UML Class Diagramming Guidelines

UML class diagrams show the classes of the system, their inter-relationships, and the operations and attributes of the classes. Class diagrams are typically used, although not all at once, to:

- Explore domain concepts in the form of a domain model
- Analyze requirements in the form of a conceptual/analysis model
- Depict the detailed design of object-oriented or object-based software

A class model is comprised of one or more class diagrams and the supporting specifications that describe model elements including classes, relationships between classes, and interfaces. There are guidelines for:

1. General Guidelines

Because class diagrams are used for a variety of purposes - from understanding requirements to describing your detailed design - you will need to apply a different style in each circumstance. This section describes style guidelines pertaining to different types of class diagrams.

Figure 1. Analysis and design versions of a class.

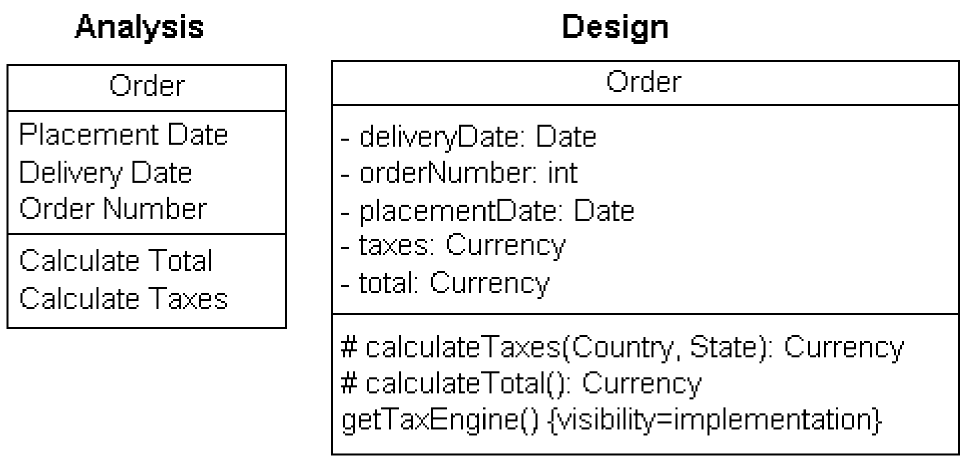
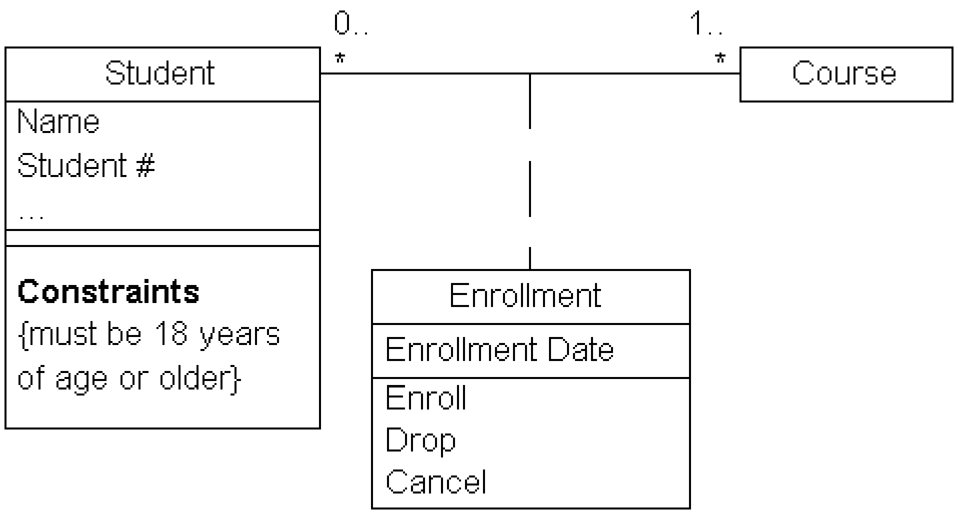


Figure 2. Modeling association classes.



