Modeling with the Unified Modeling Language (UML) – A Tutorial

Contributed by Dr. Richard Cartwright, Principal Software Architect, Quantel Ltd.

Published 16 July, 2015

Note: This article is being made available as part of the activities of the Joint Task Force on Networked Media. The author is solely responsible for its content. While the information contained herein may be of interest to those following the work of the JT-NM, this document is published by its author, and is not being published by the JT-NM itself.

Why Modeling?

By modeling the elements of human experience that are monetized by content businesses, and by doing so using a common language that is understood widely in the IT industry, a new breed of interoperable networked-media systems can be designed and built. The result of this modeling exercise will be a concise, reusable architectural framework that is captured both as:

- a few pages of paper;
- a computer-based model that can automatically be transformed into skeleton code.

The value of models is as much about providing a common language with which to explain what is being modeled in an IT context, facilitating easier dialogue between the worlds of IT and media professionals, as it is about a JT-NM user-story-derived Reference Architecture models.

The power of abstraction

Abstractions are powerful – and there's nothing better to show how powerful than to use an example based on something familiar: the FPGA (Field Programmable Gate Array). In the hardware design of an FPGA, logic units with well-known behavior are assembled into circuits. From the perspective of a non-expert human, the function of the circuit will not be obvious from its design. Complex circuits may be combined into reproducible FPGA designs with known functionality understood by a human. In the development of software using high-level programming languages, the aim is to work with elements of human experience and remain as far removed from the detail of the underlying hardware and machine code instructions as possible. The process of encapsulating underlying detail is known as *abstraction*.

All high-level programming languages, such as object-oriented, functional or interpreted programming languages, use abstraction to different degrees to make programs easier to write and maintain. Using these abstractions, elements of human experience can be classified in terms of their properties, behaviors and relationships. As most languages are text-based, a computer program may be several hundreds of pages long and not every developer is skilled in reading every language. Therefore, following the abstractions, and determining the underlying design directly from computer code can be difficult. The Unified Modeling Language (UML) provides a means to graphically represent common abstractions used in high-level languages so that the design of a computer program can be expressed concisely and independently of any specific programming language.

UML can also be used to design interfaces between computer programs when the internal functionality of these programs is hidden. The programs are used as "black box" components. These interfaces are often realized as XML[1] or JSON[2] documents, with design patterns such as SOAP[3] and/or REST[4] used to implement the protocol of these interfaces. Components can be wired together to form overall software system designs, expressing common integration patterns.

A Reference Architecture model for networked media is one step in abstraction higher again than models used directly in software design. A Reference Architecture expresses common abstract concepts and inter-relationships that will require further development to be implemented in software or to be used to define an interface. Derivation from or mapping to the abstractions of this Reference Architecture model will provide a means to compare implementations, and is a step along the road towards interoperability. The Reference Architecture models also provide a common practical starting point from which software can be developed using known techniques.

Software as model as software as …

Many books have been written about using UML as a complete solution to modeling an application and completing a design before anyone sits down to code it (e.g., the *rational development process*). Using modeling in this way has several risks:

- The model is over-designed and choices are made at the UML level that do not map well onto constraints of the development environment, such as programming languages, databases, legacy applications or best practices.

[1] XML is eXtensible Markup Language, http://www.w3.org/XML/.
[2] JSON is JavaScript Object Notation, http://json.org.
[3] SOAP is Simple Object Access Protocol, http://searchsoa.techtarget.com/definition/SOAP.
[4] REST is REpresentative State Transfer, http://www.restapitutorial.com/lessons/whatisrest.html.

- A waterfall approach[5] is adopted with no iterative feedback loop into the models. An agile approach of code early, build prototypes and fail fast would be more appropriate.

In reality, UML has become one of many tools that are used in software development and has value at design stages, which in an agile process is at the beginning of every sprint[6]. UML models can be sketched as part of a shared design workshop, often on a white board, and taken by developers back to their desk to turn into code. At that point, the model becomes the code and the UML can be stored in the software repository for reference. At the next design or review session, relevant parts of the model may be sketched again, with UML used as a communication tool to illustrate necessary design changes. For simple projects, UML may not be needed at all.

