not important to model
    - An activity is an operation that takes time to complete
      - it can be interrupted
Basic UML Statechart Diagram

Object lifecycle - General Model

Handling depends on specific request type

void:offHook ();
{busy = true;
obj.reqDialtone();
...};

Create/Initialize Object

Handle Request

Terminate Object
Object lifecycle and State Machines

- Direct mapping:

  - **Handle Event**
  - **Create/Initialize Object**
  - **Wait for Event**
  - **Terminate Object**

![Diagram showing state transitions for a lamp](image)

Actions on Entry to States

- Entry and exit actions
  - **Entry Actions**: Performed whenever the state is entered
    - Notation: `entry/action name`
    - Useful when all transitions entering a state share a common action
  - **Exit Actions**: Performed whenever the state is exited.
    - Notation: `exit/action name`
    - Useful when all transitions leaving a state share a common action.
Order of Actions: Simple Case

- **Exit** actions *prefix* transition actions
- **Entry** action *postfix* transition actions

Resulting action sequence:

`printf("exiting");`

`printf("to off");`

`lamp.off();`

Resulting action sequence:

`printf("exiting");`

`printf("to off");`

`lamp.off();`

State ("Do") Activities

- An activity is **associated with a state**
  - The notation: *"do/activity name"*
- It forks a concurrent thread that executes until:
  - the activity completes or
  - the state is exited
    - through an outgoing transition
  - the activity can be a **continuous** operation
    - e.g., "do: display main screen"

Error

`entry/printf("error!");`

`do/while (true) alarm.ring();`
Conditions

- A state is often associated with the value of an object satisfying some conditions
  - A condition is a Boolean function of object value
    - e.g., the amount within predefined limits
- A guarded transition
  - A transition labeled by an event with a condition
    - A guarded transition fires when its event occurs and the condition is true
    - It showed as a Boolean expression in brackets following the event name
      - e.g., insert card [readable]

Guards

- Conditional execution of transitions
  - guards must be side-effect free
State Generalization

- In a nested state diagram, a state in a higher-level is a form of **generalization** of the states in its lower-level state diagram.
- If it is in state $X$,
  - it is in exactly one of the state $X_1; \ldots ; X_n$

Concurrency

**Aggregation concurrency**

- A state diagram for an aggregate is a collection of state diagrams, one for each element.
- The elements are inherently concurrent
  - Objects can change state independently or interact
- **Synchronization** constraints are implemented by
  - conditions in guarded transitions
"Car" Components: Example

Statecharts: Recap

- Used to model event-driven (reactive) behavior
  - well-suited to the server model inherent in the object paradigm
**Activity Diagram**

- Captures dynamic behavior *(activity-oriented)*
  - **Purpose**
    - Model workflows
    - Model operations
  - **Transitions** are caused by internal events
  - The objective
    - To help understanding the flow of work *(operations)*
      - that an object or component performs
      - often associated with several classes especially concurrent activities

**Kinds of Action States**

- **Action (State)**
  - Action

- **Submachine (State)**
  - Submachine

- Just like their state machine counterparts (simple state and submachine state) except that
  - **transitions** are caused by internal events and
  - are taken when the step is finished,
**Example**

POEmployee

\[\text{sortMail()}\]
\[\text{deliverMail()}\]

\[\text{POEmployee.sortMail} \rightarrow \text{POEmployee.deliverMail}\]

\[\text{deliverMail}\]

\[\text{Check Out Truck} \rightarrow \text{Put Mail In Boxes}\]

**Activity Graph as Method**

- All activity graphs are methods for operations
- Application is completely OO when all action states invoke operations
**Coordinating Steps**

- Initial state
- Final state
- Branch/merge
- Fork
  - join

**Coordinating Steps (Cont.)**

- For modeling conventional flow chart decisions.

```
Calculate Cost ➔ [cost < $50] ➔ Charge Account
       |                      |                      |
       |                      |                      |
       |                      |                      |
       |                      |                      |
       |                      |                      |
       |                      |                      |
       [cost >= $50] ➔ Get Authorization ➔ Charge Account
```
**Representing Concurrency**

- Concurrency is shown using forks, joins and rendezvous.

  - A **fork** has one incoming transition and multiple outgoing transitions.
    - The execution splits into multiple concurrent threads.

  - A **join** has multiple incoming transitions and one outgoing transition.
    - The outgoing transition will be taken when all incoming transitions have occurred.
    - The incoming transitions must be triggered in separate threads.
    - If one incoming transition occurs, a wait condition occurs at the join until the other transitions occur.

  - A **rendezvous** has multiple incoming and multiple outgoing transitions.
    - Once all the incoming transitions occur, all the outgoing transitions may occur.

