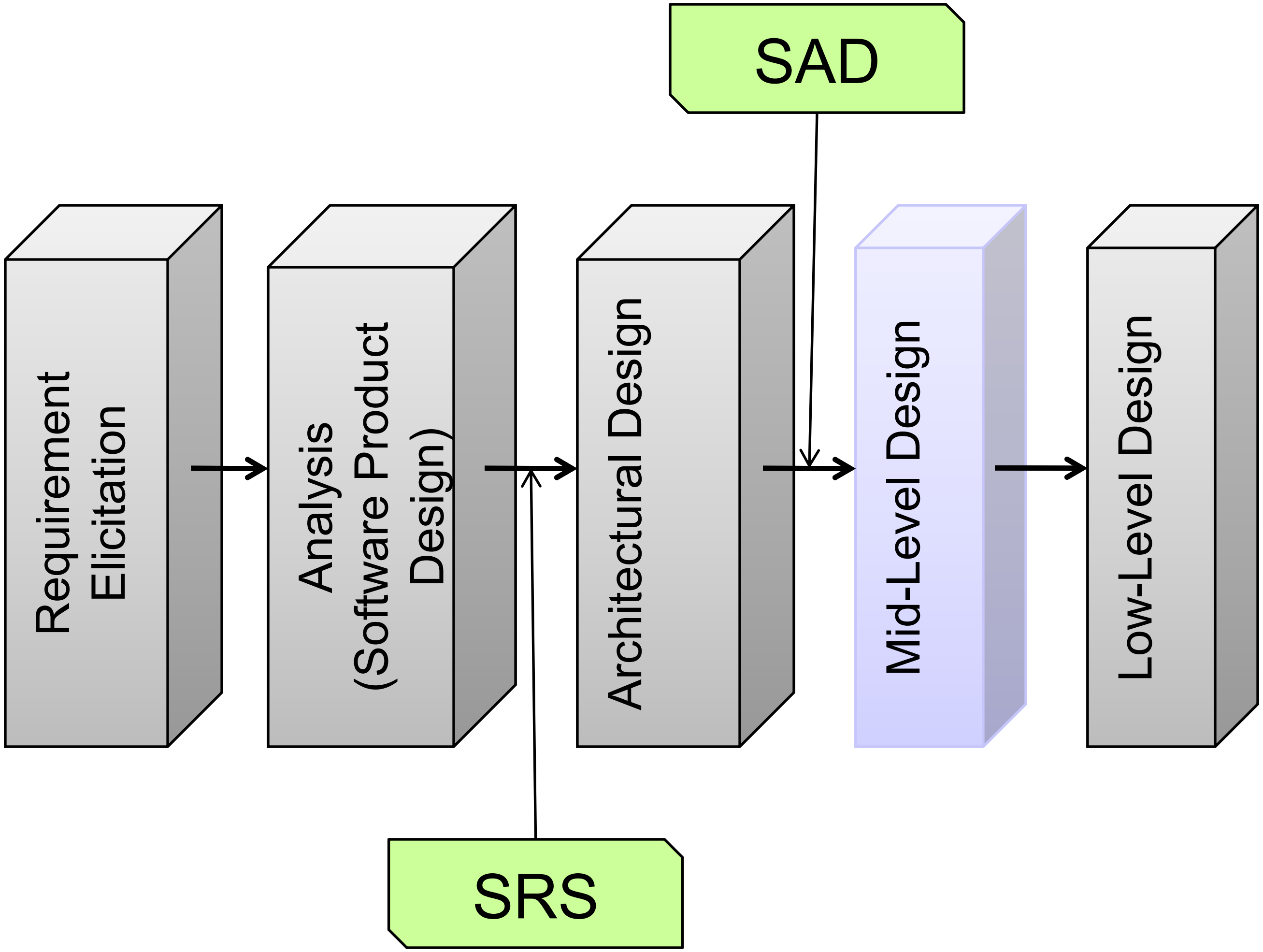
Introduction to Software Engineering

ECSE-321

Unit 10 – Mid-level Design

Detailed Design (Mid-Level Design)



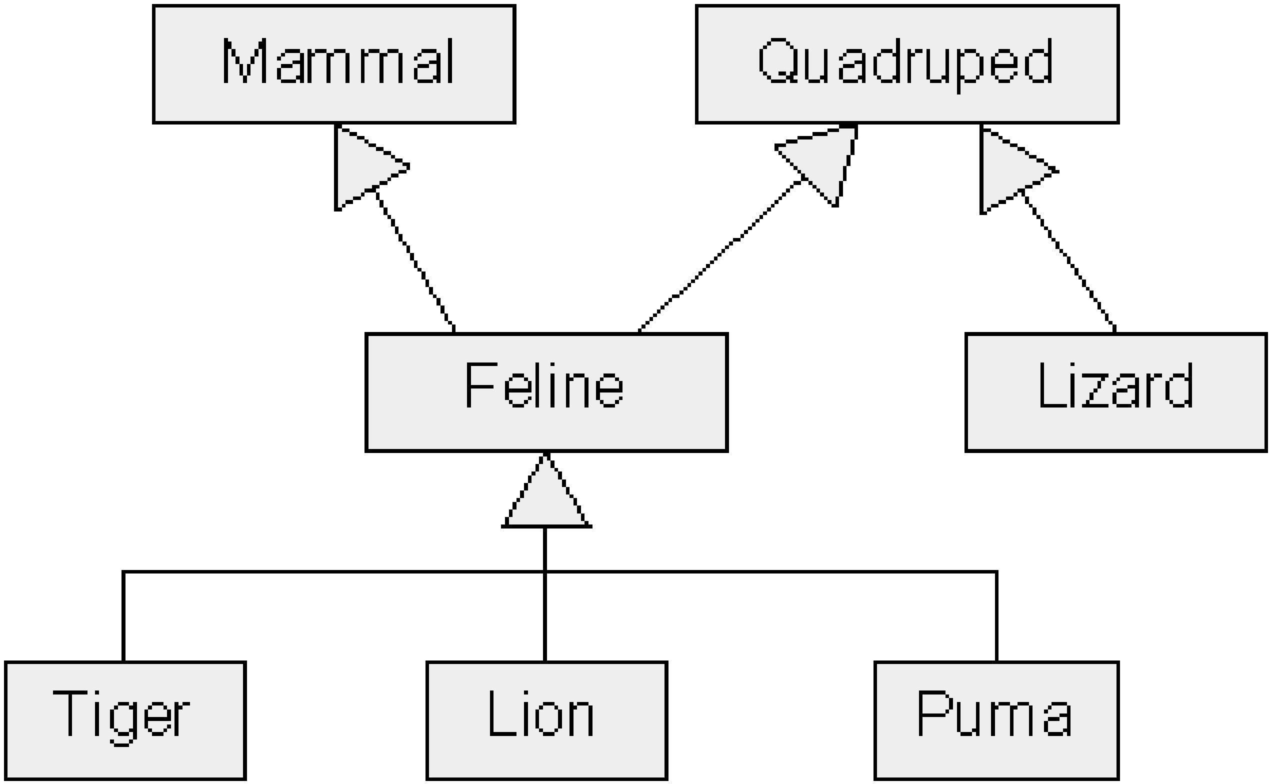
Additional UML Notation

- Mid-level design uses UML notation
- Some of notation – revision of previously discussed ones
- Additional variations for old notations

Generalization

- **Generalization** is the UML relation that holds between one model element (the *parent*) and another (the *child*) when the child is a *special type* of the parent.
  - Represented by a hollow triangle and lines
  - Triangle attaches to the parent and lines to the children
- Generalization is used in UML class diagrams to model inheritance.

Generalization Example



Generalization versus Association

- Generalization is a relation between classes.
- Associations represent relations on sets of class instances designated by the associated classes.
- Generalization is *not* a kind of association. They
  - Never have multiplicities,
  - Never have rolenames,
  - Never have names (they already have a name: generalization).

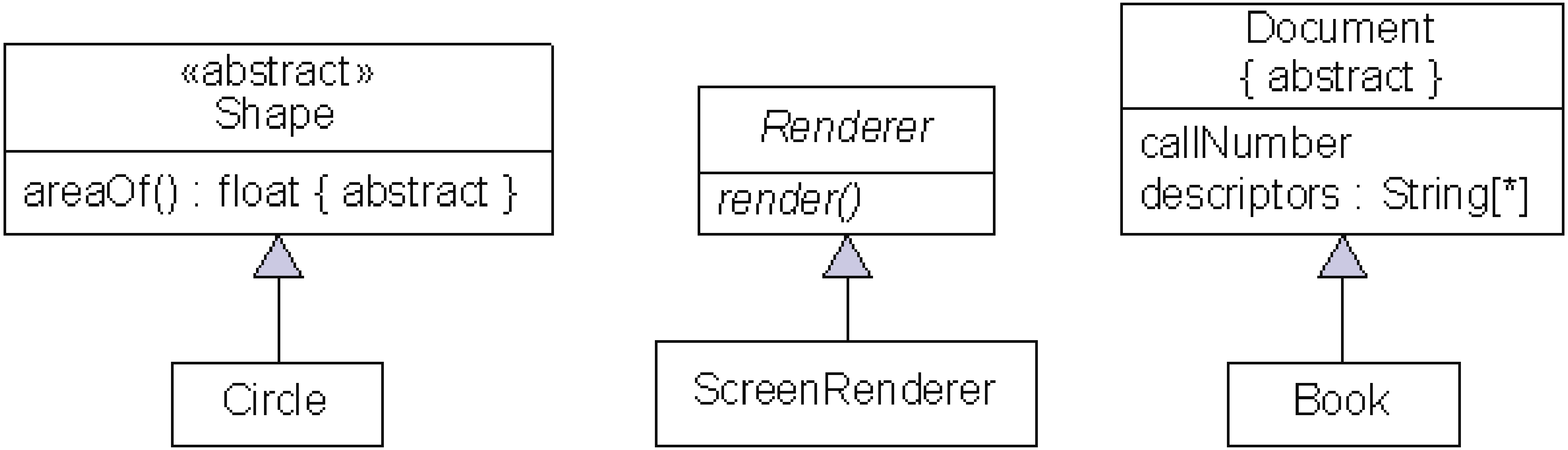
Abstract Operations and Classes

- An **abstract operation** is an operation without a body; a **concrete operation** has a body.
- An **abstract class** is a class that cannot be instantiated; a **concrete class** can be instantiated.
- A class
  - Must be abstract if it has an abstract operation;
  - May be abstract even if it has no abstract operations.

