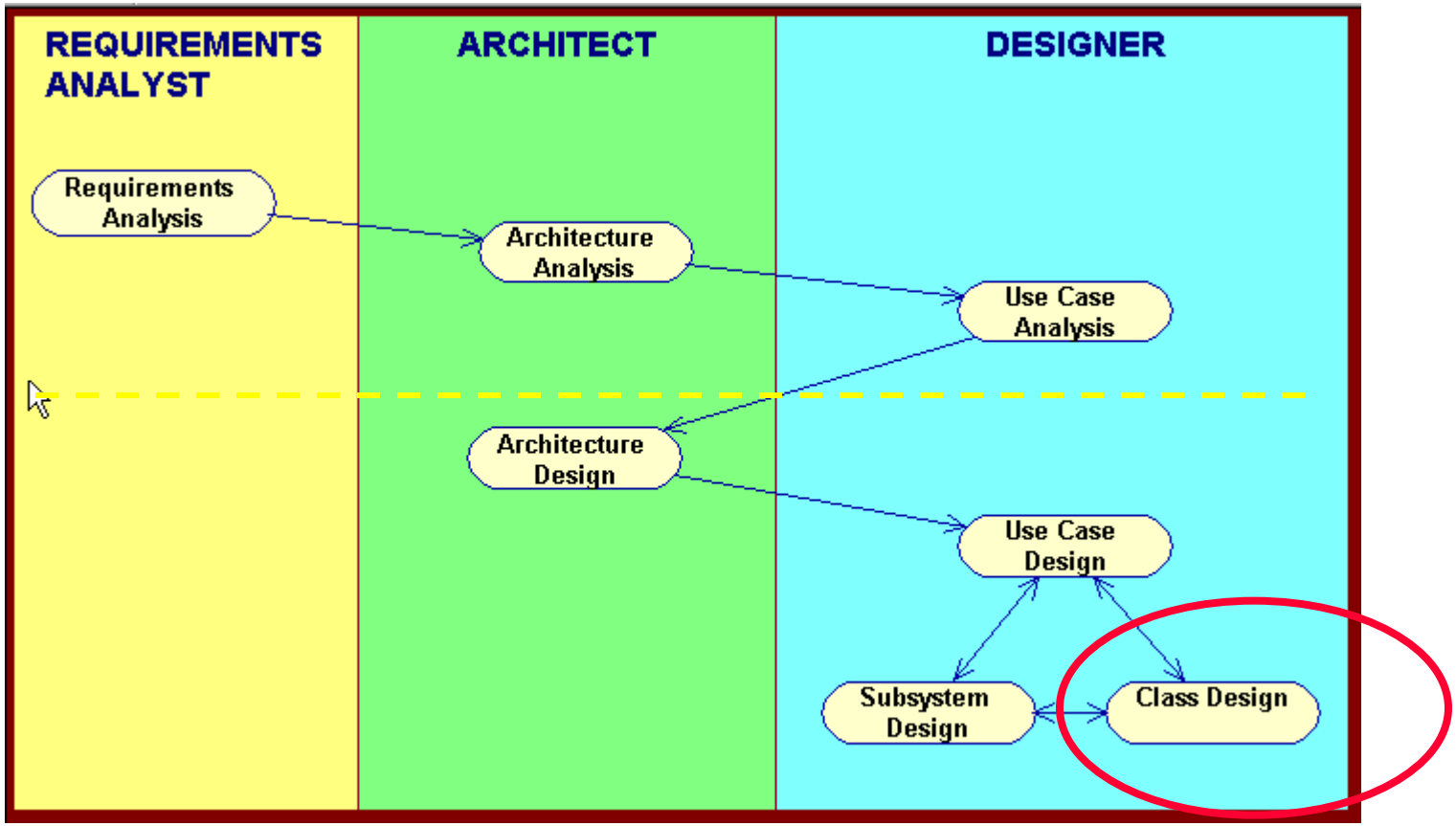
# Class Design

## *Finely Tuned Awareness Attends to the Details*

# Basic RUP OOAD Activities



# Class Design: Objectives

- Add additional classes and relationships needed to support implementation
  - Mechanisms
  - Patterns
  - Environment and language constraints
- Refine operations, attributes and relationships as needed for implementation

# MUMSched Class Design Conventions

1. Web Page Design and Navigation is part of Class Design. Show your web pages in your Use Case VOPCs.
2. Your VOPCs should match your code. Show inheritance hierarchies and the messages needed for your use case.
3. As a group we do **one class design** for the major **entities** that are used by many use cases (e.g. Entry, Block, Course, Section, Faculty, etc.)

