Associations and Inheritance

Dr. Isaac Griffith Idaho State University

Outcomes

Idaho State University | Computer Science

After today's lecture you will:

- Have and understanding of the different types of UML Diagrams
- Understand the types of relationships between classes/objects
- Be capable of using these relationships in Class Diagrams
- Be capable of translating basic Class Diagrams to working code.
- Understand the basic patterns of class collaborations

Associations

CS 2263

Association

Idaho State University | Computer Science

- **Association** - a relation among two or more classes describing a group of links with common structure and semantics
  - Implies an object of one class is making use of an object of another class
  - Indicated by a solid line connecting two classes
  - Does not imply there is always a link
  - Does imply a persistent, identifiable connection
- Can represent physical or conceptual relationships

Implementing Associations

- Implementation can take several forms:
  - Class A stores a key (or several keys) that uniquely identifies an object of class B
  - Class A stores a reference(s) to object(s) of class B
  - Class A has a reference to an object of class C, which, in turn is associated with an unique instance of class B
- The first and second are direct associations
- The third is indirect association

Example Class Diagram

Example Implementation

```
import java.util.*;

public class Student {
  private List<Section> enrolledIn;
  // remaining fields and methods ...
}

public class Section {
  private List<Student> students;
  private Course belongsTo;
  // remaining fields and methods ...
}

public class Course {
  private List<Section> sections;
  // remaining fields and methods ...
}
```

Directionality

Idaho State University | Computer Science

- Associations are assumed to be **bi-directional**
- We change this by placing an open arrow on an end
- Additionally, we can name an association by placing a descriptive name centered along and above the line
  - This name also typically implies direction
- Examples:
  - A student **enrolls** in a section, which **belongs to** a course
  - We can invert this to be: a course **has** sections that **enroll** students

Roles

- Typically the ends of associations will have named **roles**
- For example:
  - Section has zero or more `students` of type `Student`
  - Course has zero or more `sections` of type `Section`
- During implementation role names become field names with the type of the role end type

Arity of Relationships

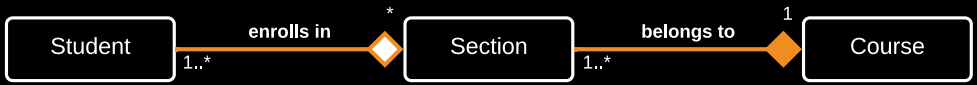
Idaho State University | Computer Science

- Arity, also called **multiplicity**, of a relationship between Class A and Class B denotes how many objects of Class A exist for each object of Class B
- Arity can be **one-one**, **one-many**, or **many-many**
- It is normally depicted as two numbers separated by ".." where the left denotes the minimum and the right the maximum
  - 1 only one (min = 1, max = 1)
  - 0..1 zero or one
  - 0..* zero to many
  - * zero to many
  - 1..* one to many
  - 2..10 two to ten

Containment Relationships

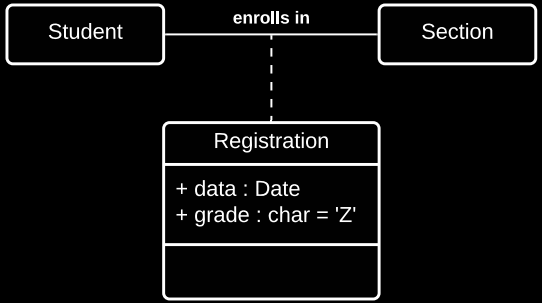
Idaho State University | Computer Science

- Objects are often composed of other objects
- In UML this relationship is denoted by a special form of association, **containment**
- There are two forms of containment:
  - **Aggregation** - An association where the object of class A is “made up of” objects of class B. Indicates a whole-part relationship.
    - Denoted by an open diamond on the class A side
  - **Composition** - An aggregation in which each instance of the part belongs to only one instance of the whole, and that the part cannot exist except as part of the whole.
    - Denoted by a filled in (black) diamond on the class A side



Association Classes

- Associations with additional information
  - qualifying its nature
  - describing its properties
- When the information is not useful outside the context of the relationship
  - We treat the association itself as a class
- For example when students enroll in a section, a registration record is created
  - This record stores the registration date and grade
  - Yet, the record does not make sense if a particular student doesn't enroll in a given section



