Module 3 – Advanced Features: Part I - Structural Diagrams
3 basic building blocks of UML - Diagrams

Graphical representation of a set of elements. Represented by a connected graph: Vertices are things; Arcs are relationships/behaviors.

5 most common views built from UML 1.x: 9 diagram types.

**UML 1.x: 9 diagram types.**

**Structural Diagrams**
*Represent the static aspects of a system.*
- Class;
- Object
- Component
- Deployment

**Behavioral Diagrams**
*Represent the dynamic aspects.*
- Use case
- Sequence;
  Collaboration
- Statechart
- Activity

**UML 2.0: 12 diagram types**

**Structural Diagrams**
- Class;
- Object
- Component
- Deployment
- Composite Structure
- Package

**Behavioral Diagrams**
- Use case
- Statechart
- Activity

**Interaction Diagrams**
- Sequence;
  Communication
- Interaction Overview
- Timing

©Lawrence Chung
Class Diagrams

Structural Diagrams

- Class:
  - Object
  - Component
  - Deployment
  - Composite Structure
  - Package
Class Diagram

The basis for all object modeling
All things lead to this

• Most common diagram.
• Shows a set of classes, interfaces, and collaborations and their relationships (dependency, generalization, association and realization); notes too.
• Represents the static view of a system (With active classes, static process view)

Three modeling perspectives for Class Diagram

- **Conceptual**: the diagram reflects the domain
- **Specification**: focus on interfaces of the software (Java supports interfaces)
- **Implementation**: class (logical database schema) definition to be implemented in code and database.

*Most users of OO methods take an implementation perspective, which is a shame because the other perspectives are often more useful.* — Martin Fowler
Classes

Names

Account
balance: Real = 0

Attributes

simple name - start w. upper case
default value

Operations

may cause object to change state

<<constructor>>
+addAccount()

<<process>>
+setBalance(a: Account)
+getBalance(a: Account): Amount
...

<<query>>
isValid(loginID: String): Boolean

signature

ellipses for additional attributes or operations

stereotypes to categorize

Bank
Customer

Java::awt::Polygon

only the name compartment, ok

path name = package name ::package name::name
Responsibilities

- anything that a class knows or does (Contract or obligation)

- An optional 4th item carried out by attributes and operations.
- Free-form text; one phrase per responsibility.
- Technique - CRC cards (Class-Responsibility-Collaborator); Kent Beck and Ward Cunningham’89

- A collaborator is also a class which the (current) class interacts with to fulfill a responsibility

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>-- handles deposits</td>
</tr>
<tr>
<td>-- reports fraud to managers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Customer</th>
<th>Account</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opens account</td>
<td>Knows interest rate</td>
<td>Knows balance</td>
</tr>
<tr>
<td>Knows name</td>
<td>Knows balance</td>
<td>Handles deposits</td>
</tr>
<tr>
<td>Knows address</td>
<td>Knows balance</td>
<td>Reports fraud to manager</td>
</tr>
</tbody>
</table>

©Lawrence Chung
Scope & Visibility

- **Instance** Scope — each instance of the classifier holds its own value.
- **Class** Scope — one value is held for all instances of the classifier (underlined).

<table>
<thead>
<tr>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>header : FrameHeader</td>
</tr>
<tr>
<td>uniqueID : Long</td>
</tr>
<tr>
<td>+ addMessage( m : Message ) : Status</td>
</tr>
<tr>
<td># setCheckSum()</td>
</tr>
<tr>
<td>- encrypt()</td>
</tr>
<tr>
<td>- getClassName()</td>
</tr>
</tbody>
</table>

- Public - access allowed for any outside classifier (+).
- Protected - access allowed for any descendant of the classifier (#).
- Private - access restricted to the classifier itself (-).
- (using adornments in JBuilder)
Multiplicity

Using Design Pattern

Singleton
- instance
+ getInstance():Singleton

NetworkController
consolePort [ 2..* ] : Port

ControlRod

Using Design Pattern

public class Singleton {
    private static Singleton instance = null;

    private Singleton() {}  
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}

©Lawrence Chung
Relationships

Window

open()
close()

Event
dependency

generalization

ConsoleWindow DialogBox Control

association

Controller EmbeddedAgent

generalization (multiple inheritance)

<<interface>>
URLStreamHandler

openConnection() parseURL() setURL() toExternalForm()

SetTopController
authorizationLevel

startUp() shutDown()