The fluidity of being able to move from software to model and back again means that the most common type of UML models are class diagrams used for data modeling. These allow business analysts and software developers to iteratively design common data sets.

To understand UML models is to understand programming concepts such as object-oriented design. The rest of this sub-section provides a brief tutorial on these concepts.

Some of the more business-oriented models introduced in later versions of UML have gained less traction because they require additional translation steps between model and software artifacts.

The networked media Reference Architecture model provided by JT-NM aims to provide a set of high-level sketches, parts of which would otherwise have to be developed at the beginning of a development project anyway. Developed from a set of industry-derived user stories in combination with expert knowledge and experience, the models provide a familiar-to-developers common basis for projects, a model of the basic concepts of the media domain and pointers to aspects that should not be overlooked.

Classification and instantiation

At the heart of UML is building data models that are relevant to the application and its domain. One of the key distinctions to make is the difference between the instance of an *object* and its classification as a *class*. An object is an entity that can be experienced, for example an analogue clock. Each clock is unique and yet a collection of clocks can be observed to have a number of common properties, such as the radius of the clock face, the size and style of the

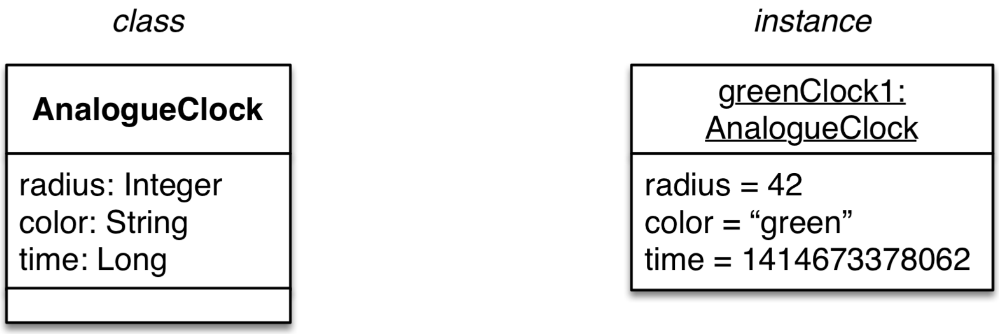
[5] “Waterfall” refers to a software development methodology that does one part of the process at a time and then falls into the next and when it is completed moves on to the next like water going over a waterfall. http://en.wikipedia.org/wiki/Waterfall_model.

[6] A *sprint* is a relatively short period of time – for example 2 or 3 weeks -- in which a defined amount of feature or function is developed and tested. Successive sprints are referred to as *iterative development* and are used to build *releases* of software.

numbers, the time that it is currently displaying, etc. A class describes a collection of possible objects using a set of their defining properties.

A running computer program stores instances of the objects it manipulates as small regions of addressable memory, with values for each of the properties. Anything from zero or one up to millions of objects of the same class may be present in memory at any one time. The actual computer program does not have each object separately represented. The program specifies the class and the programming language provides a means to stamp out new instances of the object. For example, consider a world clock application that displays several analogue clocks objects, one for each time zone selected by a user.

Similarly, UML models concentrate on the class of objects rather than the objects themselves. It is possible to represent object instances in UML. However, this is rarely done, as you would end up drawing thousands of them.



```
class AnalogueClock(
  radius: Int,
  color: String,
  time: Long)
```

```
val greenClock1 = new AnalogueClock(
  42,
  “green”,
  System.currentTimeMillis)
```

The left of the figure shows a UML class for an analog clock. The class is split into three boxes, where the top box is the name of the class, the middle box contains a list of its properties, each with name and type. Details of the bottom section are provided in the behavior section. Below the class is a representation of it as a class written in the *Scala* programming language. To the right is a UML instance of an analog clock and below it the way in which an instance of the object is created in the language.

