Unified Modeling Language (UML)

An extremely expressive language.

We’ll use only a small part of the language, Class Diagrams, to represent basic OO design.
Example: Class Person

<table>
<thead>
<tr>
<th>Data members</th>
</tr>
</thead>
<tbody>
<tr>
<td># name : String []</td>
</tr>
<tr>
<td># dob : String</td>
</tr>
<tr>
<td># gender : String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Person(name: String[], dob: String, gender: String)</td>
</tr>
<tr>
<td>+ getName(): String []</td>
</tr>
<tr>
<td>+ setName(name: String[]): void</td>
</tr>
<tr>
<td>+ getDob(): String</td>
</tr>
<tr>
<td>+ setDob(dob: String): void</td>
</tr>
<tr>
<td>+ getGender(): String</td>
</tr>
<tr>
<td>+ setGender(gender: String): void</td>
</tr>
<tr>
<td>+ toString(): String</td>
</tr>
</tbody>
</table>
Notation

Data members:

name: type

Methods:

methodName(param1: type1, param2: type2, ...): returnType

Visibility:

- private
+ public
# protected
~ package

Static: underline
Example: Inheritance

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td># name : String []</td>
</tr>
<tr>
<td># dob : String</td>
</tr>
<tr>
<td># gender : String</td>
</tr>
<tr>
<td>+ Person(name: String[], dob: String, gender: String)</td>
</tr>
<tr>
<td>+ getName(): String []</td>
</tr>
<tr>
<td>+ setName(name: String[]) : void</td>
</tr>
<tr>
<td>+ getDob() : String</td>
</tr>
<tr>
<td>+ setDob(dob: String) : void</td>
</tr>
<tr>
<td>+ getGender() : String</td>
</tr>
<tr>
<td>+ setGender(gender: String) : void</td>
</tr>
<tr>
<td>+ toString(): String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>- studentNum : String</td>
</tr>
<tr>
<td>- studentCount : int = 0</td>
</tr>
<tr>
<td>+ Student(name: String[], dob: String, gender: String, studentNum: String)</td>
</tr>
<tr>
<td>+ getStudentNum(): int</td>
</tr>
<tr>
<td>+ toString(): String</td>
</tr>
<tr>
<td>+ getStudentCount(): int</td>
</tr>
</tbody>
</table>
Notation (cont’d)

Abstract method: *italic*

Abstract class: *italic* or `<abstract>`

Interface: `<interface>`

Relationship between classes:

- Inheritance
- Interface
Example: Abstract Class

```
«abstract»
Grade

+ gpa() : double

LetterGrade
- grade: String
+ LetterGrade(grade: String)
+ gpa() : double
+ toString() : String

NumericGrade
- grade: int
+ NumericGrade(grade : int)
+ gpa() : double
+ toString() : String
```
Example: Abstract Class

```
+ VALID_GRADES : String[] {readOnly}
+ toLetter(grade : int) : String
+ gpa() : double

- grade: String
  + LetterGrade(grade: String)
  + gpa() : double
  + toString() : String

- grade: int
  + NumericGrade(grade : int)
  + gpa() : double
  + toString() : String
```
Design Patterns

A **design pattern** is a general description of the solution to a well-established problem using an arrangement of classes and objects.

Patterns describe the shape of code rather than the details. They’re a means of communicating design ideas.

They are not specific to any one programming language.

You’ll learn about lots of patterns in CSC301 (Introduction to Software Engineering) and CSC302 (Engineering Large Software Systems).
Iterator Design Pattern

Context

- A container/collection object.

Problem

- Want a way to iterate over the elements of the container.
- Want to have multiple, independent iterators over the elements of the container.
- Do not want to expose the underlying representation: should not reveal how the elements are stored.
Iterator Design Pattern: Solution

```
+ first()
+ next()
+ isDone()
+ currentItem()
```

```
+ createIterator(): Iterator
```

Returns instance of YourIteratorClass.
Iterator Design Pattern: Java
Iterator: Example in Java

```
<interface>
Iterator<T>
+</interface>

+ hasNext(): boolean
+ next(): T

AddressBookIterator

<interface>
Iterable<T>
+</interface>

+ iterator(): Iterator<T>

AddressBook

+ iterator(): Iterator<Contact>

Returns instance of AddressBookIterator.
```