1. Identify Responsibilities on Domain Class Diagrams.
2. Indicate Visibility Only On Design Models.
3. Indicate Language-Dependent Visibility With Property Strings.
4. Indicate Types Only On Design Models.
5. Indicate Types On Analysis Models Only When The Type is an Actual Requirement.
6. Design Class Diagrams Should Reflect Language Naming Conventions. In Figure 1 you see that the design version of the *Order* class uses names that conform to common Java programming conventions such as *placementDate* and *calculateTaxes()*.
7. Model Association Classes On Analysis Diagrams. Figure 2 shows that association classes are depicted as class attached via a dashed line to an association - the association line, the class, and the dashed line are considered one symbol in the UML.
8. Do Not Name Associations That Have Association Classes.
9. Center The Dashed Line of an Association Class.

2. Class Style Guidelines

A class is effectively a template from which objects are created (instantiated).Although in the real world Doug, Wayne, John, and Bill are all student objects we would model the class *Student* instead. Classes define attributes, information that is pertinent to their instances, and operations, functionality that the objects support. Classes will also realize interfaces (more on this later). Note that you may need to soften some of the naming guidelines to reflect your implementation language or software purchased from a third-party vendor.

Figure 3. The OrderItem class with and without scaffolding code.

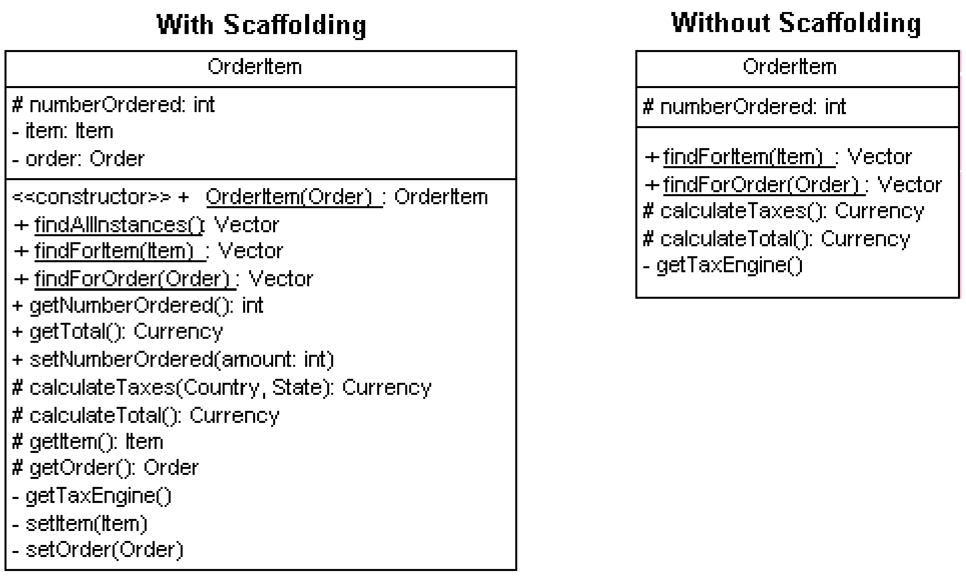

With Scaffolding
OrderItem
# numberOrdered: int
- item: Item
- order: Order
<<constructor>> + OrderItem(Order) : OrderItem
+ findAllInstances(): Vector
+ findForItem(Item) : Vector
+ findForOrder(Order) : Vector
+ getNumberOrdered(): int
+ getTotal(): Currency
+ setNumberOrdered(amount: int)
# calculateTaxes(Country, State): Currency
# calculateTotal(): Currency
# getItem(): Item
# getOrder(): Order
- getTaxEngine()
- setItem(Item)
- setOrder(Order)
Without Scaffolding
OrderItem
# numberOrdered: int
+ findForItem(Item) : Vector
+ findForOrder(Order) : Vector
# calculateTaxes(): Currency
# calculateTotal(): Currency
- getTaxEngine()

Figure 4. Indicating the exceptions thrown by an operation.