Association

Different classes of objects are related to one another in different ways, including:

- fairly loose association of "uses a", "communicates with" or "is related to" relationships, e.g., the parent-child relationships between people;

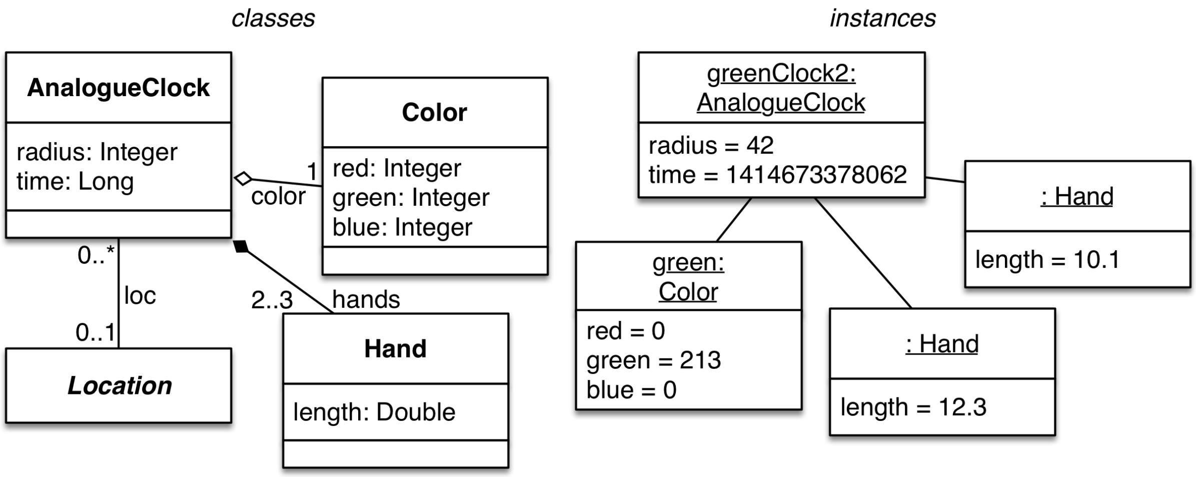
- aggregation associations representing a stronger relationship, such as "owns a" or "has a" relationships, such as between a car chassis and its engine, or a Lego™ model and its constituent bricks;

- composition associations representing aggregations with co-incident lifetime, whereby destroying the owner destroys what it owns, "is made of" e.g. the relationship between a human and its brain.

To illustrate these relationships, our clock model is improved as follows:

1. In the previous clock example, a free text string specifies the color of the clock. To extend the example, at a clock manufacturer it is necessary to make many clocks to a fixed color palette. Our model can be refactored to include a separate *Color* class.
2. The clock is made of a long hand for minutes and a short hand for hours, with some clocks having a third hand for seconds. At manufacture, these hands are permanently attached to the clock.
3. The location of a clock is tracked within the factory.

These changes alter the model as shown in the figure below.



```
class Color(red: Int,
  green: Int, blue: Int)
abstract class Location
class Hand(length: Double)
class AnalogueClock(
  radius: Int,
  color: Color,
  hands: Set[Hand],
  time: Long,
  loc: Option[Location] = None)
```

```
val green = new Color(0, 213, 0)
var greenClock2 = new AnalogueClock(
  42,
  green,
  Set(new Hand(10.1), new Hand(12.3)),
  System.currentTimeMillis)
```