Using and Representing Abstract Classes and Operations

- Abstract classes force their subclasses to implement certain operations.
- Abstract classes are represented in UML by
  - Italicizing their names,
  - Stereotyping them «abstract» or
  - Giving them an {abstract} property.
- Abstract operations are represented in UML by
  - Italicizing their specification or
  - Giving them an {abstract} property.

Abstract Class and Operation Examples

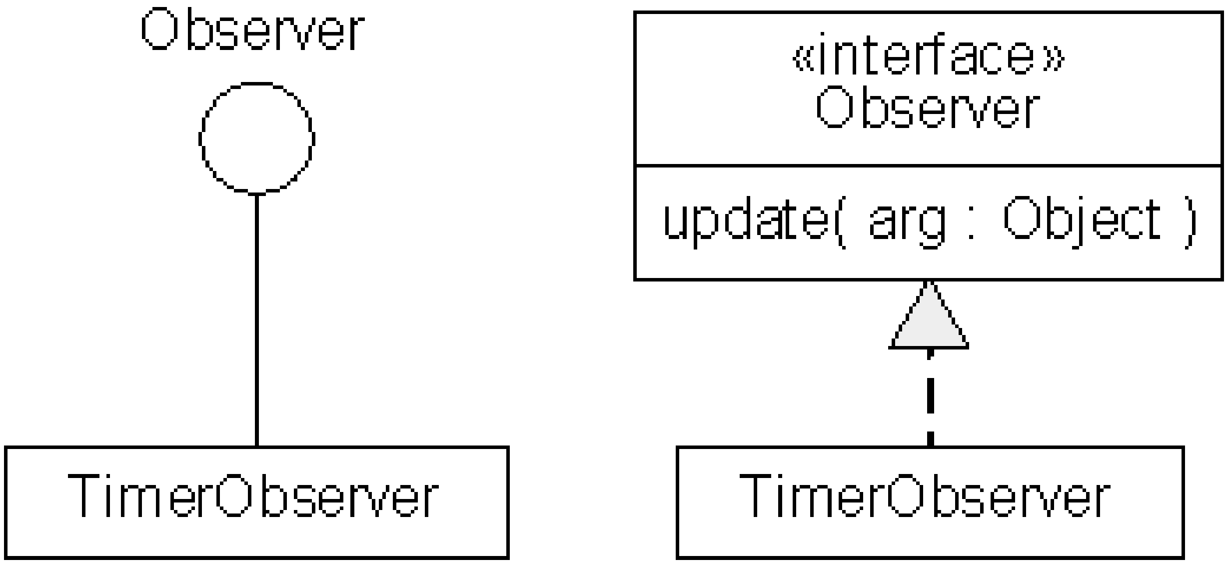


Interfaces

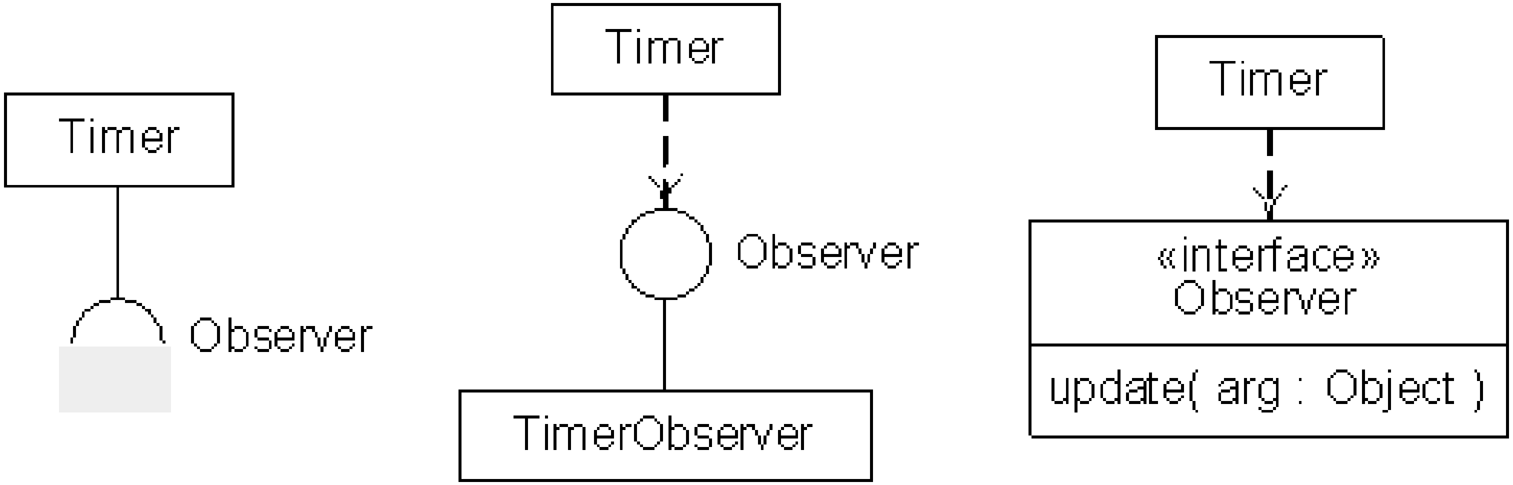
A UML **interface** is named collection of public attributes and abstract operations.

- *Provided interfaces*—realized by a class or component and represented by
  - A ball symbol or
  - A stereotyped class icon with a realization connector
- *Required interfaces*—needed by a class or component and represented by
  - A socket symbol or
  - A dependency arrow to a ball symbol or
  - A dependency arrow to a stereotyped class icon

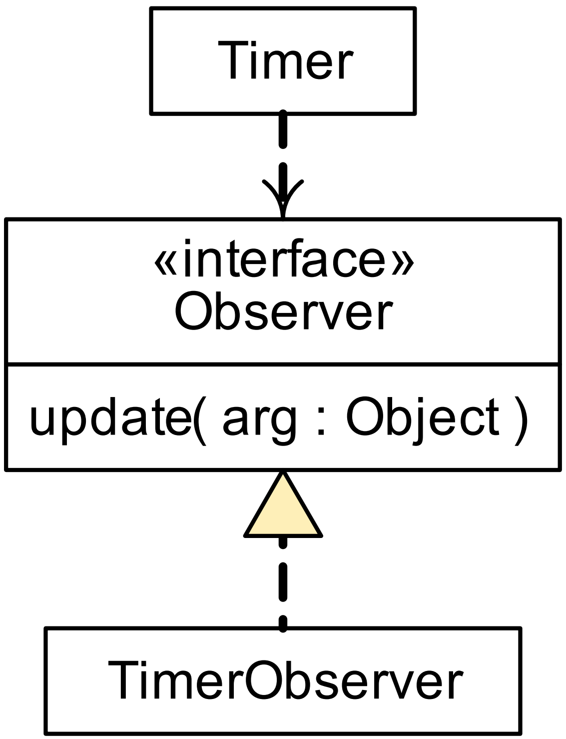
Provided Interface Notations

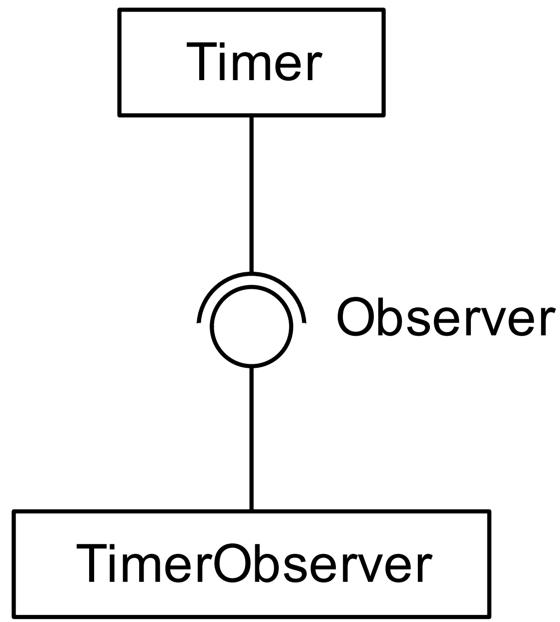


Required Interface Notations



Module Assembly Notations

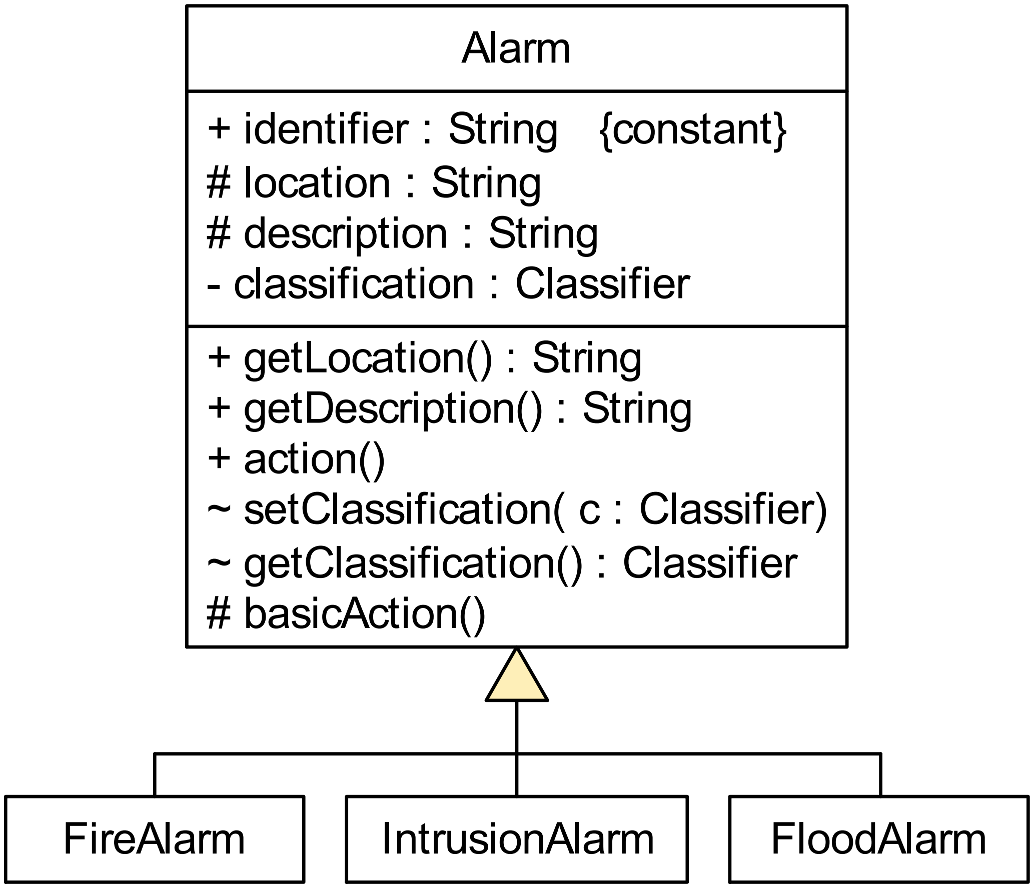




UML Feature Visibility

- *Public*—Visible anywhere that the class in which it appears is visible; denoted by +.
- *Package*—Visible anywhere in the package containing the class in which it appears; denoted by ~.
- *Protected*—Visible in the class in which it appears and all its sub-classes; denoted by #.
- *Private*—Visible only in the class in which it appears; denoted by -.

