MODULE

# UML Class Diagram Syntax

*Monash Software Engineering staff*
*Updated 9 August 2019*

## What is a *class diagram*?

A *class diagram* is a diagram that shows the structure of all or part of an object-oriented program. It shows you:

- which classes are present,
- what their names are, and
- how these classes relate to one another.

A class diagram may also show:

- *attributes* or *instance variables*,
- *class variables* or *static variables*, and
- the *methods* or *functions* available in each class.

### How are class diagrams used?

Class diagrams are useful whenever you need to think about, discuss, or explain the structure of a system. This can happen when you're designing a new system, but it can also happen when you're analysing the conceptual structure of an existing system, or trying to understand the features of the domain you're working in.

If they've been done properly, it's easy to understand a class diagram. This means that you can show them to clients as well as colleagues – and that's useful if you're using the diagram to help your team understand a client's business domain and you need them to check that your understanding is correct. Business analysis, system architects, and software developers all use class diagrams as part of their job.

## Classes

In UML, classes are represented as a box. Class boxes must contain the name of the class; optionally, they may also contain other information about the class. We'll look at that later.

## What sort of things can be classes?

Classes represent things that you might want to model. These might be physical objects or abstract concepts. Here's some examples:

- **Physical objects:** Aeroplane
- **Specifications:** Product specification
- **Places:** Shop
- **Transactions:** Sale
- **Transaction Line Items:** Sale line item
- **Roles of people:** Cashier
- **Containers:** Bin
- **Things in a container:** Item
- **Other computer systems:** Credit card authorisation system
- **Abstract Qualities:** Hunger
- **Organisations –** Sales Department
- **Events:** Meeting, Flight
- **Rules and policies:** Refund policy
- **Catalogues:** Product catalogue
- **Records:** Receipt, Ledger
- **Financial instruments:** Line of credit
- **Manuals:** Repair manual

Typically, classes represent *things* rather than processes, so their names are nouns or noun phrases rather than verbs. If you find yourself designing a system in which a class has a name that is a verb or verb phrase, such as `CalculateSalesTax`, you should pause and consider: what is it that should perform this action? What kind of entity in your system should be capable of responding to such a message? In this case, perhaps there should be a `SalesTax` class that has a `calculate()` method.

If you're coming to object-oriented analysis and design from a background in structured analysis and design, you might be used to thinking in terms of functionality and process rather than classes, objects, and structure. If this is the case you might need to make a conscious effort to expand your thinking to encompass this new style.