---

**Example**

Example of a process flow diagram showing the construction process:
- **Install Foundation**
- **Install Electricity in Foundation**
- **Install Frame**
- **Install Electricity in Frame**
- **Build Frame**
- **Put On Roof**
- **Install Walls**
- **Install Electricity Outside**
- **Inspect**
Swimlanes

- The activity of an object might involve invocation of other objects.
  - The partition of activities among the existing objects can be explicitly shown using swimlanes

- Partitions are a grouping mechanism
  - Swimlanes are the notation for partitions
    - They do not provide domain-specific semantics

Activity Diagrams – an example with swimlanes

[Diagram showing flow of activities for handling a course registration request]
**Coordinating Steps**

- Synch state is inherited from state machines but used mostly in activity graphs.
- Provides communication capability between parallel processes.

**Convenience Features (Synch State)**

- Forks and joins do not require composite states.
- Synch states may be omitted for the common case (unlimited bound and one incoming and outgoing transition).
When to Use Activity Diagrams

- Use activity diagrams when the behavior you are modeling ...
  - does not depend much on external events
  - mostly has steps that run to completion
    - rather than being interrupted by events
  - requires object/data flow between steps
  - is being constructed at a stage when you are more concerned with
    - which activities happen,
      - rather than which objects are responsible for them (except partitions possibly)

Activity Diagrams: Recap

- Use Activity Diagrams for applications that are primarily control and data-driven, like business modeling ...
  - ... rather than event-driven applications like embedded systems
- Purpose
  - Model workflows
  - Model operations
**OO Software Requirements & Design Process**

- Requirement capture using Use Cases
  - Use Case Diagrams
  - Use Case Description
- **Analysis & Design Model**
  - Class Diagrams
  - Data Dictionary
  - Interaction Diagrams
  - Statechart Diagrams
  - Activity Diagrams
- Implementation Model
- Deployment Model

**Workflows and Models**

- Requirements
- Analysis & Design
- Implementation
- Test
Component Diagram

- Captures the **physical structure** of the implementation

- Components may be
  - implemented by **artifacts**
    - e.g., binary, executable, or script files (such as html)
Components and Classes

- Component is the physical implementation of
  - a set of classes

Components and Interfaces

- The component realizes the interface is connected to the interface using a full realization relationship
- A component accesses the services of the other component through its interface
Component Diagram

Example

Deployment Diagram

- Captures the topology of a system's hardware
- Shows the configuration of
  - run-time processing elements and the software components,
    - processes and objects that live on them
**Nodes and Components**

- A component may be deployed by a node or more
  - classes might be implemented by a component or more

**Deployment Diagram**

*Example (Cont.)*
Deployment Diagram

Example (Cont.)

- Client
  - videoStoreServer:AppServer
    - Catalog
    - ShoppingCart
  - OpenSourceBrowser

- DBServer
  - VideoStore:DB

OO Software Requirements & Design Process

- Requirement capture using Use Cases
  - Use Case Diagrams
  - Use Case Description
- Analysis & Design Model
  - Class Diagrams
    - Data Dictionary
  - Interaction Diagrams
  - Statechart Diagrams
  - Activity Diagrams
- Implementation Model
- Deployment Model

Architecture-centric
- Patterns
- Frameworks
**Architectural Patterns: Frameworks**

- The notion of patterns can be applied to software architecture
  - These are called
    - architectural patterns, or
    - frameworks, or
    - architectural styles
  - Each allows you to design flexible systems using components
    - The components are as independent of each other as possible

**Architectural patterns**
- Layered
- Model View Controller (MVC)
- Distributed
  - Client Server
  - Broker
- Pipes and filters

**Multi-Layer Style**

- In a layered system,
  - each layer communicates only with the layer immediately below it
  - Each layer has a well-defined interface used by the layer immediately above
    - The higher layer sees the lower layer as a set of services

- A complex system can be built by superposing layers at increasing levels of abstraction
  - It is important to have a separate layer for the UI
  - Layers immediately below the UI layer
    - provide the application functions determined by the use-cases
  - Bottom layers provide general services
    - e.g. network communication, database access
**Multi-layer Style**

*Example*

- User Interface
- Application Logic
  - OS Access
  - DB Access
  - Network Communication

**Example: 2-tier & 3-tier Architecture**

- Graphical User Interface
- Business Object Model
- Relational Database/External

**Design Patterns**
**Multi-layer Example**

Typical layers in an operating system:
- Kernel
- File System
- User Account Mngt.
- Application Programs
- Screen Display Facilities

**Deployment Diagram Example (Cont.)**

Diagram showing:
- Presentation
- Application
- Database

Client: `videoStoreServer:AppServer`

DBServer:

```
<database>
<Session>
<ShoppingSession>
<Catalog>
<ShoppingCart>

<videoStoreServer:AppServer>
```

```bash
videoStoreServer
```
Complex Internet system

Client
Dynamic HTML, JavaScript, Java plug-ins, source code enhancements

Server
Java, C, C++, JavaScript, CGI

Application Server
Java, C, C++, JavaBeans, CORBA, DCOM

Fulfillment System
Financial System
Inventory System
RDBMS Server

The Multi-Layer Design Principles

Divide and conquer:
- The layers can be independently designed

Increase cohesion:
- Well-designed layers have layer cohesion

Reduce coupling:
- Well-designed lower layers do not know about the higher layers
- the only connection between layers is through the interfaces (API)

Increase abstraction:
- you do not need to know the details of how the lower layers are implemented

Increase reusability:
- The lower layers can often be designed generically.
The Multi-layer Design Principles

Increase reuse:
  - You can often reuse layers built by others that provide the services you need.