+ findAllInstances(): Vector {exceptions=NetworkFailure, DatabaseError}

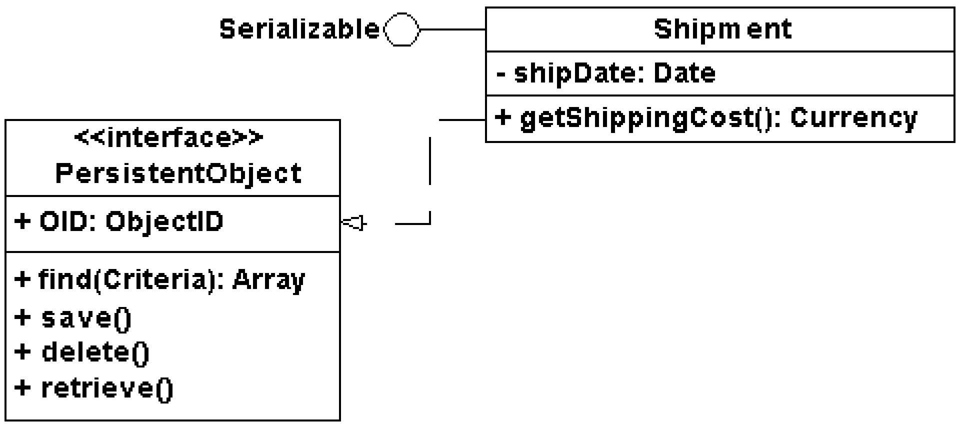
1. Use Common Terminology for Names
2. Prefer Complete Singular Nouns for Class Names
3. Name Operations with a Strong Verb

4. Name Attributes With a Domain-Based Noun
5. Do Not Model Scaffolding Code. Scaffolding code refers to the attributes and operations required to implement basic functionality within your classes, such as the code required to implement relationships with other classes. Figure 3 depicts the difference between the *OrderItem* class without scaffolding code and with it.
6. Never Show Classes With Just Two Compartments
7. Label Uncommon Class Compartments
8. Include an Ellipsis (. . .) At The End of Incomplete Lists
9. List Static Operations/Attributes Before Instance Operations/Attributes
10. List Operations/Attributes in Decreasing Visibility
11. For Parameters That Are Objects, Only List Their Type
12. Develop Consistent Method Signatures
13. Avoid Stereotypes Implied By Language Naming Conventions
14. Indicate Exceptions In An Operation's Property String. Exceptions can be indicated with a UML property string, an example of which is shown in Figure 4.

3. Interfaces

An interface is a collection of operation signature and/or attribute definitions that ideally defines a cohesive set of behaviors. Interfaces are implemented, "realized" in UML parlance, by classes and components - to realize an interface a class or component must implement the operations and attributes defined by the interface. Any given class or component may implement zero or more interfaces and one or more classes or components can implement the same interface.

Figure 5. Interfaces on UML class diagrams.



1. Interface Definitions Must Reflect Implementation Language Constraints. In Figure 5 you see that a standard class box has been used to define the interface *PersistentObject* (note the use of the <> stereotype).
2. Name Interfaces According To Language Naming Conventions
3. Apply "Lollipop" Notation To Indicate That A Class Realizes an Interface
4. Define Interfaces Separately From Your Classes
5. Do Not Model the Operations and Attributes of an Interface in Your Classes. In Figure 5 you'll notice that the *Shipment* class does not include the attributes or operations defined by the two interfaces that it realizes.
6. Consider an Interface to Be a Contract

4. Relationship Guidelines

For ease of discussion the term relationships shall include all UML concepts such as associations, aggregation, composition, dependencies, inheritance, and realizations - in other words, if it's a line on a UML class diagram we'll consider it a relationship.

Figure 6. Shipping an order.

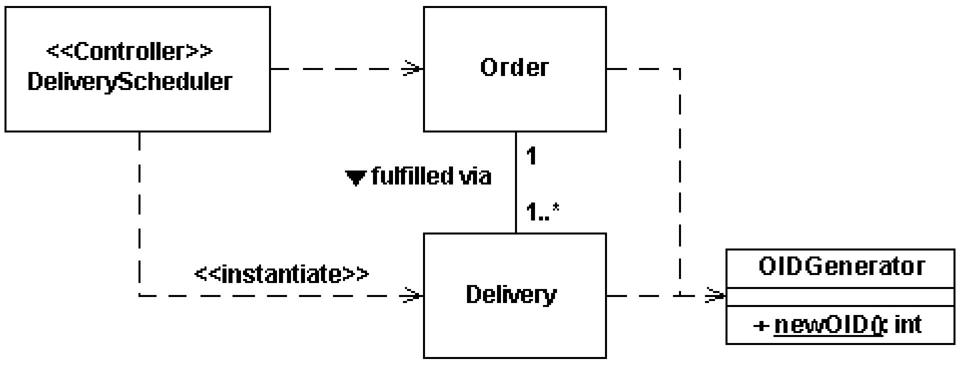


Figure 7. Modeling an order.