Collaboration Patterns

CS 2263

Object Selection

- When it comes to defining objects within a domain model, there are four categories:
  - **People:** Generally perform actions within a given context (such as a role)
    - Each context should be modeled with a more specific type of person
    - Additionally, we can personify non-human entities such as companies (they becom "Actors")
  - **Places:** Actions that take place in a location should be modeled with a place
  - **Things:** Things are the typical target of an action.
    - Often represented by a general type definition and a more specific form of the thing.
  - **Events:** These tie people, places, and things together in a moment of time (or range of time)

Object Collaboration Patterns

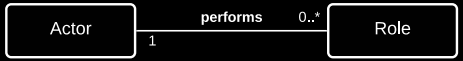
Idaho State University | Computer Science

- Objects tend to work together in order to model and fulfill requirements
- These relationships, or collaborations, follow 12 general patterns
- Within each pattern, each object is a “pattern player” and has specific responsibilities
  - responsibilities in regard to defining behavior
  - responsibilities in regard to enforcing constraints

People and Places Patterns

Idaho State University | Computer Science

Actor – Role



- Role represents an Actor in a specific context
- `Actor` may know zero or more `Roles`
  - Though the set of `Roles` is typically unique
- `Role` may only know one `Actor`

Examples

- Customer – Buyer
- Customer – Seller
- Person – LibStaff
- Person – LibUser

OuterPlace – Place

- `Place` is a location where things happen
- `OuterPlace` may know one or more `Places`
- `Place` may know at most one `OuterPlace`
- Relationship may be hierarchical

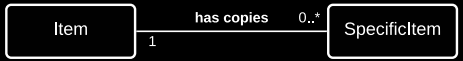
Examples

- Region – Office
- Country – Branch
- Branch – Room

Things Patterns

Idaho State University | Computer Science

Item – SpecificItem



- `SpecificItem` is a specific representation of a generic `Item`
- `Item` may know zero or more `SpecificItems`
- `SpecificItem` must know exacly one `Item` and cannot exist without it.

Examples

- Book – BookCopy
- PhoneModel – Phone
- Video – VideoTape

Assembly – Part

- `Assembly` is an aggregation of `Parts`
- `Assembly` must know at least one `Part`
- `Part` may know only one `Assembly` and cannot exist outside of an `Assembly`

Examples

- Parcel – Content
- Book – Chapter

Things Patterns

Idaho State University | Computer Science

Container – Content

- `Container` is a collection of `Content`
- `Container` may know zero or more `Contents` (can be empty)
- `Content` may know at most one `Container`
  - Can exist outside `Container` or move to another `Container`
- Relationship may be hierarchical

Examples

- BookShelf – Book
- FileCabinet – File

Group – Member

- `Group` is a collection of `Members`
- `Group` knows zero or more `Members` (can be empty)
- `Member` knows zero or more `Groups`
- Relationship may be hierarchical

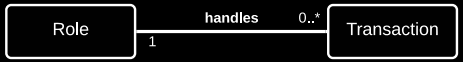
Examples

- Team – Student
- Club – Member

Events Patterns

Idaho State University | Computer Science

Role – Transaction

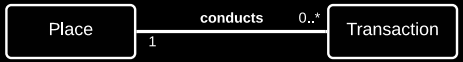


- `Role` handles `Transactions`
- `Role` knows of zero or more `Transactions`
- `Transaction` knows of only one `Role`

Examples

- Customer – Order
- Student – Register
- Buyer – Payment

Place – Transaction



- `Place` conducts `Transactions`
- `Place` knows of zero or more `Transactions`
- `Transaction` knows of only one `Place`

Examples

- Branch – Withdrawal
- Counter – Purchase

Events Patterns

Idaho State University | Computer Science

SpecificItem – Transaction

- `SpecificItem` is involved in `Transactions`
- `SpecificItem` knows of zero or more `Transactions`
- `Transaction` knows of only one `SpecificItem`

Examples

- VideoCD – Rent
- BookCopy – Loan

CompositeTransaction – LineItem

- `CompositeTransaction` is a collection of `LineItems`
- `CompositeTransaction` contains one or more `LineItems` (cannot be empty)
- `LineItem` knows of one `CompositeTransaction` and cannot exist outside of the `CompositeTransaction`

