Unit 3: Client View of a Component — Values

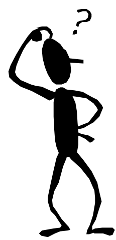
Preview of Coming Attractions

In this unit be sure to look for

- client view
- implementer/maintainer view
- the elements of a client view
- mathematical modeling
- abstract value and abstract state space
- type and subtype
- interface and class
- constraint
- programming language type
- constructor and initial value

What Can It Do for Me?

It may seem obvious, but let's say it anyway: Before a software developer can use a component, the developer must understand the component! Specifically, the developer must understand ***what*** the component can do and ***how*** to use it. Understanding what a component can do and how to use it is called the *client view* of a component.



We did not say that a software developer must, before using a component, understand *how* a component does what it does. And, there is a good reason why not. Imagine what it would be like if everyone who wanted to surf the web first had to read and understand the millions of lines of source code that explain in complete detail *how* a browser does its job. Or imagine what it would be like if everyone who uses e-mail first had to read and understand the millions of lines of source code that explain in complete detail *how* e-mail software does its job. Under those circumstances, cyberspace would be sparsely populated.

Of course, those software engineers who develop and maintain a software component do need to be aware of how the component does its job. Understanding *how* a component

does its job is called the *implementer/maintainer view* of a component. Eventually we'll look at components from the implementer/maintainer view, but for now we are concerned only with the client view.

Still Awake?

3-1. Name two things that you understand how to use and, yet, you do not understand the internal details of how these things work.

3-2. Consider two possibilities. First, people understand all details of how something works before they use it. Second, people figure out what to do to use something before trying to figure out all details of how something works. As a general rule, which of these two possibilities do you think is most prominent in society? Can you think of any reason why the same should not apply to using software components?

What to Tell a Client?

When programmers, acting as clients, use a component (or primitive type) in a piece of software, they declare variables corresponding to the component and then initialize them to some value of the component type. For example, looking back at Code Segment 2-2, we see the four variable declarations

```
SimpleReader input;
SimpleWriter output;
String yourName;
int yourAge;
```

Obviously, the programmer of Code Segment 2-2 must understand how to use the variables `input`, `output`, `yourName`, and `yourAge`. There are three things that must be described so that clients understand ***what*** a component can do and how to use variables of its type:

- *the values a variable can assume*
- *the initial value of a variable or how a variable can be initialized*
- *the operations that can be applied to a variable of a given type*

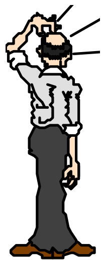
That's it. Together, these three things make up the ***interface*** of a component.

How to Tell a Client?

For primitive types (those types that are built-in to the Java language), the language itself describes what values a variable can assume, what the initial value will be, and what kinds of operations can be used to manipulate a variable of a given type. For instance, a variable of type **`int`** can take an integer value in the range $-2^{31} \ldots 2^{31}-1$, its initial value is undefined in general, and you can apply the usual arithmetic operations, +, –, *, /, etc. However, for components (i.e., non-primitive types), the language itself does not provide this information. We need some other way and place to describe these essential aspects of new components.

Clients are people and when a software component is described to a person, it would be kind of silly to speak in a programming language only. After all, programming languages are intended for communicating with computers, not people! This leaves us with a slight problem: What language should be used to describe components to clients, who are people?

Before giving you our surprising answer to this question, let's make an observation. When describing a component to a client, the description must be *understandable*. If it is



not, the client may not choose to use the component or, worse yet, may use the component incorrectly, possibly leading to software failure and even to loss of life, if the software happens to be safety critical. At the same time, we cannot sacrifice *precision* for understandability. Ambiguity in a description could have the same consequences as misunderstanding.

To address the twin concerns of precision and understandability, we will use the language of mathematics to describe software components from the client perspective. Mathematics is well known for precision and, with a little practice, it will prove to be understandable as well.