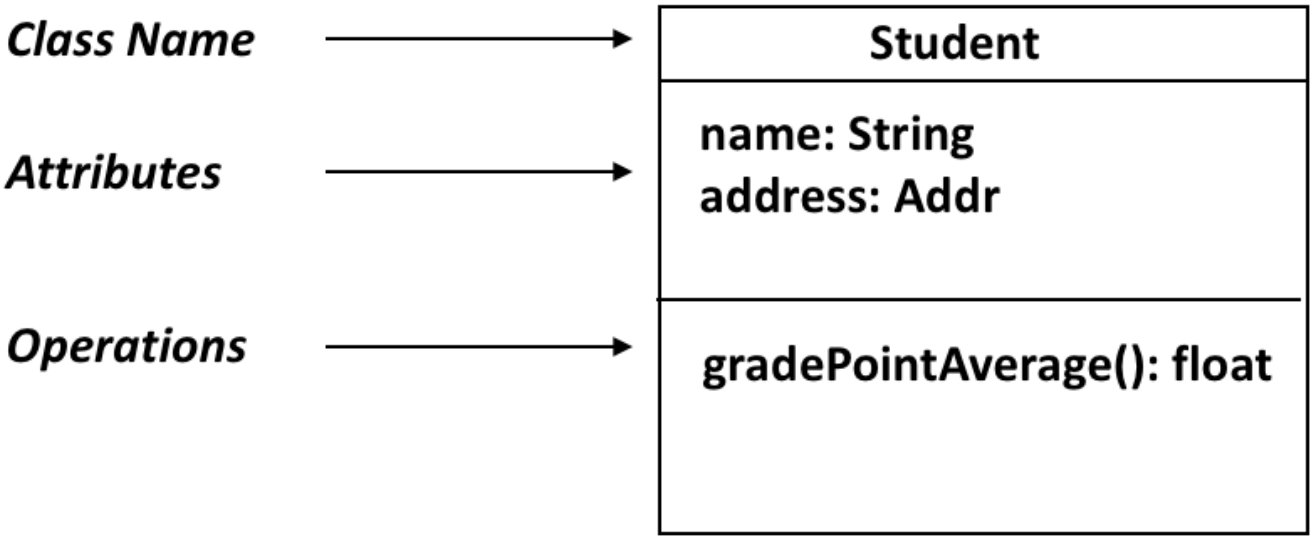
## Attributes and operations

Not all class diagrams will show the attributes and operations of all classes. If you're using UML for analysis, or if you've only just begun to design your system, you won't have decided yet what attributes and operations will be present. If you're using UML notation to help you make an architectural decision, or as a basis for a

discussion with a colleague about the best way to approach a tricky design problem, you should only put as much detail into your diagram as you need in order to support your goal.

UML *can* be used to produce detailed design documentation that could be used as a basis for implementation, or to document a codebase that already exists, but software architects and developers use it for many other purposes too. When you're thinking about how much detail you need to include in a diagram, or indeed in any other piece of documentation, you should always begin by considering how readers are going to use that documentation.

If you do need to show the class's attributes or operations (methods, functions, or features), you can put this information inside the class box.




Name at the top, then attributes, then operations.

Attributes represent the data that each object in the class will be keeping track of. Attributes answer the question, "**what does this class know about?**"

Operations represent the functionality that each object in the class can perform – the responsibilities that it has, and the messages that it knows how to respond to. Operations answer the question, "**what can this class do?**"

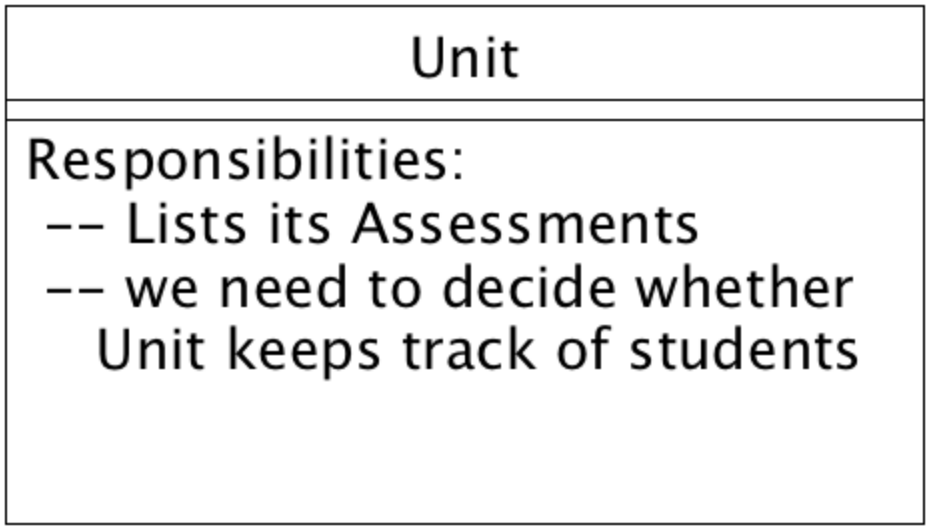
Note that the types of the attributes and the return types of the methods, if you specify them, go after the names and are separated from them by a colon. If an operation has parameters, you should list the types of each parameter inside the round brackets. You don't have to show the names of the parameters, but if you do, you should format them in the same way as the attributes:

```
operationName(parameterName1 : Type1, parameterName2 : Type2) : ReturnType
```

Attributes and operations may also have their *visibility* shown. If the name of the operation has a minus sign (-) in front of it, it is private and can only be accessed from within the class. If it has a plus sign (+), it is public and can be accessed from other parts of the system.

Other kinds of visibilities are also used, especially for detailed designs that are intended to be implemented in a programming language, but the visibilities available to you, and the meanings of these visibilities, will depend on the language of implementation. Both Java and C++ have a *protected* visibility (denoted by a hash mark, #). In C++ that restricts visibility to the class and its subclasses only, but protected attributes in Java are also accessible to other classes in the same package. At the same time, Java has a *package* visibility (represented by a tilde, ~) that C++ lacks entirely.