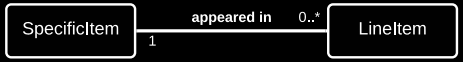
Examples

- Order – OrderDetail
- Performance – Event

Events Patterns

Idaho State University | Computer Science

SpecificItem – LineItem

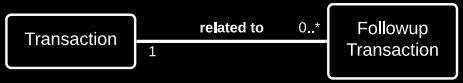


- `SpecificItem` appears within `LineItems`
- `SpecificItem` knows of zero or more `LineItems`
- `LineItem` knows of a single `SpecificItem`

Examples

- VideoCD – RentDetail
- Product – OrderDetail

Transaction – FollowupTransaction



- `Transaction` is related to `FollowupTransaction`
- `Transaction` knows of zero or more `FollowupTransactions`
- `FollowupTransaction` knows of a single `Transaction`

Examples

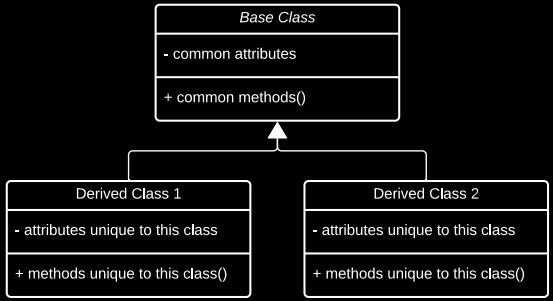
- Order – Payment
- Reserve – Purchase

Inheritance

CS 2263

Inheritance

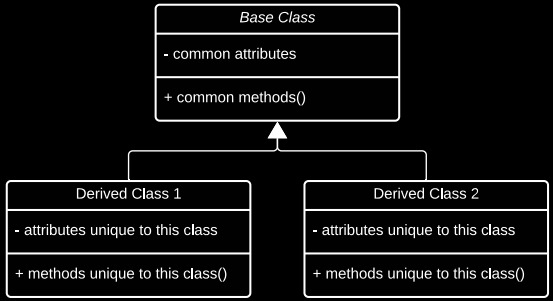
- When two classes have both
  - A great deal of similarity
  - Significant differences
- Association is not applicable
  - Too similar to be captured by association
  - Differ too much that genericity cannot be employed
- If we have two such classes `C1` and `C2`
  - We extract the common aspects into class `B`
  - Reducing the size of `C1` and `C2`
  - Capturing only the unique properties and methods for each



Inheritance

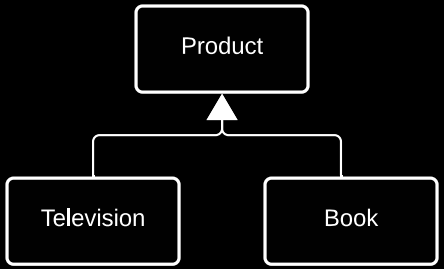
Idaho State University | Computer Science

- This is **inheritance**, where `C1` and `C2` **inherit** from `B`
- `B` is the `baseclass` or `superclass` and `C1` and `C2` are the **derived classes** or **subclasses**
- Superclasses are generalizations or **abstractions**
  - we move toward a more general type (upward movement)
  - we denote **specialization** toward more specific classes (downward movement)
  - this forms a **hierarchy**



Example Hierarchy

Hierarchy



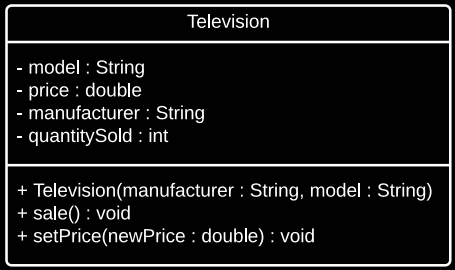
Implementation

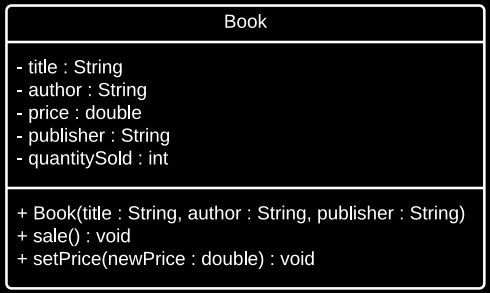
```
public class Product {
  // functionality of product
}

public class Television extends Product {
  // functionality that is unique to televisions
  // modifications
}

public class Book extends Product {
  // functionality that is unique for books
  // modifications
}
```

