

Chapter 42

Water Pics and Sketch: Unifying Diagrams and Code – UML

IN THIS CHAPTER

- ◆ Using Water Sketch for quick diagrams
- ◆ Visualizing objects in multiple views
- ◆ Viewing primitive objects, field types, and classes
- ◆ Viewing relationships between objects
- ◆ Viewing calls, logic and control flow
- ◆ Viewing a state machine
- ◆ Comparing Water Pics and UML

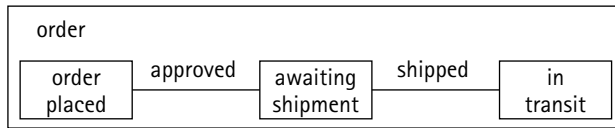
WATER PICS IS A GRAPHICAL NOTATION LANGUAGE to describe the structure of Water programs in a pictorial representation. Water Pics has a standard set of symbols and conventions to help improve the communication among people building software systems.

Many existing notation systems, such as UML, have multiple types of diagrams and notations. Water Pics is a simplified notation that provides a clean mapping to every Water concept. Some people understand things best through pictures, while others prefer code. Water Pics provides the best of both because code and pictures can be mixed freely: It lets you see code with pictures, as well as see pictures with code. Water Pics can also be used to create sketches. A Water Sketch is a rough picture used to quickly describe some requirement or design. A businessperson might draw a rough sketch, which is turned into a precise picture by adding details. Adding details to a Water Sketch transforms it into a picture, but the original structure of the sketch is maintained.

Refer to the end of the chapter for a high-level comparison between UML and Water Pics.

Using Water Sketch for Business

Water Sketch was designed for business people to sketch business processes and applications. A sketch can be very quick because it conveys the basic structure of systems without precision. A business analyst or developer can add precision to a sketch while keeping the original look.



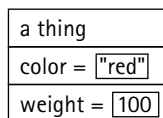
A Water Sketch consists of only labeled lines and boxes – no other notation is required. A labeled box represents some kind of object. The object could be a class, instance, or method, but the distinction may not be important in the sketch. A line represents some type of relationship between boxes. It could represent data, flow, views, or a state transition.

A Water Sketch gets transformed into a detailed Water Pic as ambiguity is removed and precision is added.

Visualizing Objects

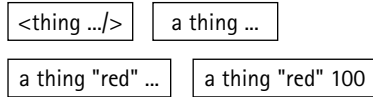
A box represents an object, and the label in the upper-left corner of the box is the name of the object. The following code and picture show an instance of `thing` with two fields: `color` and `weight`.

```
<thing color="red" weight=100/>
```



Each field of an object can be shown as a horizontal row. The row has a field key and field value. The field key is italicized and shown to the left of the value. An equal symbol appears after the key. The value is another object, therefore the object that is the value of the field appears as a box.

The previous example shows a detailed view of an object. An object can be shown in a smaller view by hiding one or more fields.



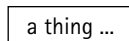
The three dots indicate that information is missing.

The angle brackets indicate that the object is an instance. The call name is `thing`, which means that the object is an instance of `thing`.

The box around the object is optional if you use angle brackets to delimit the object.

`<thing .../>`

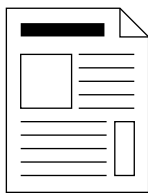
An instance can also be represented without the angle brackets by prefixing a class name with the word *a*. The following picture, for example, represents an instance of `thing`.



The smallest view of an object is represented by a tiny, solid, black square. An optional label can be put to the right of the square.

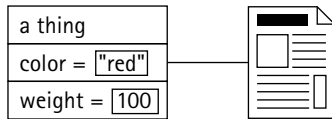


Some objects appear to a user on a screen. A box with a dog-eared, upper-right corner indicates that object is shown in a display view. For example, a page object might have a reduced-size image of a Web page in its box.

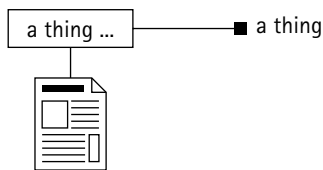


Object views and identity

An object can be viewed in multiple ways in a single Water Pic. For example, an object can appear in a display view and a detail view. The next two boxes represent different views of the same object. A line attached to the outer border of each view indicates that the views are showing the same object.



The same notation can be used between any number of views.



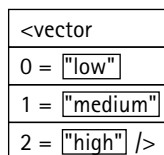
This notation is very useful for scaling a Water Pic. If a larger view does not fit in a particular area, a small view can be substituted while keeping the identical meaning.