Increase flexibility:
  - You can add new facilities built on lower-level services, or replace higher-level layers.

Design for portability:
  - All the dependent facilities can be isolated in one of the lower layers.

Design for testability:
  - Layers can be tested independently.

Model-View-Controller (MVC) Style

- to separate
  - the user interface layer
  - from other parts of the system

- The main elements
  - The model
    - contains the underlying classes whose instances are to be viewed and manipulated
  - The view
    - contains objects used to render the appearance of the data from the model in the user interface
  - The controller
    - contains the objects that control and handle the user's interaction with the view and the model
  - The Observable design pattern is normally used to separate the model from the view
The MVC Architecture

UI Example

MVC: Example
MVC: Example

The MVC Architecture Example

- View: viewed by actor
- Controller: receives actor events
- Model: notify about changes
  
  create and update
  
  modify
  
  viewed by actor
  
  receives actor events
  
  notify about changes
The MVC

**Design Principles**

**Divide and conquer:**
- The three components can be somewhat *independently designed*

**Increase cohesion:**
- The components have *stronger layer cohesion* by separating view and controller

**Reduce coupling:**
- The *communication channels* between the three components are *minimal*

**Increase reuse:**
- The *view* and *controller* normally make extensive use of reusable components for various kinds of UI controls

**Design for flexibility:**
- It is usually quite *easy to change the UI* by changing the *view*, the *controller*, or both

**Design for testability:**
- You can *test the application separately* from the UI

---

**Distributed Architecture**

[Diagram showing a distributed architecture with clients, LAN, WAN, and servers connected]
Client-Server & Multi-tier Styles

• There is at least one component
  • that has the role of server
    o waiting for and then handling connections
  • that has the role of client
    o initiating connections in order to obtain some service.

• A further extension is the Peer-to-Peer pattern
  • A system composed of various software components that are distributed over several hosts.

Client-Server

Peer-to-peer
Physical Application Architecture

The distributed architecture

**Design Principles**

Divide and conquer:
- Dividing the system into client and server processes where
  - Each can be **separately developed**.

Increase cohesion:
- **The server** can provide a **cohesive service** to clients.

Reduce coupling:
- There is usually **only one communication channel** exchanging simple messages.

Increase abstraction:
- Separate **distributed components** are often good abstractions.

Increase reuse:
- It is often possible to find suitable frameworks on which to build good distributed systems
  - However, client-server systems are often very application specific
The distributed architecture

Design Principles

Design for flexibility:
- can be easily reconfigured by adding extra servers or clients

Design for portability:
- You can write clients for new platforms without having to port the server.

Design for testability:
- You can test clients and servers independently.

The Broker Style

- Transparently distribute aspects of the software system to different nodes
  - An object can call methods of another object without knowing that this object is remotely located
  - CORBA and Web-services are a well-known open standards that support borkering.
Example of a Broker system

The Broker Architecture

Design Principles

Divide and conquer:
- The remote objects can be independently designed.

Increase reusability:
- It is often possible to design the remote objects so that other systems can use them too.

Increase reuse:
- You may be able to reuse remote objects that others have created.

Design for flexibility:
- The brokers can be updated as required, or the proxy can communicate with a different remote object.

Design for portability:
- You can develop clients for new platforms while still accessing brokers and remote objects on other platforms.

Design defensively:
- You can provide careful assertion checking in the remote objects.
The Pipe-and-Filter Framework

- A stream of data, in a relatively simple format, is passed through a series of processes
  - Each of which transforms it in some way.
  - Data is constantly fed into the pipeline.
  - The processes work concurrently.
  - The architecture is very flexible.
    - Almost all the components could be removed.
    - Components could be replaced.
    - New components could be inserted.
    - Certain components could be reordered.

Example of a pipe-and-filter system
**The Pipe-Filter Architecture**

*Design Principles*

Divide and conquer:
- The separate processes can be independently designed.

Increase cohesion:
- The processes have functional cohesion.

Reduce coupling:
- The processes have only one input and one output.

Increase abstraction:
- The pipeline components are often good abstractions, hiding their internal details.

Increase reusability:
- The processes can often be used in many different contexts.

Increase reuse:
- It is often possible to find reusable components to insert into a pipeline.

---

**The Pipe-Filter Architecture**

*Design Principles*

Design for flexibility:
- There are several ways in which the system is flexible.

Design for testability:
- It is normally easy to test the individual processes.
OO Software Requirements & Design Process

- Requirement capture using Use Cases
  - Use Case Diagrams
  - Use Case Description
- Analysis & Design Model
  - Class Diagrams
    - Data Dictionary
  - Interaction Diagrams
  - Statechart Diagrams
  - Activity Diagrams
- Implementation Model
- Deployment Model

Architecture-centric
- Patterns
- Frameworks