Class Structure

Idaho State University | Computer Science

- We can consider two entities in isolation without worrying about similarities



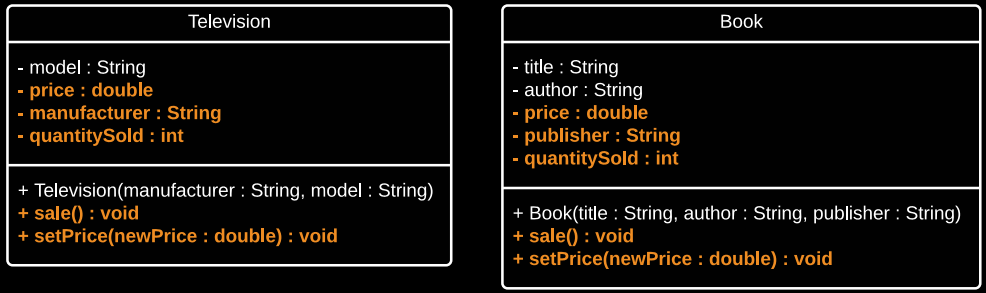


- Now notice the similarities

Class Structure

Idaho State University | Computer Science

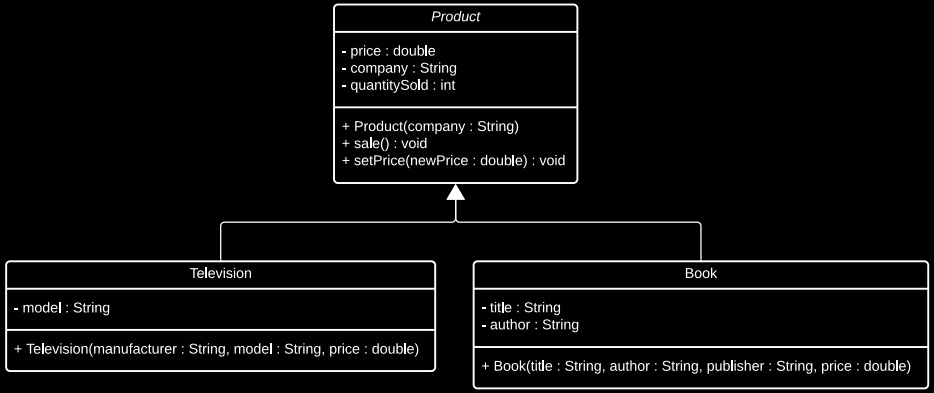
- We can consider two entities in isolation without worrying about similarities



- Now notice the similarities

Class Structure

- OO provides the power to utilize these similarities by reducing the `Television` and `Book` classes by having them extend a common `Product` class



Implementation

Idaho State University | Computer Science

Product

```
public class Product {
  private String company;
  private double price;
  private int quantitySold;

  public Product(String company, double price) {
    this.company = company;
    this.price = price;
  }

  public void sell() {
    quantitySold++;
  }

  public void setPrice(double newPrice) {
    price = newPrice
  }

  public String toString() {
    return "Company: " + company + " price: " +
      price + " quantity sold " + quantitySold;
  }
}
```

Implementation

Idaho State University | Computer Science

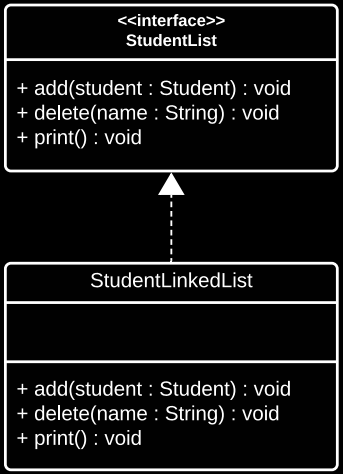
Television

```
public class Television extends Product {

  private String model;

  public Television(String manufacturer,
      String model, double price) {
    super(manufacturer, price);
    this.model = model;
  }

  public String toString() {
    return super.toString() +
      " model: " + model;
  }
}
```