Feature Visibility Example



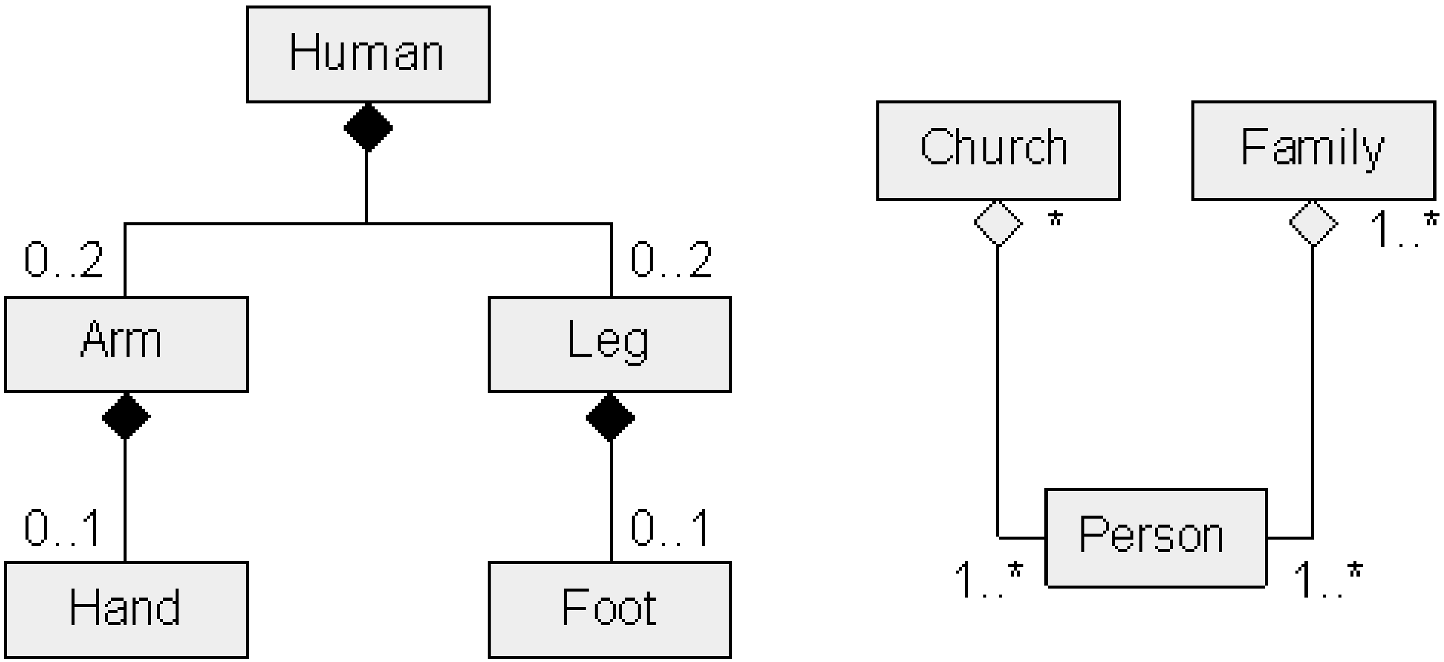
Class and Instance Variables and Operations

- An **instance variable** is an attribute whose value is stored by each instance of a class.
- A **class variable** is an attribute whose value is stored only once and shared by all instances.
- An **instance operation** must be called through an instance.
- A **class operation** may be called through the class.
- In UML class variables and operations are called *static*.
  - Indicated by underlining an attribute's or operation's specification

Aggregation and Composition

- The **aggregation association** represents the part-whole relation between classes.
  - Denoted by a solid diamond and lines
  - Diamond attaches to the aggregate (whole) while lines attach to the parts
  - May have all association adornments
- The **composition association** is an aggregation association in which each part can be related to only one whole at a time.
  - Denoted by a hollow diamond and lines

Aggregation and Composition Examples



Class Diagram Heuristics

- Never place a name, rolenames, or multiplicities on a generalization connector.
- Use the «abstract» stereotype and {abstract} property to indicate abstract classes and operations when drawing diagrams by hand; use italics when drawing diagrams on the computer.
- Use the interface ball and socket symbols to abstract interface details and a stereotyped class symbol to show details.

Class Diagram Heuristics…

- Show provided interfaces with the *interface ball symbol* or the stereotyped class symbol and a realization connector.
- Show required interfaces with the *interface socket symbol* or dependency arrows to stereotyped class symbols or interface ball symbols.
- Avoid aggregation and composition.

Mid-Level Design

Mid-level design is the activity of specifying software at the level of medium-sized components, such as compilation units or classes, and their properties, relationships, and interactions.

- DeSCRIPTR specification
- Design patterns (more on this later)

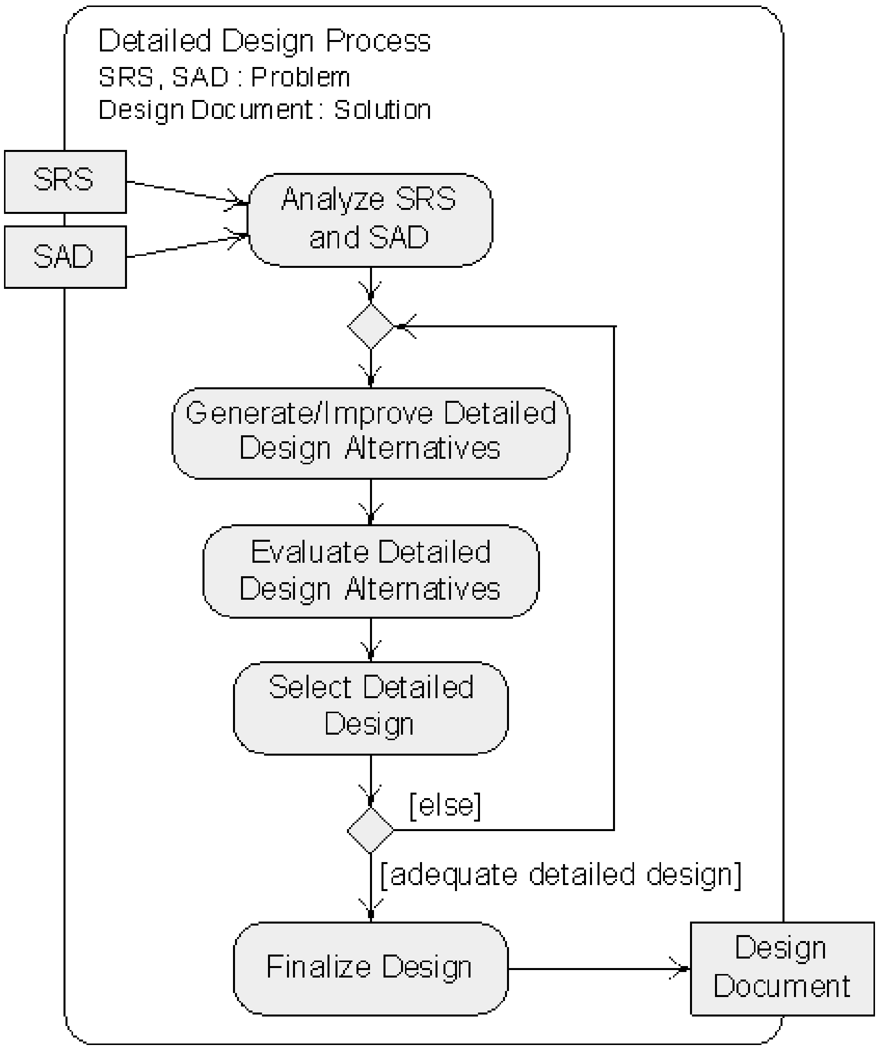
Low-Level Design

Low-level design is the activity of filling in the small details at the lowest level of abstraction.

DeSCRIPTR specifications plus PAID

- *Packaging*—Placing code into compilation units, libraries, packages, etc.
- *Algorithms*—Sometimes specified
- *Implementation*—Visibility, accessibility, association realization, etc.
- *Data structures and types*—Sometimes specified

Detailed Design Process



Detailed Design Document

- A **design document** consists of a SAD and a **detailed design document** (DDD)
- A DDD template