If you're at a stage of design where you're thinking about what each class is going to need to be able to do, but you haven't yet figured out how to break that functionality down into methods, one possibility is to write a description of the class's responsibilities in English (or another natural language) instead of writing down method names and return types. This isn't legal UML, but it's a very handy technique when you're trying to design a complicated system.



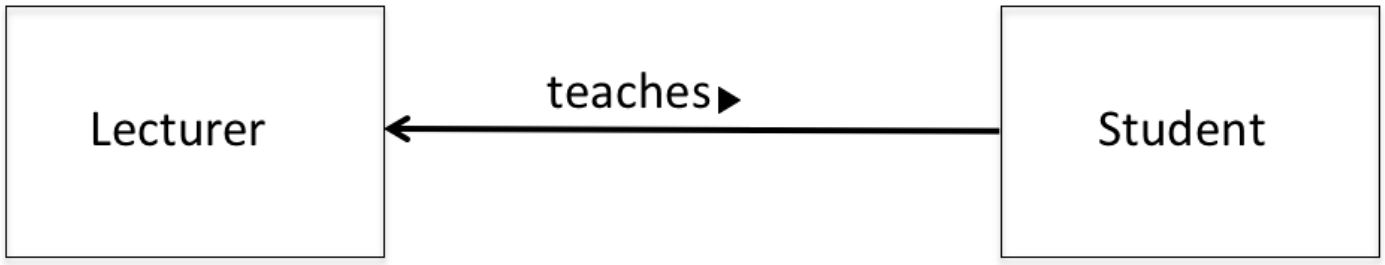

## Associations

Associations represent the *relationships* between classes. They are represented as a line connecting the two related classes, and at their simplest, that's all they are.

If the association has a navigability (i.e. an arrow on one end) it indicates that one class knows about the other but not vice versa. It's very handy to introduce navigabilities into your diagram as early as possible. If you're doing analysis, it will help you understand the responsibilities of each class and how they interact,

and if you're doing design it'll help you think about the dependencies between your classes and the strength of their coupling.

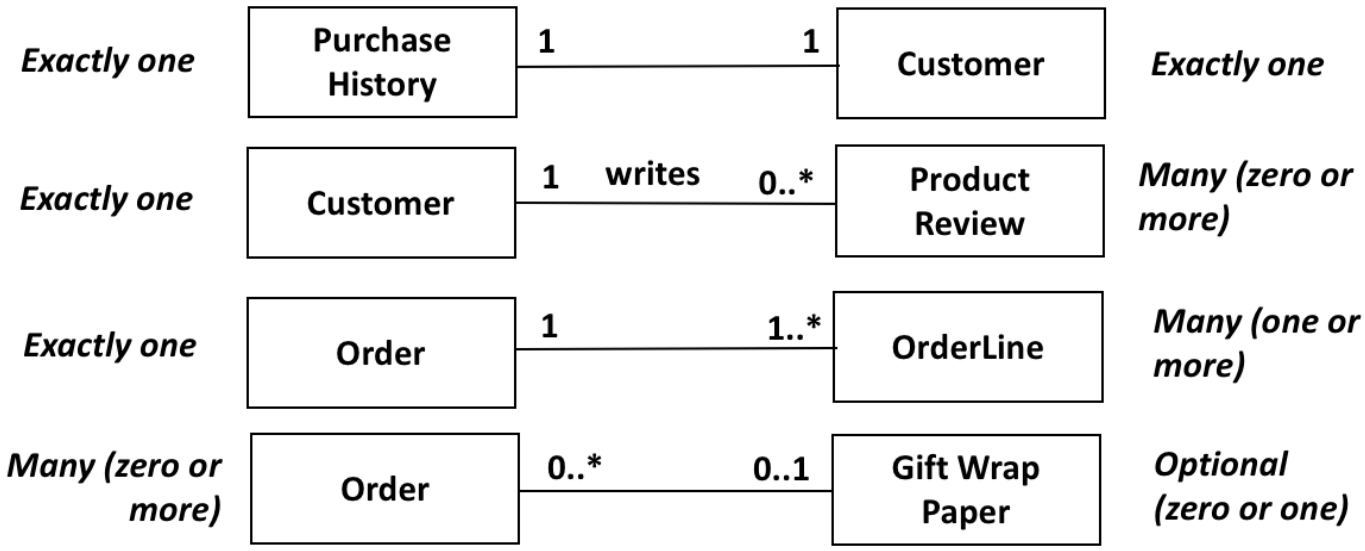
You can give an association a name if it helps you understand the nature of the relationship. This will require some thought, though – something like ***has*** is not a good name! It'll help you more if you use something more descriptive.