<<interface>>
PowerManager

<<friend>>
ChannelIterator

realization

association navigation

stereotyped dependency
Dependency

• A change in one thing may affect another.
  • The most common dependency between two classes is one where one class <<use>>s another as a *parameter to an operation*.

```
AudioClip

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>record(m: Microphone)</td>
</tr>
<tr>
<td>start()</td>
</tr>
<tr>
<td>stop()</td>
</tr>
</tbody>
</table>

Microphone
```

```
CourseSchedule

| addCourse(c: Course) |
| removeCourse(c: Course) |

Course
```

Usually initial class diagrams will not have a significant number of dependencies in the beginning of analysis but will as more details are identified.
Dependency – Among Classes

• Eight Stereotypes of Dependency Among Classes
  – bind: the source instantiates the target template using the given actual parameters
  – derive: the source may be computed from the target
  – friend: the source is given special visibility into the target
  – instanceof: the source object is an instance of the target classifier
  – instantiate: the source creates instances of the target
  – powertype: the target is a powertype of the source; a powertype is a classifier whose objects are all the children of a given parent
  – refine: the source is at a finer degree of abstraction than the target
  – use: the semantics of the source element depends on the semantics of the public part of the target
Dependency – Among Use Cases

- Two Stereotypes of Dependency Among Use Cases:
  - extend: the target use case extends the behavior of the source
  - include: the source use case explicitly incorporates the behavior of another use case at a location specified by the source

![Diagram](image-url)
Generalization

• Four Standard Constraints
  – *complete*: all children in the generalization have been specified; no more children are permitted
  – *incomplete*: not all children have been specified; additional children are permitted
  – *disjoint*: objects of the parent have no more than one of the children as a type
  – *overlapping*: objects of the parent may have more than one of the children as a type

• One Stereotype
  – *implementation*: the child inherits the implementation of the parent but does not make public nor support its interfaces
Generalization – Among Actors

- **Place Order**
  - **Extension points**
  - **Additional requests:** after creation of the order

- **Grant Credit**

**Salesperson**

- Can do only "Place Order"

**Sales Manager**

- Can do both "Place Order" and "Grant Credit"
Associations

- Represent conceptual relationships between classes
  (cf. dependency with no communication/message passing)

**What’s the difference between attributes and associations?**
Associations – A Question

• How would you model the following situation?

“You have two files, say homework1 and myPet, where homework1 is read-accessible only by you, but myPet is write-accessible by anybody.”

You could create two classes, File and User. Homework1 and MyPet are files, and you are a user.

**Approach 1:** Now, would you associate the file access right with File?

**Approach 2:** Or, would you associate the file access right with User?
Associations – Links

- A link is a semantic connection among objects.
- A link is an instance of an association.

Association generalization is not automatic, but should be explicit in UML 1.x (not in UML 2.0)
Associations – Link Attributes

• Link Attributes

The most compelling reason for having link attributes is for-many-to-many relationships

• Association Class

• With a refactoring (“reification”)

©Lawrence Chung
Modeling Structural Relationships

- Considering a bunch of classes and their association relationships

The model above is from Rational Rose. How did the composite symbol (◆) get loaded versus the aggregation? Use the Role Detail and select aggregation and then the “by value” radio button.

©Lawrence Chung
Modeling Structural Relationships

**Aggregation**
- structural association representing “whole/part” relationship.
- “has-a” relationship.

**Composite**
Composite is a stronger form of aggregation.
Composite parts live and die with the whole.
Composite parts may belong to only one composite.

Can aggregations of objects be cyclic?
Association – Qualification

Qualifier, cannot access person without knowing the account #

Bank

account #

* 0..1

Person

Chessboard

rank:Rank

file:File

1

1

Square

WorkDesk

jobId : int

* 0..1 returnedItem

©Lawrence Chung
Association – InterfaceSpecifier

**Realization**

- A semantic relationship between classifiers in which one classifier specifies a contract that another guarantees to carry out.
- In the context of *interfaces* and *collaborations*.
- An interface can be realized by many classes/components.
- A class/component may realize many interfaces.
Modeling a Logical Database