# Class Design Topics

- Refine classes
  - Remaining analysis classes
  - Operations and methods
  - Attributes
- Refine relationships
  - Associations
    - Dependencies vs. Associations
    - Multiplicity
    - Aggregation and Composition
    - Navigation
  - Generalizations
- Review

# Refine Classes Topics

- Remaining analysis classes
- Operations and methods
- Attributes

# How Many Classes Are Needed?

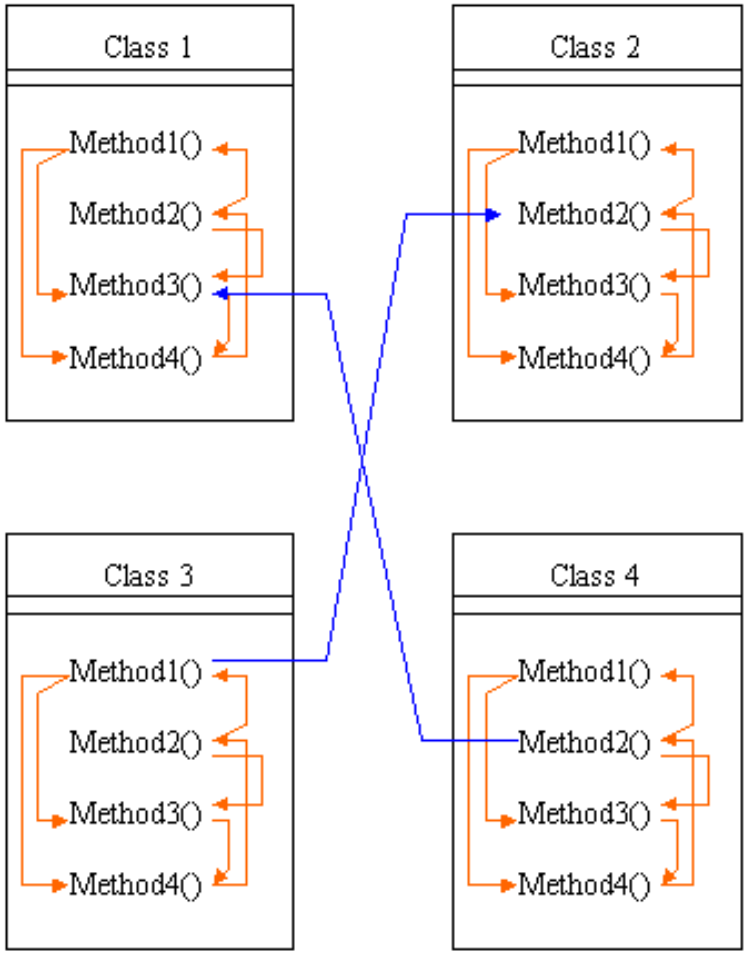
- Many simple classes
  - Encapsulates a small portion of the system intelligence
  - More reusable
  - Easier to implement
- A few complex classes
  - Encapsulates a large portion of the system intelligence
  - Less reusable
  - Harder to implement