If present, a small triangle next to the name shows you which direction to read the label. In this diagram, Students know about their Lecturer (i.e. each Student is keeping track of its associated Lecturer) but Lecturers don't know their Students. The little arrow next to the word ***teaches*** shows you that Lecturer teaches Student, not the other way around. Getting these notations the wrong way around is a common mistake.

We distinguish between associations and dependencies because dependencies are a weaker relationship. In code, an association is usually implemented as an attribute: if A has an association arrow pointing at B, then A contains an attribute of type B, or a collection of these. This is a strong relationship because *any* method in A can potentially access this attribute, and therefore need to be changed if B's public interface changes. That means that we can use the class diagram to get a rough estimate of how expensive it's likely to be to make different kinds of changes.

Each end of an association can have also show its *multiplicity*. This indicates how many objects of each class are involved in the association. For example:



You should steer clear of using variables in multiplicities, e.g. 0..N rather than 0..* – it's not legal UML, and it can be confusing unless you make it very clear what those variables mean.

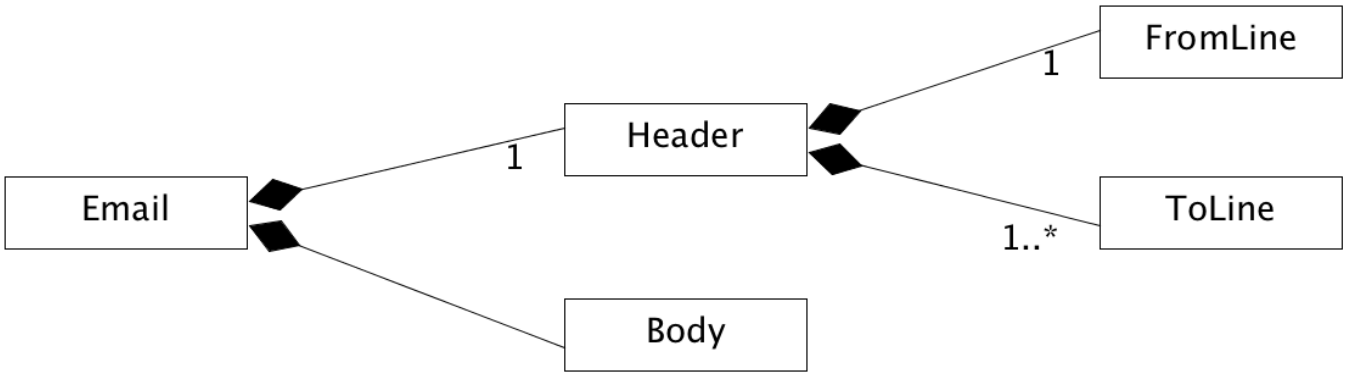
## Aggregate associations

UML 2.4.1 defines two kinds of aggregate association: *composite aggregation* and *shared aggregation*. Composite aggregation is shown as a filled (black) diamond and shared aggregation is shown as an unfilled (white) diamond.

The UML 2.4.1 standard says that composite entities can't share members but aggregate entities can. That is, if an object is part of one composition, it can't be part of another. For the time being, we'll go with this definition – but as we'll see later, there's some controversy around it.

**Composite aggregation** often appears in software systems, because many composite objects appear in real life: a dog is a composite of a head, a body, a tail and four legs; an email is composed of a header and a text message; the header is composed of a From: line, a To: line, etc.

Composite aggregation is the kind of aggregation that exists *between a whole and its parts.*




The three most important characteristics of composite aggregation as opposed to shared aggregation are:

- the composite object does not exist without its components,
- at any time, each component may be part of only one composite, and
- component objects are *likely* to be of mixed types (although this isn't *always* the case.)

**Shared aggregation** is also a familiar concept from real life: a forest is an aggregate of trees, and a flock is an aggregate of sheep. It's so common that the word "aggregation", by itself, usually refers to shared aggregation.

