Object Oriented Analysis and Design

Deepika Kamboj
Assistant professor
School of Computer Science
UPES

Requirement Modelling

Requirement Modelling is the process of **capturing, analyzing, and representing** the needs and expectations of users, stakeholders, and the system in a structured way before system development begins. It turns "what the user wants" into **clear, detailed, and testable specifications** that developers and designers can understand.

Purpose:

- Define what the system should do and how it should behave.
- Ensure a common understanding among stakeholders, developers, and testers.

When: During the *Requirements Analysis* phase of the SDLC.

Why Important: Misunderstanding requirements is a major cause of project failure; modelling reduces this risk.

Visual Representation Tools

- **Use Case Diagrams:** Show system interactions with users.
- **Context Diagrams:** Define system boundaries.
- **Data Flow Diagrams (DFD):** Model data movement.
- **Entity-Relationship Diagrams (ERD):** Model data structure.
- **State Transition Diagrams:** Model system states and changes.
- **Class Diagrams:** For object-oriented systems.

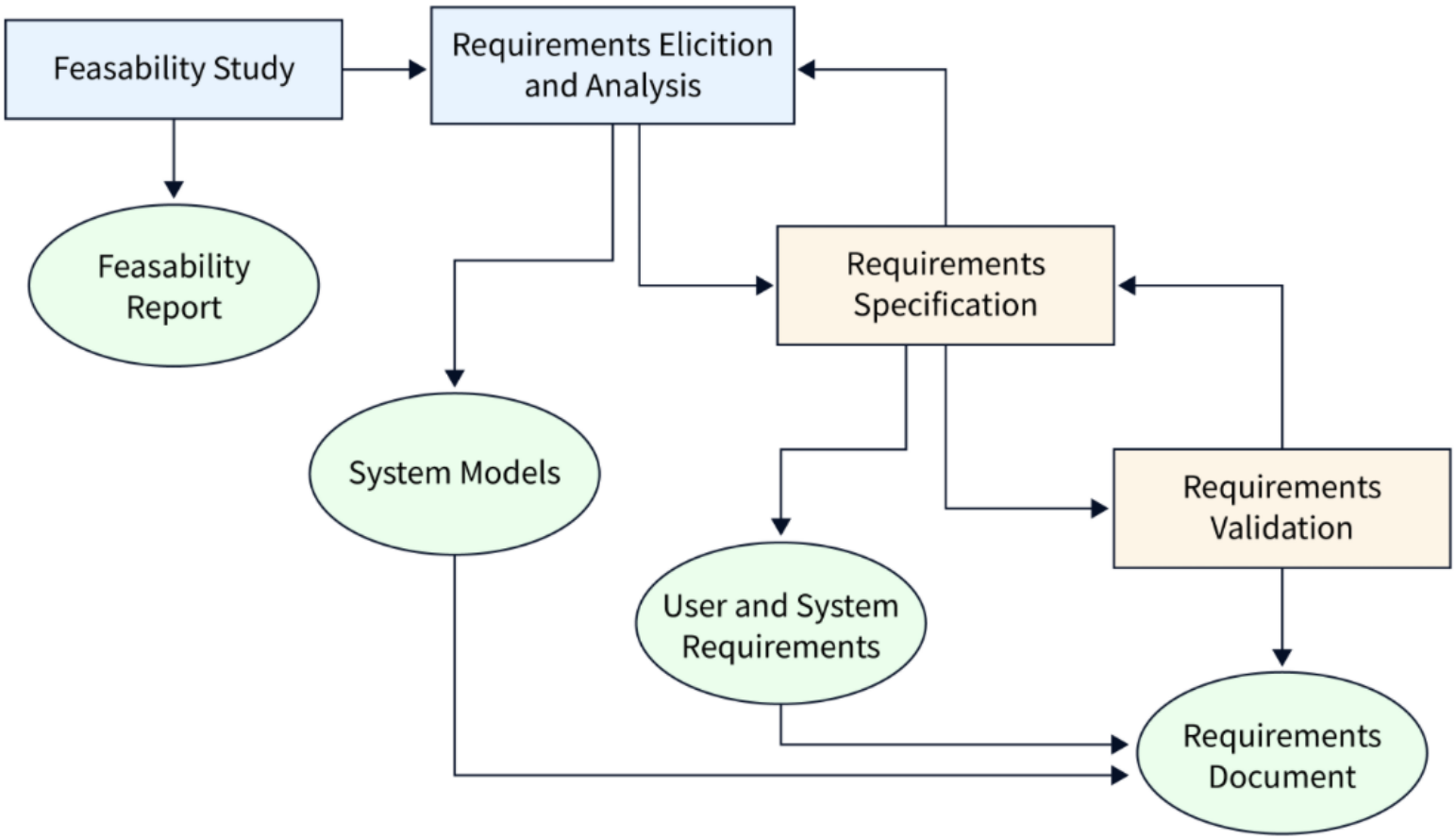
Why Requirement Modelling Matters

- Clarifies scope and avoids misunderstandings.
- Serves as a blueprint for design and development.
- Facilitates communication among team members.
- Helps in estimating cost, time, and resources.
- Supports testing by providing acceptance criteria.

Requirements Engineering

- Systematic process of discovering, analyzing, documenting, and managing requirements
- Ensures the system meets stakeholder needs and business goals
- Involves:
  - Elicitation – gathering needs from stakeholders
  - Analysis – resolving conflicts and prioritizing
  - Specification – writing clear, testable requirements
  - Validation – checking correctness and completeness
  - Management – handling changes over time
- Focuses on the **process of developing and controlling requirements**, not their representation

Diagram Representation



Feasibility Study

- **Purpose:** Evaluate the feasibility of the proposed software project.
- **Objective:** Determine if the project is technically, economically, and operationally viable.
- **Aspects considered:**
  - **Technical Feasibility:** Can the system be developed with available technology and resources?
  - **Economic Feasibility:** Is the project cost-effective in terms of costs and benefits?
  - **Operational Feasibility:** Will the system work effectively within the existing organization and processes?

Requirement Elicitation and Analysis

- **Purpose:** Gather and analyze requirements from stakeholders (users, customers, domain experts).
- **Goal:** Understand system needs and translate them into software requirements.
- **Techniques:**
  - **Interviews:** Direct discussions with stakeholders.
  - **Surveys:** Questionnaires to collect opinions and preferences.
  - **Workshops:** Group sessions for brainstorming and collaboration.
  - **Observation:** Monitoring users to understand workflows and needs.
  - **Prototyping:** Preliminary version of the system to elicit feedback.
- **Analysis:** Check for consistency, completeness, and feasibility.
- **Conflict resolution:** Resolve ambiguities and organize requirements into categories (e.g., functional, non-functional).

Software Requirement Specification

- **Purpose:** Document gathered and analyzed requirements formally.
- **Output:** The Software Requirement Specification (SRS) document, which acts as a blueprint for development.
- **Components of SRS:**
  - **Functional Requirements:** Define specific functions and features of the system.
  - **Non-functional Requirements:** Define constraints and qualities (e.g., performance, security).
  - **User Interfaces:** Describe how users will interact with the system.
  - **Data Requirements:** Specify details of data storage, processing, and management.

Software Requirement Validation

- **Purpose:** Validate documented requirements to ensure they reflect stakeholders' needs and expectations.
- **Goal:** Identify and correct errors, conflicts, or misunderstandings before development begins.
- **Techniques for Validation:**
  - **Reviews:** Expert reviews and walkthroughs of the SRS to detect issues.
  - **Prototyping:** Build a working model to confirm requirements with stakeholders.
  - **Simulation:** Simulate system behavior to verify alignment with documented requirements.

Software Requirement Management

- **Purpose:** Ongoing process of maintaining and tracking requirements throughout the software development lifecycle.
- **Goal:** Ensure changes are controlled, traceable, and communicated effectively.
- **Key Activities:**
  - **Version Control:** Manage different versions of the requirements document.
  - **Change Control:** Document and evaluate requirement changes and their impacts.
  - **Traceability:** Link requirements to design, implementation, and testing.
  - **Prioritization:** Assign importance levels to guide development focus.
  - **Communication:** Keep stakeholders informed about changes and updates.

Challenges and Risks in Requirement Engineering

- **Incomplete Requirements:** Stakeholders may omit details, leading to missing functionality.
- **Ambiguous Requirements:** Vague language → different interpretations, miscommunication, and design flaws.
- **Changing Requirements:** Evolving needs/market conditions cause scope creep, delays, and cost overruns.
- **Conflicting Requirements:** Stakeholder priorities may clash, requiring negotiation and trade-offs.
- **Poor Traceability:** Weak tracking makes it hard to ensure all requirements are implemented and tested.

Tools in Requirement Engineering

- **Document Editors:** MS Word, Google Docs, LaTeX – for requirement documents, use cases, and user stories.
- **Spreadsheets:** Excel, Google Sheets – for requirement lists, matrices, and traceability tables.
- **Diagramming Tools:** Visio, draw.io, Lucidchart – for use case, data flow, activity, and ER diagrams.
- **Requirements Management Tools:** IBM DOORS, Jama Connect, Helix RM – for organizing, tracking, and version control.
- **Prototyping Tools:** Balsamiq, Axure, Sketch – for UI mockups and interactive prototypes.

Tools in Requirement Engineering

- **Modeling Languages:** UML – for standardized diagrams (use case, class, sequence, state).
- **Collaboration Tools:** Confluence, SharePoint, MS Teams – for document sharing, feedback, and discussions.
- **Elicitation Techniques:** Surveys, interviews, questionnaires, workshops – for gathering requirements.
- **Validation/Verification Tools:** Static analysis, simulation – for checking correctness and consistency.
- **Traceability Tools:** Maintain links across requirements, design, implementation, and testing.

Advantages of Requirements Engineering

- **Customer Satisfaction:** Software meets user needs, enhancing satisfaction.
- **Early Issue Detection:** Identify problems early, preventing costly errors later.
- **Cost and Time Savings:** Reduce rework and scope creep, saving resources.
- **Effective Planning:** Clear requirements enable accurate project planning and estimation.
- **Traceability:** Links requirements to components, aiding impact analysis during changes.
- **Validation and Verification:** Serve as a foundation for ensuring correctness.
- **Collaboration:** Stakeholder engagement fosters teamwork and shared understanding.

Advantages of Requirements Engineering

- **Scope Management:** Prevent uncontrolled expansion and manage scope effectively.
- **Risk Mitigation:** Facilitate risk identification and mitigation planning.
- **Quality Assurance:** Lead to higher-quality products and informed testing.
- **Regulatory Compliance:** Help meet industry standards and regulations.
- **Documentation and Maintenance:** Provide reference for maintenance and future updates.
- **User Involvement:** Ensure user needs are considered and addressed.
- **Change Management:** Structured approach to handle requirement changes.

Disadvantages of Requirements Engineering

- **Ambiguity:** Poorly defined requirements cause confusion and misinterpretation.
- **Scope Creep:** Uncontrolled additions expand scope, leading to delays and cost overruns.
- **Changing Requirements:** Shifts during development cause rework, delays, and conflicts.
- **User Involvement Challenges:** Difficulties in engaging users can lead to missed requirements.
- **Overemphasis on Documentation:** Excessive focus slows development and reduces agility.

Disadvantages of Requirements Engineering

- **Miscommunication:** Gaps between stakeholder expectations and developer understanding.
- **Time-Consuming Process:** Extensive requirement gathering can delay project initiation.
- **Resistance to Change:** Stakeholders may resist updates due to attachment to initial requirements.
- **Dependency on Tools:** Overreliance on management tools creates risks if tools fail or age out.
- **Neglecting Non-Functional Requirements:** Critical aspects like performance and security may be overlooked.

Requirement Engineering Approaches

- **Waterfall Model:** Sequential, linear approach with complete documentation before moving ahead. Difficult to accommodate changes once a phase is completed.
- **Agile Methodology:** Iterative and incremental approach. Prioritizes working software over documentation and adapts to changing requirements.
- **Prototyping:** Builds a preliminary version for stakeholder feedback. Useful for unclear or evolving requirements, but may cause scope creep if unmanaged.

Requirement Engineering Approaches

- **Spiral Model:** Combines iterative development with risk management. Iterations are divided into phases, each addressing risks. Suitable for projects with high uncertainty and evolving requirements.
- **Use Case-driven Approach:** Focuses on interactions between users and the system. Expresses requirements as use cases and scenarios. Helps capture functional aspects and user behavior clearly.

Basic Structure Modelling (OOAD)

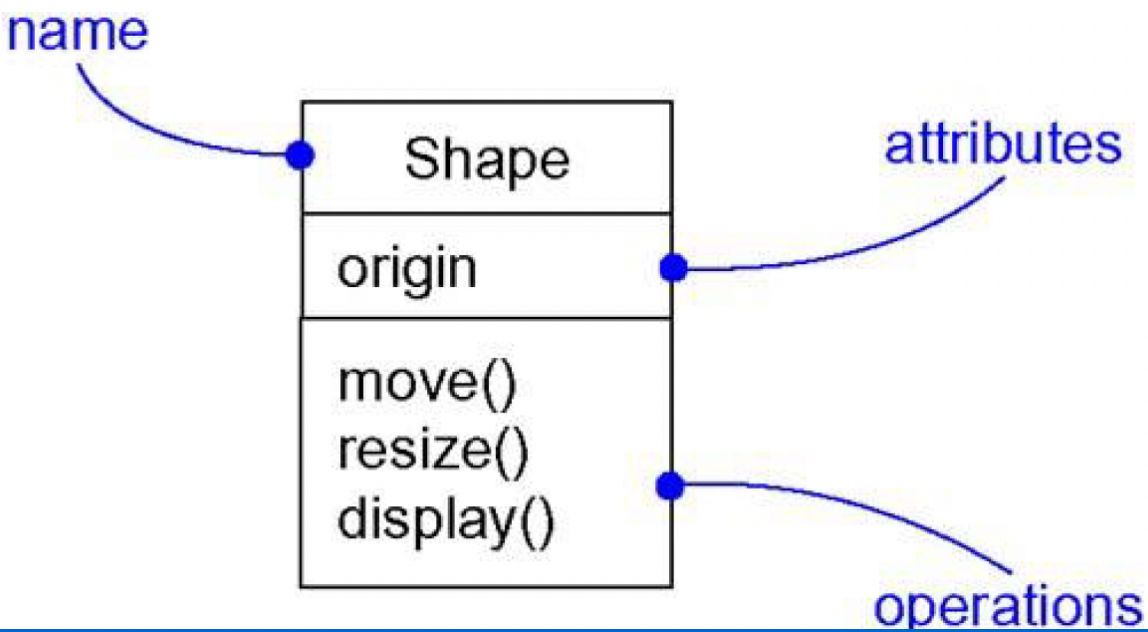
Definition: Represents the **static structure** of a system. Focuses on **fundamental building blocks** (classes, objects, interfaces) and their **relationships** (associations, generalizations, dependencies). Acts as a **blueprint** of how entities are organized and connected, independent of dynamic behavior.

Key Elements:

- **Classes** → Define the data and operations (what the system is).
- **Relationships** → Show how classes interact or depend on each other.
- **Common Mechanisms** → Reusable concepts like naming, visibility, constraints.
- **Diagrams** → UML notations that visually represent the structure.
- **Class Diagram** → The primary static model that binds all the above together.

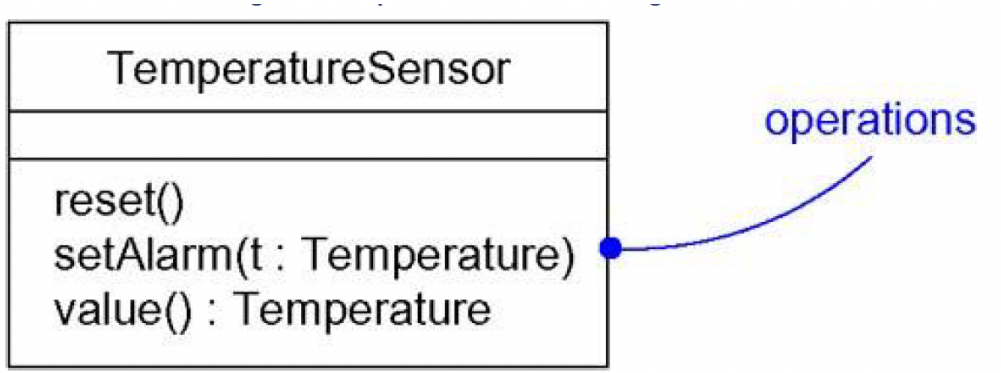
Class: Terms and Concepts

- A **class** is a description of a set of objects that share the same **attributes**, **operations**, **relationships**, and **semantics**.
- Graphically, a class is rendered as a **rectangle** divided into compartments:
  - Top: **Name**
  - Middle: **Attributes**
  - Bottom: **Operations**



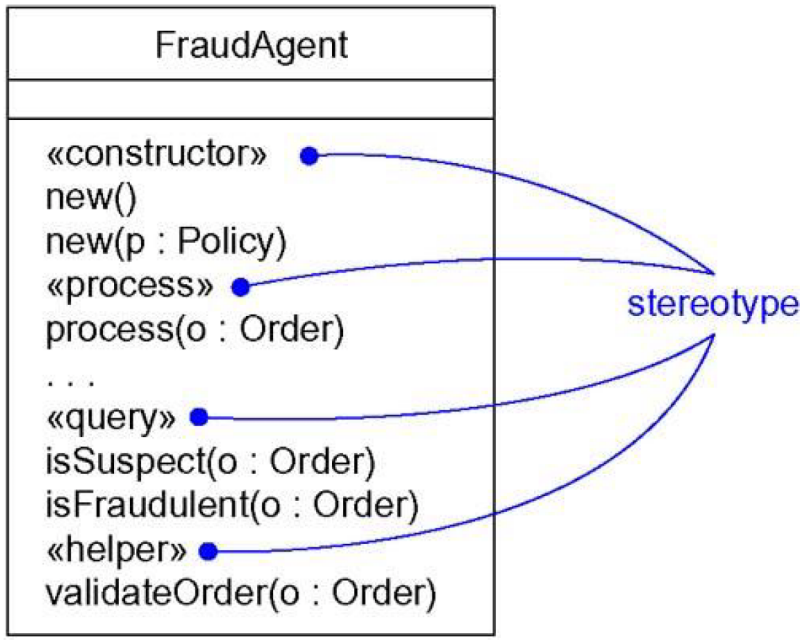
Organizing Attributes and Operations

- When drawing a class, you don't have to show every attribute and every operation at once.
- In most cases, you **can't** (too many) and you **shouldn't** (only relevant ones should be shown).
- You can **elide a class**, showing only some or none of its attributes and operations.
- An empty compartment doesn't mean there are no attributes/operations, only that they were not shown.
- You can indicate more attributes or operations exist by ending the list with an ellipsis **(...)**.