**Guidelines:**

**1. A class should have a single well focused purpose which unifies its responsibilities, attributes, and methods.**

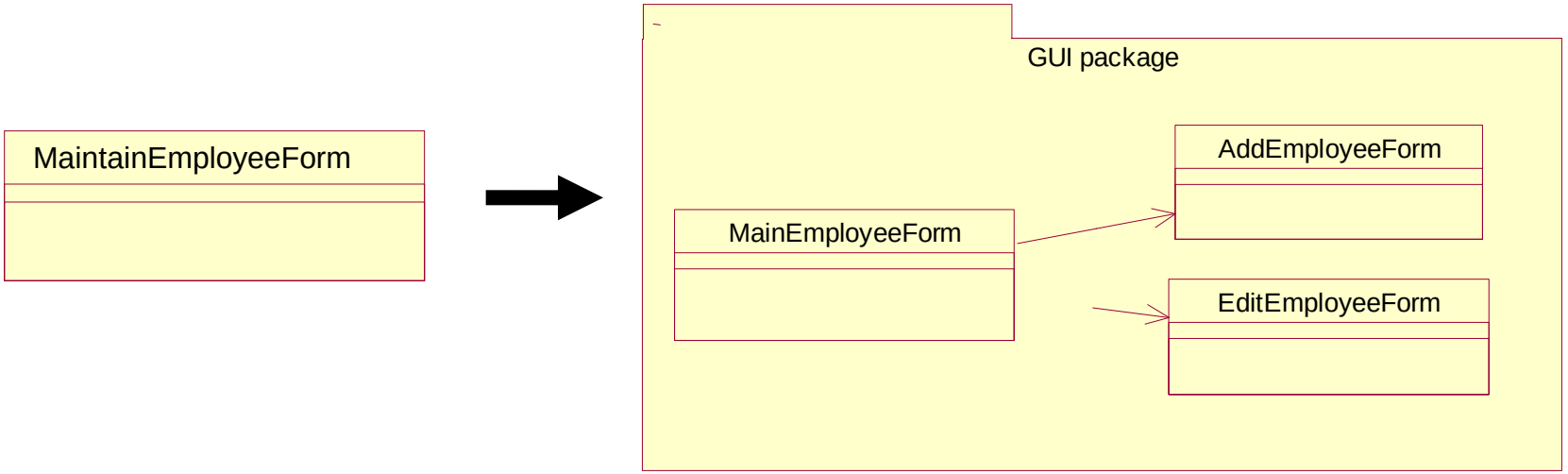
# Good Class Design

Hi Cohesion: maximum interaction within a class.
Low Coupling: minimal interaction between classes.



# Designing Boundary Classes

1. User Interface boundary classes
   - During Class Design, we work out the design details for the GUI.

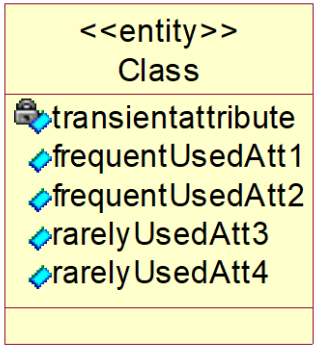



2. External system interface boundary classes
   - External systems are usually modeled as subsystems and interfaces.

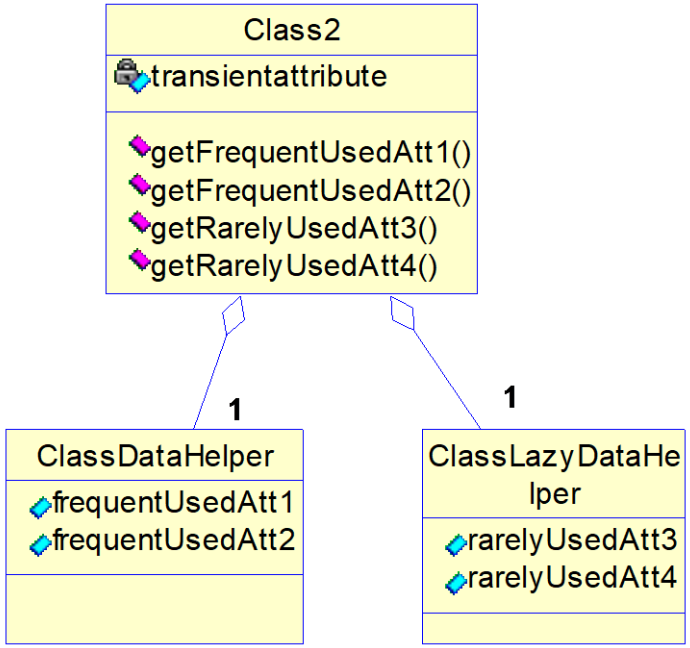
# Designing Entity Classes

- Entity classes are often persistent
- Persistent entity classes may need some refactoring to enhance performance.

**Analysis**




**Design**



# Class Exercise

- As a group review the VOPC design for generating a schedule.
  - Any controller classes to add?
  - Any entity classes to add?
  - Any attributes, associations, or dependencies to add?

# Refine Classes Topics

- Refine any remaining analysis classes
- Refine operations and methods
- Refine attributes

# Refine Operations

- Purpose: Now is the time to “get the details right”. During analysis we defined general responsibilities. During design we have become more and more definite about “how” these responsibilities are implemented. In Class Design we make final decisions about the details of operations.