A vector

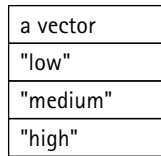
A `vector` is an object with integer keys.

```
<vector "low" "medium" "high"/>
```

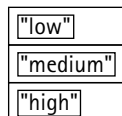
The preceding line of code describes the following view:



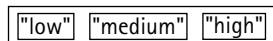
The field keys can be optionally omitted when showing a vector.



The box label is also optional.

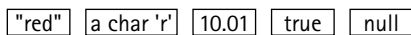


The values can be either stacked or adjacent.

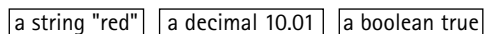


Primitive objects

Primitive values are also shown as boxes. The following figure shows a string "red", a decimal number 10.01, a Boolean true, a character <char 'r'/>, and the null object.



For primitive values, the class of the object is optional, but could be supplied.



The equivalent code view is as follows:

```
<string>red</> <char 'r'/> <decimal 10.01/> boolean.true null
```

Understanding Classes and Fields

A class is an object and is represented by a box. The label for the box is the name or path of the class. The following figure shows a `person` class:

`person`

Here are two views of the same `integer` class:

`number.integer` — `integer`

The complete picture for a class including inheritance and methods is shown later in the chapter.

Fields and relationships

Two objects are related if an object with a field has another object as the field value.

a person
name = "Mike"
father = a person "Steve"

Here, an instance of `person` is shown within the `father` field.

Moving the compact view of the field value outside of the container and leaving behind a tiny view of the object can represent the same relationship. The two views show the identical object, and therefore are linked with a line.

a person
name = "Mike"
father = ■

— a person "Steve"

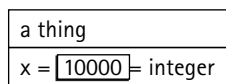
If the box is too small to fit the field key, it can be moved outside the box and put on the line that represents the value of the field.

a person "Mike" ■ father = — a person "Steve"

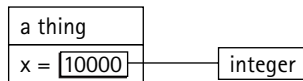
Field type

Every field has a type in addition to a key and value. The field type describes the type of values that are permissible values of the field.

A field type is represented as a box within the field of an object. The object that is the type for the field is shown directly to the right of the box. The following example describes a `thing` with a single field named `x`, holding a value of `10000`, and a field type of `integer`:

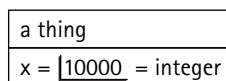


Alternatively, the type can be shown as an object outside the box that is attached to the field type box with a line.

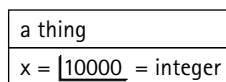


If there is no type specified, the default field type `thing` is used. This means that any `thing` can be the value of the field.

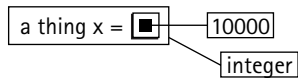
To help tell the difference between the box that represents an object from the box that represents the field type, the field type box has shading that makes it appear indented or inset. Alternatively, the left and bottom sides of the box can be thicker.



The top and right sides of the field type border can also be omitted as well.

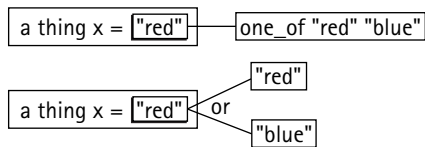


A compact view of an object can show the field type and field value at the same time, because the field type box can contain a small square representing the value.

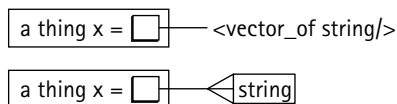


A few field types have optional special notations to improve readability.

If the field type is `one_of`, it can be shown in the standard notation or an alternative notation using `or` or `one of` between separate lines.



The field type of `vector_of` also has an alternative notation using crow's feet.



In Water, the key of a field is also an object. Typically, the key is either a `string` or an `integer`, but it could be any object. Instead of viewing a key as a box, the name of the key is italicized and has an equal symbol after the name. If the key of a field is something other than a `string` or an `integer`, then a box should surround it to make it clear that it is not a regular key.

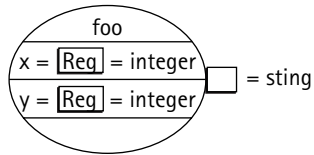
Visualizing Logic and Flow

A method is represented as a circle. The label for the circle is the name or path of the method. By convention, the parameters of the method are on the left, and the return value appears on the right.

```

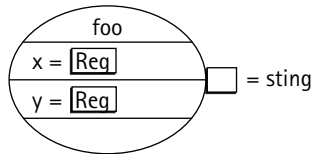
<defmethod foo x=required=integer y=required=integer _return_type=string>
  x.<concat y/>
</defmethod>
  
```

A detailed Water Pic for the Water code is as follows:

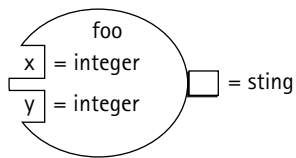


The method's `_return_type` is shown as a box attached to the outside of the method.