The idea of using mathematics to describe something of interest is called *mathematical modeling*, an idea that is centuries old. Simply stated, mathematicians create formal, mathematical descriptions (models) of a phenomenon of interest. Then, the models can be manipulated using mathematical techniques in order to better understand the phenomenon and to make predictions about its behavior. A famous example is $e = mc^2$, an equation that models the relationship between energy and mass. In software development, you have been using the idea of mathematical modeling without even realizing it. For example, consider an expression like `i + j`, where `i` and `j` are variables of type **`int`**. All but the most warped among us think of `i` and `j` as plain old integers, like 82 and 9947, and think of + as being the usual addition of integers. But this is the client perspective, because the real story of *how* the computer is representing integer values (for example, 32-bit twos-complement representation) and adding them through electronic circuitry is quite different.

Still Awake?

3-3. Give two additional examples of mathematical models. The examples need not be from computer science.

3-4. Why do you think scientists and engineers use mathematics to describe or model things that are of interest to them? Explain.

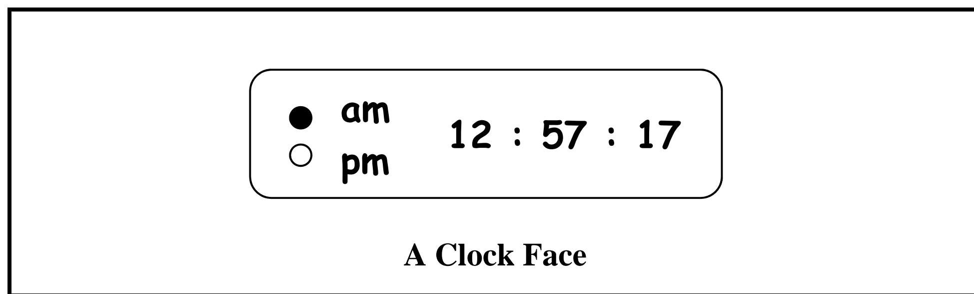
Describing Values to a Client – An Example

Let's look at a description of a software component for a clock that keeps time using am and pm (as opposed to a 24-hour or military clock). (We know, you're thinking boring, boring, boring! But consider this: A software component for keeping the time should be about the same complexity as a component for keeping the date. And yet, do you remember all of the fuss over Y2K? Well, you probably don't and may not even know what it is, but you may want to look it up on Wikipedia—http://en.wikipedia.org/wiki/Year_2000_problem.) Remember, from the client perspective, we need only describe the values a clock variable can assume, the initial value of a clock variable, and the operations that can be applied to a clock variable. In

this unit we'll describe the values a clock variable can assume, including the initial value, and in the next two units we'll describe the operations.

The value of a variable, when viewed through the mathematical model for the variable's type, is called its *abstract value*. The set all of abstract values that a variable can assume is called the *abstract state space* of the type. So, what we are about to do is to describe the abstract state space of an am-pm-clock component; that is, the values that an am-pm clock can assume as seen through the eyes of a client.

To get started, let's consider a typical clock face:



A Clock Face

Our task is to capture, through a mathematical model, the important features of such a clock face. The clocks we have in mind show the hour, minute, and second and whether it is am or pm. Thus, the state of such a clock is a particular value for the hour, a particular value for the minute, a particular value for the second, and a particular value for whether it is am or pm. So, in this case, the mathematical model will be fairly straightforward to write down.

Interface for AM_PM_Clock: Part 1 of a Client Description

The background is set and we're ready to look at our first client description of a component. The entire description is presented in two parts: Part 1 is presented in this unit and it is concerned with the mathematical model for describing values of am-pm

clock variables. Part 2 is presented in the next unit, and it is concerned with the operations that can be applied to am-pm-clock variables. Here we go!

```
public interface AMPMClock
extends Standard<AMPMClock>
```

AMPMClock number component with primary methods.

Mathematical Subtypes:

```
  HOUR_MODEL is integer
   exemplar h
   constraint 1 <= h <= 12

  MINUTE_SECOND_MODEL is integer
   exemplar m
   constraint 0 <= m <= 59

  AM_PM_CLOCK_MODEL is (
    hours:   HOUR_MODEL
    minutes: MINUTE_SECOND_MODEL
    seconds: MINUTE_SECOND_MODEL
    am:      boolean
   )
```

Mathematical Model (abstract value and abstract invariant of this):

```
  type AMPMClock is modeled by AM_PM_CLOCK_MODEL
```

Constructor(s) (initial abstract value(s) of this):

```
  default:
   ensures
    this.hours = 12 and
    this.minutes = 0 and
    this.seconds = 0 and
    this.am = true

// AMPMClock TO BE CONTINUED...
```

public interface signifies a *client* description; AMPMClock is the component name.