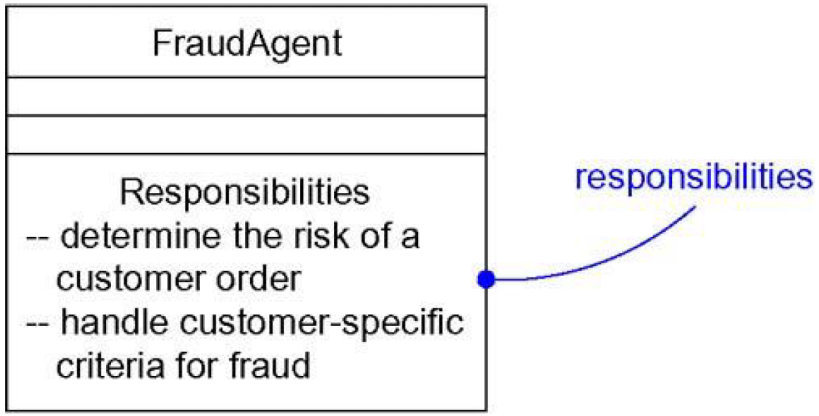
Using Stereotypes for Organization

- To better organize long lists of attributes and operations, you can prefix each group with a **descriptive category**.
- This is done by using **stereotypes** (e.g., «constructor», «process», «query», «helper»).
- Stereotypes improve clarity and help group related operations.



Responsibilities

- A **responsibility** is a contract or obligation of a class.
- All objects of a class share the same state and behavior.
- Attributes and operations are the means to fulfill responsibilities.
- Examples:
  - **Wall** → knows height, width, thickness.
  - **FraudAgent** → processes orders, detects fraud.
  - **TemperatureSensor** → measures temperature, raises alarms.
- Graphically shown in a separate compartment at the bottom of the class icon.



Basic Structure Element: Relationships

Terms & Concepts

- A **relationship** is a connection among things/classes.
- In OO modeling, the three most important are: **dependencies** (using/trace/refine/bind), **generalizations** (parent → child), **associations** (structural links among instances).
- Purpose: model how the vocabulary elements of the system *collaborate* and compose larger structures.

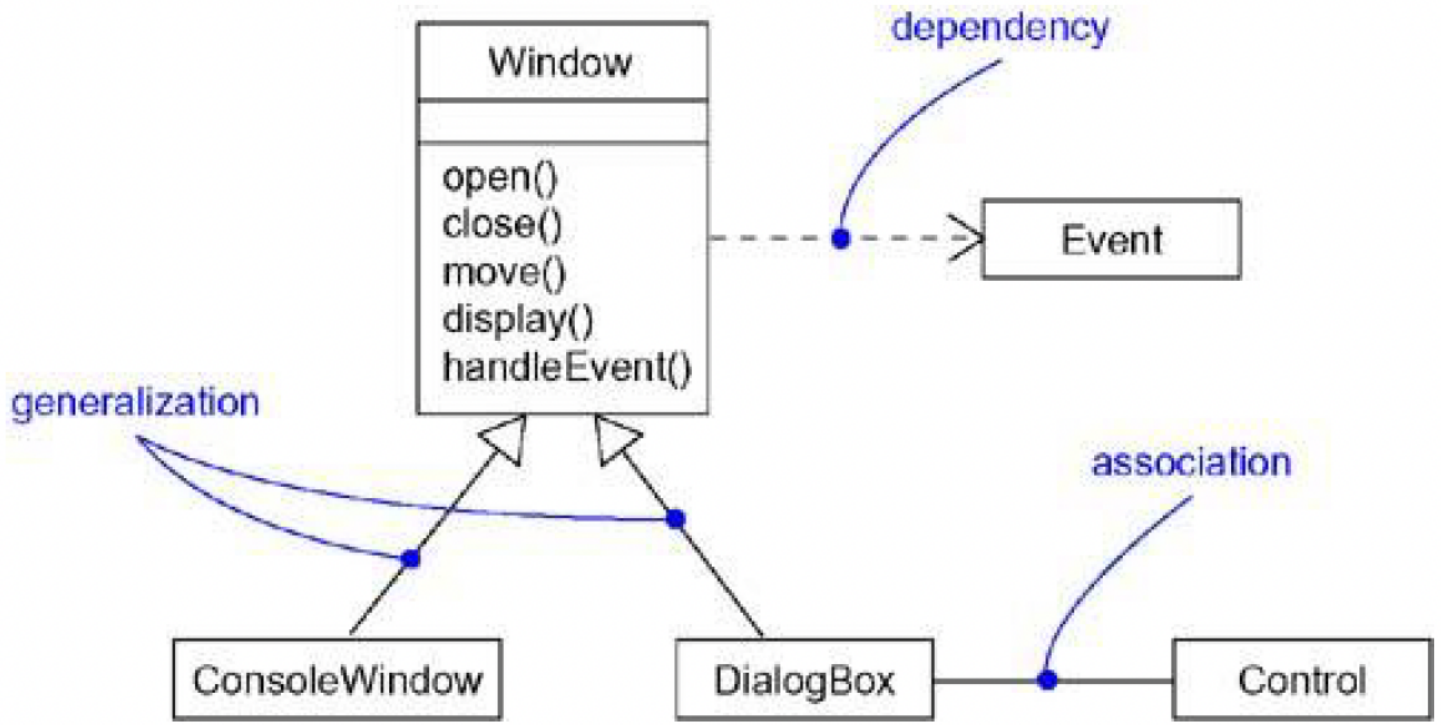
Getting Started (intuition)

- Like a house: walls–doors–windows–cabinets–lights connect structurally; there are kinds (bay, patio) ⇒ generalization.
- Balance is key: over-engineer ⇒ tangled web; under-engineer ⇒ lost richness.

Notation (UML)

- Rendered as **paths/lines** with distinct styles to distinguish dependency, generalization, association.

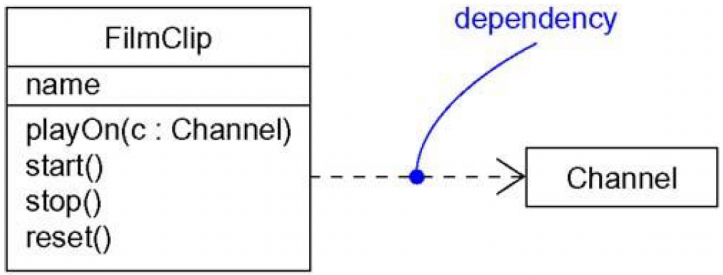
Relationships — UML Notation



- **Dependency** (dashed arrow) — Window uses Event.
- **Generalization** (hollow triangle) — ConsoleWindow, DialogBox specialize Window.
- **Association** (solid line) — DialogBox associated with Control.

Dependency Relationship

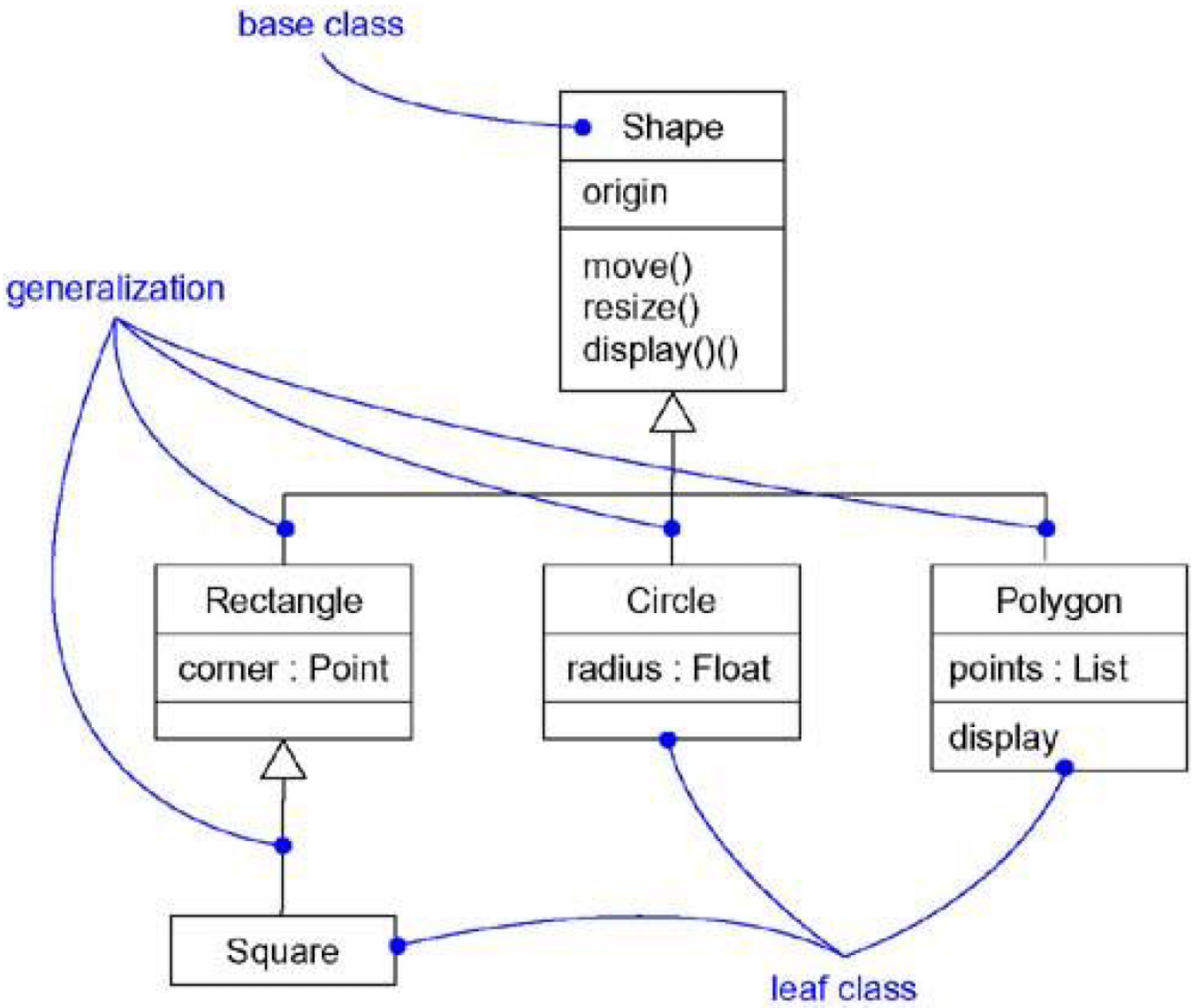
- A **dependency** is a **using relationship**: a change in one element may affect another that uses it, but not necessarily the reverse.
- **Graphical notation**: a **dashed directed line**, pointing to the element being depended on.
- Common use: between **classes**, when one class uses another as an argument in an operation signature.
- Example: `Window` depends on `Event`, `FilmClip` depends on `Channel`.
- Dependencies can also apply to **notes, packages**, etc.
- Can be named, but usually stereotypes are preferred to distinguish flavors of dependencies.



Generalization Relationship

- A **generalization** relates a general thing (**parent/superclass**) to a more specific kind of thing (**child/subclass**).
- Also called an **"is-a-kind-of"** relationship.
  Example: `BayWindow` is-a-kind-of `Window`.
- Meaning: the **child is substitutable for the parent**, but not the reverse.
- Child **inherits attributes & operations** from the parent.
  May add new features or **override** parent operations (**polymorphism**).
- **Graphical notation**: solid line with hollow triangle pointing to parent.
- Special terms:
  - **Base/Root class** → no parents, but has children.
  - **Leaf class** → no children.
  - **Single inheritance** → exactly one parent.
  - **Multiple inheritance** → more than one parent.

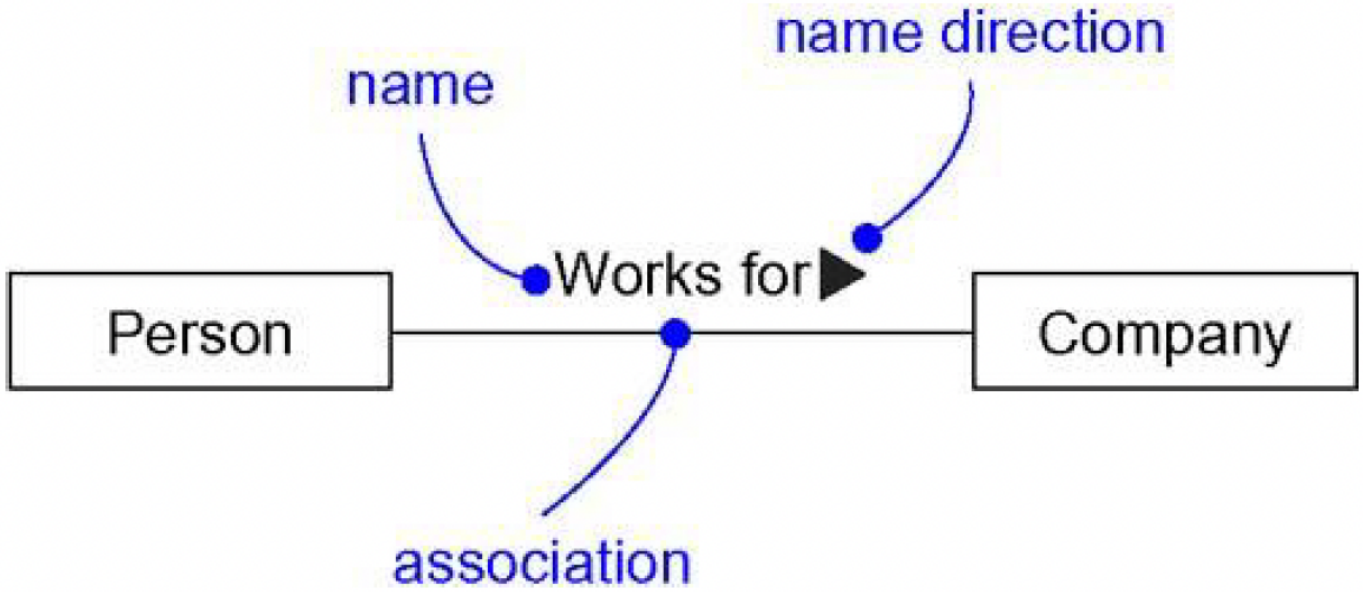
Generalization — UML Notation



Association Relationship

- An **association** is a structural relationship that connects objects of one class to objects of another.
- Allows navigation between objects of the connected classes.
- Can be:
  - **Binary association** → connects two classes.
  - **N-ary association** → connects more than two classes.
  - **Reflexive** → both ends circle back to the same class.
- **Graphical notation**: a solid line between classes.
- Associations may have:
  - A **name** describing the relationship.
  - A **name direction triangle** to avoid ambiguity.
- Use associations to show **structural relationships among objects**.

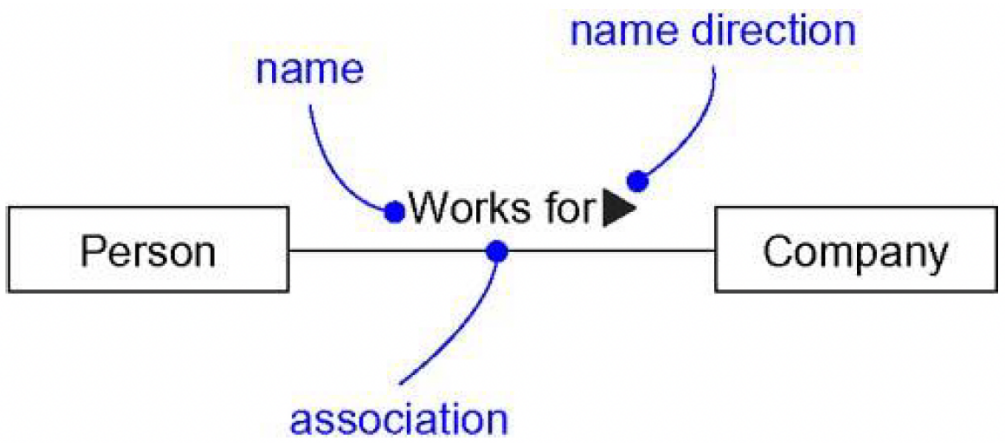
Association — UML Notation



- Example: `Person` ↔ `Company`.
- Association name: **Works for**.
- Name direction triangle indicates how to read the label.