By convention, the method's parameters are shown on the left side of the circle.



If the parameter's key does not fit within the circle, it can be put outside the circle.



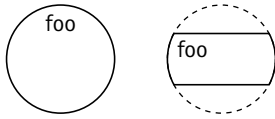
A compact Water Pic is a circle with just the name of the method.



A tiny Water Pic is a solid dot. The name of the method could optionally be shown adjacent to the dot.

- •foo

A circle with the top and bottom chopped off is equivalent to a full circle. This shape is often used for a method because it takes up less space.



Method as a box

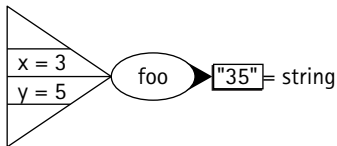
A method is an object, so a method could also be represented as a box where the label indicates it is a method.

a defmethod foo
x = <u>Required</u> = integer
y = <u>Required</u> = integer

A Water method is also a Water contract. The edge of the circle represents the contract because the edge describes the input parameters and the output return type. The interior of the circle represents the implementation of the method. The notation for method implementation is described later in this chapter.

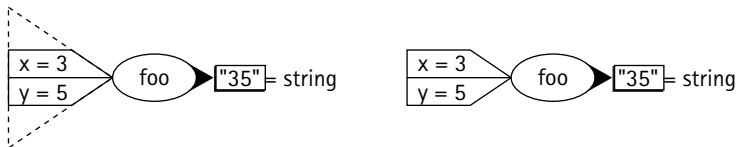
Call

The call to a method or class takes the shape of a triangle. The call arguments are fields of the call. The following Water Pic shows a call to `foo` with two arguments: `x=3` and `y=5`.



The return value of the call can be shown as the output of the method.

A triangle with the top and bottom chopped off is equivalent to a full triangle. This shape takes up much less space than a full triangle.



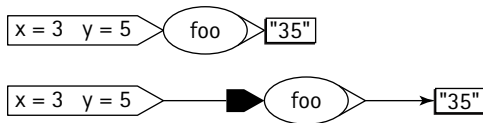
The tiny view of a call is a solid black arrow.



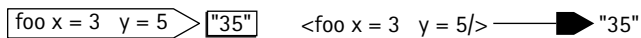
As with all views, the tiny view of a call can be linked to a detailed view with a single line.



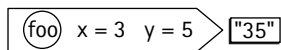
The next two Water Pics are equivalent:



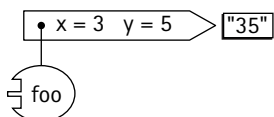
A call can be shown in code view as well. The following two lines have the same meaning:



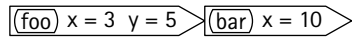
A Water Pic for the call object can be used in place of the call name.



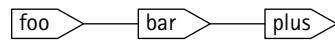
The call object could be shown as a tiny view.



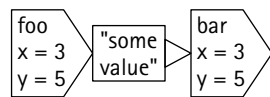
A Water path containing multiple calls can be represented in a series of adjacent calls. The flow of control and data are combined into a single data flow.



A line could optionally connect the calls. For the compact view of a call, the arguments can be hidden.



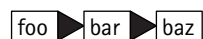
Intermediate values in the path can be represented.



A path of fields can also be represented.



As with all compact views, the full path to the object is optional.

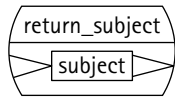


Visualizing an Implementation

The picture of a method's implementation can be shown within the detailed view of a method.

```
<defmethod return_subject>
  _subject
</defmethod>
```

The following Water Pic is the same as the preceding call to `defmethod`:

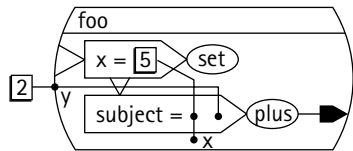


The executional flow through the method starts with a single arrow inside the circle.

The arrow points to some Water expression. In the preceding example, the arrow points to the `subject`, which is a local variable. The arrow out of `subject` points to the edge of the circle, which means the method exists there.

Any local variables are shown as a line attached to the edge of the circle.