An aggregation is the kind of association that exists *between a group and its members.*


The three most important characteristics of shared aggregation are:

- the aggregate object may potentially exist without its constituent objects (although not necessarily in a useful state),
- at any time, each object may be a constituent of more than one aggregate (e.g. a person may belong to several clubs), and
- constituent objects are typically of the same class (but, again, that's not always the case).

### Controversy

Unfortunately, there is disagreement about the exact meanings of these terms. For example, Coad and Yourdon state that an organisation *is* an aggregation of its employees, while Rumbaugh says that an organisation *is not* an aggregation of its employees.

The experts do not agree on how to tell these kinds of aggregation apart. The UML standard says that shareability is the key qualification but the standard is only useful insofar as people follow it – and most people have not read the UML standard.

This makes it hard to tell what kind of aggregation to use. If you're in doubt, remember that the purpose of *any* documentation, including UML class diagrams, is **to communicate with somebody**. Notation is only useful insofar as it serves this purpose. Remember that both composite and shared aggregation are just different kinds of association – an association line will usually be sufficient for any informal or intermediate class diagram.

## Association classes

Sometimes, you'll find you need to keep track of information about the association itself. For example, in a system that keeps track of library books, the *borrows* relationship between a borrower and a book will have a due date that the system needs to remember. Association class give you a way to represent these kinds of information and allow you to add attributes, operations and other features to associations.

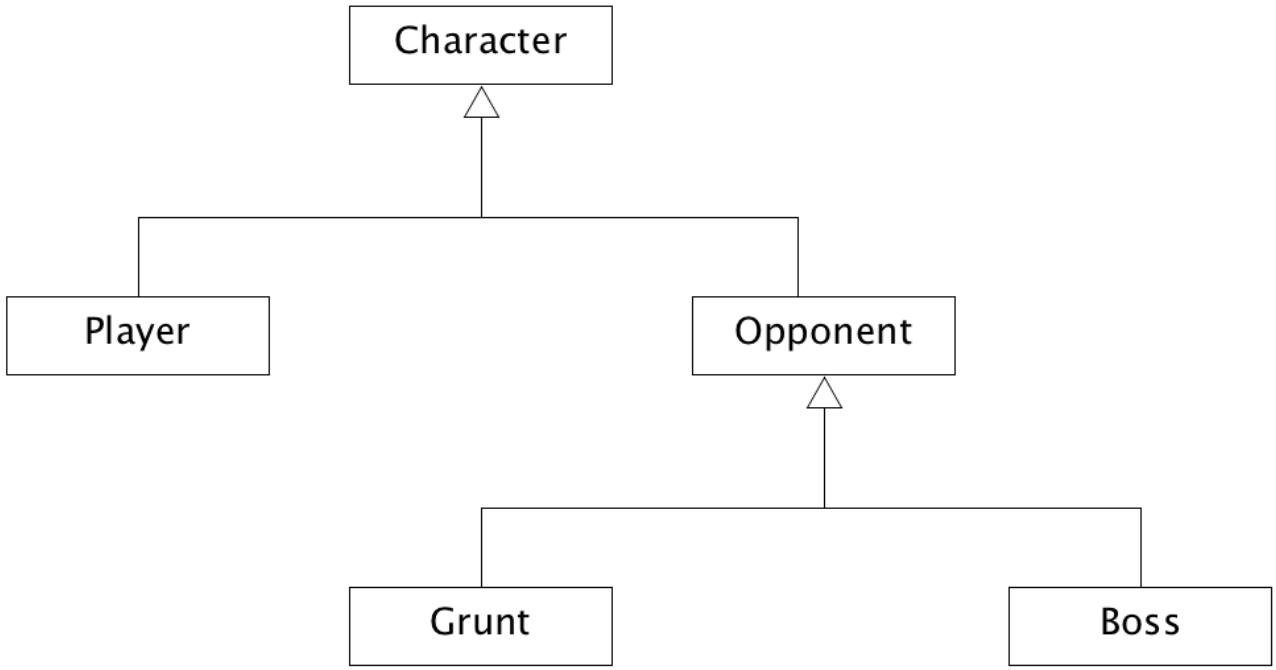
There can be many such classes in your design, and there can be many instances of each class (that is, in our library example, there can be many borrowings of books.) However, there can only be one instance of the

association between any two participating objects – a borrower can't borrow a book if they already have it out on loan.

## Generalisation

Generalisation in UML is used to represent the same kinds of relationship as inheritance in object-oriented programming languages.

