Modeling a Class’s Internal Structure

Composite Structures

- Models how objects create a big picture
  - How objects work together in a class
  - How objects achieve a goal
- Examples
  - Internal structures
  - Ports
  - Collaborations
- Recall aggregation (has-a) and composition (part-a) relationships
- Composite structure offer an alternate way of showing the relationships

When Class Diagrams Won’t Work

- Requirements
  - BlogEntry consists of an Introduction and a MainBody (composition)

Then, what about this?
When Class Diagrams Won’t Work (cont.)

- An example runtime view

Desired runtime view

Connectors

- Specify relationships between parts with multiplicity
- Enables communication between parts
  - Dynamic or run-time link between parts

Class Internal Structure: Parts of a Class

- Specification format
  - Parts: `<rolename>` : `<type>` in solid outline
    - A part is a set of instances that may exist in a single instance of the containing class at runtime.
- Multiplicity
- Comparison

Properties

- Referenced through `association`
- Shared with other classes in the system
- Notation
  - Dashed outline
- Example
Complex Relationships Between Parts

- Parts of the same type can play multiple roles

Runtime view

- Port
  - A point of interaction between a class and the outside world.
  - Represent distinctive ways of using a class
  - Commonly, associated with interfaces

- Provided interface
  - When a class realizes an interface

- Required interface
  - When a class requires another class or component that realizes the interface

Showing Patterns with Collaborations

- Collaborations show object interaction
  - Similar to object diagram but focused on representing the “role” of objects and textual description of their behavior
  - Good candidate for representing design patterns (solutions to common problems in software design)
  - Represented using a large dashed-lined oval enclosing all participants
  - Example: Chain of Responsibility (COR) pattern

Showing How a Class is Used (cont.)

- Example
Showing Patterns with Collab. (cont.)

- Alternate representation
- Notes
  - Collaborations are temporary runtime relationships between objects
  - An object is not bound to its role in a collaboration (can play different roles in different collaborations)
  - Objects in a collaboration are not owned by the collaboration; they can exist before and after the collaboration.

Need for High-Level Architectural Design

- Rare to jump directly from requirements to detailed class design
- Components
  - An encapsulated, reusable, and replaceable part of your software
  - Used to organize a system
  - Range in size from relatively small, up to a large subsystem
  - In UML, a component can do the same things a class can do.
- Examples
  - Loggers, XML parsers, online shopping carts

Component Diagram: Notations

- Drawn as a rectangle with the <<component>> stereotype
- Provided vs. Required interfaces
  - Ball and socket symbols; or
  - Stereotype notation
Showing Components Working Together

- Component dependencies

CMS Example

- CMS Example with Details

Classes that Realize a Component

- Model classes that implement functionality of a component
  - Similar to composite structure
  - Use the <<realizations>> compartment inside a component

Preferred Ports and Internal Structure

- Components can also have ports and internal structure.
Ports and Internal Structure (cont.)

- **Delegation connectors**
  - Show the connection between ports and internal parts that realize or use the component’s interfaces

![Delegation connectors diagram]

- **Assembly connector**
  - Show the connection between two components – one that requires an interface provided by the other component

![Assembly connector diagram]

Black-Box and White-Box Component View

- Classification depends on whether parts inside a component is modeled in a diagram.
  - Black-box view shows key components and their connection
  - White-box view shows how a component achieve its functionality through the classes it uses
- Use black-box to focus on large-scale architectural concerns in a component diagram.

![Black-box and White-box Component View diagram]

Organizing Models into Packages

Package Diagram

Lecture 25-26  Dr. Ilchul Yoon

Sources: Learning UML 2.0 (by R. Miles and K. Hamilton)
Manage System Complexity

- A software may contain a large number of classes
- Impose structure by organizing classes into groups
  - In UML, model classes with packages
    - OO languages already have similar concepts – e.g., package in Java, namespace in C#
  - Model package dependencies
  - Package diagram can organize any UML element – not just classes (e.g., use-cases)
- Other advantages
  - Easier to find classes
  - Divide tasks between developers
  - Access control

Namespaces and Visibility

- A UML package establishes a namespace.
  - To access an element in another package, need to specify fullscoped name – format: packageName:className
  - Namespace inheritance
    - Elements in a nested package can refer to elements in the containing package withoutscoping
    - Different to Java.
- Element visibility
  - Public
    - Elements with public visibility are accessible outside the package.
  - Private
    - Elements with private visibility are available only to other elements inside the package.
Importing and Accessing Packages

- If a package imports another package,
  - Elements in the importing package can use elements in the imported package without full-scoped name.
- Model with `<import>` stereotype annotated dependency arrow
- Can import specific element in another package

Importing and Accessing Packages (cont.)

