Umeå University
Department of Computing Science

Object-Oriented Programming Methodology 7.5 p
5DV133

OU1 Clock

Submitted 2016-04-07
Author: Lorenz Gerber (dv15lgr@cs.umu.se lozger03@student.umu.se)
Instructor: Anders Broberg / Niklas Fries / Adam Dahlgren / Jonathan Westin / Erik Moström / Alexander Sutherland

1 Introduction

Aim of this laboration was to implement a model of a digital clock and an alarm clock in the object oriented language Java. The class design was mostly defined in the specifications [**?**]. After implementing *Clock* and *Alarm* classes, a *main* method had to be written to provide test cases for the implemented classes. Additionally, it was also recommended to write JUnit tests for all classes.

2 Usage Instructions

The files provided in an zip archieve: *ClockApp.java, Clock.java, ClockTest.java, NumberDisplay.java, NumberDisplayTest.java, Alarm.java, AlarmTest.java*. To build from the command line, *javac ClockApp.java* is invoked. To run the program *java ClockApp*. The main program *ClockApp* does not take any input. On execution it writes output to the screen. The implications of the printout will be described further in the section *testing*.

3 System Description

First Class Responsibility Collaborators *CRC* Tables were drawn for all classes (*Table* **??**, **??**, **??**). The specification did not demand an actual running clock, just a model that would have all the functionality. Hence the advancement of the time had to be done manually in the main program.

The UML class diagrams shown here are basically the same as those given in the assignment (*Figure* **??**). However, the *Alarm* class was added. The *Clock* aggregates two *NumberDisplays*. The *Alarm* inherits this from the *Clock* but aggregates two additional *NumberDisplays* to keep track of the alarm time. This could also have been solved as primitive datatype properties in the *Alarm* class. Here it was however decided to reuse the components, and keep it flexible if further functionality should be implemented later on.

Table 1 *'Class Responsibility Collaborators' tables for Clock*

Class: Clock	
Responsibilities	**Collaborators**
knows hours	NumberDisplay
knows minutes	
show time	
set time	
advance minutes	

3.1 Checking for wrap around

In the specifications it was given that *didWrapAround* is a method. There are different ways how this check can be implemented. The most simplistic is to check whether minutes is 0 and deduce from it that it wrapped around on the last *minutes.increment*. Another way would be to add a boolean property variable that can be set to true when actual wrapping happens.

2(**??**)