- Class diagrams to provide more semantics
- From a general class diagram, first identify classes whose state must be persistent (e.g. after you turn off the computer, the data survives, although the program doesn’t).
- Create a class diagram using standard tagged value, (e.g. \{persistent\}).
- Include attributes and associations.
- Use tools, if available, to transform logical design (e.g., tables and attributes) into physical design (e.g., layout of data on disk and indexing mechanisms for fast access to the data).

```
School
  { persistent}
  name : Name
  address : String
  phone : Number
  addStudent() removeStudent() getStudent() getAllStudents() addDepartment() removeDepartment() getDepartment() getAllDepartments()

Student
  { persistent}
  name : Name
  studentId : Number

Department
  { persistent}
  name : Name
  addInstructor() removeInstructor() getInstructor() getAllInstructors()

Course
  { persistent}
  name : Name
  courseId : Number

Instructor
  { persistent}
  name : Name

Is mapping 1-1?
```
Forward/ Reverse Engineering

- translate a collaboration into a logical database schema/operations
- transform a model into code through a mapping to an implementation language.

Steps
- Selectively use UML to match language semantics (e.g. mapping multiple inheritance in a collaboration diagram into a programming language with only single inheritance mechanism).
- Use tagged values to identify language.

```java
public abstract class EventHandler {
    private EventHandler successor;
    private Integer currentEventId;
    private String source;

    EventHandler() {}
    public void handleRequest() {}
}
```

- translate a logical database schema/operations into a collaboration
- transform code into a model through mapping from a specific implementation language.
Object Diagrams

Structural Diagrams

- Class;

**Object**

- Component
- Deployment
- Composite Structure
- Package
Instances & Object Diagrams

- “instance” and “object” are largely synonymous; used interchangeably.

- **difference:**
  - instances of a **class** are called **objects or instances**; but
  - instances of other abstractions (components, nodes, use cases, and associations) are not called objects but only **instances**.

*What is an instance of an association called?*

Object Diagrams

- very useful in debugging process.
  - walk through a scenario (e.g., according to use case flows).
  - Identify the set of **objects** that collaborate in that scenario (e.g., from use case flows).
  - Expose these object’s states, attribute values and links among these objects.

*Are use cases objects?*

©Lawrence Chung
Instances & Objects - Visual Representation

- **named instance**: myCustomer
- **anonymous instance**: : Multimedia :: AudioStream
- **multiobject**: : keyCode
- **orphan instance (type unknown)**: t : Transaction
- **active object** (with a thicker border; owns a thread or process and can initiate control activity): agent : c : Phone [WaitingForAnswer]
- **instance with current state**: myCustomer
  - id : SSN = “432-89-1738”
  - active = True

©Lawrence Chung
Instances & Objects - Modeling Concrete Instances

- Expose the stereotypes, tagged values, and attributes.
- Show these instances and their relationships in an object diagram.

Instances & Objects - Modeling Prototypical Instances

- Show these instances and their relationships in an interaction diagram or an activity diagram.
Instances & Objects – More Examples

call ::= label [guard] ["*" ] [return-val-list "="] msg-name "(" arg-list ")"

client

servers

:Server

aServer:Server

1: aServer := find(criteria)

2: process(request)

d: Directory

contents: File

addFile(f:File)

1: addElement(f)

d: Directory

contents: File

secureAll()

1*: changeMode(readOnly)

d: Directory

f: File

*:

1: sort()

d: Directory

contents: File

list()

:Server

aServer:Server

name = “Erin” employeeID = 4362 title = “VP of Sales”

:ContactInformation

address = “1472 Miller St.”

manager

erin : Person

uss : Department

name = “US Sales”

rd : Department

name = “R&D”

s : Department

name = “Sales”

c : Company

©Lawrence Chung
Component Diagrams

Structural Diagrams
- Class;
  Object
- **Component**
- Deployment
- **Composite Structure**
- Package
Component Diagram

- Shows a set of components and their relationships.
- Represents the static implementation view of a system.
- Components map to one or more classes, interfaces, or collaborations.

Mapping of Components into Classes

- LoanOfficer.dll
- CreditSearch
- LoanPolicy

Components and their Relationships

- Registrar.exe
- Student.dll
- Course.dll
Big demand, hmm...

