An Introduction To UML Class Diagrams

Classes

1. Represent a user-created or defined data type
   a. they are an abstract data type (ADT)
   b. they implement data hiding and encapsulation
   c. they establish a new scope
   d. the UML graphic shows class name, attributes, and operations

2. Class Name (first section)
   a. class name (italic) abstract class
   b. class name (straight) concrete class

3. Attributes (second section)
   a. [visibility] name [multiplicity] [: type] [= initial_value] [[property_string]]
   b. synonyms
      i. data
      ii. variable
      iii. instance variable, instance field, or instance data (Java)
      iv. data member (C++)
      v. state

4. Operations or behaviors (third section)
   a. [visibility] name ((parameter_list)) [: return_type] [[property_string]]
   b. synonyms
      i. method (Java)
      ii. function
      iii. member function (C++)
      iv. service
      v. sending a message to an object is equivalent to calling one of an object’s methods
   c. special purpose operations
      i. accessor returns an attribute attrType getAttr()
      ii. mutator sets an attribute void setAttr(attrType attr)
      iii. Example:
         (1) String name; // attribute
         (2) String getName(); // accessor
         (3) void setName(String name); // mutator

5. Legend
   a. scope/visibility
      i. + public Program scope
      ii. − private Class scope
      iii. # protected Inheritance hierarchy scope
      iv. default (i.e., friendly) Package scope
   b. ownership
      i. underlined class ownership; one value shared by the class and all instances: static
      ii. not underlined instance ownership; each instance or object maintains its own, distinct value
   c. properties
      i. changeable attribute may be modified
      ii. addOnly for attributes with multiplicity greater than one: values may be added but once added, a value cannot be removed
      iii. frozen attribute’s value may not be changed once initialized: const or final

6. Stereotypes— improve the organization of long lists of attributes and operations
   a. «constructor»
   b. «process»
   c. «query»
   d. «helper»
A class with an accessor (setter) or mutator (getter) operation (method) exposes a (in a controlled way) a portion of its interior. Together/J denotes such a class (1) by adding a small rectangle to the upper left of the class diagram and (2) by combining the attribute (data) name and the method name in a fourth section of the class diagram. Compare the following class diagram with the generated code.
Class Relationships

1. Inheritance or Generalization
   a. relationship between a class and a more refined version
      (1) Base Class Derived Class
      (2) Superclass Subclass (Java preferred)
      (3) Parent Class Child Class
      (4) Generalization Specialization
      (5) Ancestor Descendant
   b. A subclass inherits all of the features (attributes and behaviors) of the superclass (i.e., a child has everything that the parent has but may not be able access it directly)
   c. Definitions
      i. abstraction for sharing similarities among classes while preserving their differences
         (1) superclass and subclass share all features defined in the superclass
         (2) subclass alone has the features defined in the subclass
      ii. mechanism for code reuse
      iii. conceptual simplification by reducing the number of unique features (i.e., helps control complexity)
      iv. an instance of a subclass is simultaneously an instance of all its ancestor classes (i.e., it contains sub-objects instantiated from all of its ancestor classes); this implies that when a subclass is instantiated that a chain of constructor calls runs from the top of the inheritance hierarchy to the class being directly instantiated
      v. overriding—subclass defines operation with the same name and the same signature
   d. inheritance is a relationship between classes and is called an “is a” relationship (in terms of the example below, Employee is a Person)
   e. generalization is called an “is a kind of” relationship (generalization includes inheritance but also permits relationships between other entities such as packages)
   f. when the subclass overrides one or more methods in the superclass, the relationship is often called a simile or an “is like a” relationship

```java
/* Generated by Together */

public class Employee extends Person {
}
```
2. **Association**
   a. a connection between peers
   b. shows structural relationships, which allows objects to call each others methods (i.e., operations are invoked along association pathways)
   c. associations between objects need not be permanent nor must they be created at the same time that the objects are instantiated; the objects may be instantiated at different times (i.e., they have distinct lifetimes or existences)
   d. navigation may be bidirectional or directed; the association name indicates the forward direction
   e. may be one-to-one, one-to-many (illustrated below), or many-to-many
   f. a class may have a connection with itself
   g. implemented as object references in Java (i.e., attributes that reference other objects) and pointers in C++
   h. adornments are optional
      i. name and direction (note that Together/J employs separate, directed lines or arrows for each association and does not use the standard UML direction adornment)
      ii. role
      iii. multiplicity
         1. 1 exactly 1
         2. 0..1 0 or 1
         3. 1..* 1 or more
         4. 0..* 0 or more
         5. 2..4 2, 3, or 4 (other numbers are permitted)
         6. 0..* 0 or more

```java
/* Generated by Together */

class Contractor {
   /**
    * @label Works On
    */
   private Project worksOn;
}

/* Generated by Together */

class Project {
   /**
    * @label Personnel
    * @clientCardinality 1
    * @supplierCardinality 1..*
    */
   private Contractor[] personnel;
}
```
3. **Dependency, using, or delegation**
   a. temporary or ephemeral form of unidirectional association
   i. dependency one class relies on the specification of another class
   ii. using one class uses the services (operations, methods) of another class
   iii. delegation one class delegates part of its responsibilities to another class
   b. implemented as local or parameter objects to an operation or method (i.e., the association ends when the operation completes)
   c. at the implementation level, dependency differs from association by where the reference variable is defined (it is defined as an instance variable in association and as a local/parameter variable in dependency)
   d. dependencies can be named but are usually not

```java
/* Generated by Together */

public class FooBar {
    /** @link dependency */
    /*#Graphics lnkGraphics;*/
    public void paint(Graphics g) {
        g.drawString("Hello world", 20, 40);
    }

    /** @link dependency */
    /*#StringTokenizer lnkStringTokenizer;*/
    public void searchEntries(String record) {
        for (String line = db.getFirst(); line != null; line = db.getNext()) {
            StringTokenizer parser = new StringTokenizer(line, ":");
            while (parser.hasMoreTokens()) {
                String token = parser.nextToken();
                . . .
            }
        }
    }
}
```
4. **Aggregation**