We say that a particular class is a *subtype* (or *subclass*) of another class if all instances of the first class are also instances of the second class. For example, in an on-line store such as Amazon, `Book` is likely to be a subclass of `Product`. That means that all the attributes and operations that a `Product` has, such as `price` and `addToCart()`, will also be part of `Book`.




At the implementation level, generalisation is *usually* delivered using inheritance, but you can also use other mechanisms such as delegation. That means that you don't have to avoid generalisations in your analysis or early design, even if you're not using an OO language. You can decide on a mechanism for implementation later, once you're sure you've gotten the model figured out.

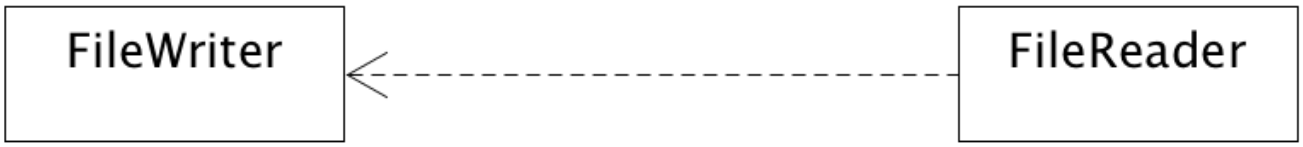
## Dependencies

An association between two classes implies that there's an ongoing relationship between instances of those classes. In code, associations are usually implemented as attributes. But sometimes your classes will have an impact on one another, even if there's no ongoing association relationship between them.

A *dependency* is a relationship which indicates that a change in specification of one thing may affect another

thing that uses it, but not necessarily the reverse. You should use dependency when you want to show one thing using another, but there isn't an association between them.

In UML, a dependency is shown as a dotted line. There's almost always a navigability arrow on one end.



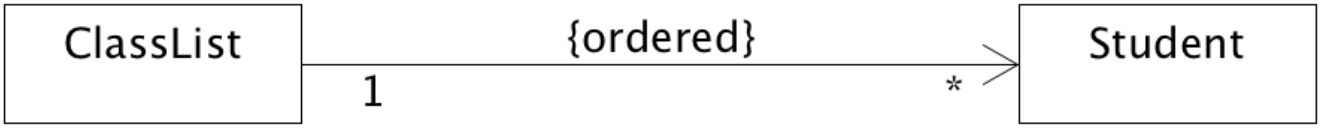

Dependencies can arise because a class uses another *explicitly*, for example as a local variable or parameter, or *implicitly*, such as the return type of a call to an object of another class. Explicit dependencies are easy to spot in code because you'll need to declare the types of those parameters or variables, but implicit dependencies can be hard to track down. For example, consider a method that reads information from a file. It depends on the function (or system) that writes that information – if the writer's file format is changed, then the reader will need to be changed to accommodate it. That's a relationship that can affect maintainability, so we definitely care about it, but it's not going to be easy to spot from analysing the code.

## Constraints

The elements of a class diagram (associations, attributes and generalisation, etc.) are effectively placing constraints on the system. For example, an association can indicate that an order must have a customer by having a multiplicity of 1 at the customer end. But sometimes, you'll need to specify constraints on the system that can't be covered by these structural elements.

In UML, you can simply write these constraints in natural language and place them inside curly brackets ({ }). At the conceptual level, it's best if you use simple statements, such as `{ordered}` to show that a particular collection must be ordered in some way.

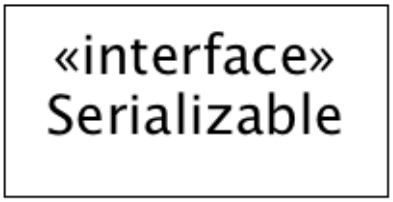
You can apply these constraints to any part of the diagram: attributes, associations, classes, etc.




## Stereotypes

Stereotypes are a general extension mechanism for the UML. They consist of a descriptive word or

phrase between *guillemets*: « »




A stereotype tells you something interesting about a class, for example that it's a Java interface. Another common stereotype is «`abstract`», which means that the class cannot be instantiated. Recent versions of UML actually support an "official" notation for abstract classes – you put the class name in *italics*. However, that doesn't work when you're hand-drawing a class diagram, so using a stereotype will easier then.

---