1. Mid-Level Design Models
2. Low-Level Design Models
3. Mapping Between Models
4. Detailed Design Rationale
5. Glossary

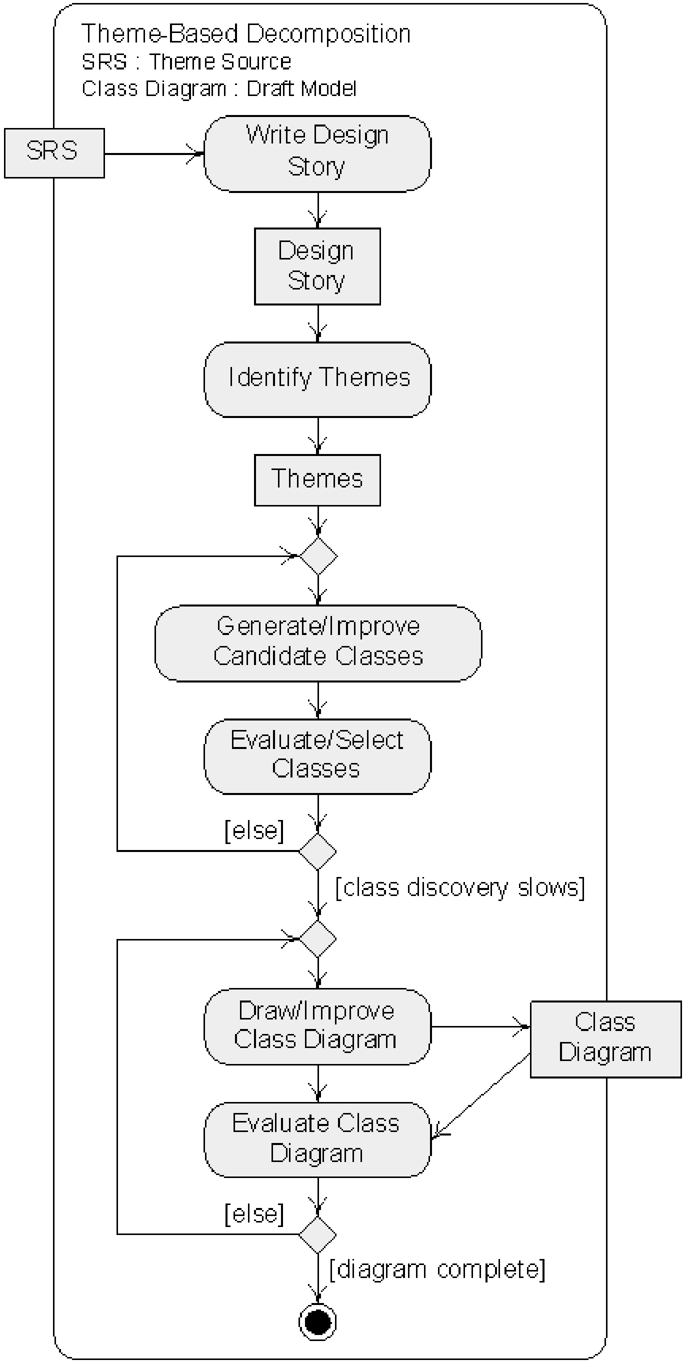
Mid-Level Generation Techniques

- *Creational*—Make a mid-level design class model from scratch
  - Functional decomposition
  - Quality attribute decomposition
  - Design themes
- *Transformational*—Change another model into a mid-level design class model
  - Similar system
  - Patterns or architectures
  - Analysis model

Generation from Design Themes

- A **design theme** is an important problem, concern, or issue that must be addressed in a design.
- Design themes can be the basis for generating a design from scratch.

Design Theme Process



Analyzing Design Stories

- Start by writing a **design story**: a short description of the application that stresses its most important aspects.
- Study the design story to identify design themes.
- List the themes
  - Functional themes
  - Quality attribute themes

Generating Candidate Classes

- Brainstorm candidate classes from the themes; list classes and their responsibilities.
  - Entities in charge of program tasks
  - Actors
  - Things about which the program stores data
  - Structures and collections
- Rationalize the classes.
  - Discard those with murky names or responsibilities
  - Rework classes with overlapping responsibilities
  - Discard those that do something out of scope

Draft a Class Diagram

- Draw the classes from the list.
- Add attributes, operations, and associations.
- Refine the class diagram.
  - Check classes for completeness and cohesion.
  - Make super-classes where appropriate.
  - Apply design patterns where appropriate.

Responsibilities

A **responsibility** is an obligation to perform a task (an **operational responsibility**) or to maintain some data (a **data responsibility**).

- Operational responsibilities are usually fulfilled by operations.
- Data responsibilities are usually fulfilled by attributes.
- Class collaborations may be involved.

Responsibility-Driven Decomposition

- Responsibilities may be stated at different levels of abstraction.
- Responsibilities can be decomposed.
- High-level responsibilities can be assigned to top-level components.
- Responsibility decomposition can be the basis for decomposing components.
  - Responsibilities reflect both operational and data obligations, so responsibility-driven decomposition can be different from functional decomposition.

Responsibility Heuristics

- Assigning responsibilities well helps achieve _high cohesion_ and _low coupling_.
  - State both operational and data responsibilities.
  - Assign modules at most one operational and one data responsibility.
  - Assign complementary data and operational responsibilities.

Responsibility Heuristics…

- Make sure module responsibilities do not overlap.
- Place operations and data in a module ***only*** if they help fulfill the module’s responsibilities.
- Place ***all*** operations and data needed to fullfill a module responsibility in that module.

Inheritance

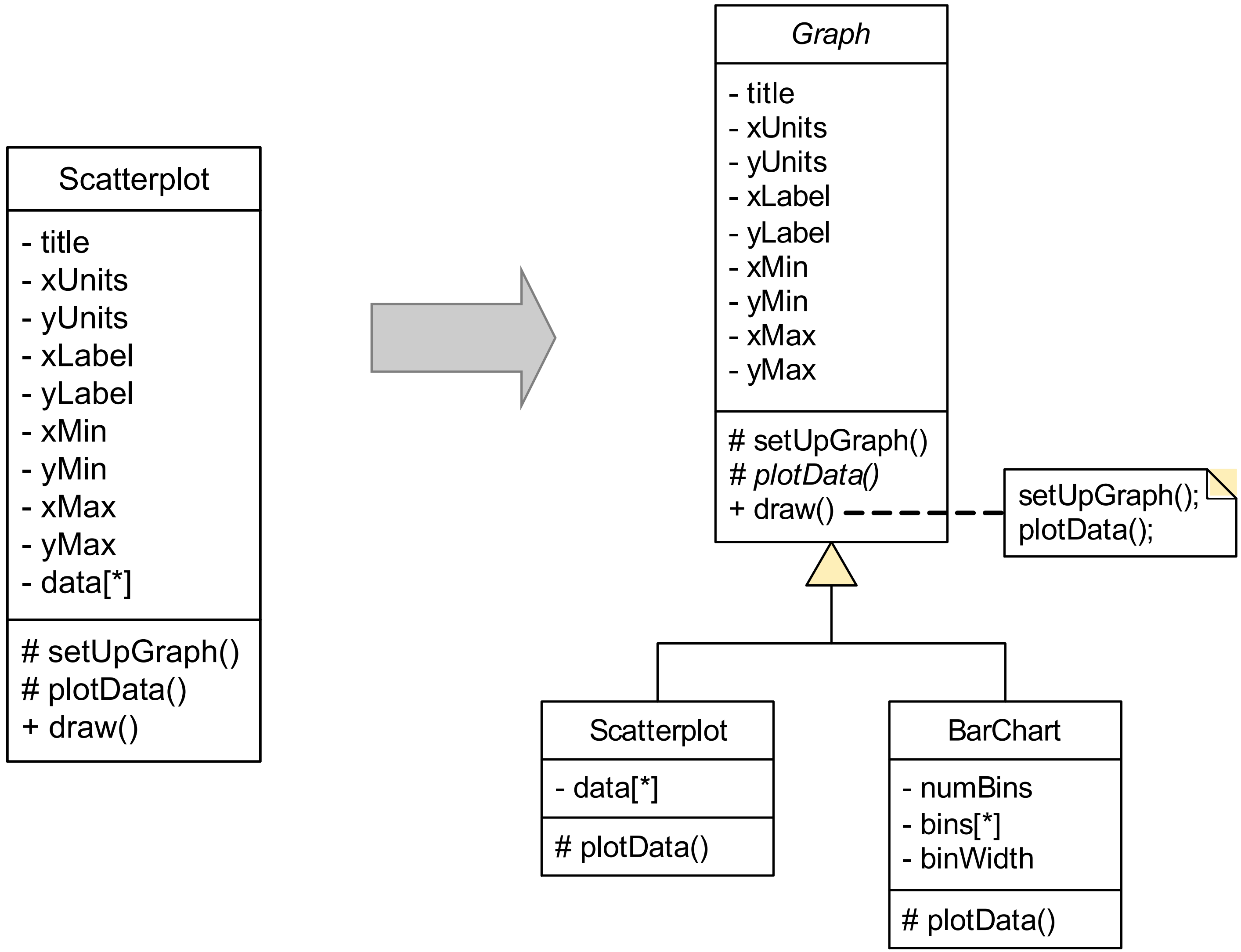
Inheritance is a declared relation between a class and one or more super-classes that causes the sub-class to have every attribute and operation of the super-class(es).

- Captures a generalization relation between classes
- Allows reuse of attributes and operation from super-classes in sub-classes

Using Inheritance Properly

- Don’t use inheritance only for reuse.
  - Confusing
  - Ugly
  - Leads to problems in the long run
- Use inheritance only when there is a generalization (kind-of) relation present.
- Reuse can often be achieved by rethinking the class structure.
  - Clear
  - Elegant
  - Robust