Association Roles

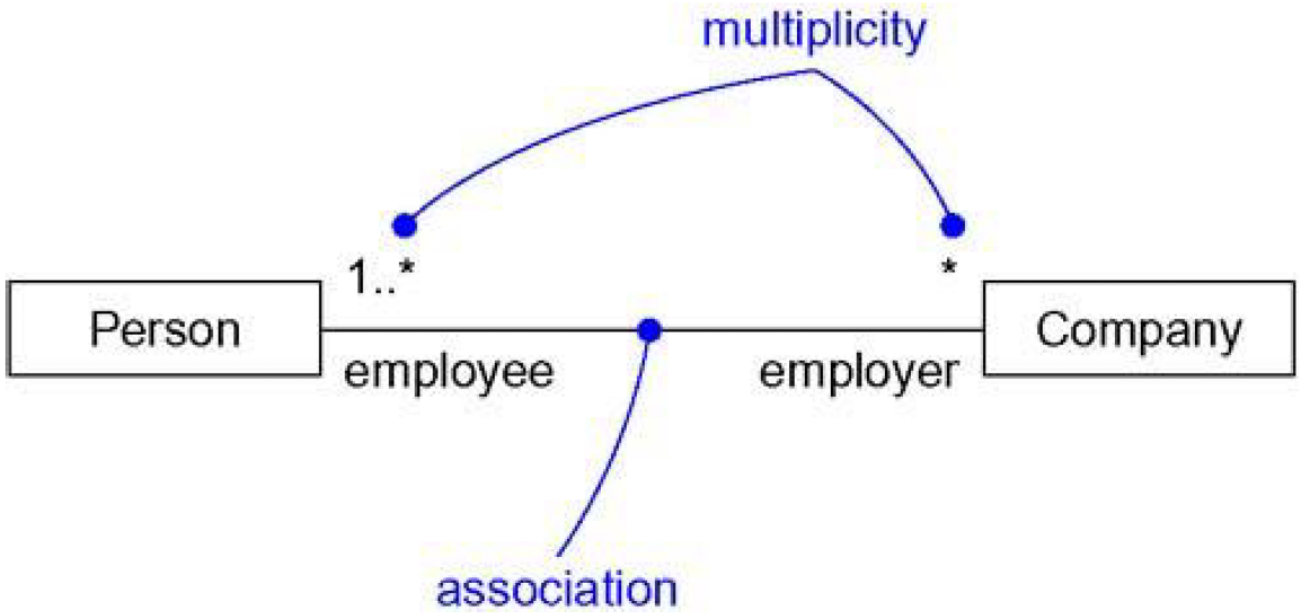
- When a class participates in an association, it has a specific **role**.
- A role is the **face a class presents** in that relationship.
- Roles can be explicitly named for clarity.
- Example:
  - `Person` plays the role of **employee**.
  - `Company` plays the role of **employer**.
- The same class can play the same or different roles in other associations.
- An **instance of an association** is called a **link**.



Association Multiplicity

- An **association** specifies a structural relationship among objects.
- **Multiplicity** defines **how many objects** may participate at each end of the association.
- Expressed as a **range** or an **exact value**.
- Common notations:
  - **1** → exactly one.
  - **0..1** → zero or one.
  - **0..*** → many (zero or more).
  - **1..*** → one or more.
  - Exact number → e.g., **3**.
- Example: In a Person–Company relationship:
  - A `Company` may employ **many** persons.
  - A `Person` works for **at least one** company.
- Complex multiplicities possible (e.g., 0..1, 3..4, 6..*).

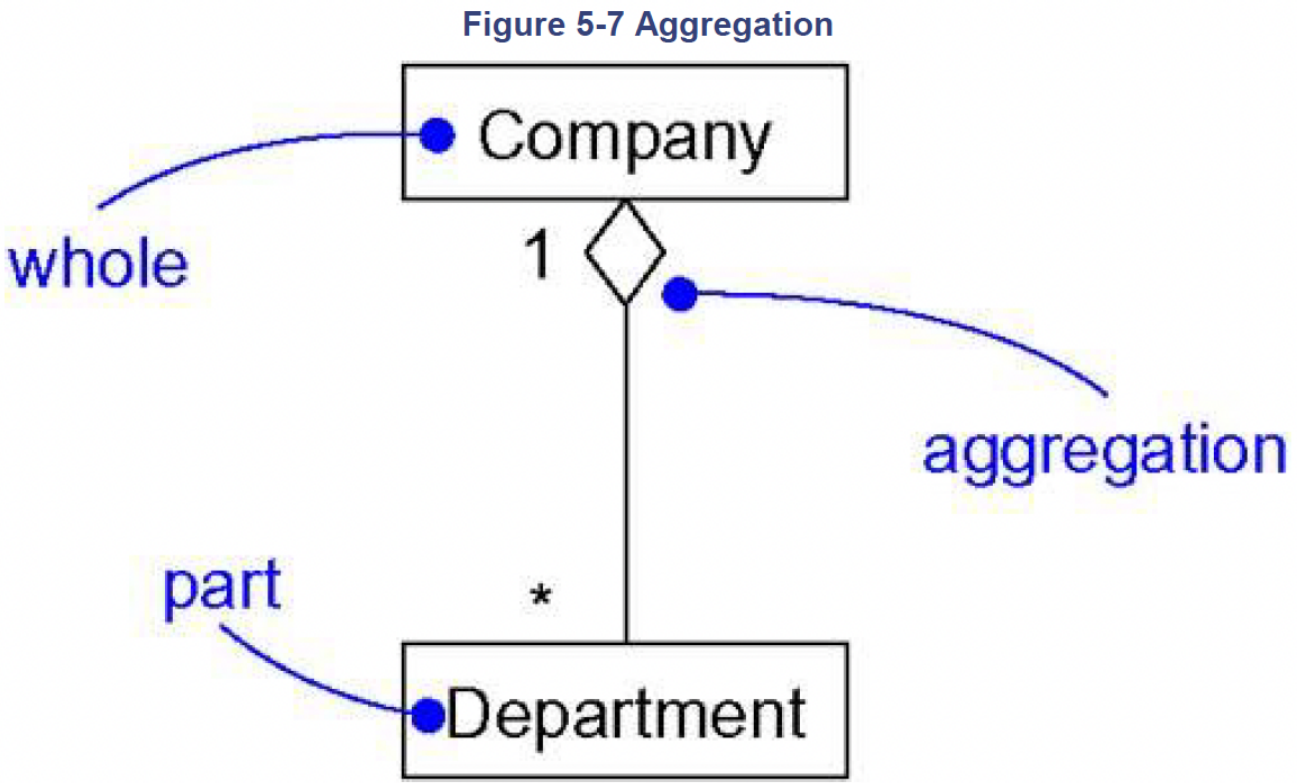
Association Multiplicity — UML Notation



Aggregation Relationship

- A plain association = relationship between **peers** (same level).
- Sometimes, one class is a **whole** and the other its **parts**.
- This is called **Aggregation**, a special form of association.
- Represents a **"has-a" relationship**:
  - A `Company` **has** Departments.
- Graphical notation:
  - Plain association + **open diamond** at the "whole" end.
- Meaning of simple aggregation is **conceptual only**.
  - Open diamond distinguishes whole vs part.
  - Does not affect navigation or object lifetimes.

Aggregation — UML Notation



- Example: `Company` (whole) ◊— `Department` (part).
- "Company has Departments" = conceptual whole–part relationship.

Common Mechanisms in UML

Definition: Fundamental concepts consistently applied across UML to ensure clarity, precision, and flexibility in modeling.

Purpose:

- Provide a **uniform way** to describe and interpret UML models
- Help in **simplifying communication** of complex system designs
- Enable **extension & customization** when needed

Types of Common Mechanisms:

- Specifications
- Common Divisions
- Adornments
- Extensibility Mechanisms

Specifications in UML

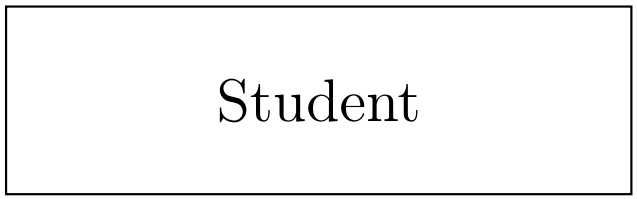
Definition: Specifications provide detailed descriptions of model elements so their role and meaning are clear.

Purpose:

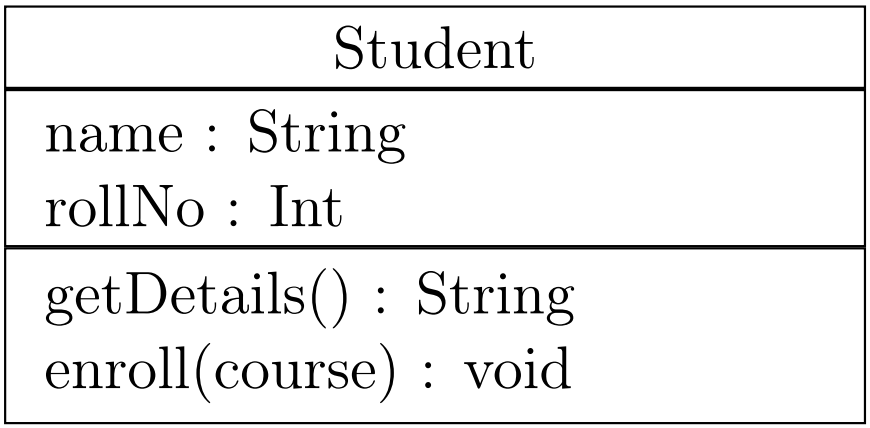
- Clarify elements with attributes, operations, signatures, and behaviors
- Make system components precise and unambiguous
- Example: Classes with rich specifications

Specifications in UML

Without Specification



With Specification



Common Divisions in UML

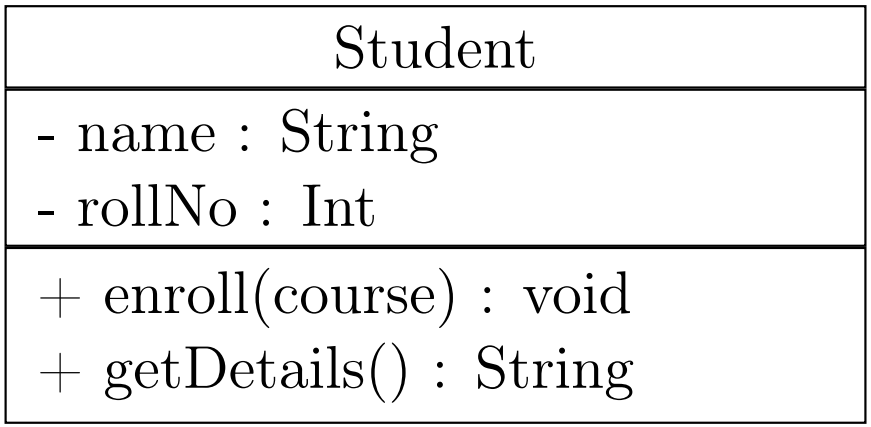
Definition: Common divisions distinguish between concepts that are closely related, ensuring clarity in UML models.

Two Key Divisions:

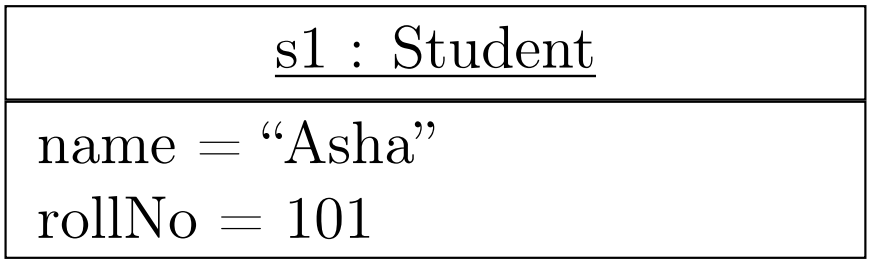
- **Abstraction vs. Manifestation** - Class = abstraction (blueprint) - Object = manifestation (instance of class)
- **Interface vs. Implementation** - Interface = contract (what must be done) - Implementation = realization (how it is done)

Common Divisions: Abstraction vs Manifestation

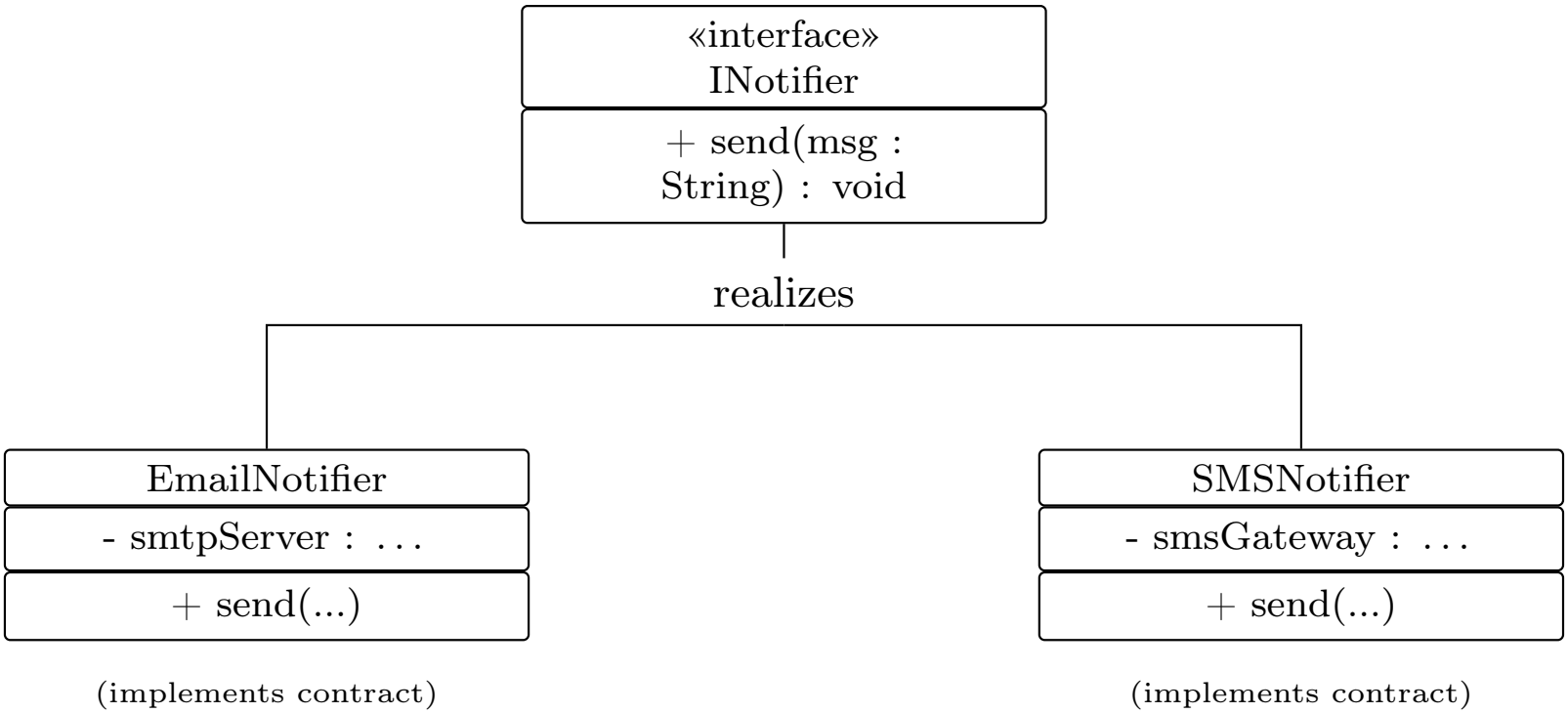
Class (Abstraction)



Object (Manifestation)



Common Divisions: Interface vs Implementation



Adornments in UML

Definition: Adornments are textual or graphical items added to a UML element's basic notation to visualize extra details from that element's specification.

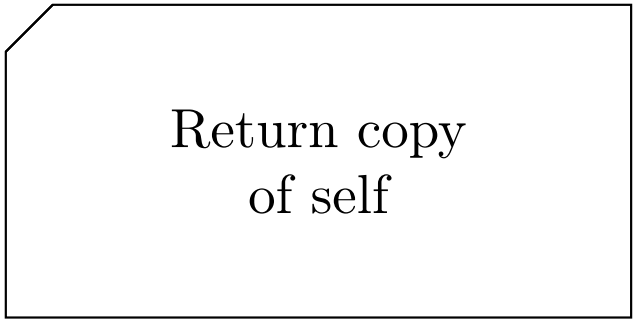
Why use them?

- Make implicit details explicit (roles, multiplicities, constraints, comments)
- Improve clarity without changing the model's semantics
- Keep diagrams concise while preserving meaning

Adornments: Association (roles & multiplicity)

- Base notation: a single line between *Person* and *Car*
- Adornments added:
  - **Role names**: *owner*, *vehicle*
  - **Multiplicity**: *1* (Person end), *0..** (Car end)

Adornments: UML Note

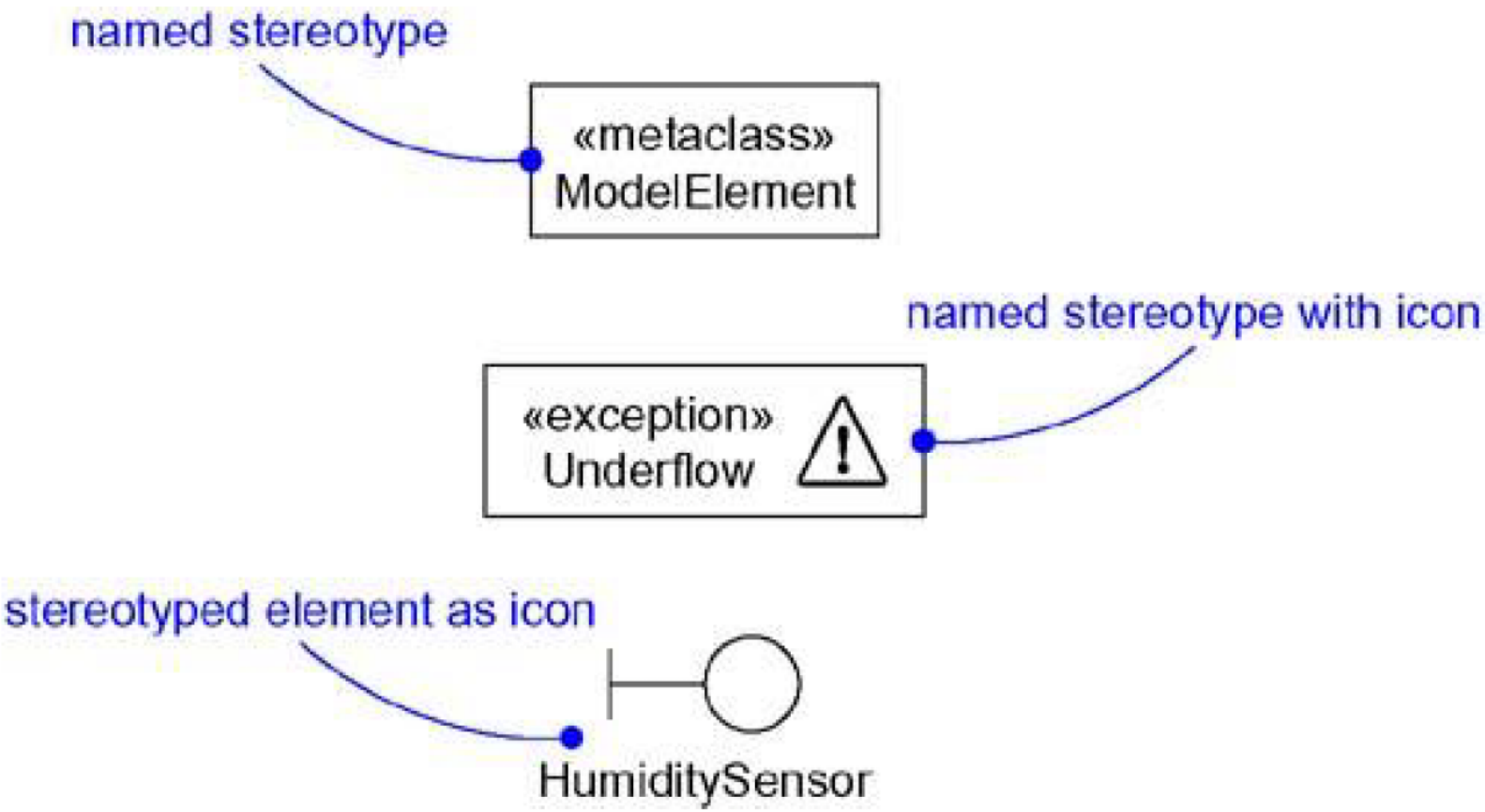


- A **Note** is a UML adornment for comments, requirements, or explanations.
- Carries **no semantic impact** — it does not change the model.