- Short history behind architecture
- Architecture still an emerging discipline
- Challenges, a bumpy road ahead

- UML and architecture evolving in parallel
- Component diagram in need of better formalization and experimentation
Component Diagram – another example

Component Diagram – another example
Component Diagram – another example

Explicit description of **interfaces**:

- provided services to other components
- requested services from other components

- An interface is a collection of 1..* methods, and 0..* attributes
- Interfaces can consist of synchronous and / or asynchronous operations
- A **port** (square) is an interaction point between the component and its environment
- Can be named; Can support uni-directional (either provide or require) or bi-directional (both provide and require) communication; Can support multiple interfaces.
- possibly concurrent interactions
- fully isolate an object’s internals from its environment
Component Diagram: UML 1.x and UML 2.0
(http://www.agilemodeling.com/artifacts/componentDiagram.htm)
Component Diagram: UML 1.x and UML 2.0

(https://www.agilemodeling.com/artifacts/componentDiagram.htm)

So, how many different conventions for components in UML2.0?
Building a Component

- simplified the ports to either provide or require a single interface
- relationships between ports and internal classes in three different ways:
  1) as *stereotyped delegates* (flow), as *delegates*, and as *realizes* (logical->physical) relationships
- Cohesive reuse and change of classes; acyclic component dependency
Component Diagram – Connector & Another Example

- a connector: just a link between two or more connectable elements (e.g., ports or interfaces)
- 2 kinds of connectors: assembly and delegation. For “wiring”
  - An assembly connector: a binding between a provided interface and a required interface (or ports) that indicates that one component provides the services required by another; simple line/ball-and-socket/lollipop-socket notation
  - A delegation connector binds a component’s external behavior (as specified at a port) to an internal realization of that behavior by one of its parts (provide-provide, request-request).

**So, what levels of abstractions for connections?**

**Left delegation:** direction of arrowhead indicates “provides”

**Right delegation:** direction of arrowhead indicates “requests”
Structured Class

- A structured class(ifier) is defined, in whole or in part, in terms of a number of parts - contained instances owned or referenced by the structured class(ifier).
- With a similar meaning to a composition relation.
- A structured classifier’s parts are created within the containing classifier (either when the structured classifier is created or later) and are destroyed when the containing classifier is destroyed.
- Like classes and components, combine the descriptive capabilities of structured classifiers with ports and interfaces.

Any difference?

Components extend classes with additional features such as
- the ability to own more types of elements than classes can; e.g., packages, constraints, use cases, and artifacts
- deployment specifications that define the execution parameters of a component deployed to a node.
Classifiers

- Classifier—mechanism that describes structural (e.g. class attributes) and behavioral (e.g. class operations) features. In general, those *modeling elements* that can have *instances* are called classifiers.
- *cf. Packages and generalization relationships do not have instances.*

**Classifiers**

- Classifier—mechanism that describes structural (e.g. class attributes) and behavioral (e.g. class operations) features. In general, those *modeling elements* that can have *instances* are called classifiers.
- *cf. Packages and generalization relationships do not have instances.*
Structured Class – Another Example

Composite class (incomplete)

- with parts, ports and connectors

```
ATM

User-Reader
  :CardReader

User-Screen
  :Screen

User-Keyboard
  :Keyboard
  :CashDispenser

User-Cash
```

part

port

connector

what kind?
Deployment Diagrams

Structural Diagrams
- Class;
  - Object
- Component
- Deployment
  - Composite Structure
  - Package
Deployment Diagram

- Shows a set of *processing* nodes and their relationships.
- Represents the static deployment view of an *architecture*.
- Nodes typically enclose one or more *components*.
Structural Diagrams - Deployment Diagram
(http://www.agilemodeling.com/artifacts/deploymentDiagram.htm)

- **Student administration application**
  - Physical nodes - stereotype *device*
  - *WebServer* - physical device or software artifact
  - *RMI/message bus*: connection type
  - Nodes can contain other nodes or software artifacts recursively
  - Deployment specs: configuration files: name and properties

- **WebServer**
  - **Student Administration**
  - **JSPs**

- **ApplicationServer**
  - OS = Solaris

- **EJBContainer**
  - **Student**
  - **Seminar**
  - **Schedule**
  - Persistence
    - **infrastructure**
    - vendor = Ambysoft

- **Mainframe**
  - OS = MVS
  - **Course Management**
  - **Legacy System**

- **DBServer**
  - OS = Linux

- **Course Management Facade**
- **Web services**
Is this better?
- More concrete
- Implementation-oriented
Composite Structure Diagrams

Structural Diagrams
- Class;
  - Object
- Component
- Deployment
- Composite Structure
- Package
Composite Structure Diagrams

(http://www.agilemodeling.com/artifacts/compositeStructureDiagram.htm)

- Depicts the internal structure of a classifier (such as a class, component, or collaboration), including the interaction points of the classifier to other parts of the system.

structured class, structured component, structured use case, structured node, structured interface, …
Variations [Rumbaugh – UML 2.0 Reference: p234]
Context Model in UML2.0 - II