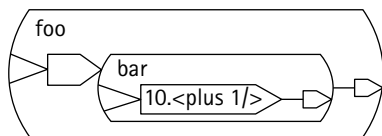
```
<defmethod foo y=2>
  <set x=5/>
  x.<plus y/>
</defmethod>
```



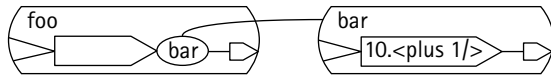
The flow to inside calls can be made:

```
<defmethod foo>
  <bar/>
</defmethod>

<defmethod bar>
  10.<plus 1/>
</defmethod>
```

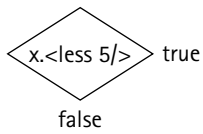


If the method nesting becomes too crowded, the detailed method can always be put outside the method and linked with a line.



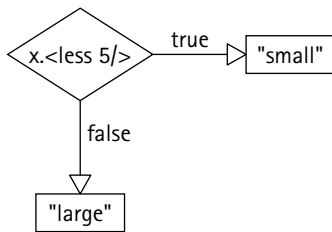
A diamond represents a condition.

`x.<less 5/>`



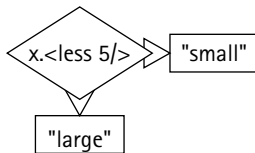
An if call is a diamond with flow arrows.

```
<if> x.<less 5/> "small"
  else "large"
</if>
```

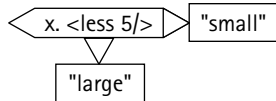


By convention, if the expression does not return `false`, the flow continues to the right. If the expression returns `false`, the flow continues down.

The lines out of the diamond are optional. The `true` and `false` labels are optional as well if the standard convention is followed.



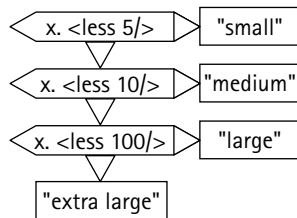
To conserve space, a diamond with the top and bottom cut off means the same thing as a full diamond.



An if statement with multiple conditions is as follows:

```

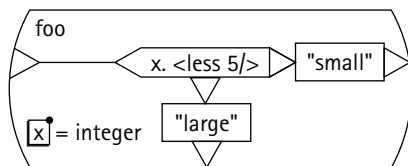
<if> x.<less 5/> "small"
      x.<less 10/> "medium"
      x.<less 100/> "large"
      else "extra large"
</if>
  
```



An if statement within a method looks like the following example and picture:

```

<defmethod foo x=required=integer>
  <if> x.<less 5/> "small"
      else "big"
  </if>
</defmethod>
  
```

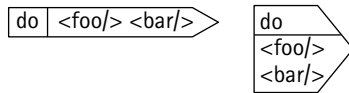


Dotted circle for do and set

A call to `do` for can be viewed in multiple ways. The following is the code view:

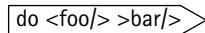
```
<do>
  <foo/>
  <bar/>
</do>
```

Two variations of the Water Pic call notation are shown below:

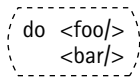


The horizontal line separates the regular arguments from the content argument. The following `do` call does not have a content argument:

```
<do
  <foo/>
  <bar/>
/>
```



A dotted circle can also be used for `do`, `set`, and `for_each`. For every method call, a new local scope is created to execute a call. The local scope is an object containing the local variables in that scope. In Water Pics, a circle with a solid border represents a method and the creation of a new local scope.



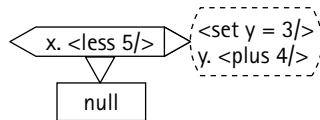
`do` can be omitted. This is often useful when showing a series of calls in the action part of `if`.

```
<if> x.<less 5/>
  <do>
    <set y=3/>
```

```

    y.<plus 4/>
  </do>
</if>

```



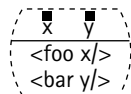
The following method calls, `do`, `for_each`, `try`, and `set`, can be represented with a dotted circle instead of a call symbol. The dotted line represents an extension to the local scope. For example, `value` and `key` are local variables available inside the scope of a `for_each`, but are not available outside the scope of `for_each`. When inside the scope of `for_each`, though, all the local variables are accessible because the call to `for_each` does not create a new local scope; it just extends the local scope for the duration of the call.

A call to `set` is shown in the following code and Water Pics. `set` can be omitted from the picture.

```

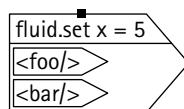
<set x=5 y=2>
  <foo/>
  <bar/>
</set>

```



Fluid variables

Fluid variables are represented by a big dot on the border of the method. Local variables, by contrast, are a small dot attached to the inside of a method. A fluid variable can be seen in any calls that originate within the content of the fluid set.