Extensibility Mechanisms in UML

- UML can be customized & extended for domain-specific needs
- Provides three main mechanisms:
  1. **Stereotypes**
     - Extend UML vocabulary
     - Mark existing elements with domain-specific meaning
  2. **Tagged Values**
     - Add metadata as keyword–value pairs
     - Example: {author = "Sharma", version = "2.0"}
  3. **Constraints**
     - Define rules/conditions that must always hold true
     - Often expressed using **OCL (Object Constraint Language)**

Stereotypes in UML



Tagged Values

- **What:** A *tagged value* is a key–value pair attached to a UML element; think of it as metadata.
- **Why:** Adds extra properties without changing the element; complements stereotypes.
- **Not an attribute:** Attributes describe instances; a tag describes the *model element* itself.
- **Notation:** `{tag = value}` placed under the element name. (You may write just the value when unambiguous, e.g., an enum literal.)
- **Examples:** Node `{processors = 3}`; Component `«library»` with `(serverOnly)`.
- **Typical uses:** Code generation / configuration metadata (e.g., target language, author, version).

Tagged Values

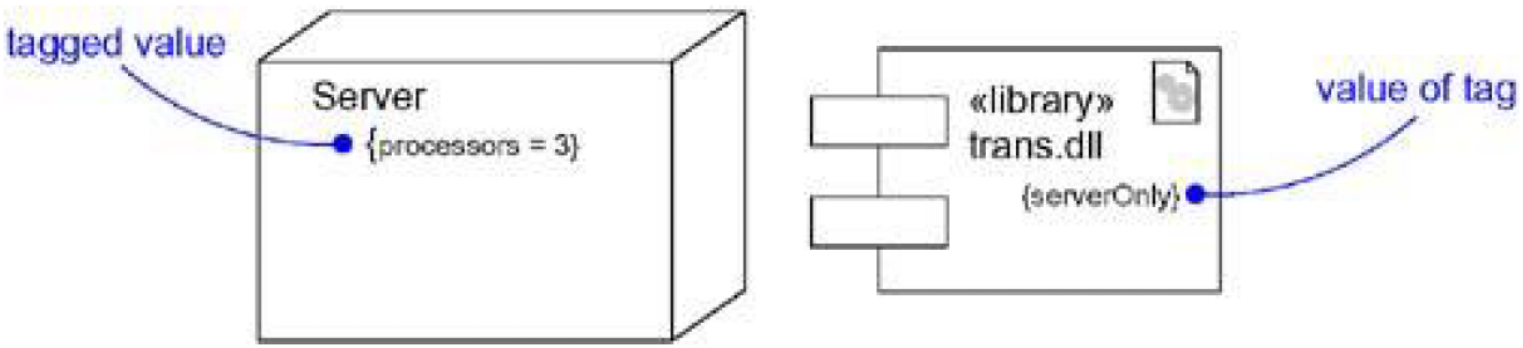


Figure: Tagged values on a *Server* node and a «library» component.

Constraints

What they are

- Rules/conditions that **must always hold** for model elements
- Extend element **semantics** (add/modify rules)

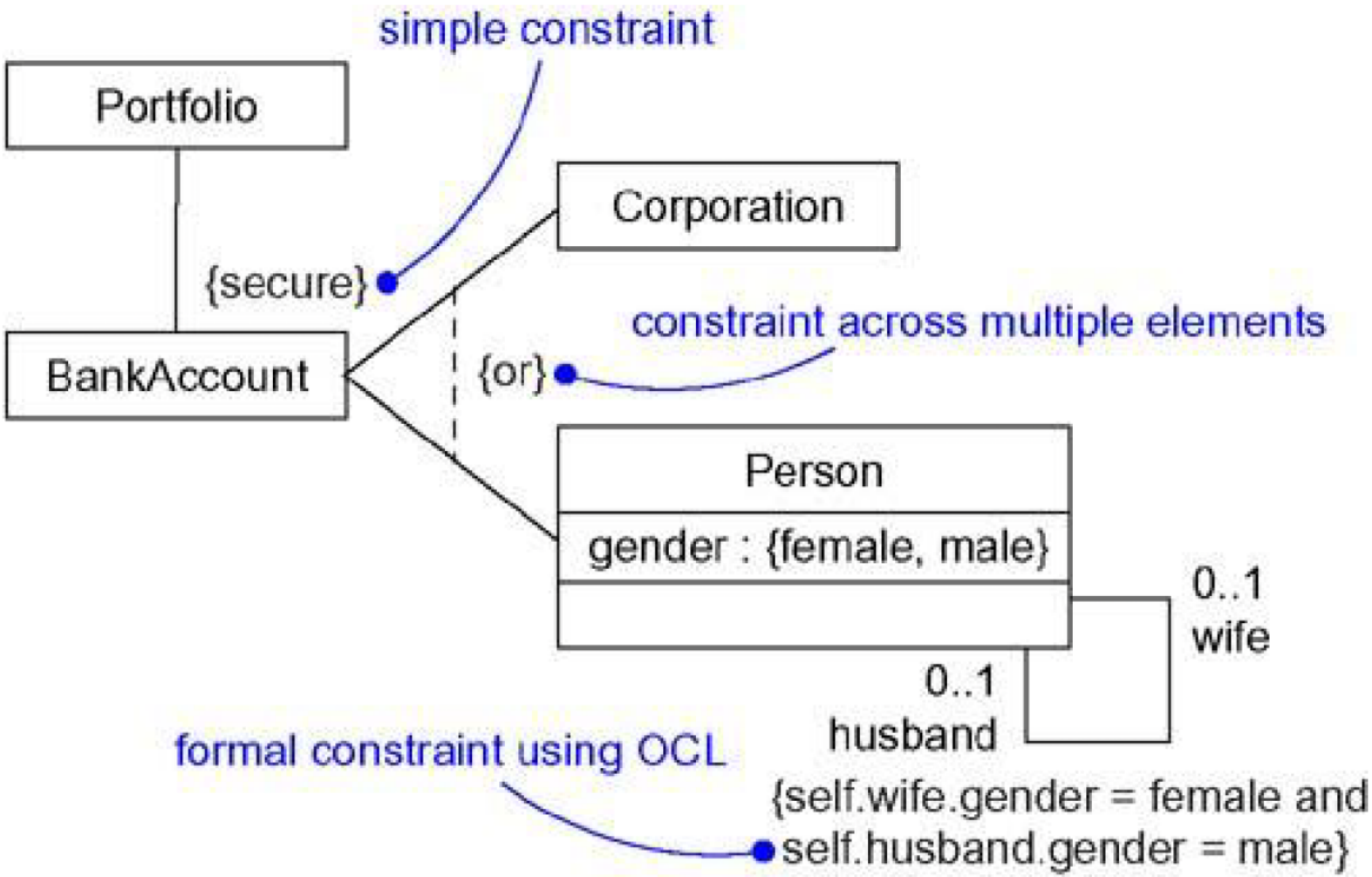
Why use

- Capture timing, safety, security, ordering, changeability, etc.
- Keep diagrams precise without new element types

Notation

- Text in {braces} near the element or linked by a dependency
- May be formalized in **OCL** (Object Constraint Language)

Constraints



Formal Constraint using Object Constraint Language (OCL)

Advanced Structural Modelling

Overview: Advanced structural modelling extends the basic structure concepts (classes, relationships, class diagrams) with richer mechanisms to capture complex designs.

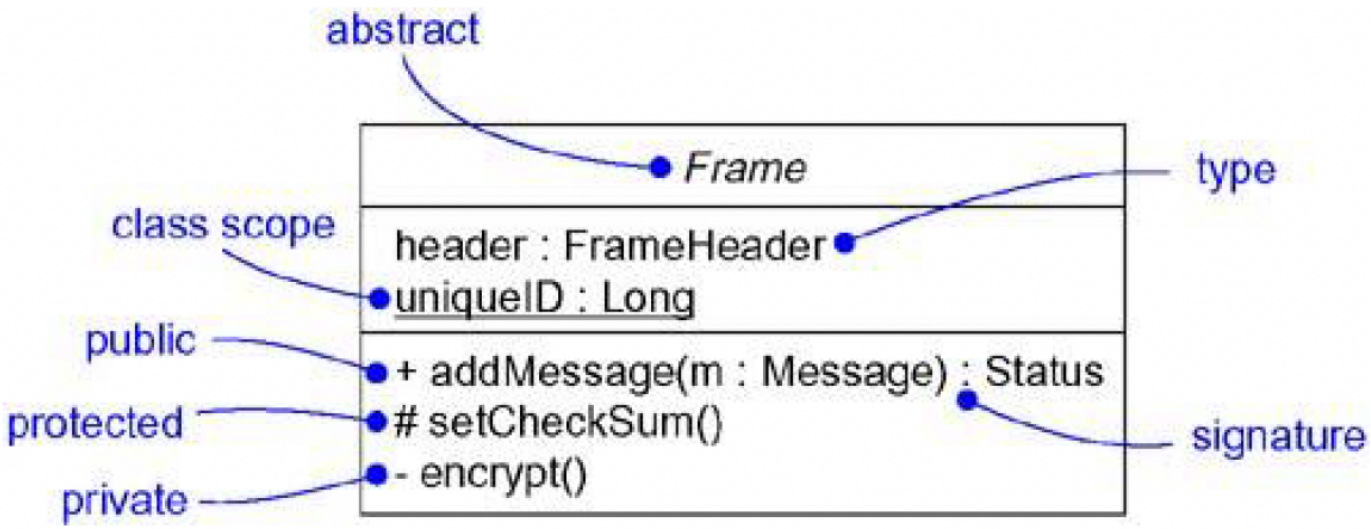
Key Elements:

- **Advanced Classes** – abstract classes, parameterized classes.
- **Advanced Relationships** – aggregation, composition, dependency, association classes.
- **Interfaces** – define contracts without implementation.
- **Types and Roles** – specify alternative views and roles in associations.
- **Packages** – group related elements for modularity and reuse.
- **Object Diagram** – snapshot of system objects and their links at a given time.

Key Ideas:

- Classes are the most important building block, but belong to a more general concept: **classifiers**.
- A **classifier** describes structural and behavioral features. Examples: classes, interfaces, datatypes, signals, components, nodes, use cases, subsystems.
- Advanced features of classes include:
  - Multiplicity, visibility, signatures.
  - Polymorphism and special properties of attributes/operations.
- UML lets you model the **semantics of a class** at different levels of detail — from conceptual responsibilities to implementation-level design.
- Choosing the right classifier is essential to accurately represent real-world abstractions.

Advanced Classes — UML Notation



- Shows advanced UML class features: abstract classes, scope, visibility, signatures, and types.
- Used to visualize, specify, construct, and document classes in detail.

Classifiers

Key Ideas:

- **Classifier** → an element that has **instances**, with **structural features** (attributes) and **behavioral features** (operations).
- Every instance of a classifier shares the same features.
- Most important classifier = **Class**, but UML provides many others.

Examples of Classifiers:

- **Classes** – e.g., Customer, Transaction.
- **Components** – e.g., Pricing component deployed on client nodes.
- **Interfaces** – define contracts (no attributes).
- **Datatypes, Signals, Nodes, Use Cases, Subsystems**.

Notes:

- Some UML elements (e.g., packages, generalizations) are **not classifiers** (no instances).
- When modeling with classifiers, you can use all advanced features (multiplicity, visibility, signatures, polymorphism, etc.).

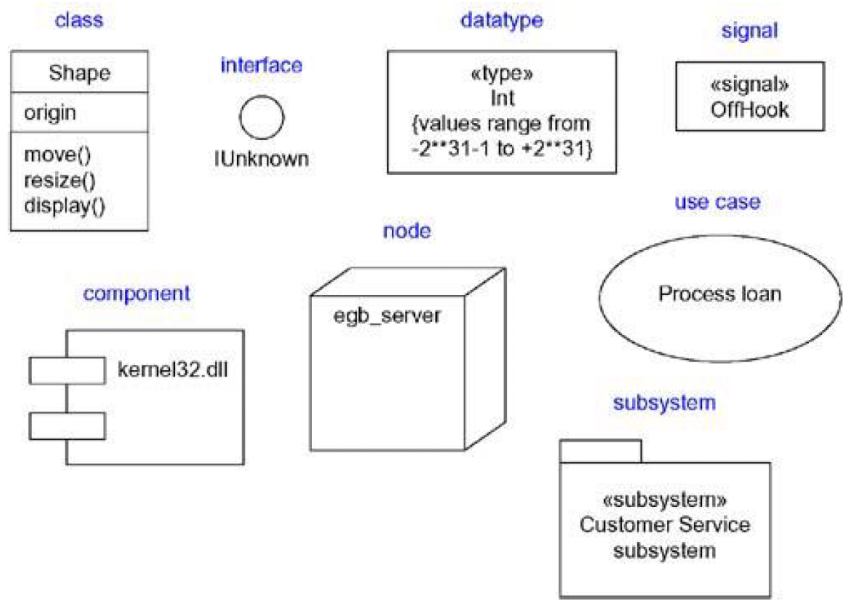
Classifiers in UML

Classifier	**Description**
Interface	Collection of operations used to specify a service of a class or component.
Datatype	Type with no identity, including primitive built-in types (e.g., numbers, strings) and enumerations (e.g., Boolean).
Signal	Specification of an asynchronous stimulus communicated between instances.
Component	Physical and replaceable part of a system that conforms to and provides realization of a set of interfaces.
Node	Physical element at run time representing a computational resource, usually with memory and processing capability.
Use Case	Description of a sequence of actions, including variants, that yields an observable result of value to an actor.
Subsystem	Grouping of elements, some of which specify the behavior offered by others contained within.

Classifier Icons in UML

Design Rationale

- A *single icon* for all classifiers is misleading: **classes** (logical) vs **components** (physical) are materially different.
- A *unique icon for every kind* is overkill: **classes** and **datatypes** are not that different.
- UML strikes a **balance**:
  - Materially different classifiers get **distinct icons**.
  - Others reuse a base icon with **keywords** (e.g., `<type>`, `<signal>`, `<subsystem>`).



Visibility in UML

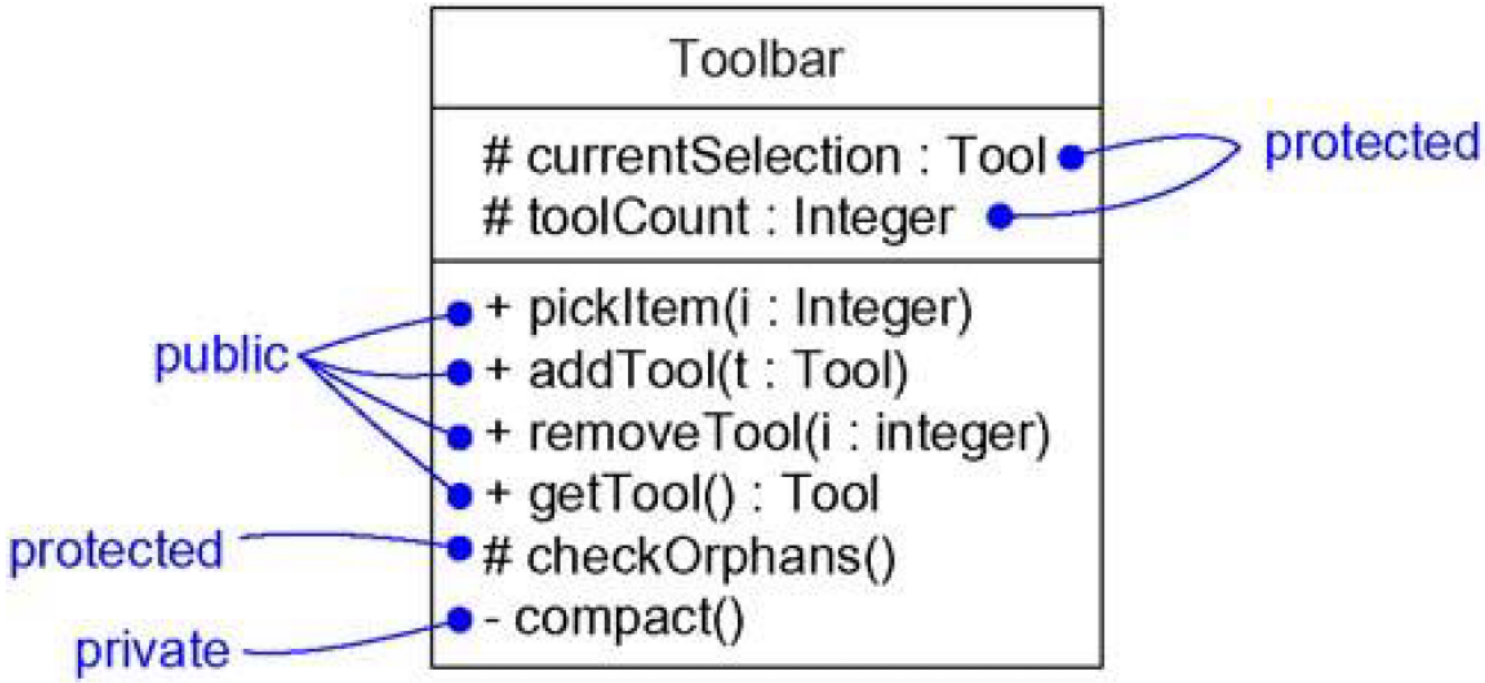
Key Ideas:

- Visibility specifies whether a feature (attribute/operation) can be used by other classifiers.
- Three levels of visibility in UML: **public**, **protected**, **private**.
- Supports **information hiding** $\rightarrow$ exposing only what is necessary, hiding implementation details.
- Matches semantics of most programming languages (C++, Java, Ada, Eiffel).
- If not explicitly specified, visibility is generally assumed to be **public**.

