Class model notation

Class model notation

Let's start by describing the notation used to specify the structure of a class.

Classes

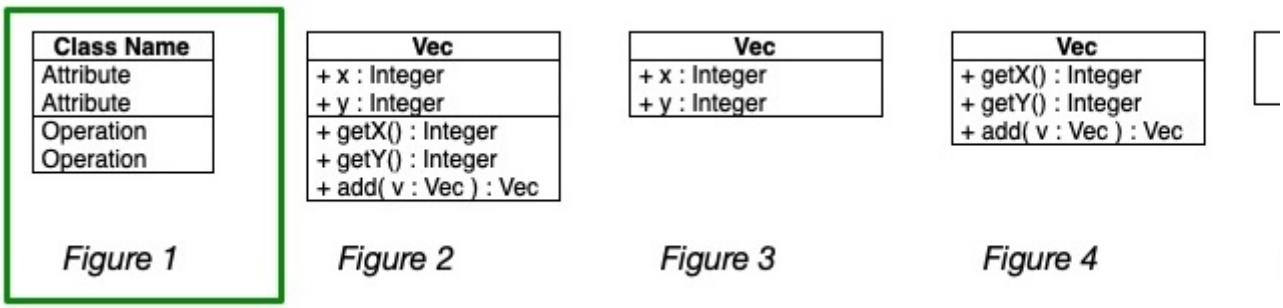


Figure 1, surrounded by a green box, shows the basic structure for a class. The general form of a class has three boxes:

1. The first box contains the **class name**. Figures 2 to 5 show examples of a class diagram using our `Vec` class from previous lecture material.
    - The class name must be unique among all of the classes in that particular scope; otherwise, it is qualified with a scope operator e.g. *Package-name*::*Class-name*, as in `Banking::CheckingAccount`.
    - The class name must be capitalized, centered or left-justified, and in bold. Note that, in general, class names are not pluralized unless the class is intended to be a container for multiple objects of the type.
    - The class name may be qualified by an optional stereotype keyword, centered and in the regular typeface, placed above the class name, and within guillemets "«»" (guillemets are

the class name, and within guillemets "≪" (guillemets are special Unicode single characters, and are **not** made up of << and >> but rather Unicode 226A and 226B) and/or an optional stereotype icon (in the upper-right corner). For example:

Some stereotypes are defined by the UML; others may be defined by the user. *We do not expect you to know or use them in this course.*

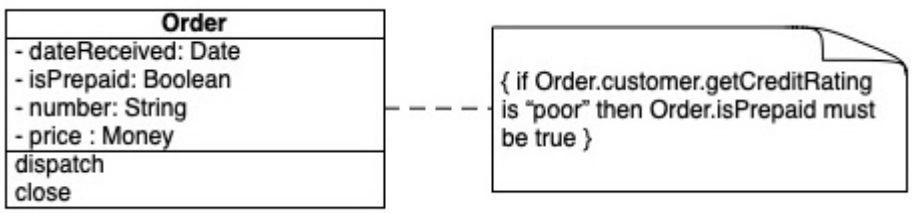
2. The second box is optional, and contains the class **attributes**.
3. The third box is also optional, and contains the class **operations**.

Either or both of the attributes or operation boxes may be omitted, in which case the separation line for the missing box isn't drawn. The class name may never be omitted. Examples of this are shown in figures 3 through 5.

Comments

A **comment** can be added to the class diagram by joining a rectangle with a bent upper-right corner ("dog-ear") to the item being annotated by a dashed line. The comment is a text string but has no effect on the model, though it may describe a constraint on the annotated item.

For example:



Attributes

Attributes are left-justified and written in the regular typeface. They are generally shown when needed, though a full description must be provided at least once. They describe the information held by an instance of the class, and may be replaced by **association ends**.