Every variable described by AMPMClock has newInstance, transferFrom, and clear operations.

Math definitions used to describe the mathematical model of AMPMClock values.

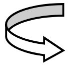
The component AMPMClock defines a new *programming type* with the same name and modeled by the math subtype AM_PM_CLOCK_MODEL.

Every variable described by AMPMClock has this initial value.

Whoa! That Was Different!

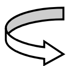
We're pretty sure that the description of the `AMPMClock` component is different from what you are used to! Don't worry. We'll carefully work our way through the description, paying close attention to the idea of mathematical modeling and how it

appears in the AMPMClock component. This example will become the model (sorry for the pun) for how we do mathematical modeling in software components.



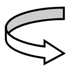
```
public interface AMPMClock
```

This line is telling us (the reader) that the component to be described is a Java **interface** (i.e., a client view description), it is publicly accessible (as it should be for a client view), and that its name is AMPMClock. In later examples we'll see that a Java **class** is used to describe the implementer/maintainer view of a component.



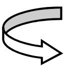
```
extends Standard<AMPMClock>
```

This line indicates that AMPMClock includes the functionality described in another component called Standard. As we will see, Standard defines three fundamental operations that will be common to the components in the course libraries: newInstance, transferFrom, and clear. The details here are really off-topic for what we are discussing, so we'll delay their explanation until later.



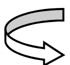
AMPMClock component with primary methods.

This is an informal comment providing a compact description of this component. As we will see in the next unit, *primary methods* are the basic operations that can be performed on variables of this component type.



Mathematical Subtypes:

This heading announces the start of the mathematical description of this component; namely, the mathematical types needed to model the values of variables of this component type. Let's look at each one of them carefully.



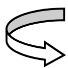
```
HOUR_MODEL is integer
 exemplar h
 constraint 1 <= h <= 12
```

This little piece of mathematics defines a mathematical model for hours. The syntax will probably look unfamiliar, but here's how you can read it. `HOUR_MODEL` is the name of a new mathematical subtype, and you can think of a subtype as just a set of values. `HOUR_MODEL is integer` says that the set of values for the subtype `HOUR_MODEL` will be all integers; that is, {0, +1, –1, +2, –2, …}. But wait a minute! In a clock, how can the value of the hour be something like 296638 or –88? Shouldn't the value of an hour be between 1 and 12 inclusive? Yes it should, and that is the purpose of the lines:

```
exemplar h
constraint 1 <= h <= 12
```

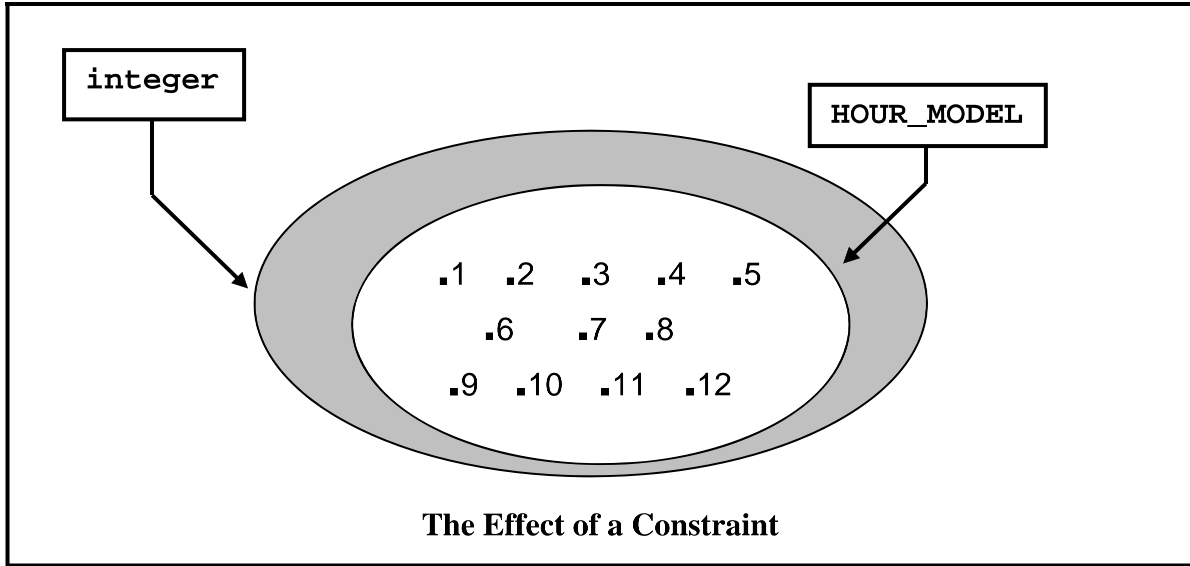
`exemplar h` just means "let `h` be a mathematical variable of type `HOUR_MODEL`". The `constraint` then specifies that the value of `h` must satisfy the condition `1 <= h <= 12`, which is exactly what we want for our mathematical model of the idea of hours.