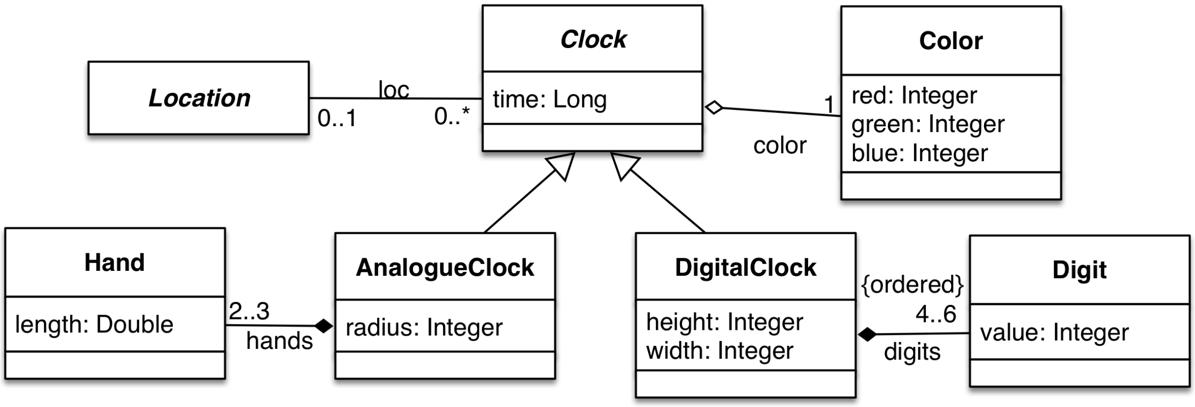
In the updated model, to track clocks as they are moved around, each clock has a loose and optional association with its location. The optionality is indicated by the "0..1" multiplicity on the line used to indicate the association "loc" between each of the analog clocks and its location. The purpose of the optionality constraints is to show that the location may not be known and to ensure that when it is, a clock can be in at most one location. A multiplicity of "0..*" is used to indicate that one location may have many clocks. The details of a location are not yet modeled and so *Location* is shown as an *abstract class* by italicizing its name.

Separating out the color into a separate class allows a set of palette colors to be defined and referenced from one or more clocks. This kind of aggregation association is shown using an unfilled diamond at the owning end. Every clock has exactly one color.

A clock is composed of two or three hands. Composition associations are represented using a line with a filled diamond at the owning end. Destroy the clock and you destroy the hands. Note that in the UML and code representations of the clock instance, the hands of the clock have no identity of their own.

Generalization

Many objects have similarities to one another and yet have some differences, where those differences can be classified. More efficient and maintainable computer programs can be written when patterns of commonality between objects can be identified through the process of generalization. Common properties for a broad classification of objects are identified in one *super class* and more specific features are added in one or more specialized *sub classes*. The sub class has all the common features of its general super class parent that it *extends* and it is not necessary to repeat them.



```
class Color(red: Int,
  green: Int, blue: Int)
abstract class Location
class Hand(length: Double)
class Digit(value: Integer)
abstract class Clock(
  color: Color,
  time: Long,
  loc: Option[Location])
```

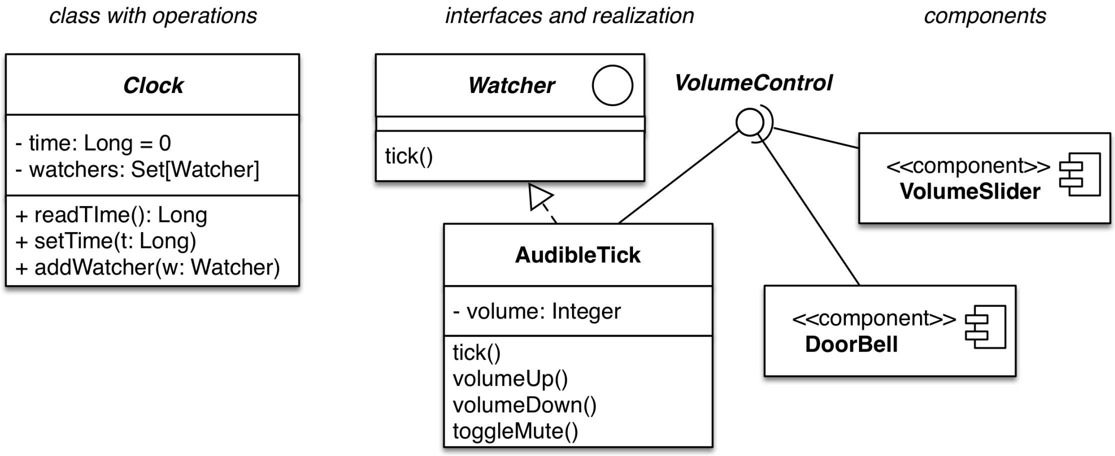
```
class AnalogueClock(radius: Int,
    color: Color, hands: Set[Hand],
    time: Long, loc: Option[Location] = None)
  extends Clock(color, time, loc)

class DigitalClock(height: Int, width: Int,
    color: Color, digits: Seq[Digit],
    time: Long, loc: Option[Location] = None)
  extends Clock(color, time, loc)
```