Inheritance Example

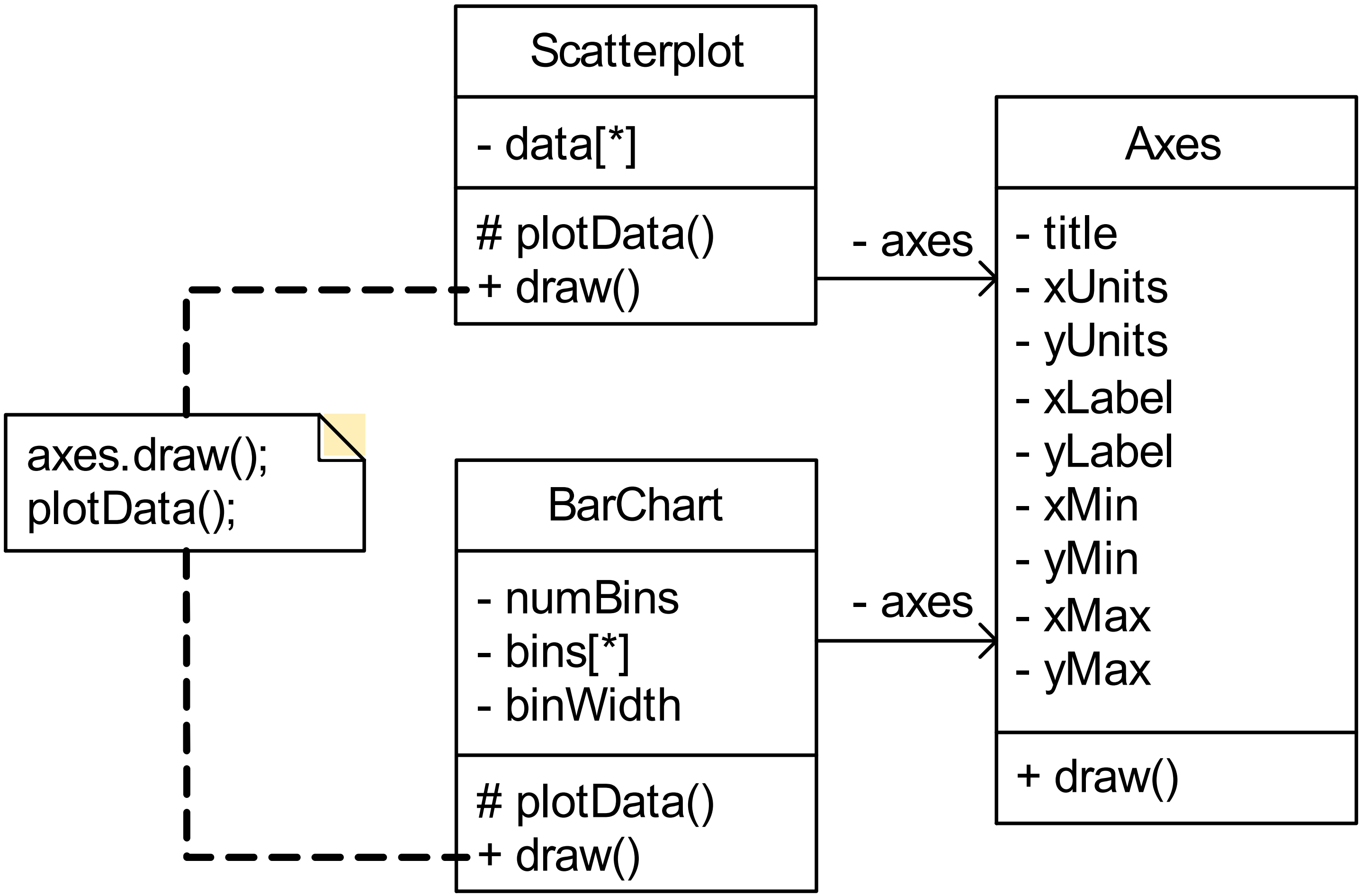


Delegation

Delegation is a tactic wherein one module (the *delegator*) entrusts another module (the *delegate*) with a responsibility.

- Allows reuse without violating inheritance constraints
- Makes software more reusable and configurable

Delegation Example



Inheritance and Delegation Heuristics

- Use inheritance only when there is a generalization relationship between the sub-class and its super-class(es).
- Combine common attributes and operations in similar classes into a common super-class.
- Use delegation to increase reuse, flexibility, and configurability.

Summary

- Detailed design is complex -> mid-level design + low-level design
- Mid-level design is captured in a DDD that includes DeSCRIPTR specifications.
- Mid-level class designs can be generated from scratch (creational) or by changing another model (transformational).

Summary…

- One creational techniques uses design themes extracted from a design story.
- One transformational techniques is to convert a conceptual model into a design class model.
- Responsibility-driven design helps designers make good decisions about class models.
- Inheritance and delegation, when used properly, lead to clear and elegant designs that increase reusability, flexibility, and configurability.

So far..

- We did static design
- What about the behavioural aspects of the problem?

- Again we use UML notation
- Review the required UML notation

Interaction Diagrams

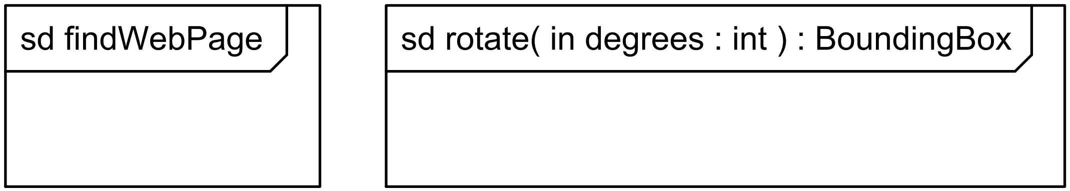
An **interaction diagram** is a notation for modeling the communication behavior of individuals exchanging information to accomplish some task.

- *Sequence diagram*—shows interacting individuals along the top and message exchange down the page
- *Interaction overview diagram*—a kind of activity diagram whose nodes are sequence diagram fragments
- *Timing diagram*—shows individual state changes over time

Sequence Diagram Frames

Frame—a rectangle with a pentagon in the upper left-hand corner called the *name compartment*.

- **sd** *interactionIdentifier*
- *interactionIdentifier* is either a simple name or an operation specification as in a class diagram



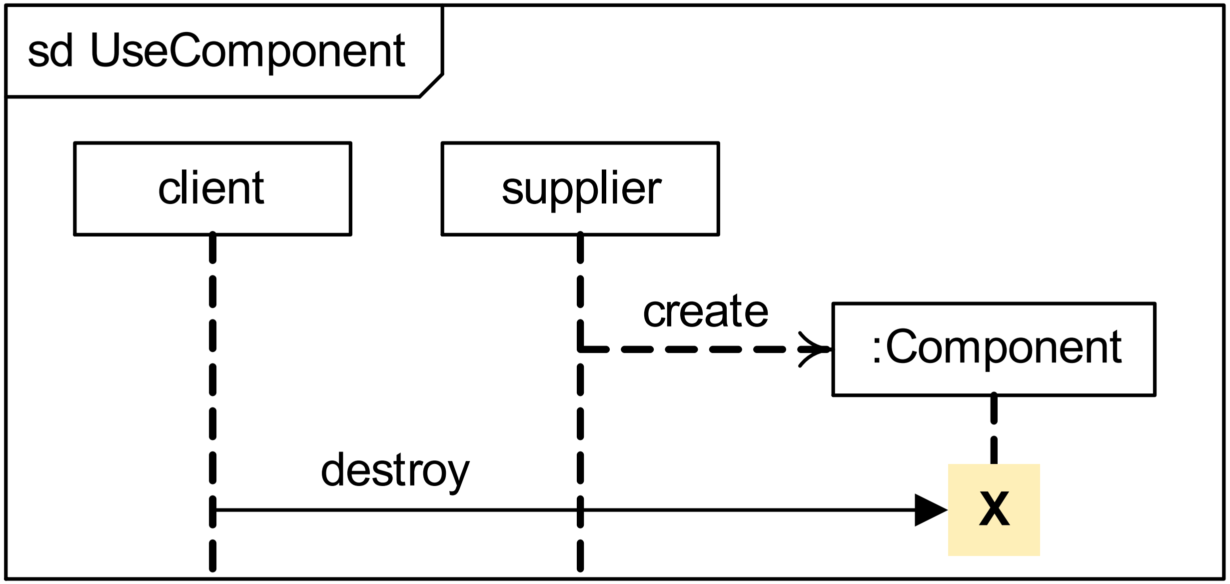
Lifelines

- Participating individuals are arrayed across the diagram as *lifelines*:
  - Rectangle containing an identifier
  - Dashed line extending down the page
- The vertical dimension represents time; the dashed line shows the period when an individual exists.

Lifeline Creation and Destruction

- An new object appears at the point it is created.
  - Not clear from UML specification
- A destroyed object has a truncated lifeline ending in an **X.**
- Persisting objects have lifelines that run the length of the diagram.

Lifelines Example



Lifeline Identifier Format

name[*selector*] : *typeName*

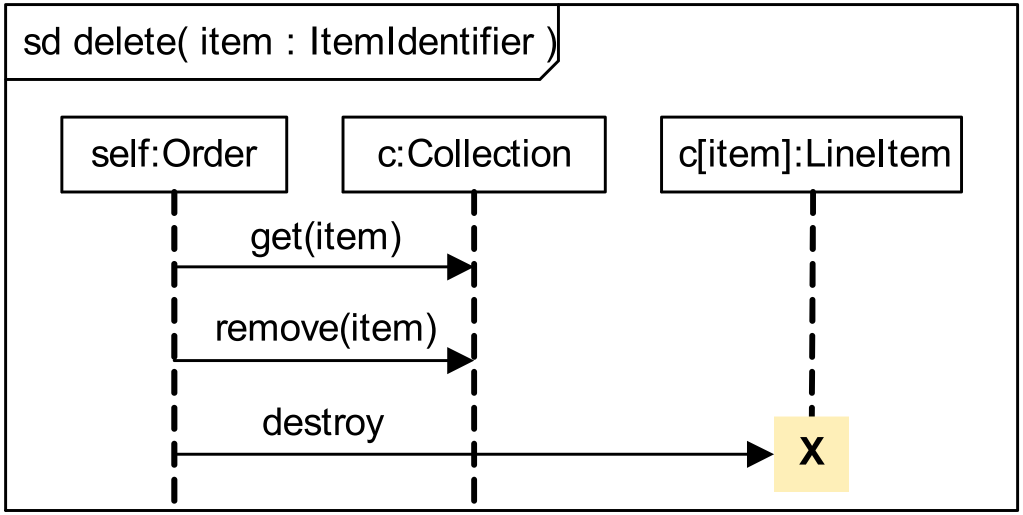
- *name*—simple name or “self”; optional
- selector—expression picking out an individual from a collection
  - Format not specified in UML
  - Optional; if omitted, so are the brackets
- *typeName*—Type of the individual
  - Format not specified in UML
  - Optional; if omitted, so is the colon
- Either *name*, *typeName*, or both must appear

Lifeline Identifier Examples

- player[i] : Player
- player[i]
- : Player
- board

Self

Used when the interaction depicted is “owned” by one of the interacting individuals