In general, the purpose of a ***constraint*** is to place additional restrictions on a set of values. Normally, the effect of a constraint is to throw away unwanted values from an original set of values. The next figure illustrates this for `HOUR_MODEL`.

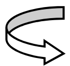


```
MINUTE_SECOND_MODEL is integer
 exemplar m
 constraint 0 <= m <= 59
```

The definition of the `MINUTE_SECOND_MODEL` is similar to the definition of the `HOUR_MODEL`, and it just provides a mathematical model for the idea of minutes and of seconds.



The Effect of a Constraint



```
AM_PM_CLOCK_MODEL is (
  hours:   HOUR_MODEL
  minutes: MINUTE_SECOND_MODEL
  seconds: MINUTE_SECOND_MODEL
  am:      boolean
 )
```

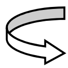
`AM_PM_CLOCK_MODEL` is an interesting model. Values for this subtype are actually ***4-tuples***; that is, each value has four parts:

- one part, named `hours`, that can assume values of math subtype `HOUR_MODEL` (`integer` values in the range 1 through 12 inclusive)
- one part, named `minutes`, that can assume values of math subtype `MINUTE_SECOND_MODEL` (`integer` values in the range 0 through 59 inclusive)
- one part, named `seconds`, that can assume values of math subtype `MINUTE_SECOND_MODEL` (`integer` values in the range 0 through 59 inclusive)
- one part, named `am`, that can assume values of type `boolean` (true or false).

Thus, we might write down a typical value of type `AM_PM_CLOCK_MODEL` as (2, 56, 24, true) meaning that the current time is 2:56:24 am. Notice that there is no constraint

clause in the definition of `AM_PM_CLOCK_MODEL` because we have no additional restrictions to impose on the value of an am-pm clock.

This completes the section of the interface where we defined three *mathematical* subtypes in order to define formally a *mathematical model* for the values that am-pm clocks can assume. That mathematical model is named `AM_PM_CLOCK_MODEL`.



Mathematical Model (abstract value and abstract invariant of this):

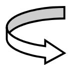
```
  type AMPMClock is modeled by AM_PM_CLOCK_MODEL
```

In this section of the interface we define a new programming type provided by the `AMPMClock` component. This is important! There is another type floating around in this component and its name is `AMPMClock`, the same name as the component name. Being the name of a component (a Java interface), `AMPMClock` is considered to be a ***programming type***, not a mathematical type.

The line

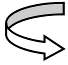
```
type AMPMClock is modeled by AM_PM_CLOCK_MODEL
```

is crucial because it links together the two types: the mathematical model `AM_PM_CLOCK_MODEL` and the programming type `AMPMClock`. The statement essentially says "Hey! If you have a variable in your program of type `AMPMClock` and would like to know what values it can assume, take a look at the definition of `AM_PM_CLOCK_MODEL`. That's where you'll find your answer."