Visibility Levels

1. public	Any outside classifier with visibility to the given classifier can use the feature; specified by prepending the symbol +
2. protected	Any descendant of the classifier can use the feature; specified by prepending the symbol #
3. private	Only the classifier itself can use the feature; specified by prepending the symbol -

Visibility — UML Example



- **Public (+)** → accessible to all (e.g., pickItem, addTool).
- **Protected (#)** → accessible to descendants (e.g., currentSelection, checkOrphans).
- **Private (-)** → accessible only within the class (e.g., compact).

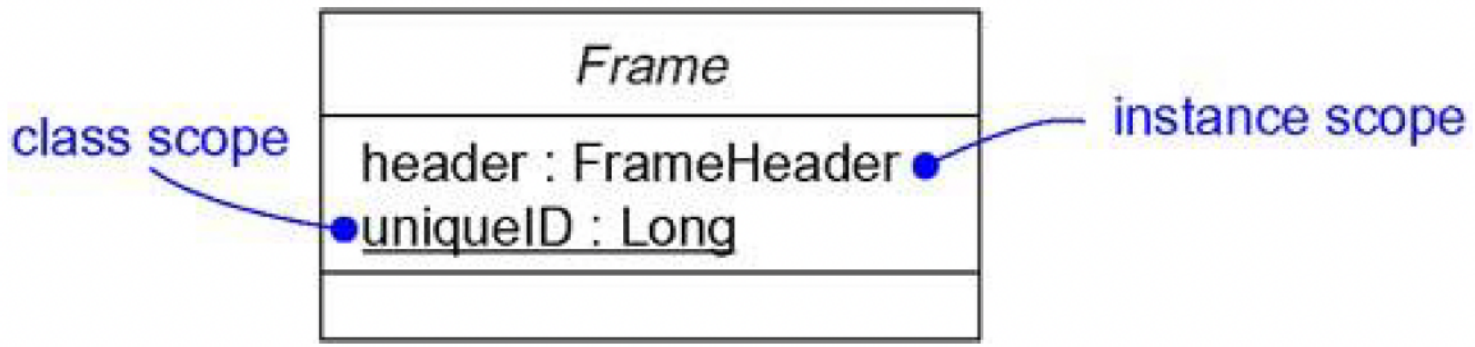
Scope of Features

Owner Scope (UML)

- Specifies whether an attribute/operation exists **per instance** or **once per classifier**.
- Two kinds of scope:
  - **Instance scope** — default; no adornment.
  - **Classifier scope** — shown by underlining the feature name.
- Most features are **instance scoped**.
- Common classifier-scoped uses: shared private data (e.g., *unique ID* generator) and factory/creation operations.
- *Note:* Classifier scoped ↔ `static` (C++).

1. instance	Each instance of the classifier holds its own value for the feature.
2. classifier	There is just one value of the feature for all instances of the classifier.

Scope — UML Notation



- Underlined feature = **classifier scoped** (shared across all instances).
- Non-underlined feature = **instance scoped** (separate per object).

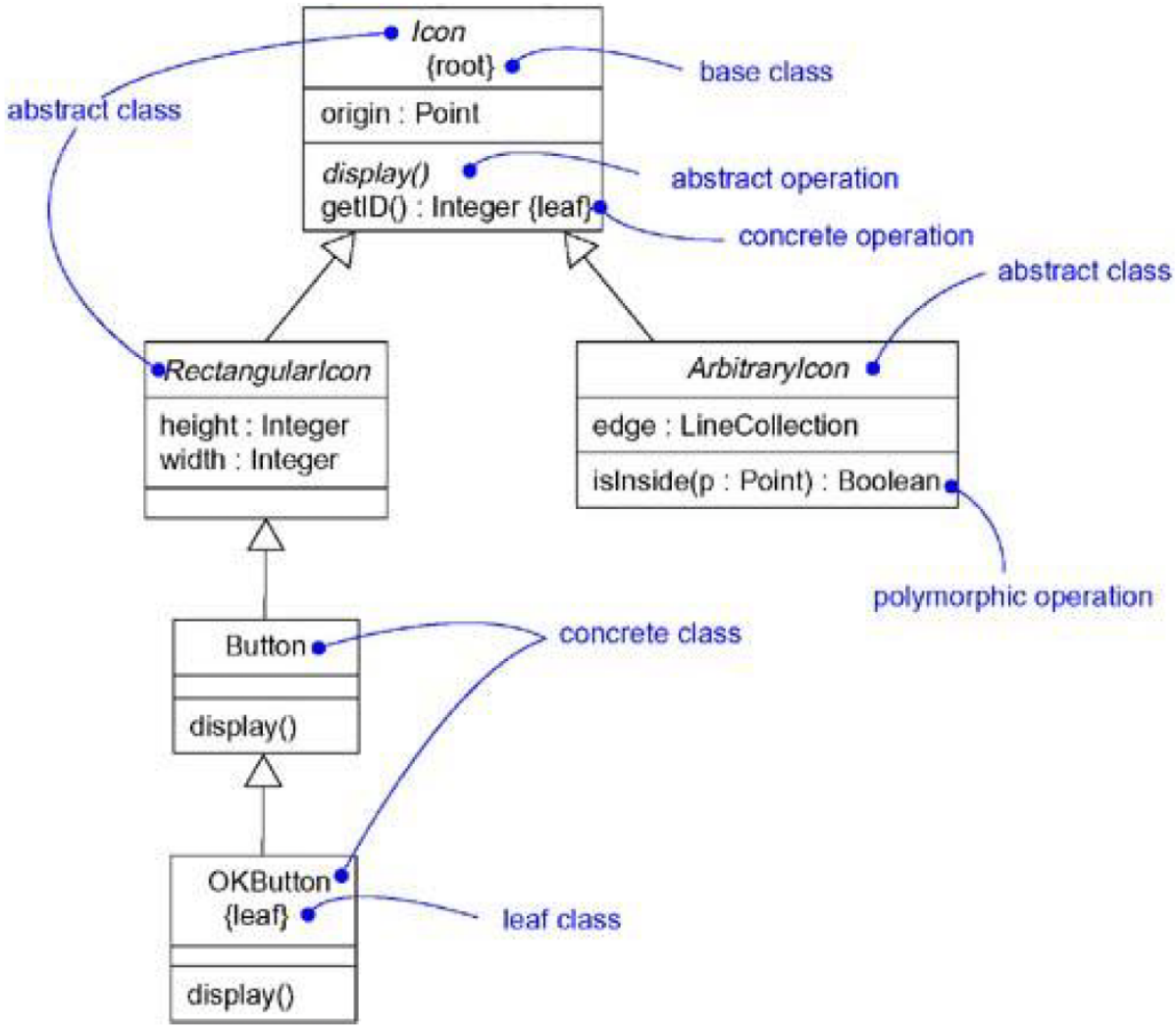
Abstract, Root, Leaf, and Polymorphic Elements

Key Ideas:

- **Abstract Class** – cannot have direct instances; name in *italics*. Example: `Icon`, `RectangularIcon`, `ArbitraryIcon`.
- **Concrete Class** – may have direct instances. Example: `Button`, `OKButton`.
- **Root Class** – has no parents; marked with property `root`. Example: `Icon`.
- **Leaf Class** – has no children; marked with property `leaf`. Example: `OKButton`.
- **Polymorphic Operation** – can be overridden in subclasses. Example: `display()`, `isInside()`.
- **Abstract Operation** – incomplete; requires subclass implementation (name in *italics*). Example: `Icon::display()`.
- **Leaf Operation** – cannot be overridden; marked with `leaf`. Example: `Icon::getID()`.

Note: Abstract operations map to C++ **pure virtual**, while leaf operations map to C++ **non-virtual**.

UML Example — Abstract, Root, Leaf, Polymorphic

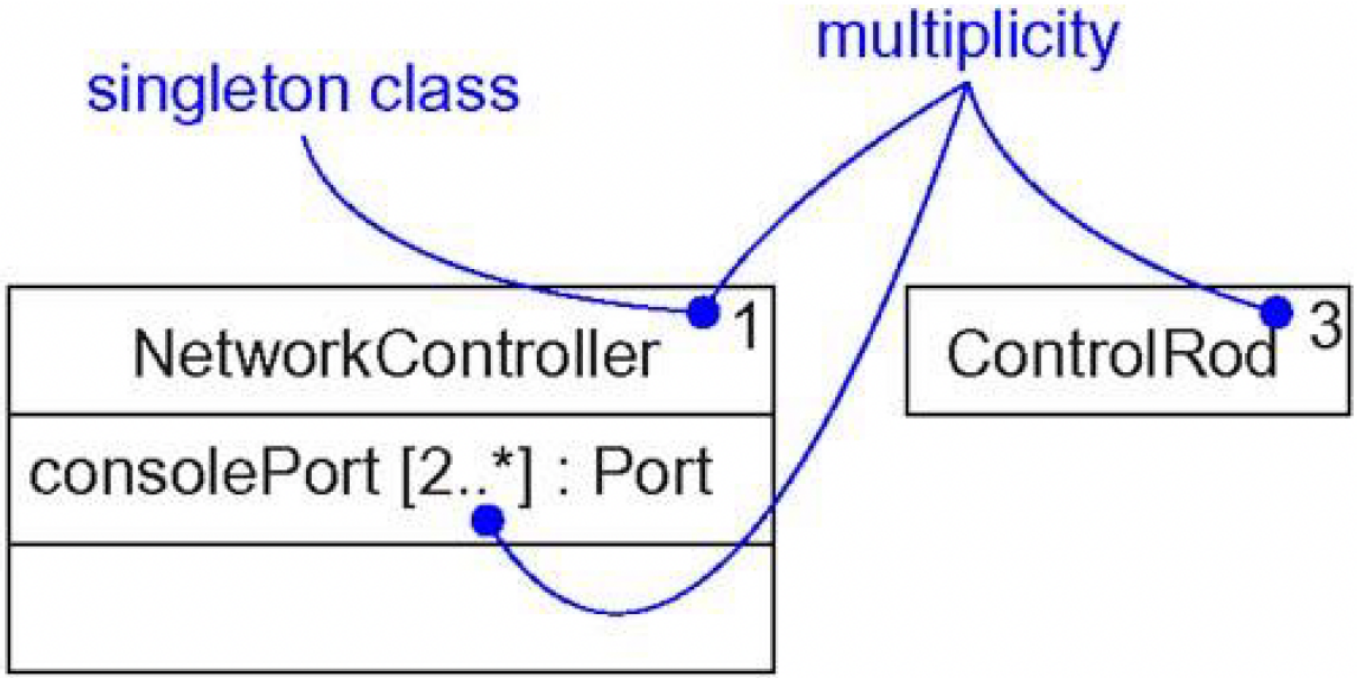


Multiplicity in UML

Key Ideas:

- Multiplicity = number of instances a class (or attribute/association) may have.
- Default: many instances.
- Restrictions:
  - **0 instances** → utility class (only class-scoped features).
  - **1 instance** → singleton class.
  - **n instances** → exact number specified.
  - **Range** → e.g., `2..*` means at least 2, unbounded above.
- Applies to:
  - **Classes** → total number of instances allowed.
  - **Associations** → allowable connections between instances.
  - **Attributes** → number of values per instance.
- UML notation: multiplicity written in the **upper-right corner** of the class, or in **brackets** next to attributes.

Multiplicity — UML Example



- **NetworkController** → singleton (only 1 instance).
- **ControlRod** → exactly 3 instances.
- **consolePort [2..*]** → every NetworkController must have at least 2 console ports.

Attributes in UML

Key Ideas:

- Attributes model the **structural features** of a class.
- Minimal form: just the attribute name.
- Extended form: **visibility**, **scope**, **multiplicity**, **type**, **initial value**, and **properties**.

Attributes in UML

Attribute Declarations:

Declaration	Meaning
`origin`	Name only
`+ origin`	Visibility and name
`origin : Point`	Name and type
`head : *Item`	Name and complex type
`name [0..1] : String`	Name, multiplicity, and type
`origin : Point = (0,0)`	Name, type, and initial value
`id : Integer {frozen}`	Name and property

Attribute Properties:

1. changeable	No restrictions on modifying the attribute's value.
2. addOnly	For multiplicity > 1, additional values may be added, but once created, values cannot be removed or altered.
3. frozen	Attribute's value cannot be changed after initialization.

Unless otherwise specified, attributes are always ***changeable****. Use* ***frozen*** *for constant or write-once attributes.*

Operations in UML

Key Ideas:

- Operations model the **behavioral features** of a class (services it provides).
- Minimal form: just the operation name.
- Extended form: specify **visibility**, **scope**, **parameters**, **return type**, **concurrency**, and other properties.
- An operation's name + parameters (+ return type) = **operation signature**.
- UML distinction: **Operation** (specification of service) vs **Method** (implementation/body).
- Inheritance & polymorphism: same operation may have multiple methods; run-time dispatch chooses the appropriate one.

UML Syntax:

```
[visibility] name (parameter-list) [:  return-type]
[= initial-value] [{property-string}]
```

Operation Declarations in UML

Examples of Legal Operation Declarations:

• `display`	Name only
• `+ display`	Visibility and name
• `set(n : Name, s : String)`	Name and parameters
• `getID() : Integer`	Name and return type
• `restart() {guarded}`	Name and property

Operation Signature:

In an operation's signature, you may provide zero or more parameters, each of which follows the syntax:

```
[direction] name : type [= default-value]
```

Parameter Directions in UML

Direction may be any of the following values:

• `in`	An input parameter; may not be modified
• `out`	An output parameter; may be modified to communicate information to the caller
• `inout`	An input parameter; may be modified

Operation Properties in UML

In addition to the *leaf* property described earlier, there are four defined properties that you can use with operations:

1. `isQuery`	Execution of the operation leaves the state of the system unchanged. It is a pure function with no side effects.
2. `sequential`	Callers must coordinate outside the object so that only one flow is active at a time. With multiple flows of control, semantics and integrity cannot be guaranteed.
3. `guarded`	The integrity of the object is guaranteed by sequentializing all calls to its guarded operations. Effectively, only one operation at a time can be invoked, reducing this to sequential semantics.
4. `concurrent`	The integrity of the object is guaranteed by treating the operation as atomic. Multiple calls may occur simultaneously and proceed concurrently with correct semantics. Concurrent operations must be designed to perform correctly with concurrent, sequential, or guarded operations.

Interfaces and Boundaries

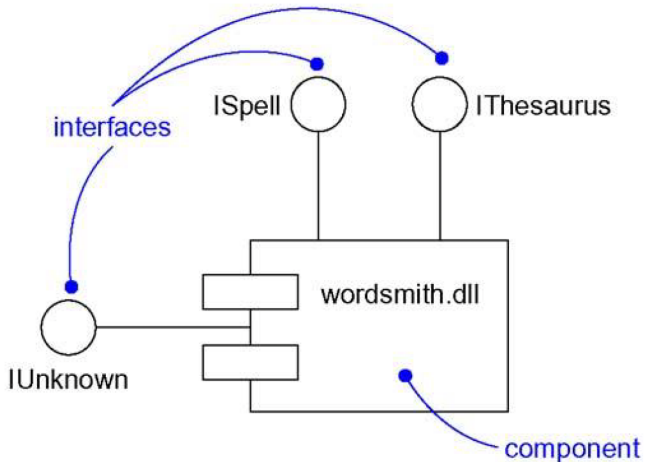
- An **interface** tells *what* something does, not *how*
- Interfaces act as clear **boundaries** between parts of a system
- Example: in buildings, walls, pipes, and doors can change without touching the foundation
- Using standard interfaces = easier to replace, reuse, and reduce cost
- In UML: an interface is a set of services a class or component provides
- Clients use the interface; actual code can be swapped anytime

Types, Roles, and Realization

- **Type** = what is promised at design time
- **Role** = how it behaves in a specific situation
- Static type = known before running; Dynamic type = actual object at runtime
- **Realization** = class or component fulfills the interface
- Good interfaces: clear, simple, and easy to understand
- Key idea: clear boundaries + simple contracts = systems easy to change and maintain

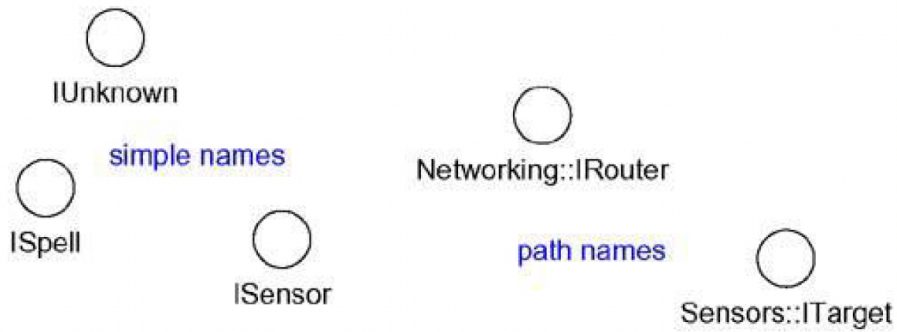
Terms and Concepts

- **Interface**: collection of operations specifying a service of a class/component
- **Type**: stereotype of a class that defines a domain of objects + operations
- **Role**: behavior of an entity in a specific context
- Graphical notation: interface as a **circle** (lollipop symbol)
- Expanded form: interface may appear as a stereotyped class with operations
- Interfaces can also specify contracts for use cases or subsystems