- Steps
  - Define operation name, signature and description
  - Choose operation visibility
  - Define operation scope
  - The first word of the operation is usually a verb, e.g.
    - openAccount()
    - printMailingLabel()

# Define Operation Signature

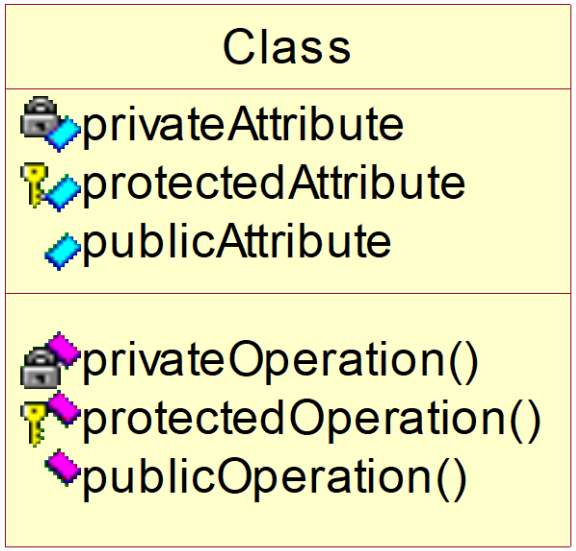
- Now in our VOPC operation signatures define all parameters, type of the parameters and the return type of the operation.

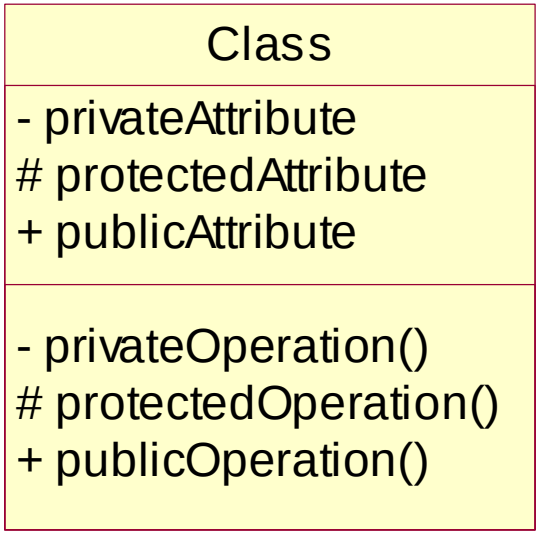
  operationName(parameter1: type, parameter2: type): returntype
- Use as few parameters as possible
  - decompose overly complex operations if necessary
- Pass objects (e.g., Book) instead of individual data fields (e.g., BookId).
- Defining operation signatures may lead to the discovery of additional classes and relationships.
  - E.g., Fees--needed info on amount, days overdue, etc

# Operation Visibility

- Visibility enforces encapsulation
- UML - 4 types of visibility (basically same as in Java)
  - public: operations are accessible by any client
  - protected: operations are only accessible by instances of the class itself, or instances of subclasses
  - private: operations are only accessible by instances of the class itself
  - package:operations accessible within the package
- What visibility should I use?
  - Look at interaction diagrams
    - message from outside object => public
    - message from subclass => protected
    - message from itself => private

# Visibility in UML



# Refine Classes Topics

- Refine any remaining analysis classes
- Refine operations and methods
- Refine attributes

# Define Attributes

- Purpose: Formalize definition of attributes

- Task List for Defining Attributes
  - Define attribute name, type and initial value
  - Choose attribute visibility
  - Specify if attribute is persistent

# Class Design Topics

- Refine classes
- Refine relationships
  - Associations
  - Generalizations
- Address non-functional requirements
- Review

Remaining analysis classes
Operations and methods
Attributes

# Refine Associations Topics

- Dependencies vs. Associations
- Multiplicity
- Aggregation and Composition
- Navigation

# Refining Associations

- Analysis assumed structural (i.e. permanent) relationships
- This may or may not be required for all analysis associations
- Could be temporary associations (that is, dependency)

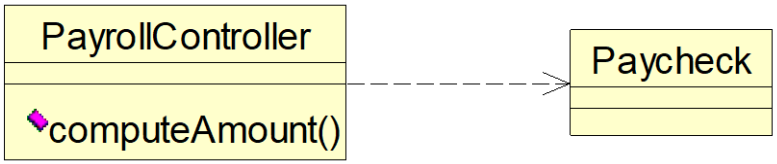
Q177-178, B69

# Dependency Relationships

- Four types of dependency relationships
  - Local variable
  - Parameter
  - Return
  - Global