- Including multiplicities on parts

Multiplicity

BankContext

:Bank

:ATM [1..100]

User-Reader

User-Screen

User-Keyboard

User-Cash

[1..10,000]
Structural Diagrams

- Class;
  Object
- Component
- Deployment
- Composite Structure

- Package
# Packages

<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>A grouping of model elements.</td>
<td>![Package Icon] Name</td>
</tr>
<tr>
<td>Import</td>
<td>A dependency indicating that the public contents of the target package are added to the namespace of the source package.</td>
<td>![Import Icon]</td>
</tr>
<tr>
<td>Access</td>
<td>A dependency indicating that the public contents of the target package are available in the namespace of the source package.</td>
<td>![Access Icon]</td>
</tr>
</tbody>
</table>

### Diagram

- **Sales**
  - **Customer**
  - **Order**

- **Warehouse**
  - **Location**
  - **Item**
    - **Stock Item**
    - **Order Item**
Packages

- Package — general-purpose mechanism for organizing elements into groups.
- Nested Elements: Composite relationship (When the whole dies, its parts die as well, but not necessarily vice versa)
- (C++ namespace; specialization means “derived”)

Visibility

- Packages that are friends to another may see all the elements of that package, no matter what their visibility.
- If an element is visible within a package, it is visible within all packages nested inside the package.
Dependency – Among Packages

• Two Stereotypes of Dependency Among Packages:
  – **access**: the source package is granted the right to reference the elements of the target package (:: convention)
  – **import**: a kind of access; the **public** contents of the target package enter the flat namespace of the source as if they had been declared in the source
An imported element can be given a local alias and a local visibility.
Modeling Groups of Elements

- Look for “clumps” of elements that are semantically close to one another.
- Surround “clumps” with a package. \textit{High cohesion & high coupling}
- Identify public elements of each package.
- Identify import dependencies.

Use Case package Diagram

- Included and extending use cases belong in the same package as the parent/base use case
- Cohesive, and goal-oriented packaging
- Actors could be inside or outside each package
Classes related through inheritance, composition or communication often belong in the same package.

A frame depicts the contents of a package (or components, classes, operations, etc.)

Heading: rectangle with a cut-off bottom-right corner, [kind] name [parameter]
Common Mechanisms

• Adornments

  Notes & Compartments

• Extensibility Mechanisms
  – Stereotypes - Extension of the UML metaclasses.
  – Tagged Values - Extension of the properties of a UML element.
  – Constraints - Extension of the semantics of a UML element.
Adornments

• **Textual or graphical items added to an element’s basic notation.**

• **Notes** - Graphical symbol for rendering constraints or comments attached to an element or collection of elements; No Semantic Impact

  Rendered as a rectangle with a dog-eared corner.

  See smartCard.doc for details about this routine.

  See [http://www.rational.com](http://www.rational.com) for related info.

Additional Adornments

• Placed near the element as
  – Text
  – Graphic

• Special compartments for adornments in
  – Classes
  – Components
  – Nodes

©Lawrence Chung
Stereotypes

- Mechanisms for extending the UML vocabulary.
- Allows for new modeling building blocks or parts.

- Allow controlled extension of metamodel classes.

- Graphically rendered as
  - Name enclosed in guillemets (<< >> )
  - <<stereotype>>
  - New icon

- The new building block can have
  - its own special properties through a set of tagged values
  - its own semantics through constraints
Tagged Values

- a (name, value) pair describes a property of a model element.
- Properties allow the extension of "metamodel" element attributes.
- modifies the semantics of the element to which it relates.
- Rendered as a text string enclosed in braces {}.
- Placed below the name of another element.
Constraints

- Extension of the **semantics** of a UML element.
- Allows new or modified rules
- Rendered in braces `{}`.
  - Informally as free-form text, or
  - Formally in UML’s Object Constraint Language (OCL):
    
    E.g., `{self.wife.gender = female and self.husband.gender = male}`

---

**Diagram:**

- Portfolio
- BankAccount
  - `{secure}`
- Corporation
- BankAccount
- Person
  - `id : {SSN, passport}`
- Department
  - `member 1..* 1 manager`
- Person

**Table:**

<table>
<thead>
<tr>
<th>Person</th>
<th>employers</th>
<th>employers</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>age: Integer</td>
<td>0..*</td>
<td>0..*</td>
<td></td>
</tr>
</tbody>
</table>