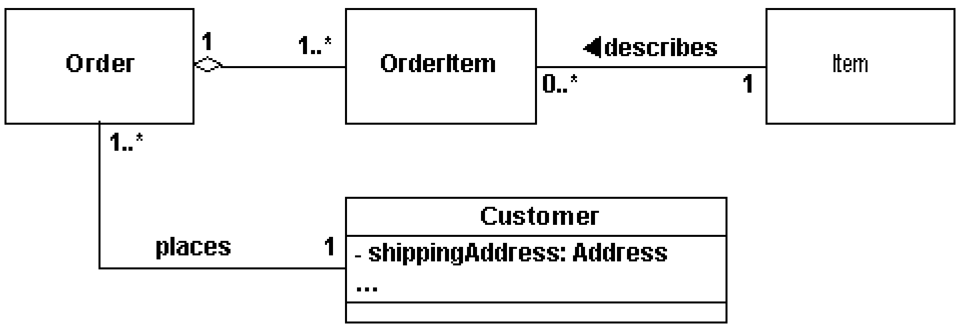


Figure 8. Professors and seminars.

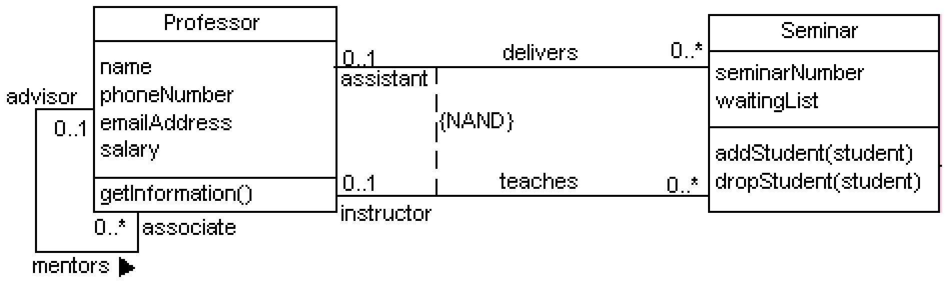
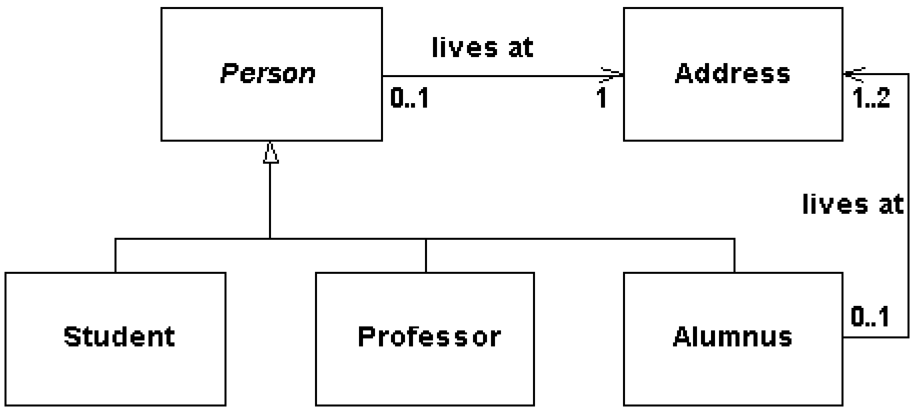


Figure 9. Modeling people at a university.