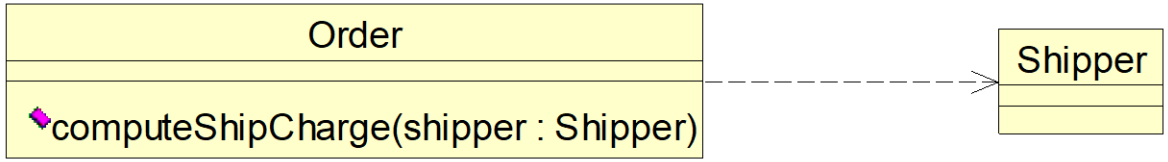
# Local Variable

- A local reference to the dependent class
- E.g. computeAmount() contains a local variable of the type Paycheck.



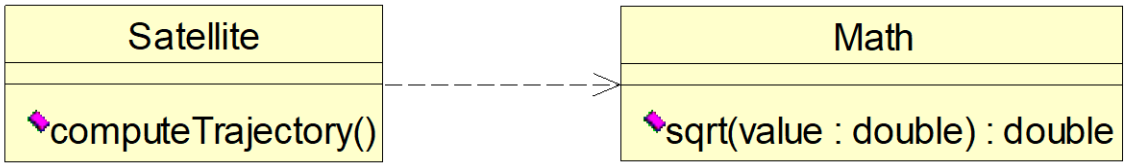
# Parameter And Return Types

- Parameter or return value of a method in the class
- E.g. computeShipCharge() takes a Shipper object as a parameter



# Global

- The dependent class is globally visible
- E.g. computeTrajectory() references the static sqrt() method of the Math class



# Dependency vs. Association

- Depends on your circumstances
  - Temporary vs. permanent
  - Cost of instantiation vs. reduced coupling
  - Usually there is no need for one class to have an association with a singleton class
  - What would be examples from MUMSched of analysis associations that we changed to a dependency during design.

# Refine Associations Topics

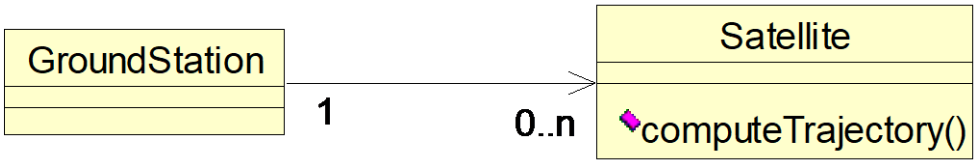
- Dependencies vs. Associations
- Multiplicity
- Aggregation and Composition
- Navigation

# Multiplicity

- How many objects (instances) of the class can be associated with one object (instance) of the other class.
- All Associations/Aggregations must have multiplicity specified
- Multiplicity of 1 or 0..1 need no further design
- Multiplicity of > 1 may need further design

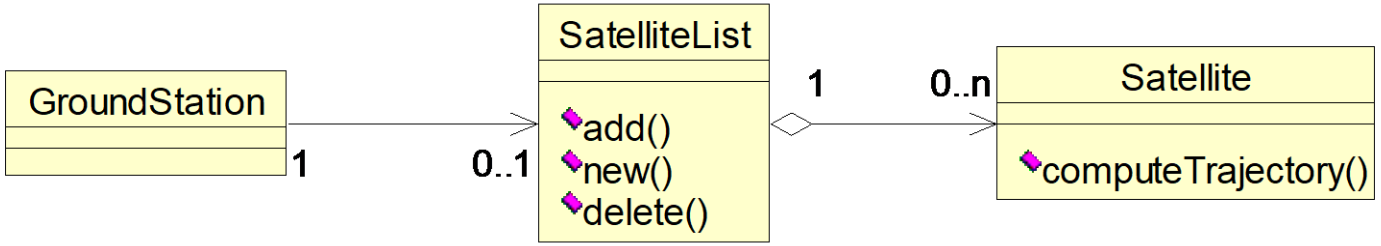
# Multiplicity Refinement

- Multiplicity > 1 needs ‘collection’ class
  - Modeled explicitly
  - Modeled as a note
  - Not modeled