Names of Interfaces

- Every interface must have a **unique name** within its package
- **Simple name**: just the interface name (e.g., IUnknown, ISpell, ISensor)
- **Path name**: package name + interface name (e.g., Networking::IRouter, Sensors::ITarget)
- Names are short nouns/noun phrases from system vocabulary
- Convention: prepend `I` for interfaces (IUnknown, ISpelling) prepend `T` for types (TCharacter, TNatural)
- Interface names can include letters, numbers, and punctuation (but not "::" except for path)



Operations in Interfaces

- An **interface** = named collection of operations (services of a class/component)
- Interfaces have **no attributes** (no structure) and **no methods** (no implementation)
- They may declare any number of operations
- Operations can include: visibility, concurrency, stereotypes, tagged values, constraints
- Normal form (circle notation) hides operations Expanded form (stereotyped class) shows them in a compartment
- Operations may be listed by name or full signature
- Interfaces can also be associated with **signals**

Example: Operations in an Interface

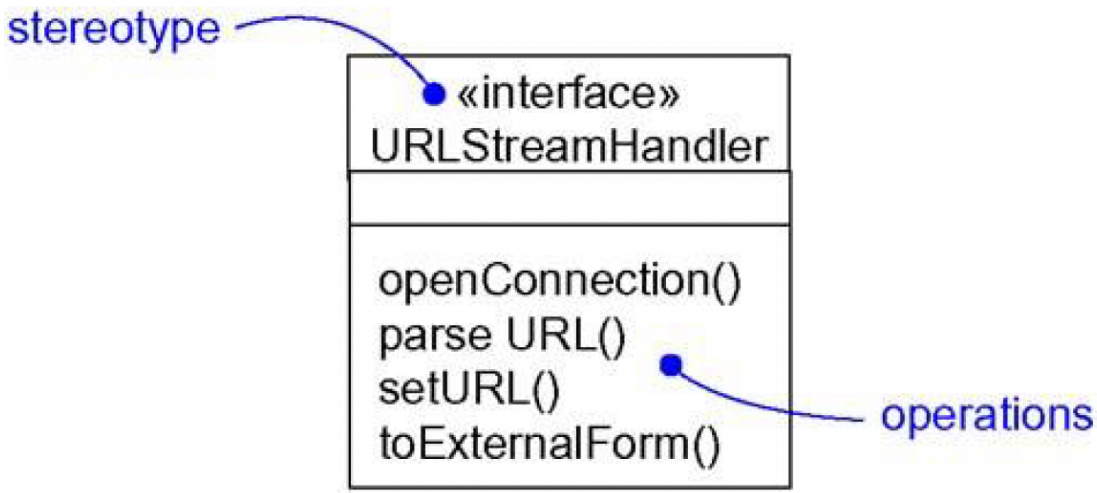


Figure: UML interface (URLStreamHandler) shown as a stereotyped class with its operations.

Relationships of Interfaces

- Interfaces can have **generalization**, **association**, and **dependency** links
- They also support **realization**:
  - Class/component commits to the contract of an interface
  - May realize many interfaces (must implement all operations)
  - May depend on many interfaces (expects others to honor contracts)
- Two notations:
  - **Simple**: lollipop (shows seam only)
  - **Expanded**: stereotyped class (shows operations)
- Realization = dashed line with open arrow to interface
- Difference from abstract class:
  - Both have no direct instances
  - Abstract class can have attributes; interface cannot

Example: Realization Relationships

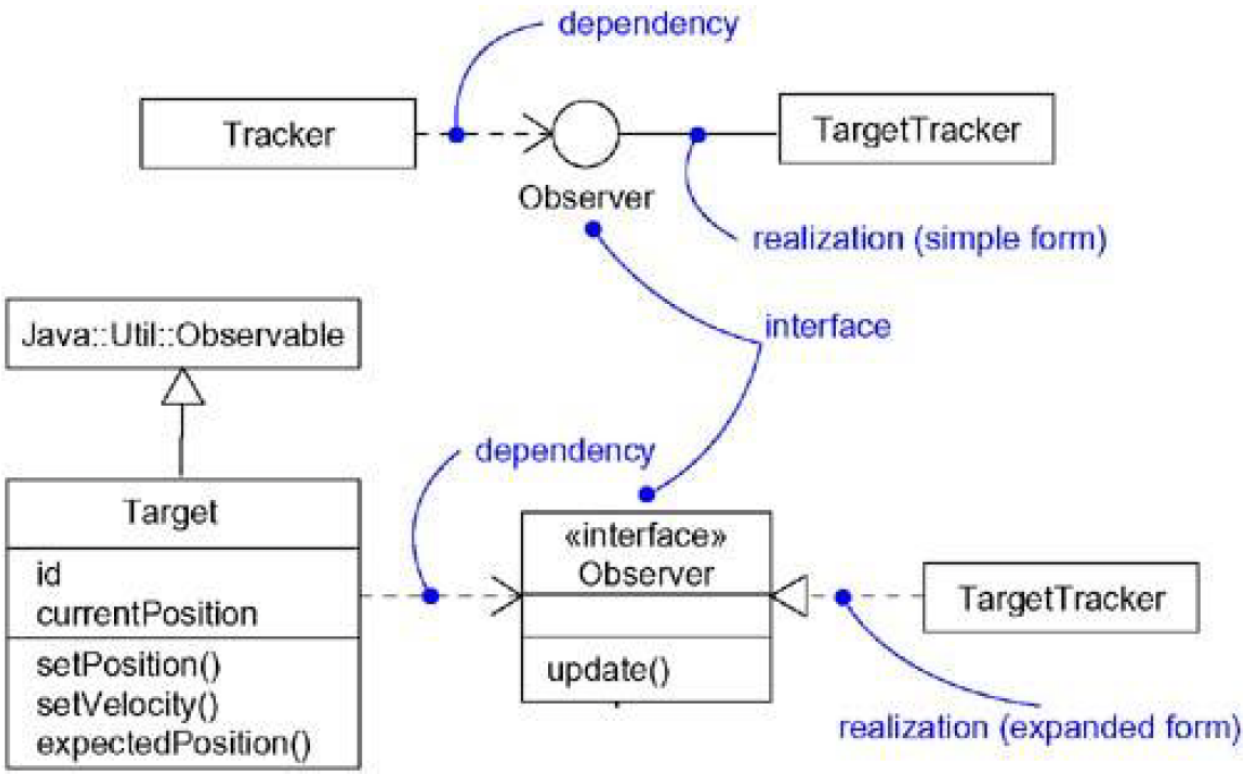


Figure: UML example showing dependency and realization (simple and expanded forms).

Understanding an Interface

- An interface shows a set of **operations** (service of a class/component)
- Operations have properties: visibility, scope, concurrency
- But operations alone may not explain full **semantics**
- To make an interface clear:
  - Add **preconditions, postconditions, invariants**
  - Attach a **state machine** → legal order of operations
  - Attach **collaborations** → expected behavior in interactions
- Optionally use **OCL** for precise formal specification
- Goal: Clients understand and use the interface *without reading the implementation*

Types and Roles

- A class may realize many interfaces → must support all contracts
- In a context, an object may show only one interface = its **role**
- **Role** = the "face" an object presents in a given situation
- Example: a Person may act as Employee, Manager, Customer, or Mother
- In UML, a role is shown by giving an association end a specific interface
- Role = context-dependent; Type = more general definition of operations (can include attributes)
- Interfaces cannot have attributes; Types may

Example: Role as Employee

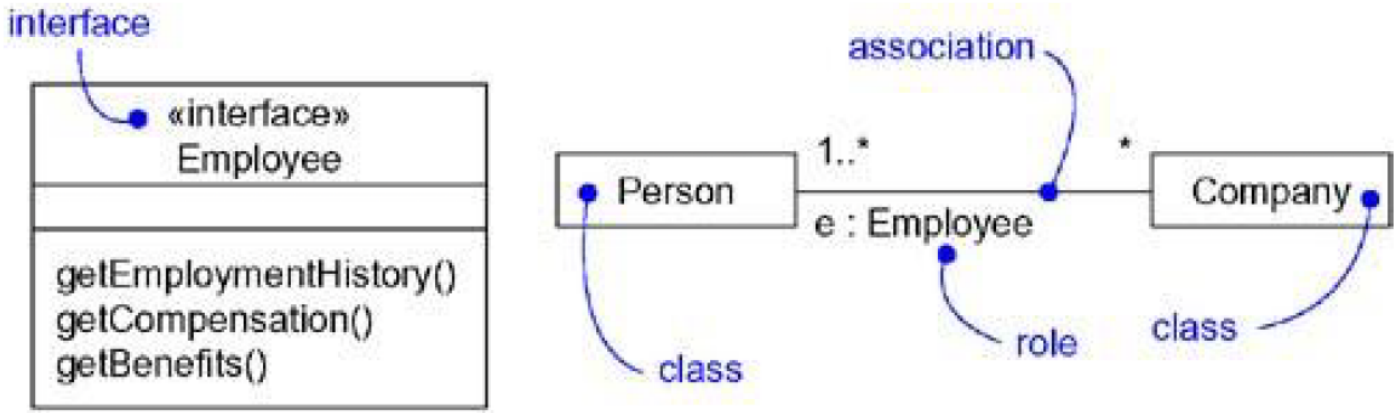


Figure: Person plays the role of **Employee** in association with Company. Only the operations of Employee (getEmploymentHistory, getCompensation, getBenefits) are visible to Company.

Packages

- **Purpose**: Organize classes, interfaces, components, and diagrams into groups
- Packages make large systems easier to **visualize, specify, construct, and document**
- Control **visibility**: some elements are public, others hidden
- Present different **architectural views** of the system
- Good packages = **cohesive** (change together) and **loosely coupled** (minimal dependencies)
- Real-world analogy: rooms → areas → buildings → complexes (chunking abstractions into larger groups)
- Packages exist in the **model**, not in the deployed system

Example: UML Package Notation

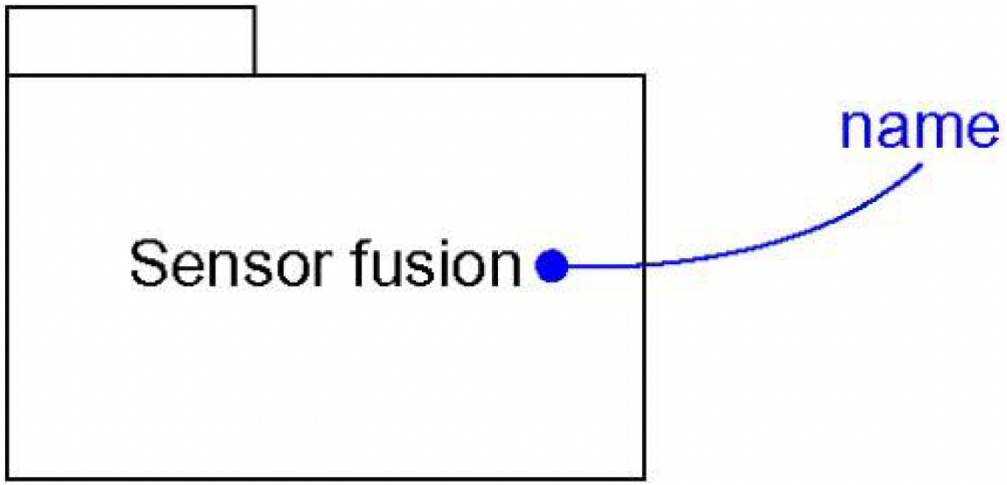


Figure: UML package shown as a tabbed folder (e.g., “Sensor fusion”).

Package Names

- Each package must have a **unique name** within its enclosing package
- **Simple name**: just the package name (e.g., Business rules)
- **Path name**: includes enclosing package (e.g., Sensors::Vision)
- Package names are short grouping nouns/phrases from the model vocabulary
- Names may include letters, numbers, punctuation (except "::")
- Packages can be shown with only a name, or with **tagged values/compartments** for details

Example: Package Names

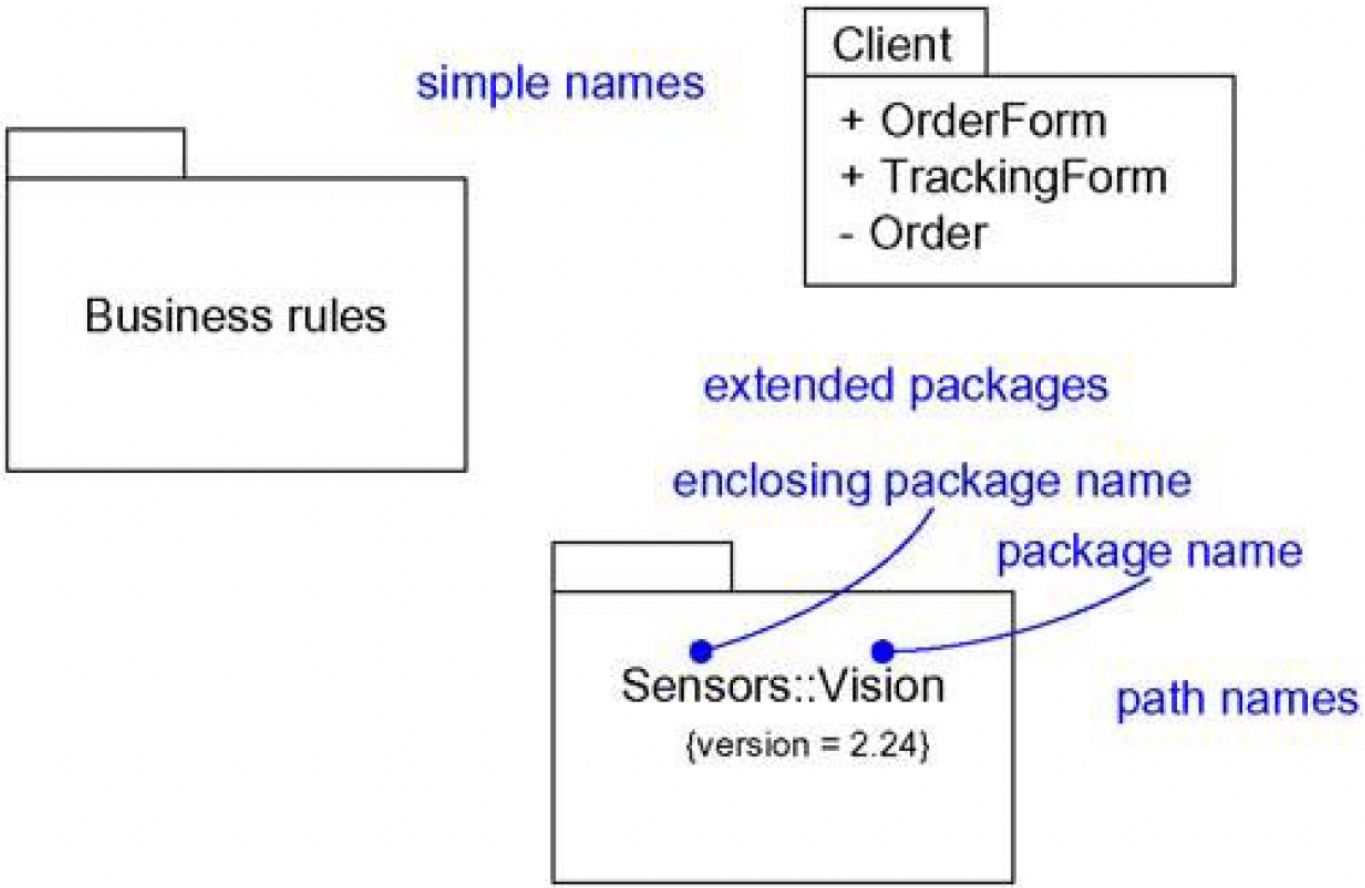


Figure: UML packages with simple names (Business rules) and path names (Sensors::Vision).

Owned Elements in Packages

- A package may own classes, interfaces, components, nodes, use cases, diagrams, even other packages
- **Ownership = composition**: if package is destroyed, owned elements are destroyed
- Each element is owned by exactly **one package**
- A package forms a **namespace** → elements of the same kind must be uniquely named
- Same name allowed if element kinds differ (e.g., Class Timer and Component Timer)
- Packages may be nested (e.g., Sensors::Vision::Camera) but deep nesting is discouraged (2–3 levels max)
- Packages help avoid flat, unmanageable models and support scaling with multiple teams

Example: Owned Elements

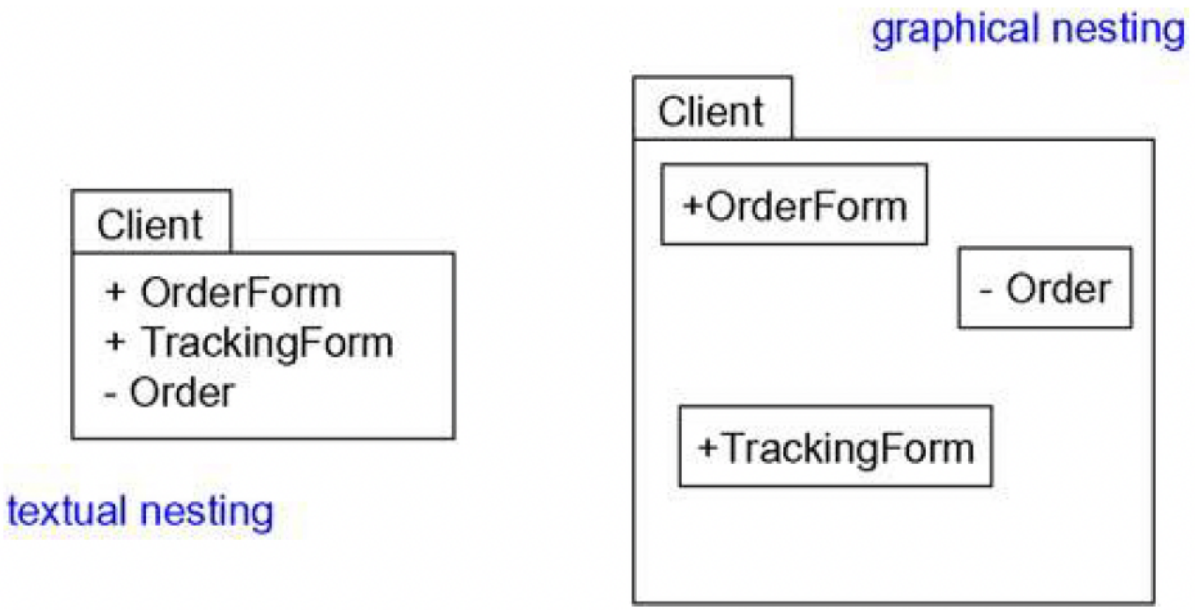


Figure: Package **Client** showing its owned elements (OrderForm, TrackingForm, Order) using textual and graphical nesting.