In the clock example, digital clocks can be introduced as another specific kind of clock alongside analog clocks. Digital clocks still have a time, color and location, but their display has a height and width as digital clocks have display digits instead of hands. Rather than representing the common properties inside both analog and digital clocks, a new *abstract* type of *Clock* is introduced, where abstract means that a generic clock can never exists as an instance in its own right. Every sub class that is a specialism of a clock, diagrammatically represented by a large unfilled arrowhead at the general end, *inherits* the properties and behaviors of the clock.

Note that generalization relationships are often shown as tree structures, with a single arrow at the root. This tree structure is the same as drawing lots of separate generalization arrows with the same root.

Behavior

Objects exhibit behaviors and can respond to stimuli. The primary means of representing application behavior in UML is by adding *operations* to classes in the bottom box. Each operation has a name, a list of parameters and the type of value that it can return. Collections of related operations can be grouped into *interfaces*, with classes said to *realize* those interfaces when they provide an implementation for each of the operations of an interface. A class may realize any number of interfaces.



```
trait Watcher { def tick(t: Long) }
  // scala interfaces are called traits
abstract class Clock {
  private var time = 0l
  private var watchers = Set[Watcher]()
  def readTime() = time
  def setTime(t: Long) = { time = t }
  def addWatcher(w: Watcher) = {
    watchers = watchers + w }
  ... }
```

```
trait VolumeControl {
  var volume: Integer
  def volumeUp() = { volume = volume + 1 }
  def volumeDown() = { volume = volume - 1 }
  def toggleMute() = { volume = -volume } }
class AudibleTick
    extends Watcher with VolumeControl {
  var volume: Integer = 5
  def tick(t: Long) = if (volume > 0) playTick()
  ... }
```

The clock example has been extended with some operations to allow time to be read, set and watched by a clock *Watcher*. Note that the time field itself has been made private so that it can only be accessed via the operations, allowing constraints to be applied. The *clock watcher* is defined as an interface that is modeled by its operations. Interfaces are represented as classes with a circle in the top right hand corner.

Different kinds of clock-watcher can be conceived; for example, an hourly chime or a photo slideshow that changes every few seconds. The one shown as a class in the diagram is an audible tick that is played every second. UML diagrams show that a class realizes an interface using an arrow with a large, unfilled head connected to the interface with a dashed line to the realizing class.

Component diagrams show how systems are composed by the interconnection of components via the one or more interfaces that they expose. Component diagrams often hide the detail of operations of an interface and the classes that are used to implement the component, showing this detail on another diagram. Components represent software modules (source code, binary code, executable, DLL[7], etc.) with well-defined interfaces that depend on one another.

The audible tick has a volume control, as does a *door bell* component, and that is specified by the *volume control* interface. In the figure, the details of the volume control interface are hidden and assumed to be on another diagram and so it has been collapsed to a single circle. The line from the circle to the class indicates that all audible tick objects need to implement to the volume control interface, as do all door bell components. The details of which class(es) implements the doorbell are not shown. As both audible ticks and doorbells have the same volume control, the same *volume slider* component can be used to control both.

Other forms of behavior diagram exist in UML and form an important part of designing application behaviors. *Sequence diagrams* define the sequence of operations that must occur between objects to achieve a specific use case. *State diagrams* show the different states that an object can be in, the events the trigger state change and the resulting behaviors – useful for defining protocols such as the states of a transcode operation. Both kinds of diagram can represent hierarchical structures and multi-threaded parallel executions.