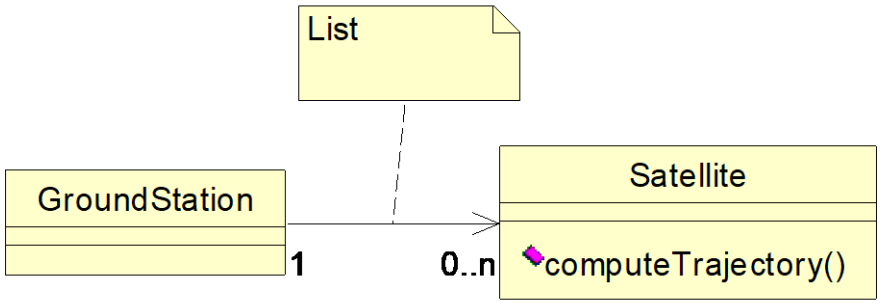
# Modeled Explicitly

- Addition of a ‘collection’ class



# Modeled as a Note

- Just indicate that a ‘collection’ will be used



# Multiplicity

- Give an example of multiplicity and your modeling approach from our MUMSched Project.
- Can you think of examples of our frameworks enforcing proper delegation and minimizing fragile code.

# Refine Associations Topics

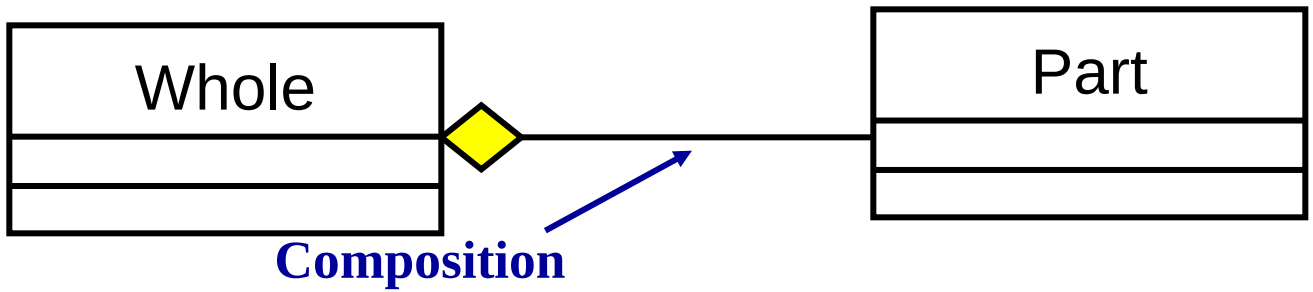
- Dependencies vs. Associations
- Multiplicity
- Aggregation and Composition
- Navigation