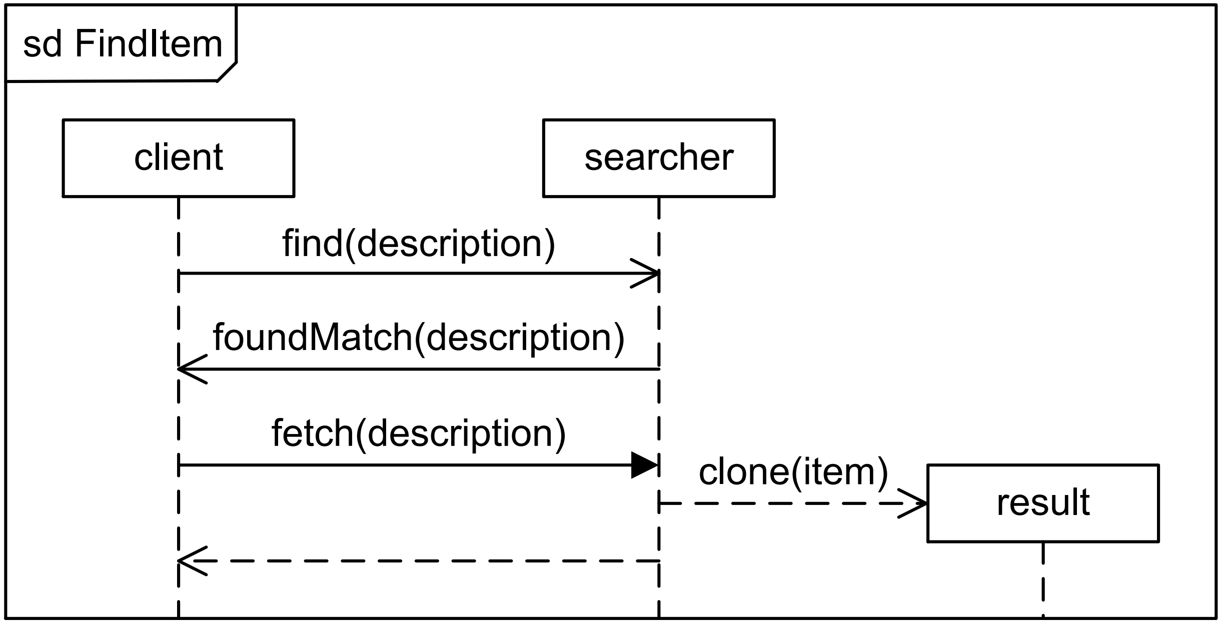
Messages and Message Arrows

- *Synchronous*—The sender suspends execution until the message is complete

- *Asynchronous*—The sender continues execution after sending the message

- *Synchronous message return or instance creation*

Message Arrow Example



Message Specification Format

variable = name argumentList

- *variable*—simple name of a variable assigned a result
  - Optional; if omitted, so is the equals sign
- *name*—simple name of the message
- *argumentList*—comma-separated list of arguments in parentheses
  - *varName = paramName*
    - = *paramName* may be omitted
  - *paramName = argumentValue*
    - = *argumentValue* may be omitted
- Message specification may be * (any message)

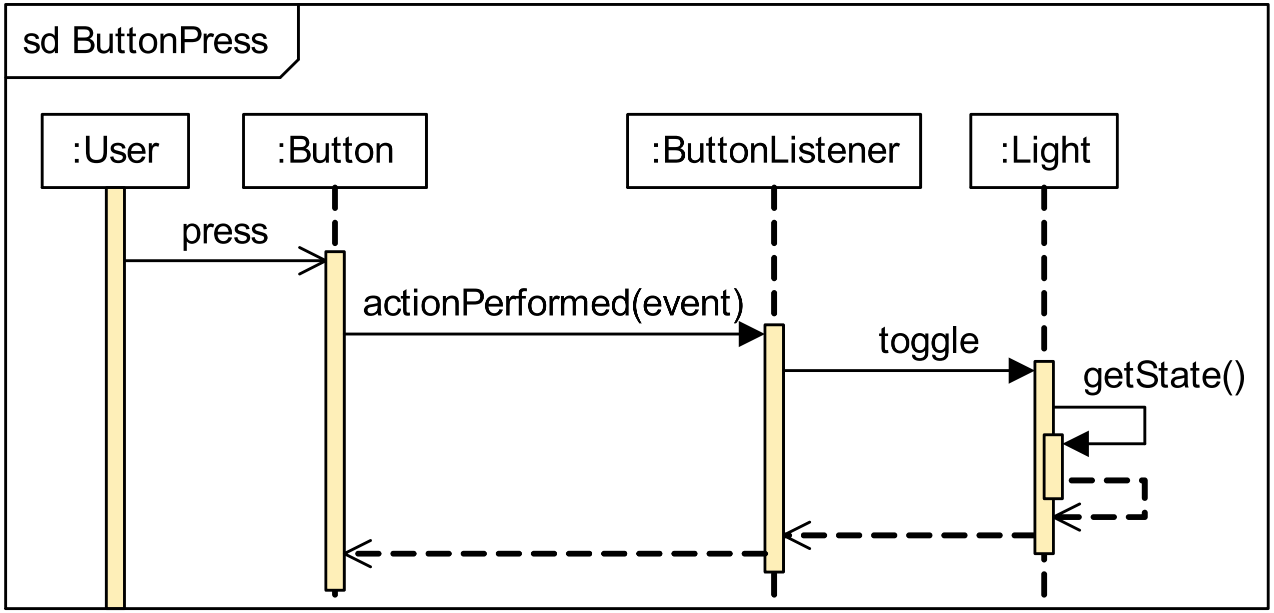
Message Specification Examples

- hello
- hello()
- msg = getMessage(helloMessage)
- x = sin(a/2)
- x = sin(angle = a/2)
- trim(result = aString)

Execution Occurrences

- An operation is **executing** when some process is running its code.
- An operation is **suspended** when it sends a synchronous message and is waiting for it to return.
- An operation is **active** when it is executing or suspended.
- The period when an object is active can be shown using an *execution occurrence*.
  - Thin rectangle over lifeline dashed line

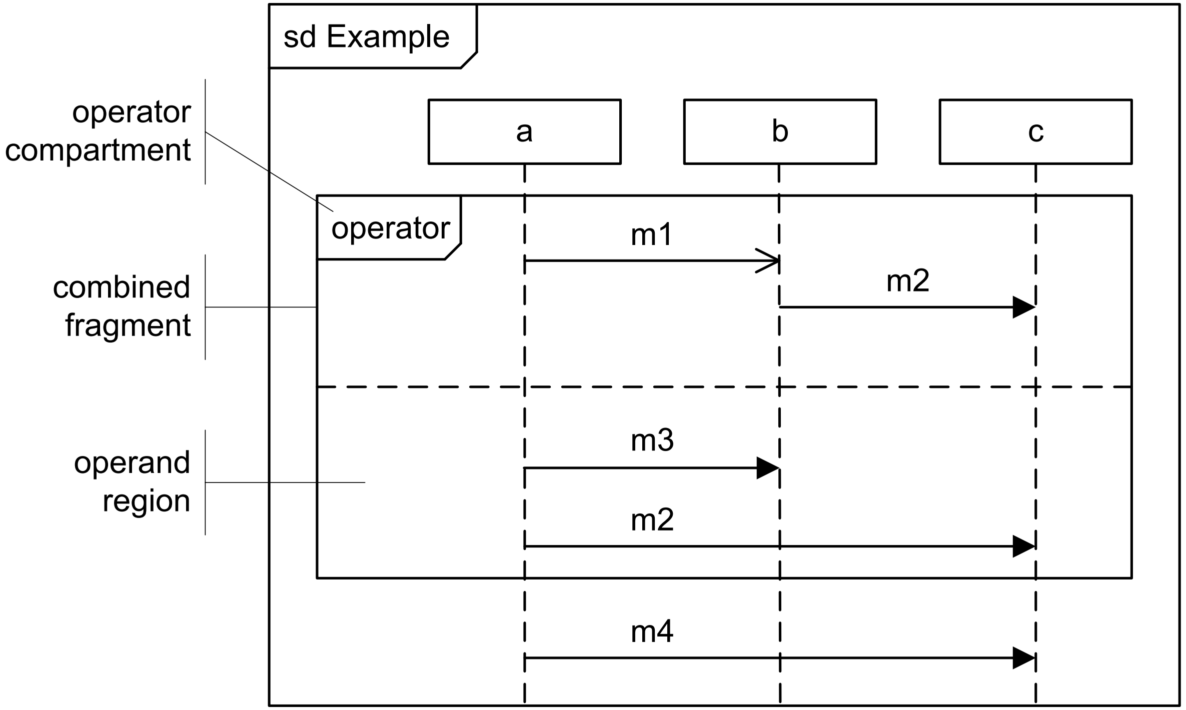
Execution Occurrence Example



Combined Fragments

- A combined fragment is a marked part of an interaction specification that shows
  - Branching,
  - Loops,
  - Concurrent execution,
  - And so forth.
- It is surrounded by a rectangular frame.
  - Pentagonal operation compartment
  - Dashed horizontal line forming regions holding operands

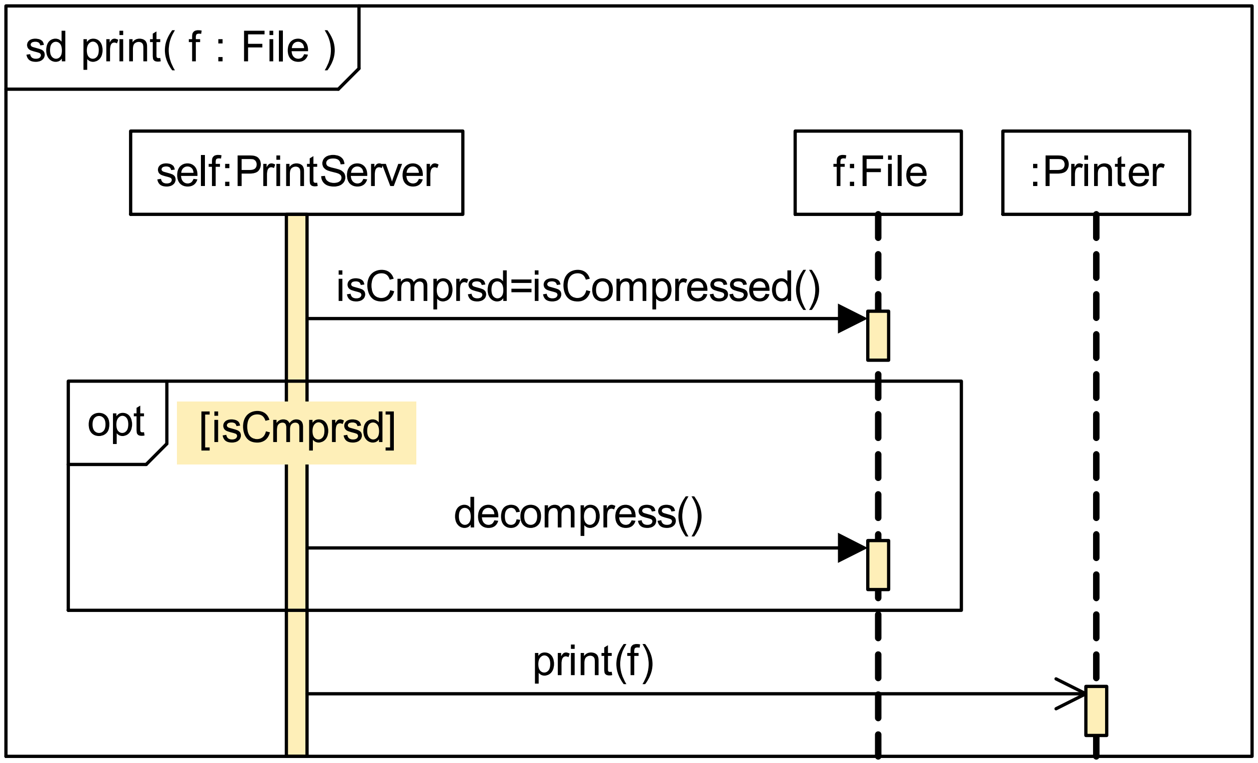
Combined Fragment Layout



Optional Fragment

- A portion of an interaction that may be done
  - Equivalent to a conditional statement
  - Operator is the keyword opt
  - Only a single operand with a guard
- A *guard* is a Boolean expression in square brackets in a format not specified by UML.
  - [else] is a special guard true if every guard in a fragment is false.