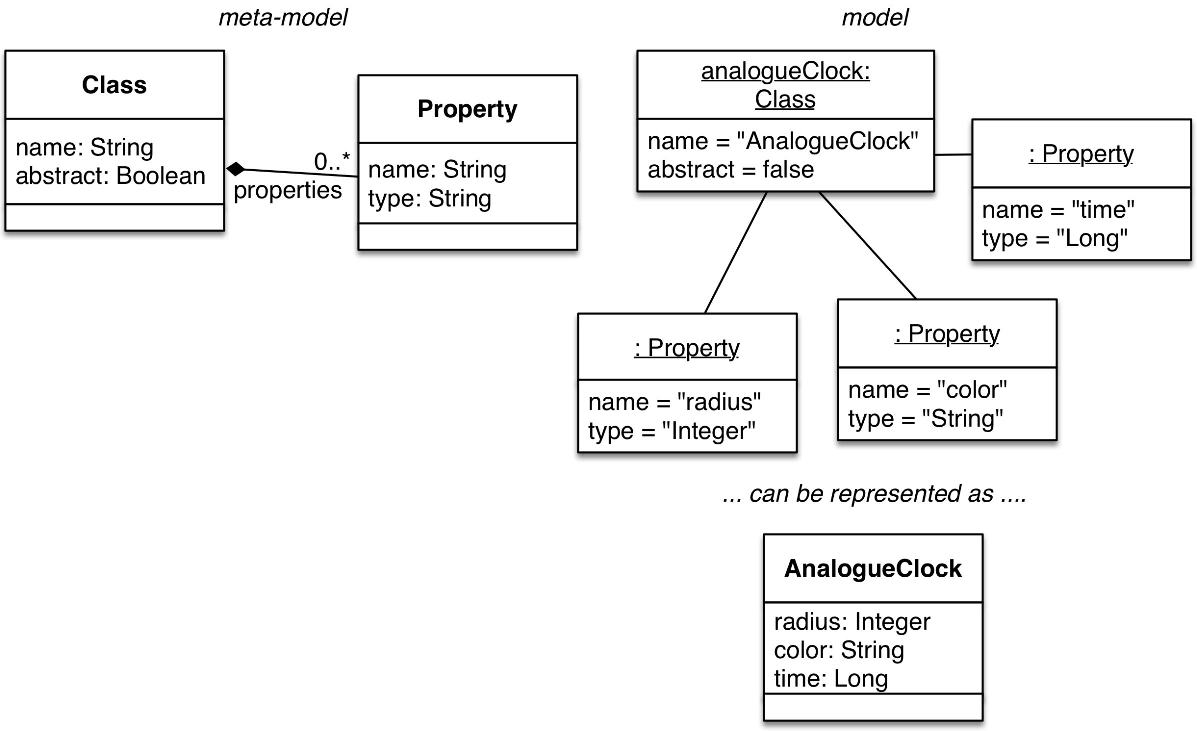
Higher-order meta-models

Meta-models and their run-time implementations are an important concept in building generalized monitoring dashboards, generalized control surfaces, software defined networks and dynamically scalable cloud-computing resources.

The capability interface that is the cornerstone of the JT-NM Reference Architecture, as introduced in the systems model and further developed in the following UML Reference Architecture model, is a form of *meta-model*. It is the means of describing an interface so that it can be advertised, discovered and subsequently consumed. Such higher-order abstractions are commonplace in computer science, are mathematical in nature and can take some getting used to.

As an example of a higher-order meta-model, consider how UML could be used to model UML. The JT-NM capability interface is just that as it is effectively a way of exposing a behavioral fragment of UML that happens to be modeled in UML.

[7] DLL is *Dynamic Link Library. http://support.microsoft.com/kb/815065.*



To follow this through with the clock example, consider a UML model for classes and their properties and how that model would be expressed as an instance. This is illustrated in the figure above.

As mentioned, meta-models and their run-time implementations are important concepts in building:

generalized monitoring dashboards where any device connected to a network can be discovered via a monitoring meta-model and have both its common (e.g. CPU, RAM) and unique (e.g. buffer capacity, number of users) telemetry dynamically added to control room displays, audit trails and alarm systems;

generalized control surfaces where any device connected to a network can describe the controls that it offers via a meta-model and these can be added to dynamic control panels or brought under the automated control of a workflow or rules engine;

software defined networks where a meta-model for paths in the network can be used to reconfigure the network to suit the content currently flowing over the network;

dynamically scalable cloud-computing resources where the computing resources are represented by a meta-model that can be used to provision more or less resource as required.