# Refining Aggregations

- Aggregation vs Composition
- Composition vs Attributes

# Review: Aggregation and Composition

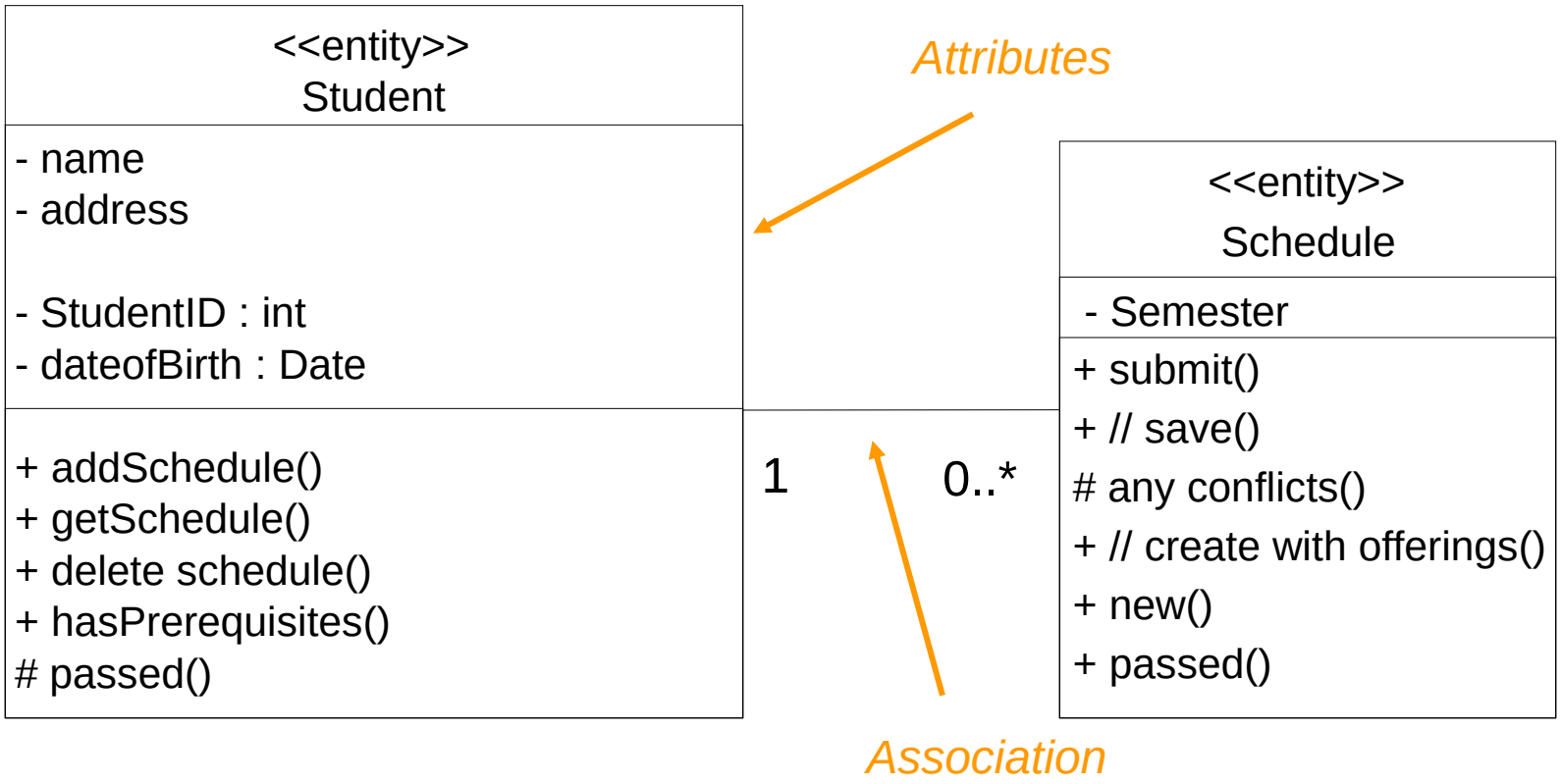
- Aggregation
  - Whole-part relationship
  - E.g., Class is an aggregation of students
- Composition
  - A form of aggregation with strong ownership and coincident lifetimes
  - The parts cannot survive the whole/aggregate
  - Example: Company has a composition relationship with Department



# Implementing Properties Using Association Or Attributes

- Use a class and association when
  - Properties need independent identities
  - Multiple classes have the same properties
  - Properties have a complex structure and properties of their own
  - Properties have complex behavior of their own
  - Properties have relationships of their own
- Otherwise use attributes

# Example: Modeling Properties: Attributes Vs Association




Did you implement  a student transcript in MUMSched for checking SectionRegistration?

# Class Exercise

- As a group create your MUMSched class diagram for your entity classes showing associations, multiplicity, attributes, and methods.

# Module Summary

1. Class design is the final design step. It focuses on individual classes in preparation for implementation.
2. Meaningful to do now that superstructure is done—We have followed an architecture centric development, driven by use case realizations
3. Class refinement straightforward--methods and attributes in correct syntax, helper classes.
4. Association refinements require a good understanding of our design.
5. Code can now be efficiently and effectively generated, integrated, reverse engineered, verified, validated, updated, maintained, and reused.

# Class Design Main Points

## CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

1) Classes are refined by defining method signatures, incorporating proper language syntax, and adding required helper classes. Now that we have the general structure for the class in place, filling in the last syntactic details is natural and effortless. This illustrates the Principle of Diving.

2) Associations are refined to accurately capture dependencies, multiplicities, aggregation, and navigation. This is a type of purification process where we remove remaining inconsistencies or inaccuracies in how we are modeling the application domain, and is made possible by all earlier design activities. This illustrates the Principle of Purification of the Path.

.

# Class Design Main Points

3. **Transcendental consciousness** is the fully expanded value of attention. We use TM to experience the broadest perspective.

4. **Impulses within the transcendental field**: On the ground of the fully expanded value of consciousness it is possible for the detailed design of creation to unfold within this field in a mistake-free manner.

5.**Wholeness moving within itself:** In unity consciousness we experience complete harmony between the finest details and the broadest perspective.

.