- Only public elements of the imported package are available in the importing namespace.
- Import relation itself can be `public <<import>>` or `private <<access>>`.
  - Similar to public and private inheritance in C++

Managing Package Dependencies

- Package dependencies reduce stability of system
  - Changes in a package can cause its dependent packages to break.
  - Especially, avoid cyclic package dependencies
- Approaches for breaking cyclic dependencies
  - Factor out a new package
  - Reduce all into one package

Using Packages to Organize Use Cases

- Rolling up use cases into higher levels of system design help organize models.
  - Allow developers to see which actors interact with which system portions

use case groups
Modeling State-Dependent Object Behavior

- Often, object behaves differently depending on its state.
- State (machine) diagrams are heavily used for modeling:
  - Real-time/mission-critical systems
  - Devices whose behavior is defined in terms of state
  - Games
- Types
  - Behavioral state machine – Show states, transitions, and behavior
  - Protocol state machine – show states and transitions

Essentials

- Diagram elements
  - States – a condition of being at a certain time
  - Events (triggers state changes)
  - Transition – state changes (trigger|guard|behavior)
  - Initial pseudo-state
  - Final state

Transition Variations

- Use guards and triggers for modeling different types of state changes
  - Guard – Boolean condition that permits or blocks the transition
  - Only trigger
  - Use of guard to model choice
  - Guarded transition
  - Combined state diagram
  - Trigger after internal behavior
Modeling Internal Behavior

- **Internal behavior**
  - Behavior that happens while an object is *in a state*
  - Format: label/behavior
  - Entry and exit behaviors are not interruptible.

- **Internal transition**
  - Transition that causes a reaction *within* a state, but *doesn't* cause the object to change states
    - Format: trigger[guard]/behavior
  - Model reactions to events that *don't* cause state changes
  - Different to self transition!
    - Do not cause entry and exit behavior to occur

Composite States

- **Used to model concurrent states**
  - Contains 1 or more state diagrams where each belongs to a region
    - Both diagrams become active at receiving events and are interrupted if an exit trigger occurs.
    - Synchronous behavior – if sub-diagrams are working at the trigger, wait for termination

Choice, Fork and Join Pseudo-states

- **Choice**
  - Used to model alternative state changes

- **Fork and join**
  - Used to model branching into concurrent states and then rejoining

Signals

- Special icons for transitions to draw attention to transitions and transition behavior
  - Transition-oriented view
  - Used to emphasize sending and receiving signals
Protocol State Machines

- State diagram without behavior
  - Focused on events and state changes
  - Used for modeling communication protocol

Example

![State diagram for eliciting communication protocol](image)

Physical view of a System

- Show how software gets assigned to hardware and how the pieces communicate

Notations

- `<device>`: a hardware node
- `<artifact>`: a software artifact
  - Can also use document icon, instead.
  - Executable files
  - Library files
  - Source files
  - Configuration files

Dependencies between Artifacts

- Use of features

- Manifest components
Node

- A hardware of software resource that can host software or related files
  - Hardware nodes
    - Server
    - Desktop PC
    - Disk drive
  - Execution environment nodes
    - OS
    - J2EE container
    - Web server
    - Application server
  - Show runtime view with node instances

Communication between Nodes

- Model the communication between hardware nodes or execution environment nodes
  - Communication stereotype should be as high-level as possible because it is about communication in your system.

Deployment Specification

- A special artifact specifying *how another artifact is deployed to a node*
  - Represented by:
    - Drawing a dependency arrow from spec to the artifact
    - Attach spec to the deployment arrow.

When to Use a Deployment Diagram?

- Useful at all stages of design process
  - At high-level design stages, model initial sketches.
    - For example:
      - Architecture – web server, app server, database
      - Clients can access through a browser or a dedicated GUI interfaces
      - Web server is protected by with a firewall.
  - At later stages, add more specifics.