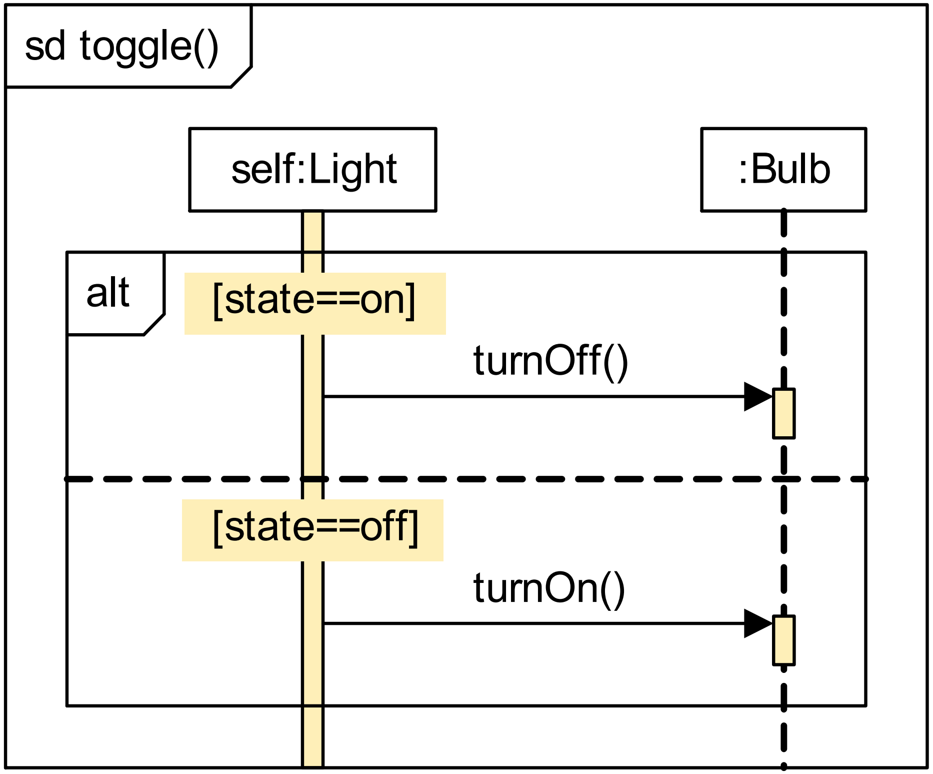
Optional Fragment Example



Alternative Fragment

- A combined fragment with one or more guarded operands whose guards are mutually exclusive
  - Equivalent to a case or switch statement
  - Operator is the keyword alt

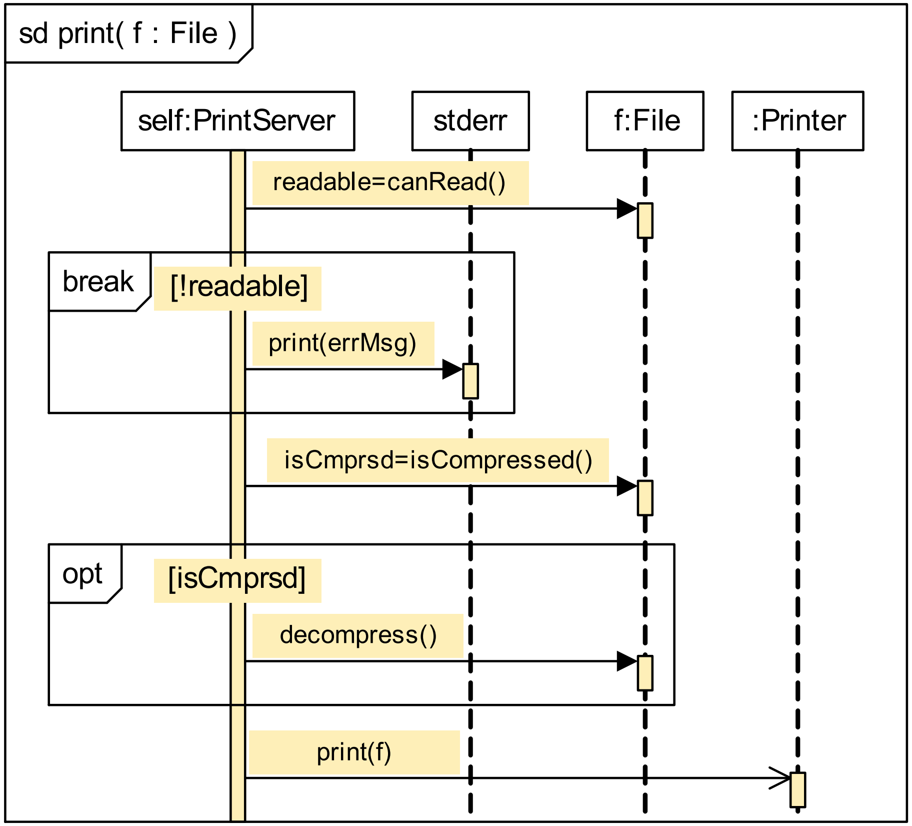
Alternative Fragment Example



Break Fragment

- A combined fragment with an operand performed in place of the remainder of an enclosing operand or diagram if the guard is true
  - Similar to a break statement
  - Operator is the keyword break

Break Fragment Example



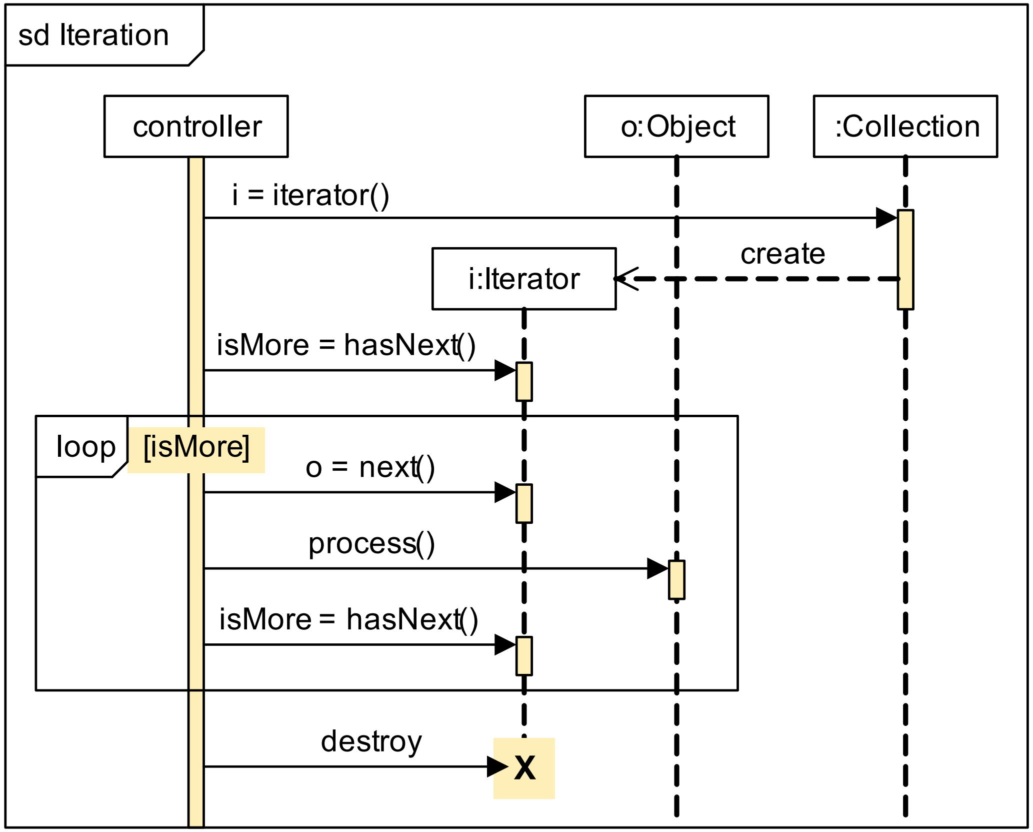
Loop Fragment

- Single loop body operand that may have a guard
- Operator has the form loop(*min*, *max*) where
  - Parameters are optional; of omitted, so are the parentheses
  - *min* is a non-negative integer
  - *max* is a non-negative integer at least as large as *min* or *; max is optional; if omitted, so is the comma

Loop Fragment Execution Rules

- The loop body is performed at least *min* times and at most *max* times.
- If the loop body has been performed at least *min* times but less than *max* times, it is performed only if the guard is true.
- If *max* is *, the upper iteration bound is unlimited.
- If *min* is specified but *max* is not, then *min*=*max*.
- If the loop has no parameters, then *min*=0 and *max* is unlimited.
- The default value of the guard is true.

Loop Fragment Example



Sequence Diagram Heuristics

- Put the sender of the first message leftmost.
- Put pairs of individuals that interact heavily next to one another.
- Position individuals to make message arrows as short as possible.
- Position individuals to make message arrows go from left to right.

Sequence Diagram Heuristics…

- Put the self lifeline leftmost.
- In a sequence diagram modeling an operation interaction, draw the self execution occurrence from the top to the bottom of the diagram.
- Name individuals only if they are message arguments or are used in expressions.

Sequence Diagram Heuristics…

- Choose a level of abstraction for the sequence diagram.
- Suppress messages individuals send to themselves unless they generate messages to other individuals.
- Suppress return arrows when using execution occurrences.
- Don’t assign values to message parameters by name.

Using Sequence Diagrams

- Sequence diagrams are useful for modeling
  - Interactions in mid-level design;
  - The interaction between a product and its environment (called *system sequence diagrams*);
  - Interactions between system components in architectural design.
- Sequence diagrams can be used as (partial) use case descriptions.

Summary of Sequence Diagrams

- Sequence diagrams are a powerful UML notation for showing how objects interact.
- Interacting objects are represented by lifelines arrayed across the diagram.
- Time is represented down the diagram.
- The exchange of messages is shown by message arrows arranged down the diagram.