Constructor(s) (initial abstract value(s) of this):

This line introduces the last section of this part of the `AMPMClock` component. This is where we describe the possible initial value(s) for new variables of type `AMPMClock` and how they can be constructed. In this case, there is only one way a client can construct an `AMPMClock` value.



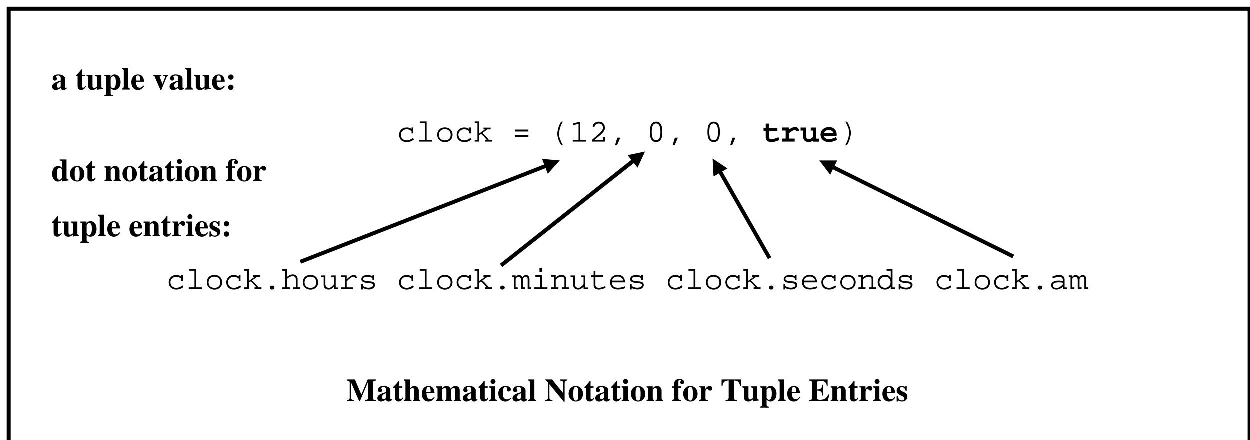
```
default:
 ensures
  this.hours = 12 and
  this.minutes = 0 and
  this.seconds = 0 and
  this.am = true
```

These lines specify the default *initial value* that a newly declared variable described by `AMPMClock` can be given. For example, if `myClock` is such a variable, then the statement

```
AMPMClock myClock = new AMPMClock1();
```

will assign the value `(12, 0, 0, true)` to myClock; that is, `myClock.hours` = `12`, `myClock.minutes` = `0`, `myClock.seconds` = `0`, and `myClock.am` = **`true`**.

(Actually, we just slipped in some additional mathematical notation, the dot notation used to refer to the individual entries in a tuple value. The next figure explains dot notation.)



Mathematical Notation for Tuple Entries

Still Awake?

3-5. Give three example values that lie in the gray-shaded region in the figure "The Effect of a Constraint".

3-6. There is no constraint in the definition of the math subtype `AM_PM_CLOCK_MODEL`. Why not?

Unit Wrap-up

Comment 1: For most of us, after looking at the picture of the clock face presented earlier in this unit, our intuition would tell us that the value of the hours portion of an am-pm clock will be one of 1, 2, 3, …, 12. Our intuition would be similar for minutes, seconds, and am. So, you might ask, what's the big deal with the mathematical model? It just says what is obvious!

True enough— this simple example of a mathematical model pretty much does state the obvious. However, for other software components, the values that variables can assume may not be obvious at all. In those instances, a mathematical model can be indispensable. Do not be misled by this simple example— it is only a warm up.

Comment 2: Comment 2 appears in the next figure. IT IS FUNDAMENTALLY IMPORTANT. Make sure you understand it completely. Put it into an accessible, long-term part of your memory!

> The client description of a component must describe the values that variables of that type can assume. Such a description can be presented in different ways. A good description will be precise (no margin for error) and understandable (by human readers). The ***mechanism*** that is used in CSE 2221/2231 to make such a precise and understandable description is ***mathematical modeling***.

Comment 2 is FUNDAMENTALLY important!

Make sure you understand it completely!