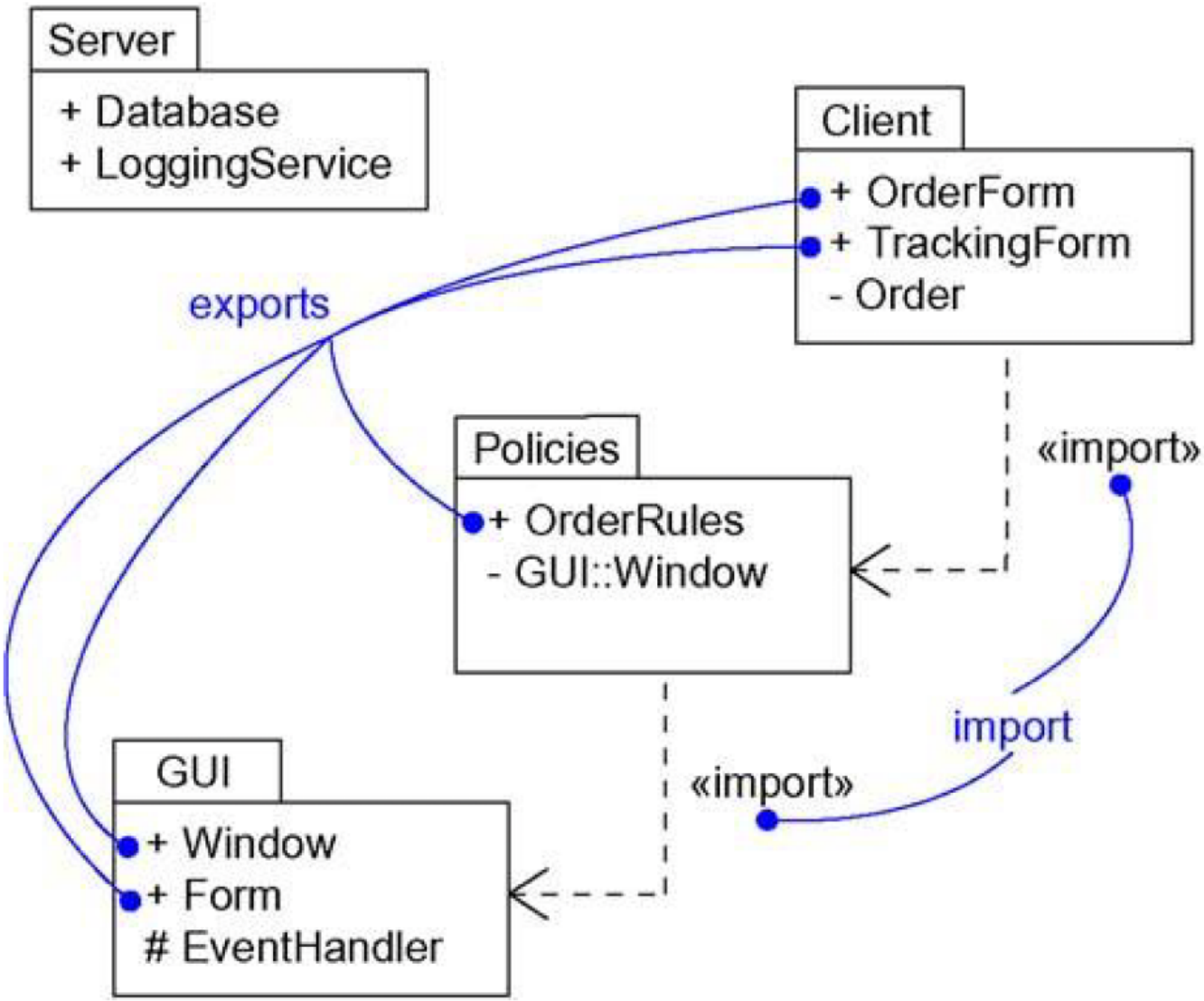
Visibility in Packages

- Elements in a package have visibility, like class members
- **Public (+)** → visible to any package that imports it (e.g., Client::OrderForm)
- **Protected (#)** → visible only to child packages (inheritance)
- **Private (-)** → visible only inside the package (e.g., Order in Client)
- Public parts of a package = the package's **interface**
- Friend packages (special case) → can see all elements regardless of visibility

Importing and Exporting

- Without packaging: all classes can see each other → chaos in large systems
- Packaging adds **boundaries** → classes in different packages cannot see each other directly
- **Exporting**: declare elements public in a package
- **Importing**: one package explicitly imports another → gains one-way access
- UML: import = dependency with stereotype `«import»`
- Two stereotypes:
  - **import**: adds target's contents to source namespace (no qualification needed; risk of name clashes)
  - **access**: requires name qualification (no namespace pollution)
- Importing controls access and reduces complexity in large models

Example: Import and Export



Exports and Imports in Packages

- **Exports** = public parts of a package (e.g., GUI exports Window, Form)
- **Protected** parts (e.g., EventHandler in GUI) are not exported
- Only packages that **import** another can see its exports
  - Policies imports GUI → sees GUI::Window and GUI::Form
  - Policies cannot see GUI::EventHandler (protected)
- Server does not import GUI → cannot access its contents
- Import/access are **not transitive**:
  - Client imports Policies (can see its exports)
  - Client does **not** automatically import GUI
- Nested packages inherit visibility of their containing packages

Generalization in Packages

- Packages can relate via:
  - **Import / Access** → bring in exported elements
  - **Generalization** → define families of packages
- Works like class inheritance:
  - Specialized package inherits public + protected elements
  - May override general elements
  - May add new elements
- Example:
  - GUI exports Window, Form, and protected EventHandler
  - WindowsGUI and MacGUI specialize GUI
  - WindowsGUI inherits Window, EventHandler; overrides Form; adds VBForm
- Substitutability principle applies: a specialized package can be used wherever a general package is expected

Example: Package Generalization

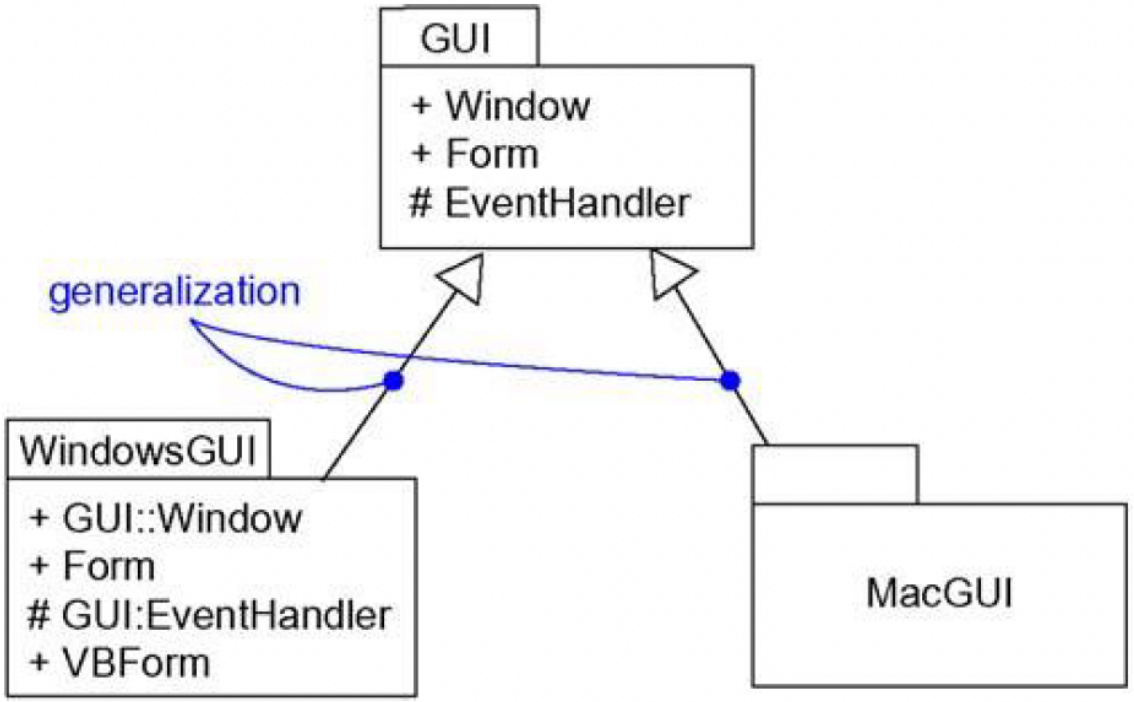


Figure: GUI package generalized into WindowsGUI and MacGUI, with inheritance of elements and extensions.

Standard Elements in Packages

- UML extensibility applies to packages: **tagged values** (extra properties) and **stereotypes** (new kinds)
- Five standard stereotypes for packages: *facade*, *framework*, *stub*, *subsystem*, *system*
- **Facade**: simplified view of a complex package (shows only selected parts)
- **Stub**: proxy for another package's contents (useful in distributed/team development)
- Subsystem/system stereotypes help model scale (independent part vs entire system)
- Import dependency is also a standard stereotype

UML Standard Package Stereotypes

Stereotype	**Meaning**
facade	Specifies a package that is only a view on some other package
framework	Specifies a package consisting mainly of patterns
stub	Specifies a package that serves as a proxy for the public contents of another package
subsystem	Specifies a package representing an independent part of the entire system being modeled
system	Specifies a package representing the entire system being modeled

Figure: UML standard stereotypes for packages.

CRC (Class, Responsibility, Collaboration)

- **CRC = Class, Responsibility, Collaboration.**
- A lightweight design tool in **object-oriented programming**.
- Helps capture and organize:
  - **Class** → the object or component.
  - **Responsibilities** → what it does.
  - **Collaborations** → how it interacts with others.
- Typically represented on simple **index cards**.
- Useful for brainstorming and early design discussions.

Analogy: Imagine building a house with Lego blocks — each block represents a part of the house. **CRC cards are like index cards for each block**, describing what it is, what it does, and how it connects.

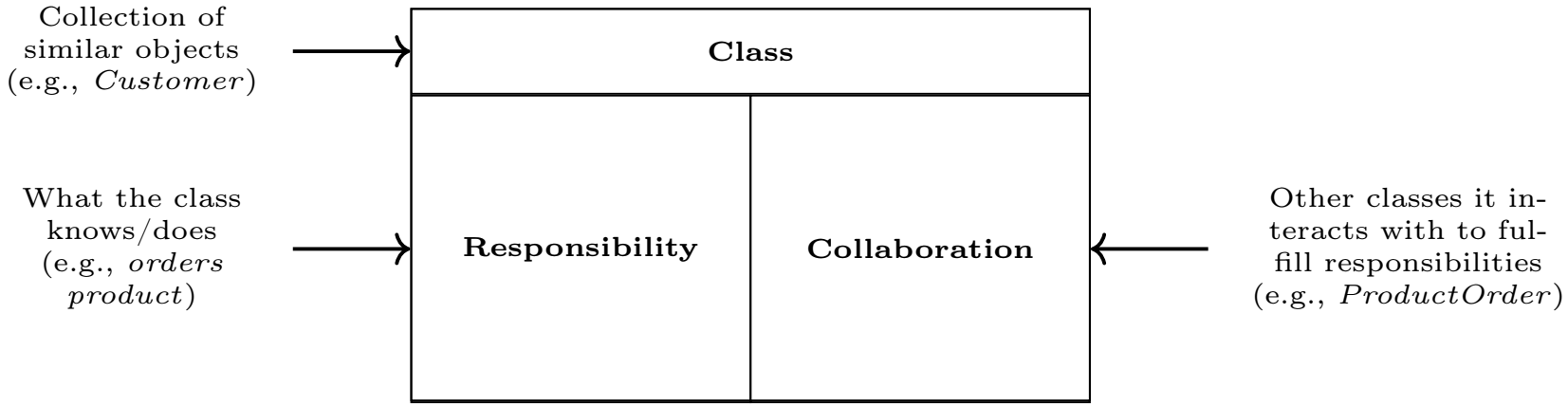
Purpose of CRC cards in software design

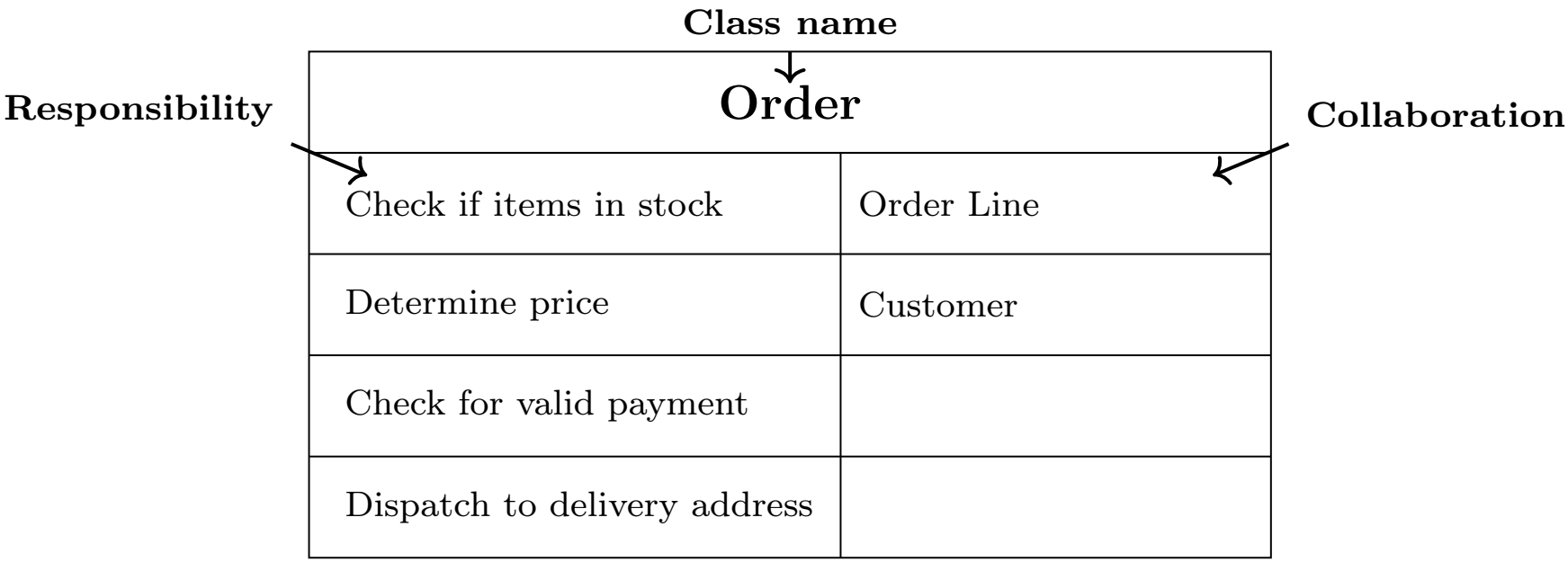
- **Clarify Class Responsibilities**
  - Define what each class is supposed to do.
  - Avoid confusion by clearly listing responsibilities.
- **Identify Collaborations**
  - Show which classes interact to fulfill responsibilities.
  - Ensure class interactions are well-defined and appropriate.
- **Encourage Communication & Teamwork**
  - Typically used in group settings (brainstorming, design meetings).
  - Promotes active participation and shared understanding.
- **Support Iterative Design**
  - Easy to update as requirements evolve.
  - Enables continuous refinement of design throughout development.

Components of CRC Cards

- **Class**
  - Represents a software entity/component in the system.
  - Encapsulates *data* (attributes/properties) and *behavior* (methods/functions).
  - On a CRC card, the **class name** is written at the top.
- **Responsibility**
  - Tasks/functions the class is responsible for performing (what it does).
  - Listed below the class name as short phrases or bullet points.
- **Collaboration**
  - Interactions/relationships with other classes to fulfill responsibilities.
  - Shows which classes this one communicates or cooperates with; listed alongside responsibilities.

CRC Card Structure




Class name
Order
Responsibility
Collaboration
Check if items in stock
Order Line
Determine price
Customer
Check for valid payment
Dispatch to delivery address

Benefits of Using CRC Cards

- **Visualization** — Tangible, visual view of classes, behavior, and relationships; makes structure easy to grasp.
- **Collaboration** — Great for group design/brainstorming; invites participation and shared understanding.
- **Clarity & Understanding** — Concise responsibilities/collaborations per class clarify purpose and roles.
- **Iterative Design** — Easy to update as requirements change; supports continuous refinement of architecture.
- **Rapid Prototyping** — Quickly create/modify/rearrange cards to explore alternatives in hierarchy and duties.

How to Create CRC Cards?

1. **Identify Classes** — Find the key entities/components in the system (e.g., Student, Course, Teacher).
2. **List Responsibilities** — For each class, note down what it does or is expected to accomplish; keep them concise.
3. **Define Collaborations** — Specify how a class interacts with others (dependencies, messages, communications).
4. **Create Cards** — Make a card for each class (physical or digital). Write class name on top, then responsibilities and collaborations.
5. **Organize Cards** — Arrange visibly (table/board) to show architecture, group related classes, or hierarchy.