Interaction Design: Process & Heuristics

- In the next set of slides we study…
  - An overview of the interaction design process
  - Alternative control styles and consider their strengths and weaknesses
  - Interaction design heuristics

Component and Interaction Co-Design

- Components cannot be designed alone because they may not support needed interactions.
- Interactions cannot be designed alone because they may rely on missing features of components or missing components.
- Components and interactions must be designed together iteratively.

Outside-In Design

- Interaction design should be mainly top-down (from most to least abstract interactions).
- The most abstract interactions are specified in the SRS and use case models.
- Starting with the interactions between the program and its environment (outside) and designing how interacting components can implement them (inside) is called **outside-in design.**

Controllers

A **controller** is a program component that makes decisions and directs other components.

Controller are important because they are the central figures in collaborations.

Control Styles

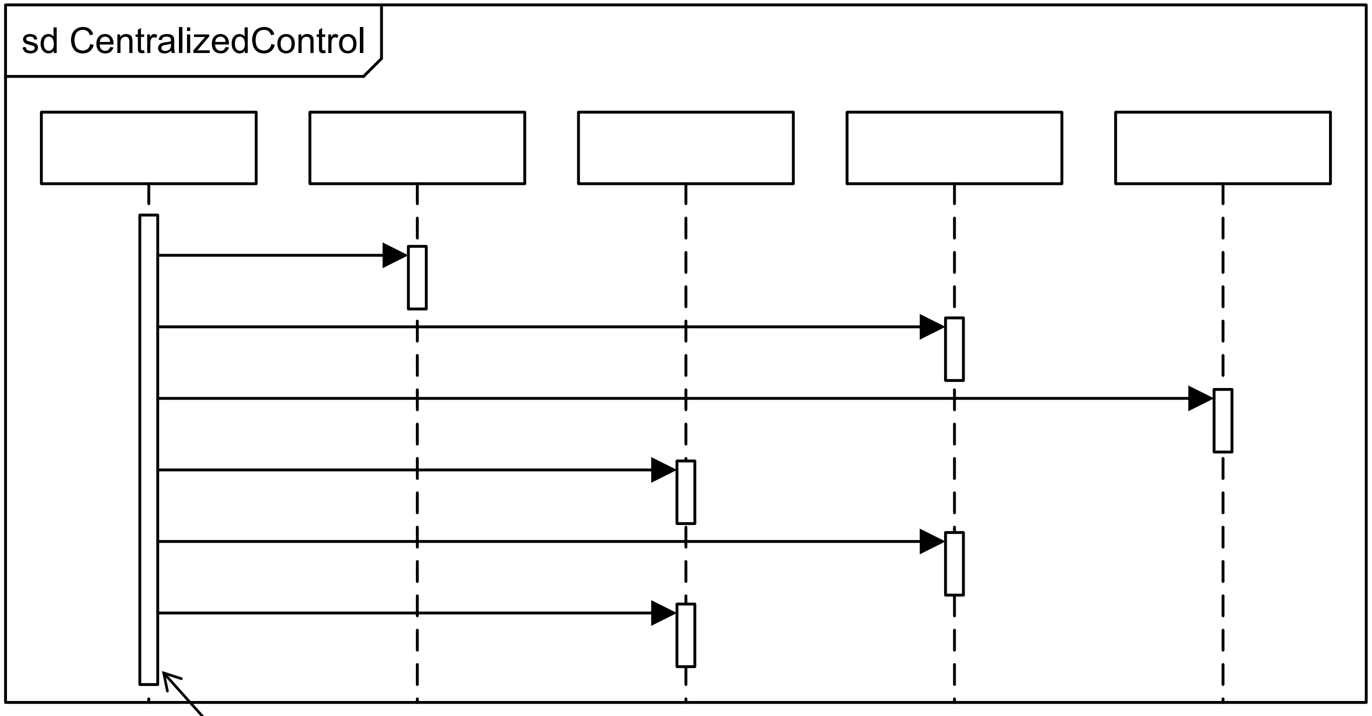
A **control style** is a way that decision making is distributed among program components.

- *Centralized*—A few controller make all significant decisions
- *Delegated*—Decision making is distributed through the program with a few controllers making the main decisions
- *Dispersed*—Decision making is spread widely through the program with few or no components making decisions on their own

Centralized Control

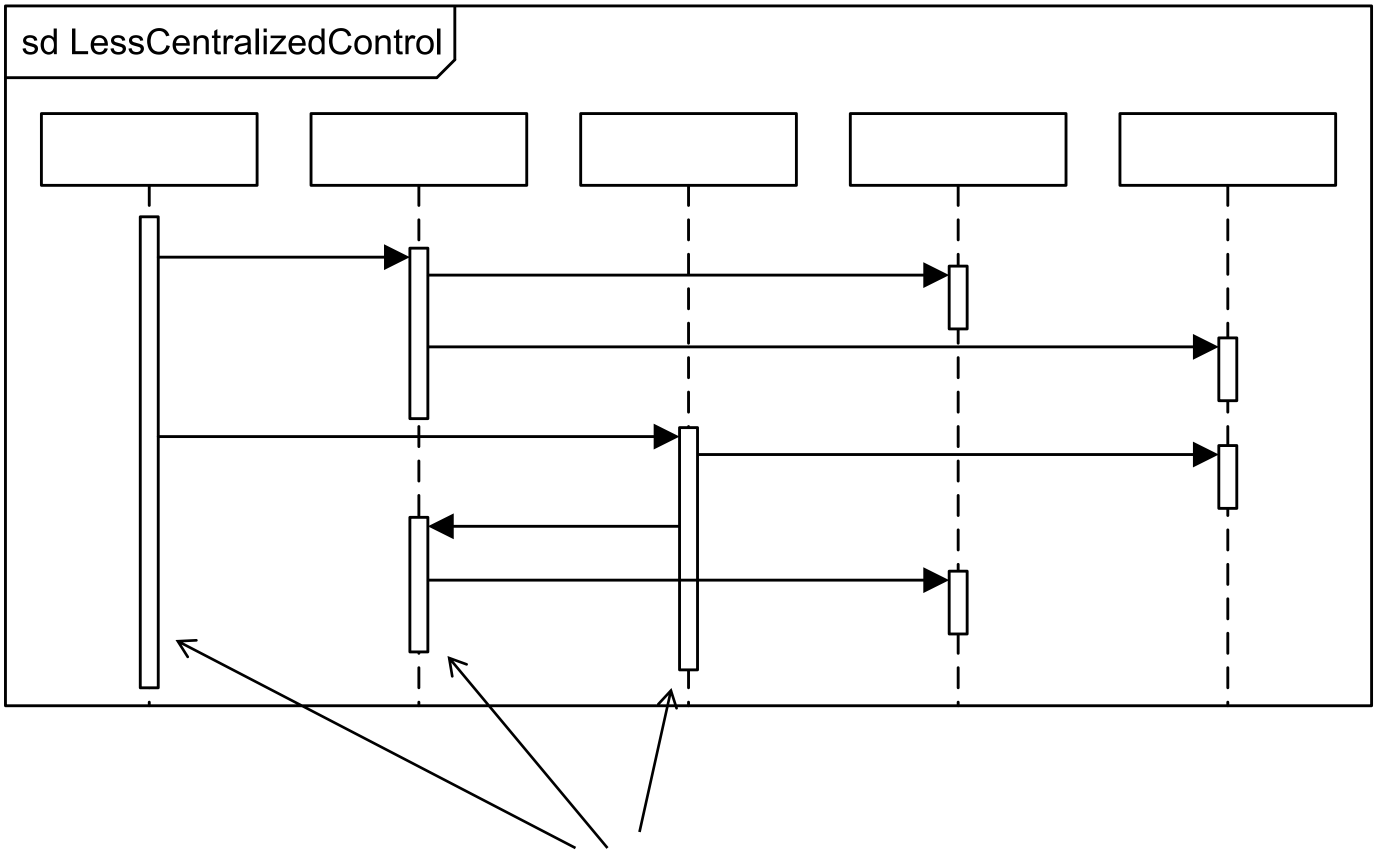
- Easy to find where decisions are made
- Easy to see how decisions are made and to alter the decision-making process
- Controllers may become **bloated**—large, complex, and hard to understand, maintain, test, etc.
- Controller may treat other components as data repositories
  - Increases coupling
  - Destroys information hiding

Centralized Control Form



Controller initiating all the interactions

Less-Centralized Control Form



More than one element initiating interactions.
Less centralized control structure.

Control Heuristics

- Avoid interaction designs where most messages originate from a single component.
- Keep components small.
- Make sure operational responsibilities are not all assigned to just a few components.
- Make sure operational responsibilities are consistent with data responsibilities.

Delegated Control

- Controller are coupled to fewer components, reducing coupling.
- Information is hidden better.
- Programs are easier to divide into layers.
- Delegated control is the preferred control style.

Have components delegate as many low-level tasks as possible.

Dispersed Control Style

- Characterized by having many components holding little data and having few responsibilities.
- It is hard to understand the flow of control.
- Components are unable to do much on their own, increasing coupling.
- It is hard to hide information.
- Cohesion is usually poor.
- Few modularity principles can be satisfied.

Avoid interactions that require each component to send many messages.

Law of Demeter

An operation of an object *obj* should send messages only to the following entities:

- The object *obj*;
- The attributes of *obj*;
- The arguments of the operation;
- The elements of a collection that is an argument of the operation or an attribute of *obj*;
- Objects created by the operation; and
- Global classes or objects.

Consequences of the Law of Demeter

- Objects send messages only to objects “directly known” to them.
- The Law of Demeter helps to
  - Hide information,
  - Keep coupling low,
  - Keep cohesion high,
  - Discourage an over-centralized control style, and
  - Encourage a delegated control style.

Remarks on Control Styles and Heuristics

- There is a continuum of control styles with centralized and dispersed on the ends and delegated in the middle.
- Different levels of centralization may be more or less appropriate depending on the problem.
- The control heuristics are in tension.

Summary of Interaction Design

- Interactions and components cannot be designed independently, so they must be designed together iteratively (component and interaction co-design).
- Interaction design should proceed top-down (outside-in).
- Controllers are important components in designing interactions.
- We can distinguish various control styles on a continuum of centralization versus distribution.

Summary…

- A delegated control style in which a few controllers make important decisions but delegate other decisions to subordinates is usually best.
- Various heuristics, including the Law of Demeter, encourage control styles that maximize information hiding and cohesion and minimize coupling.