**OCL example:**

```
self.employees.forAll(Person p | p.age >= 18 and p.age <= 65)
```
Appendix
Some Additional Material
## Classes: Notation and Semantics

<table>
<thead>
<tr>
<th>Class - Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute-name-1 : data-type-1 = default-value-1</td>
</tr>
<tr>
<td>attribute-name-2 : data-type-2 = default-value-2</td>
</tr>
<tr>
<td>operation-name-1 ( argument-list-1) : result-type-1</td>
</tr>
<tr>
<td>operation-name-2 ( argument-list-2) : result-type-2</td>
</tr>
</tbody>
</table>

### responsibilities

To model the `<<semantics>>` (meaning) of a class:
- Specify the body of each method (pre-/post-conditions and invariants)
- Specify the state machine for the class
- Specify the collaboration for the class
- Specify the responsibilities (contract)
Attributes

• Syntax
  \[ \text{[ visibility ] name [ multiplicity ] [ : type ] [ = initial-value ] [ \{property-string\} ] } \]

• Visibility
  + public; - private; # protected; \{default = +\}

• type
  – There are several defined in Rational Rose.
  – You can define your own.

  Or you can define your own: e.g. \{leaf\}

• property-string
  Built-in property-strings:
  – \textit{changeable}—no restrictions (default)
  – \textit{addOnly}—values may not be removed or altered, but may be added
  – \textit{frozen}—may not be changed after initialization

<table>
<thead>
<tr>
<th>origin</th>
<th>Name only</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ origin</td>
<td>Visibility and name</td>
</tr>
<tr>
<td>origin : Point</td>
<td>Name and type</td>
</tr>
<tr>
<td>head : *Item</td>
<td>Name and complex type</td>
</tr>
<tr>
<td>name [ 0..1 ] : String</td>
<td>Name, multiplicity, and type</td>
</tr>
<tr>
<td>origin : Point = { 0, 0 }</td>
<td>Name, type, and initial value</td>
</tr>
<tr>
<td>id : Integer { frozen }</td>
<td>Name and property</td>
</tr>
</tbody>
</table>
Operations

• Syntax
  \[[ \text{visibility} ] \text{name} [ (\text{parameter-list}) ] [ : \text{return-type} ] [ (\text{property-string}) ]\]

• Visibility
  \(+\) public; \(-\) private; \# protected; \{default = +\}

• parameter-list syntax
  \[[ \text{direction} ] \text{name} : \text{type} [ = \text{default-value} ]\]

• direction
  – \text{in}—input parameter; may not be modified
  – \text{out}—output parameter; may be modified
  – \text{inout}—input parameter; may be modified

• property-string
  – \text{leaf}
  – \text{isQuery}—state is not affected
  – \text{sequential}—not thread safe
  – \text{guarded}—thread safe (Java synchronized)
  – \text{concurrent}—typically atomic; safe for multiple flows of control
Template Classes; Primitive Types

- A template class is a parameterized element and defines a family of classes
- In order to use a template class, it has to be instantiated
- Instantiation involves binding formal template parameters to actual ones, resulting in a concrete class

Primitive Types using a class notation

<<enumeration>>

<table>
<thead>
<tr>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
</tr>
<tr>
<td>true</td>
</tr>
</tbody>
</table>

<<dataType>>

| Int |
| { value range -2**31 to +2**31-1 } |
public interface SoundFromSpaceListener extends EventListener {
    void handleSoundFromSpace(SoundFromSpaceEventObject sfseo);
}

public class SpaceObservatory implements SoundFromSpaceListener {
    public void handleSoundFromSpace(SoundFromSpaceEventObject sfseo) {
        soundDetected = true;
        callForPressConference();
    }
}

Can you draw a UML diagram corresponding to this?
Package Diagrams: Standard Elements

- **Facade** — only a view on some other package.
- **Framework** — package consisting mainly of patterns.
- **Stub** — a package that serves as a proxy for the public contents of another package.
- **Subsystem** — a package representing an independent part of the system being modeled.
- **System** — a package representing the entire system being modeled.

*Is <<import>> transitive?*
*Is visibility transitive?*
*Does <<friend>> apply to all types of visibility: +, -, #?*
Dependency – Among Objects

• 3 Stereotypes of Dependency in Interactions among Objects:
  – *become*: the target is the same object as the source but at a later point in time and with possibly different values, state, or roles
  – *call*: the source operation invokes the target operation
  – *copy*: the target object is an exact, but independent, copy of the source