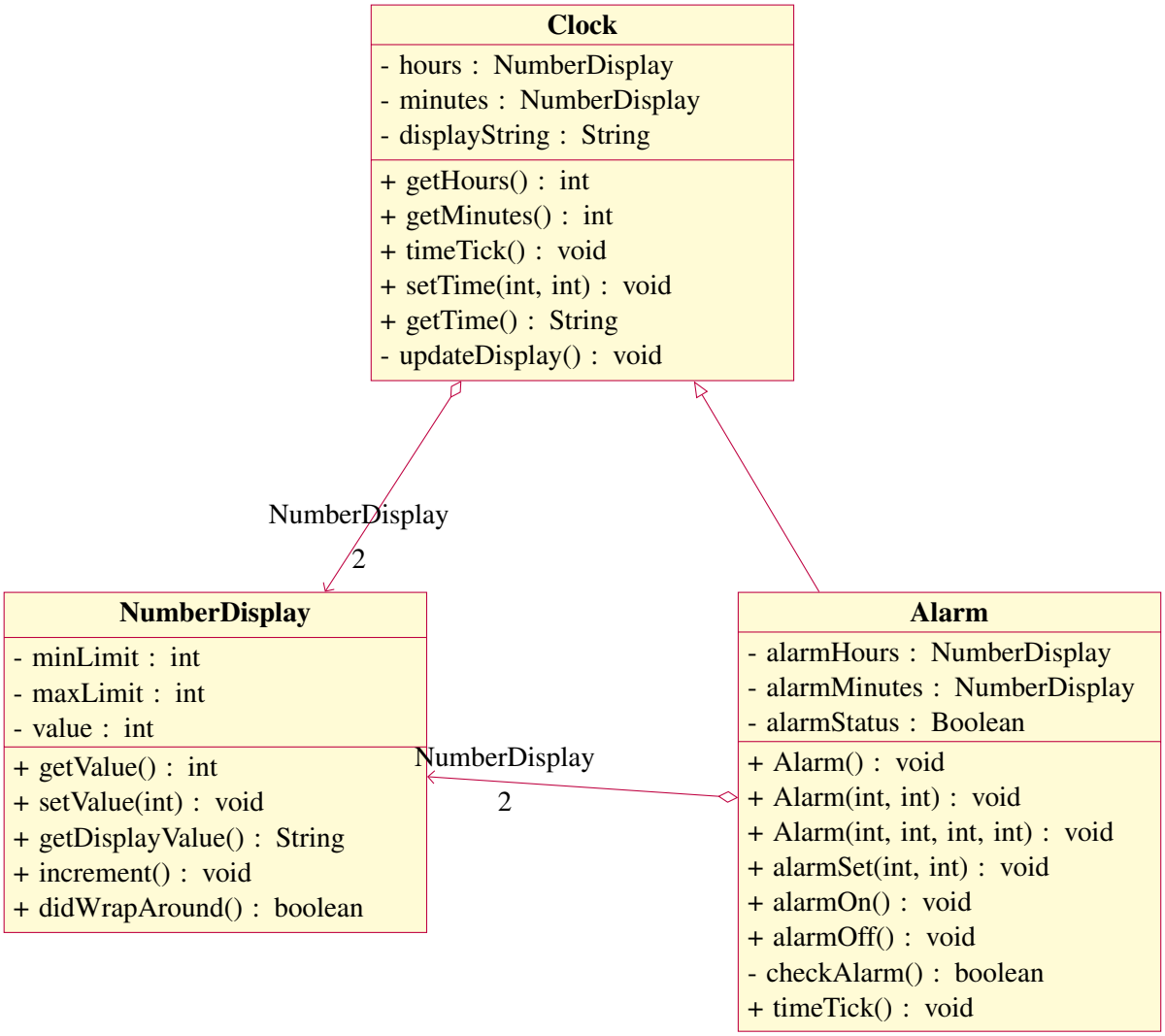

Figure 1: *Class diagrams of Clock and NumberDisplay.*

Table 2 *'Class Responsibility Collaborators' tables for NumberDisplay*

Class: Number Display	
Responsibilities	**Collaborators**
knows minLimit	Clock
knows maxLimit	
knows current value	
show value	
advance value	
check if wrapped around	

Table 3 *'Class Responsibility Collaborators' tables for Alarm.*

Class: Alarm - extends Clock	
Responsibilities	**Collaborators**
knows alarm hours	NumberDisplay
knows alarm minutes	
know if it is alarm	
set alarm	
switch on / off alarm	
alarm!	

3.2 Checking for Alarm condition

In the specifications it was given that the check for alarm should happen in the *timeTick* method. Hence, setting time or alarm to the same time will not produce an alarm event as the alarm condition will be checked after the *minutes.increment* in the *timeTick* method. Implementing the feature to check for alarm condition after both setting time or alarm time can be implemented in many ways. The determining factor is that alarm check happens in the *timeTick* method. Hence one way could be to decrease the minutes counter one minute and then call for the *timeTick* method that would also check for alarm condition in the actual set time. However, this implementation creates a complicated edge case for the value 0. When setting time to 0, it will be decreased to -1 to advance then to 1. This will however detect wrap around condition and advance the hour by one. Hence, to implement this feature, it could make sense to store wrap around condition when it actually happens in a seaparate boolean property.

4 Testing

Testing was conducted in the main program. Additionally, JUnit4 test files were written and tested. The JUnit test cases check wheter the constructors works and for the *IllegalArgumentException*. More detailed logical testing was done in the main program which included nine usecases which were all run on the *Alarm* class. For the sake of simplicity, now logic was built in to check for passed or not passed. However, after reading the specification of the classes, the expected result should be obvious.

First a new object was instantiated. Then the default time of the clock modul is written

to screen. Next, alarm time is set to clock time. Due to the specific implementation of the program, no alarm is expected. The opposite case is tested next where the clock is set to an activated alarm time. Also here, due to the specific implementation, no alarm event happens. Then incrementing the time through a full cycle with activated alarm should create an alarm event. Next, the same situation is tested with alarm deactivated. Exception handling for out of range input was implemented in all classes. Here exception handling for such a case is tested next. Finally, two edge cases are tested. First incrementing from time 00:00 to 00:01 and then from 23:59 to 00:00.