a. form of association showing subordination
b. whole/part or “has a” relationship (e.g., Employee has an Address and Employee has a Project)
c. navigation is unidirectional from the whole class to the part class
d. implemented as class scope reference (Java) or pointer (C++) variables defined in the whole class
e. parts exist independently of the whole (i.e., parts are loosely bound to the whole)
   i. parts may change during program execution
   ii. parts may belong to multiple wholes or aggregates
   iii. parts are instantiated independently of the whole and the references are established as needed (by mutator methods or possibly by constructor methods)
f. loose form of Composition (see next)
g. is transitive: if A has a B, and B has a C, then A has a C (or if C is part of B, and B is part of A, then C is part of A)
h. is antisymmetric: if A is part of B, B cannot be part of A
i. multiplicity
   i. 1 is understood
   ii. 1 exactly 1
   iii. 0..1 0 or 1
   iv. 1..* 1 or more
   v. 0..* 0 or more
   vi. 2,4 2 or 4 (other numbers are permitted)
   vii. 2..4 2, 3, or 4 (other numbers are permitted)

```java
/* Generated by Together */

public class Employee {

   /**
    * @link aggregation
    */
   private Address lnkAddress;

   /**
    * @link aggregation
    */
   private Project lnkProject;
}
```
5. **Composition**

a. tight form of aggregation (and transitively of association)

b. a whole/part or “contains a” relationship (e.g., Automobile contains a Motor and Automobile contains a Transmission)

c. navigation is unidirectional from the whole class to the part class

d. implemented as class scope reference variables (Java) or as statically instantiated (value rather than address) objects (C++) defined in the whole class

e. strong ownership (a part may belong to only one whole)

f. parts have coincident lifetime with the whole (i.e., the parts live and die with the whole)

i. parts should be instantiated in the whole constructor or when defined in class scope

ii. parts are not changed or exchanged during program execution

iii. parts with nonfixed multiplicity may be created after the whole but once created they must live and die with the whole

g. the whole is responsible for the disposition of its parts (i.e., the whole creates and disposes of its parts)

h. is transitive: if A has a B, and B has a C, then A has a C (or if C is part of B, and B is part of A, then C is part of A)

i. is antisymmetric: if A is part of B, B cannot be part of A

j. multiplicity

i. 1 is understood

ii. 1 exactly 1

iii. 0..1 0 or 1

iv. 1..* 1 or more

v. 0..* 0 or more

vi. 2,4 2 or 4 (other numbers are permitted)

vii. 2..4 2, 3, or 4 (other numbers are permitted)

k. Note the use of the “final” modifier to achieve tight binding. This is not commonly used in the literature and is not automatically generated by Together/J. This idea was originally suggested by a former student. It is syntactically correct and more faithfully represents the UML composition relationship in Java

```java
/* Generated by Together */

public class Automobile {

   /**
   * @link aggregationByValue
   */
   private final Motor lnkMotor;

   /**
   * @link aggregationByValue
   */
   private final Transmission lnkTransmission;

}
```