Showing a Class

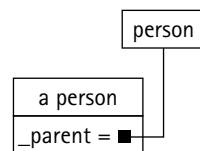
The compact view of a `Water` class is just a box with a label that is the name of a class.

```
<defclass person/>
```

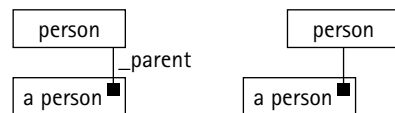


Every object has a system field with key `_parent` that holds the parent class object.

```
<person/>
```

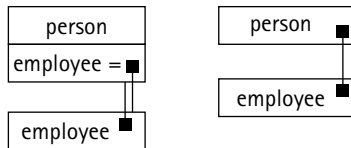


By convention, if there is a dot in the upper-right corner of an object, it represents the `_parent` of the object. The `_parent` label on the line is optional if the class is shown above the instance.



A class may have subclasses. The following shows the code and picture for the class `person` and subclass `employee`:

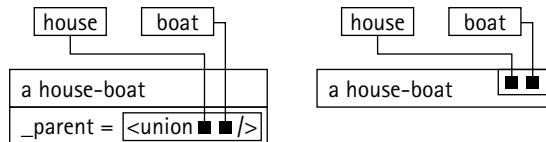
```
<defclass person>
  <defclass employee/>
</defclass>
```



The preceding picture on the left explicitly shows the `employee` field holding an `employee` subclass. The `employee` class has a `_parent` class of `person`. In a class-subclass relationship, the class has a field key that is the same name as the subclass. When this relationship exists, you can use the shorthand notation shown in the preceding picture on the right.

Water's multiple inheritance uses the `union` object to express an ordered union. The picture on the left shows an explicit `_parent` field with a union object containing a `house` and `boat` class. The picture on the right shows the shorthand notation for multiple inheritance.

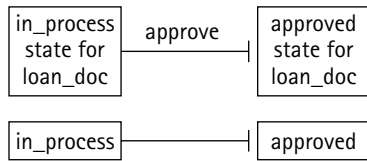
```
<<union house boat/>>
```



Viewing a State Machine

A state machine is a set of states and transitions between states. In Water, a state can be represented by a `view`. The views are on a single class. All instances that flow through the state machine inherit from that class. At any point, an instance is in a single state. An instance can only respond to specific events when in a particular state. Methods can be used to represent possible events. A view makes only a specific set of methods available in a state. When those events/methods are called, the instance might transition to other states.

Water Pics represents a state as a box. If the class of the state is `loan_doc`, and the state is named `approved`, then the label on the box is `approved state for loan_doc`.



Every state transition path has a starting state and an ending state. The transition path is shown as a line between two states. The line's label is the name of a transition method. The state transition line ends with a T, because there is no guarantee that the new state will be the ending state of the transition path.

The `init` method for the class should put a new instance at the starting state.

Water Pics and UML

UML is the leading diagram notation for software systems. UML, which supports nine different diagram types, was the result of combining the notational styles from several different notations.

Water Pics started as the diagram or picture notation for Water. Water's multi-role object system is more flexible than traditional class-instance object systems that UML was designed to support. Water Pics was designed to be able to represent everything in a Water program. Water Pics has a single notation for all pictures, but different pictures can play multiple roles. A single picture could combine the information in UML's nine different diagrams.

One or more Water Pics can represent the following UML diagrams:

- ◆ *Class inheritance*: Inheritance hierarchy between classes. Similar to the UML class diagram.
- ◆ *Class relationships*: Relationships between classes of objects.
- ◆ *Class methods*: Methods available to classes of objects.
- ◆ *Instances*: Specific instances and their relationship to other objects. Similar to the UML object diagram.
- ◆ *Component*: High-level resources and groups of classes.
- ◆ *Deployment*: Hardware and network resources and relationships.
- ◆ *Flow*: Dynamic activity and flow between objects and methods. Similar to the UML use cases.
- ◆ *Web flow*: Objects are Web pages and methods are lines between pages.
- ◆ *State machine*: States for a class. Lines are possible transition arcs between states. State machine can be expressed as a picture with nodes and arcs or as a state table.

Each type of diagram has a particular use and Water Pics provides a simple notation that can be used across all the views. Everything in Water has a corresponding representation in Water Pics, and everything in Water Pics has an equivalent code representation.

Summary

This chapter is the first published document on the Water Pics standard. Water Pics covers a wide range of diagram types using a simple, consistent notation. The Water Sketch was designed for capturing the basic form without getting bogged down in the precision of Water Pics. Since Water Pics is a one-to-one mapping to Water code, Water Pics is conceptually executable. Future tools will likely take advantage of this feature.