The general syntax for an attribute is:

```
«stereotype» visibility / name : type multiplicity initial-valu
```

where every argument except the name is optional. The `name` generally starts with a lower-case letter.

visibility	Describes the visibility of the attribute as a punctuation mark, though can be instead described in the property string. There are four markers: • + public • – private • # protected • ~ package
type	A string that describes the attribute's type. Usually written to be as language-independent as possible e.g. Boolean rather than the C++ `bool`, though in this course we may sometimes use C++-specific syntax since we're not using the UML formally. If the type is omitted, so is the preceeding colon.
/	Indicates that this attributed is derived from a parent class.
multiplicity	Describes the multiplicity constraints on the attribute i.e. how many of these values will be held. It is enclosed in square brackets (`[ ]`). It is usually omitted if the value is exactly 1 since that's the commonest case. It will consist of either a string specifying the exact number (e.g. `[ 3 ]`), a string specifying a precise range (e.g. `[ 3..10 ]` or `[ 0..1 ]`), an unknown number (e.g. `[ * ]`), or a range with a specified lower bound but no known upper bound (e.g. `[ 2..* ]`). Note that `[ 0..1 ]` implicitly represents a pointer in C/C++. If the upper bound is greater than 1, you can specify ordering and uniqueness on the values in the property string
initial-value	Specifies the default initial value as an equal sign followed by the value e.g. = `0`
property	List of comma-separated strings surrounded by `{}`. Defaults to `{changeable}`, but can be specified as `{readOnly}` to indicate that it's a constant. Can be used to annotate multiplicity e.g. `ordered` (elements in collection follow an order), `unordered` (items in collection are not in order), `bag` (collection of elements that may have multiple copies of elements), `seq` (elements of the set form an ordered sequence), `set` (collection of unique elements) and list (ordered variable-length collection i.e. ordered bag).

If the attribute is `static`, the name and type strings are underlined.

Some examples of attribute declarations are:

```
- colours: Saturation [3]
# points : Point [2..*] {ordered,set}
size : Area = (100, 100)
+ name : String [0..1]
+ name : String {readOnly}
```

Operations

Operations are left-justified and written in the regular typeface. They are generally shown when needed, though a full description must be provided at least once.

In this course, since all classes will have constructors, destructors, accessors (getters), and mutators (setters), we will not generally specify them in our UML class diagram since we'll assume their presence. Specifying them, however, clearly tells the implementer what is expected in terms of parameters and return-types, and is thus a useful element when drawing up the model.

The general syntax for an operation is:

```
«stereotype» visibility name ( parameter-list ) : return-type {
```

where every argument except the name and parameter list is optional. The `name` generally starts with a lower-case letter.

visibility	Same as for the attribute.
return-type	A string containing a comma-separated list of names that describes the operation's return type. (Some languages allow multiple return values.) If the return type is omitted, so is the preceeding colon and the return type is equivalent to the C/C++ `void`.
parameter-list	Comma-separated list of parameters, enclosed in parentheses. (We will often use C++ syntax for this information rather than the classic UML notation.) Each parameter is of the form: *direction* `name : type` *multiplicity = default-value* • *direction*: specifies direction of information flow and is optional ◦ `in`: input parameter, passed by value (default)

	◦ `out`: output parameter with no input value; final value is available to caller ◦ `inout`: input parameter that may be modified and whose final value is available to the caller ◦ `return`: return value of a call (equivalent to out but available for use inline) • If the *default value* is omitted, so is the preceeding equal sign.
`property`	List of comma-separated strings surrounded by `{}`. We will rarely use this element, though the ability to specify `abstract`/`virtual` or `static` may be useful if your UML drawing software doesn't let you italicize or underline. • A list of raised exceptions may be shown by specifying a comma-separated list of them after the keyword `exception`. • Instead of italicizing an abstract operation, the keyword `abstract` may be placed in the property string. • Instead of underlining a static operation, the keyword `static` may be placed in the property string. • The property `isQuery=false` indicates that this operation doesn't change the state of the object. It defaults to a value of `false`. A value of `true` doesn't guarantee that the state changes only that it may change