Book

```
public class Book extends Product {

  private String title;
  private String author;

  public Book(String title, String author,
      String publisher, double price) {
    super(publisher, price);
    this.title = title;
    this.author = author;
  }

  public String toString() {
    return super.toString() + " title: "
      + title + " author: " + author;
  }
}
```

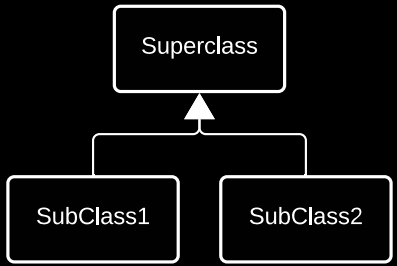
Interface Realization



- The relationship between an interface and the implementing class is called **realization**
- Depicted in UML as a dotted line with a large open arrowhead pointing to the interface.

Polymorphism

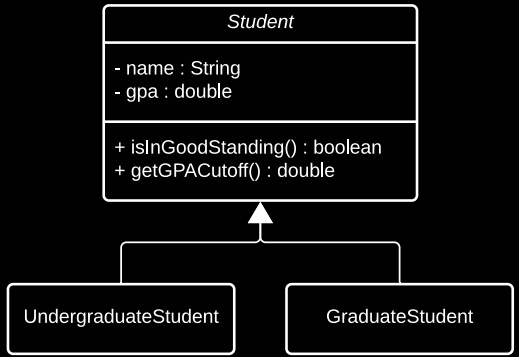
- In general the following polymorphic rules apply to any type hierarchy
  1. Any object of type `SubClass1` or `SubClass2` can be store in a reference of type `SuperClass`
  2. No object of type `SubClass1` (`SubClass2`) can be stored in a reference of type `SubClass2` (`SubClass1`)
  3. A reference of type `SuperClass` can be cast as a reference of type `SubClass1` or `SubClass2`



- Thus, a reference is able to point to objects of different types as long as the actual types of these objects conform to the type of reference.
- The above rules are the rules of **conformance**

Polymorphism

- Polymorphism will result in a loss of information at run time
- But, it will allow us to invoke methods without knowing the specific object we are dealing with
- This is provided via **Dynamic Binding**
  - If we have the hierarchy depicted, dynamic binding allows
    - Us to have a variable of type `GraduateStudent`
    - on which we can call `getGPACutoff()`
  - Similarly, if we override `isInGoodStanding()` in `UndergraduateStudent`
    - If we have a variable of type `Student` but it is an instance of `UndergraduateStudent`
    - If we call `isInGoodStanding()` it will know to call the one in `UndergraduateStudent`
    - But a call to `getGPACutoff()` will be called in `Student`



The `Object` Class

Idaho State University | Computer Science

- In Java, the superclass of all classes is `java.lang.Object`
- Thus from polymorphism we know that a variable of type `Object` can hold a reference to any type
- The following is legal:

```
Object any;
any = new Student();
any = new Integer(4);
any = "Some String";
```

Genericity

Idaho State University | Computer Science

- Mechanism for creating entities that vary only in the types of their parameters.
- Can be associated with any entity (class or method) that requires type parameterization.
- A generic entity replaces types with placeholders (**generic parameters**)
- Because of this generic entities are **not fully specified** and thus cannot be directly instantiated
- Generics relieve us of the need to classes that require the use of `Object` and require and overuse of casting and exception handling

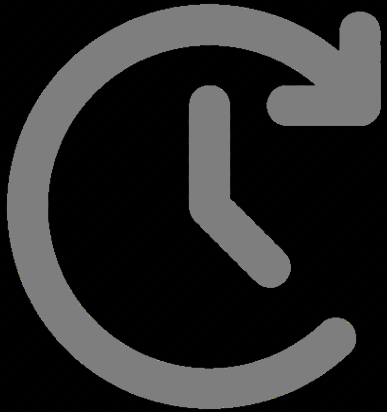
Genericity Example

Idaho State University | Computer Science

```
public class Stack<E> {
  public void push(E item) {
    // code to push item onto stack
  }

  public E pop() {
    // code to pop item from stack
  }
}

Stack<Integer> stack = new Stack<>();
```

For Next Time

- Review Chapters 2 and 3 from the book
- Review this Lecture
- Come to Class
- Continue working on Homework 02



Are there any questions?