1. Model Relationships Horizontally
2. Collaboration Indicates Need for a Relationship
3. Model a Dependency When The Relationship is Transitory
4. Depict Similar Relationships Involving A Common Class As A Tree. In Figure 6 you see that both *Delivery* and *Order* have a dependency on *OIDGenerator*. Note how the two dependencies are drawn in combination in "tree configuration", instead of as two separate lines, to reduce clutter in the diagram.
5. Always Indicate the Multiplicity
6. Avoid a Multiplicity of "*"
7. Replace Relationships By Indicating Attribute Types. In Figure 7 you see that Customer has a *shippingAddress* attribute of type *Address* - part of the scaffolding code to maintain the association between customer objects and address objects.
8. Do Not Model Implied Relationships
9. Do Not Model Every Single Dependency
10. Center Names on Associations
11. Write Concise Association Names In Active Voice
12. Indicate Directionality To Clarify An Association Name
13. Name Unidirectional Associations In The Same Direction
14. Word Association Names Left-To-Right
15. Indicate Role Names When Multiple Associations Between Two Classes Exist
16. Indicate Role Names on Recursive Associations
17. Make Associations Bi-Directional Only When Collaboration Occurs In Both Directions. The *lives at* association of Figure 9 is uni-directional.
18. Redraw Inherited Associations Only When Something Changes
19. Question Multiplicities Involving Minimums And Maximums

6. Inheritance Guidelines

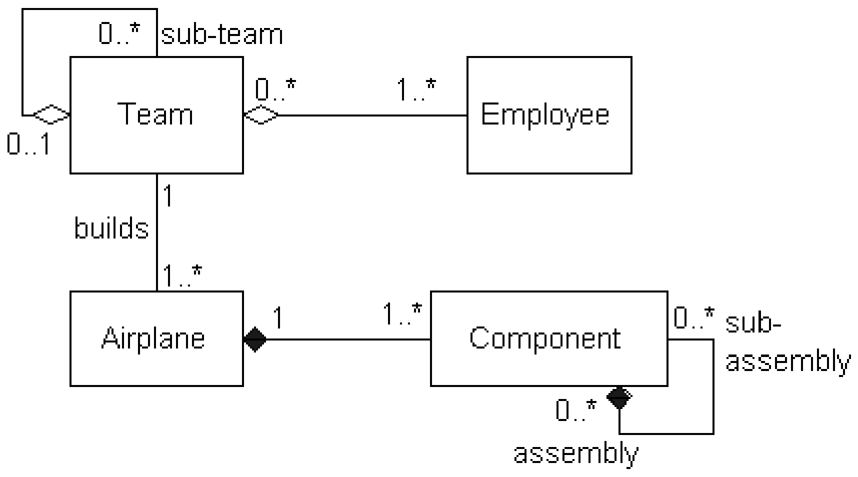
Inheritance models "is a" and "is like" relationships, enabling you to easily reuse existing data and code. When "A" inherits from "B" we say that "A" is the subclass of "B" and that "B" is the superclass of "A."Furthermore, we say that we have "pure inheritance" when "A" inherits all of the attributes and methods of "B." The UML modeling notation for inheritance is a line with a closed arrowhead pointing from the subclass to the superclass.

1. Apply the Sentence Rule For Inheritance
2. Place Subclasses Below Superclasses
3. Beware of Data-Based Inheritance
4. A Subclass Should Inherit Everything

7. Aggregation and Composition Guidelines

Sometimes an object is made up of other objects. For example, an airplane is made up of a fuselage, wings, engines, landing gear, flaps, and so on.A delivery shipment contains one or more packages. A team consists of two or more employees. These are all examples of the concept of aggregation, which represents "is part of" relationships.An engine is part of a plane, a package is part of a shipment, and an employee is part of a team.Aggregation is a specialization of association, specifying a whole-part relationship between two objects.Composition is a stronger form of aggregation where the whole and parts have coincident lifetimes, and it is very common for the whole to manage the lifecycle of its parts. From a stylistic point of view, because aggregation and composition are both specializations of association the guidelines for associations apply.

Figure 10. Examples of aggregation and composition.



1. Apply the Sentence Rule for Aggregation
2. You Should Be Interested In Both The Whole And The Part
3. Depict the Whole to the Left of the Part
4. Apply Composition to Aggregates of Physical Items
5. Apply Composition When the Parts Share The Persistence Lifecycle With the Whole
6. Don't Worry About Getting the Diamonds Right