Architectural Modelling

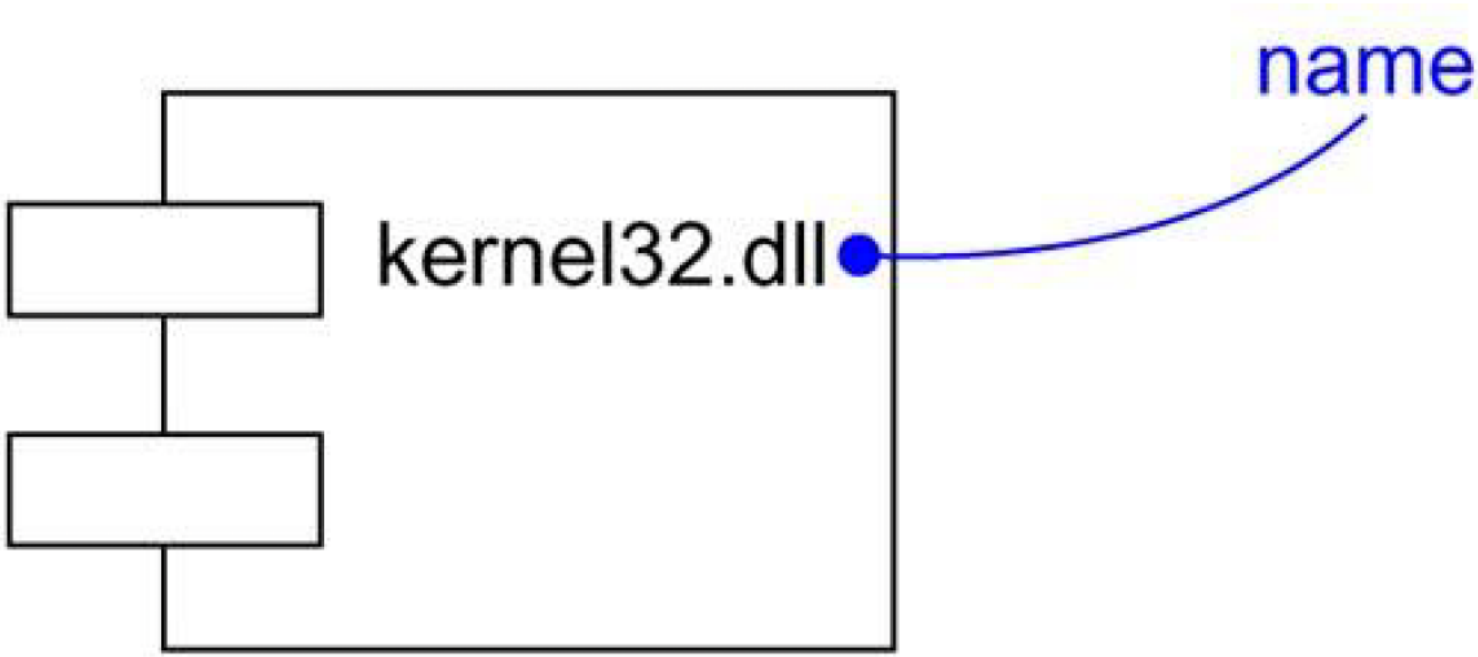
Component

- **Definition:** A component is a *physical, replaceable part* of a system.
- **Role:** Conforms to and realizes a set of interfaces.
- **Purpose:** Used to model physical things that may reside on a node:
  - Executables, libraries
  - Tables, files, documents
- **Nature:** Represents the *physical packaging* of logical elements (classes, interfaces, collaborations).
- **Quality:** Good components define crisp abstractions with well-defined interfaces $\rightarrow$ enable easy replacement/upgrade.

Component

- **Logical ↔ Physical Bridge:**
  - Interfaces specified in logical model are realized by components.
  - Interfaces bridge logical and physical views.
- **Examples in Software:**
  - Object libraries, executables
  - COM+ components, Enterprise Java Beans
  - Tables, files, documents
- **In UML:**
  - Components are rendered as a rectangle with small tabs.
  - Canonical notation independent of OS or language.
  - **Stereotypes** extend notation for specific kinds of components.
- **Reality Check:** Physical things live in the *world of bits*, residing on nodes, executed directly or indirectly in a system.

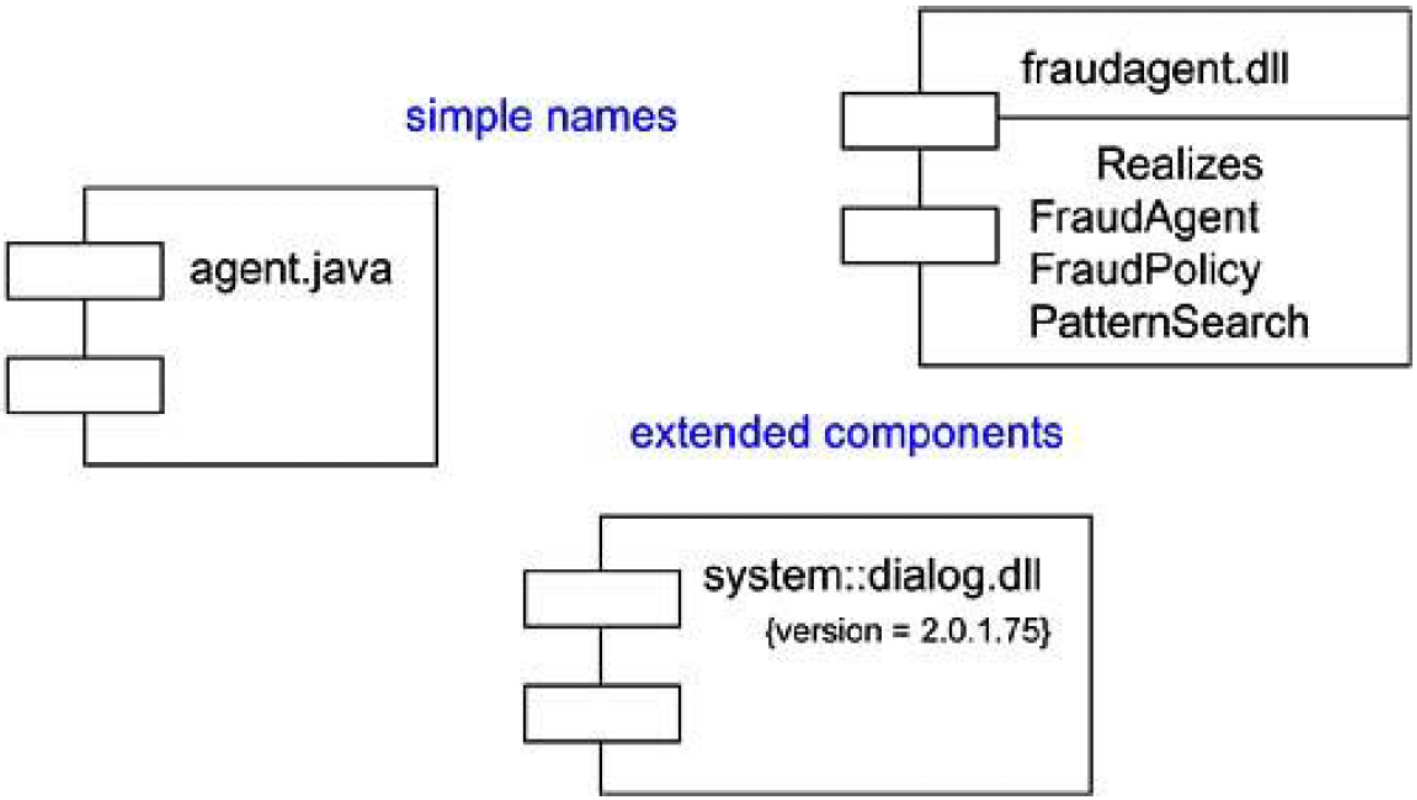
Component Example



Component Names

- A component name must be **unique** within its enclosing package.
- Every component must have a **name** that distinguishes it from others.
- A name is a **textual string**:
  - **Simple name:** name alone
  - **Path name:** name prefixed by package name
- Components are typically drawn showing only their name.
- Just as with classes, components may include:
  - **Tagged values**
  - **Additional compartments** (to expose details)
- Component names can include letters, numbers, and punctuation (except reserved separators like colon).
- In practice, names are short nouns or noun phrases, often with **extensions** (e.g., `.java`, `.dll`).

Component Names

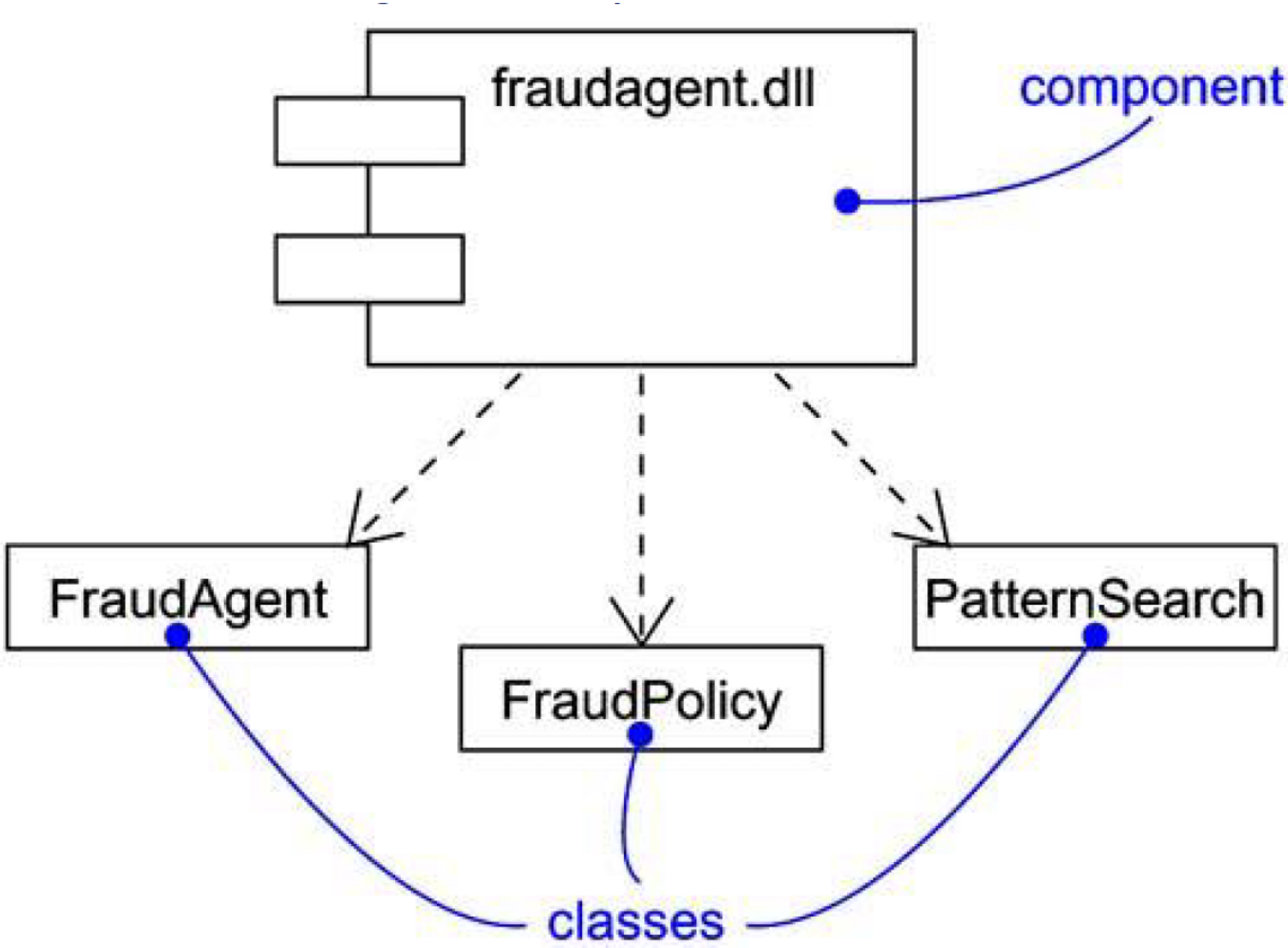


Simple and Extended Components

Components and Classes

- **Similarities with Classes:**
  - Both have names and may realize interfaces.
  - Both participate in dependency, generalization, and association relationships.
  - Both may be nested, have instances, and participate in interactions.
- **Key Differences:**
  - **Classes** → logical abstractions. **Components** → physical things that live on nodes.
  - Components = physical packaging of logical elements (classes, collaborations).
  - Classes have attributes and operations directly. Components expose operations only via interfaces.
- **Modeling Decision:**
  - If it lives on a node → model as a **component**.
  - Otherwise → model as a **class**.
- **Relationship:** Components are physical implementations of logical elements. This can be shown with **dependency relationships**.

Components and Classes

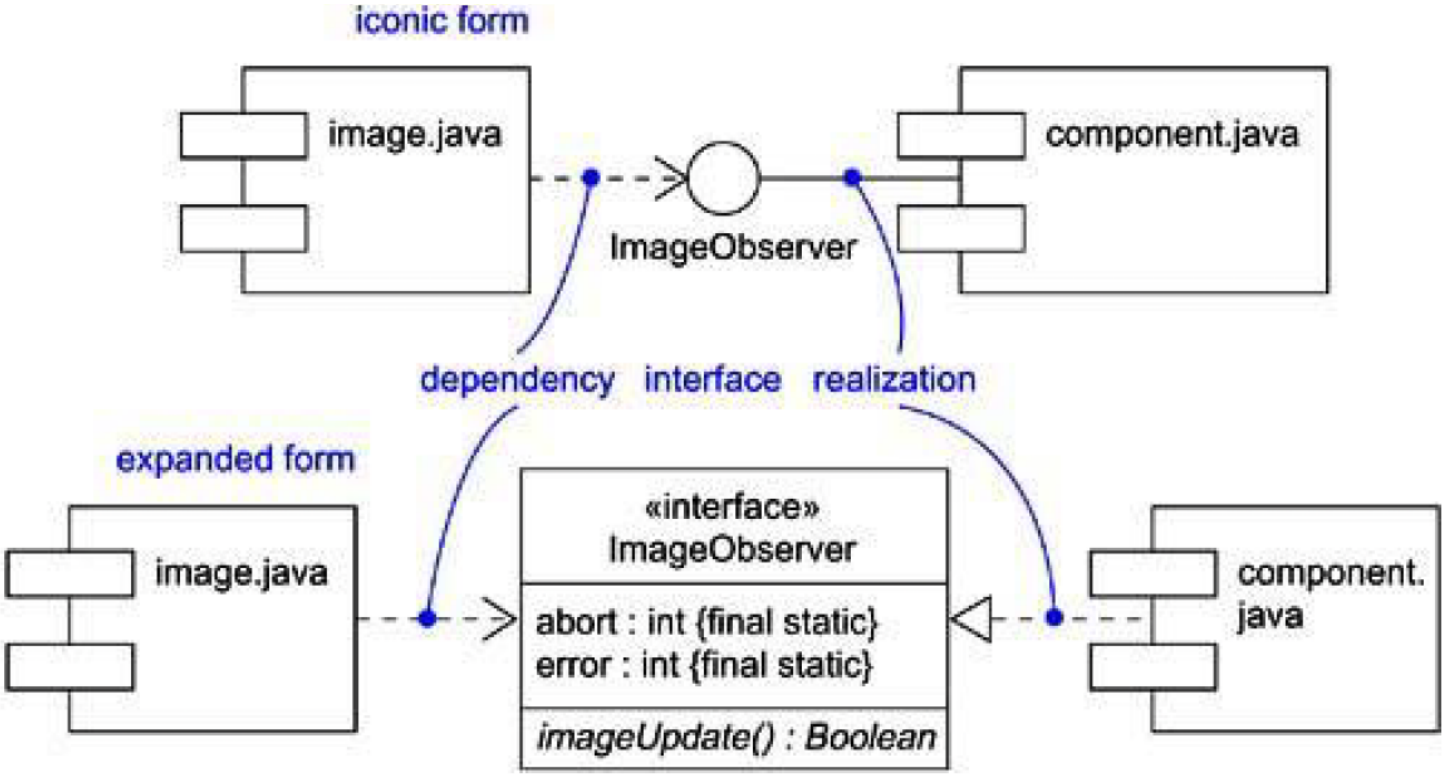


Components and Classes

Components and Interfaces

- **Interface:** A collection of operations specifying a service of a class or component.
- **Importance:** Interfaces are the glue that binds components together (used in COM+, CORBA, EJB, etc.).
- **Modeling:**
  - Export interfaces → provided by a component as services.
  - Import interfaces → required by a component to function.
  - A component may both import and export multiple interfaces.
- **Benefits:**
  - Components can be replaced as long as they realize the same interface.
  - Breaks direct dependency between components.
  - Enables location-independent, distributed system deployment.
- **Notation:**
  - **Elided form:** interface shown as a small circle (iconic).
  - **Expanded form:** interface box revealing operations.
  - Realization → component connected to interface.
  - Dependency → component uses interface services.
- **Note:** Interfaces span both logical and physical boundaries.

Components and Interfaces



Components and Interfaces

Overview of C++

- **C++** was developed by **Bjarne Stroustrup** at Bell Labs (early 1980s) as an extension of the C language.
- Retains all core **C (procedural)** features: functions, control structures, pointers, direct memory access.
- Adds **object-oriented** concepts: classes, objects, inheritance, polymorphism, encapsulation.
- This mix enables:
  - **Procedural style** for low-level, efficient code.
  - **OOP style** for modularity, reusability, scalability.
- Often called **multi-paradigm**: supports procedural, object-oriented, and **generic programming** (templates).

Difference Between C and C++

Aspect	**C (Procedural)**	**C++ (OOP Extension)**
Paradigm	Procedural (step-by-step)	Multi-paradigm: Procedural + OOP + Generic
Programming Style	Functions, structured code	Classes, Objects + Functions
Data Security	No encapsulation, global access	Supports encapsulation (private/public)
Code Reusability	Functions only	Functions + Inheritance, Polymorphism
Complex Apps	Less suited for very large projects	Better for large projects (OOP)
Libraries	Limited (stdio.h)	Rich (iostream, STL)
Approach	Top-down (functions first)	Bottom-up (objects first)
Polymorphism	Not supported	Supported (compile + runtime)
Example Use	OS, drivers, kernels	Software design, games, simulations