Introduction

Purpose: Illustrate design steps for a single data-abstracting class. Different internal representations possible. No complicated abstraction function.

Method

The development follows the general Design by Contract pattern:

1. Informal requirement analysis;
2. Design class interface;
3. Add contracts;
4. Design internal representation;
5. Implement methods.

Step 1: Informal Requirements

Design and code a class TimeOfDay which maintains the time of day. It should have the following operations:

- Creation and initialization to 00:00:00;
- Queries/Inspection
  - current time in integers representing hours, minutes and seconds;
  - current time in seconds since 00:00:00;
  - current time in string of the form hh:mm:ss;
- Commands/Modification
  - reset to 00:00:00;
  - set to given hours, minutes, seconds;
  - set to given seconds since 00:00:00;
  - advance time by a number of seconds ("modulo length of day");
  - tick, i.e. advance time by one second ("modulo length of day");
Step 2: Design Class Interface

The informal requirements naturally lead to a class interface of the following form, where the operations have been grouped according to their main role, i.e.

- construction and initialization
- queries ("observable entities"), i.e. inspection of the current state
- commands, i.e. changes of the current state

```java
public class TimeOfDay {
    // Constructors
    public TimeOfDay() {
    }

    // Queries
    public int hours() {
    }
    public int minutes() {
    }
    public int seconds() {
    }
    public int totalSeconds() {
    }
    public String toString() {
    }

    // Commands
    public void setHMS(int h, int m, int s) {
    }
    public void setTotalSeconds(int ts) {
    }
    public void advance(int s) {
    }
    public void reset() {
    }
    public void tick() {
    }
}
```

The methods `hours`, `minutes`, and `seconds` are primitive queries with obvious meanings. The method `totalSeconds` expresses the current time in seconds since 00:00:00. The method `toString` returns the current time in a string of the form hh:mm:ss.

These methods, which provide different pieces of information about the current time, are not independent. Their exact relationships will be discussed in Step 3.

The method `setHMS` sets the current time to given hours, minutes and seconds. The method `setTotalSeconds` sets the current time in seconds since 00:00:00. The method `advance` advances the current time with a given number of seconds. The method `reset` resets the current time to 00:00:00. The method `tick` advances the current time by 1 second.

The precise effect of these procedures on the observable entities will be described in Step 3.

Step 3: Add Contracts

The queries `hours`, `minutes`, `seconds`, and `totalSeconds` provide different pieces of information about the current time. They may be considered as primitive queries. They are not independent, however. It is possible to express `totalSeconds` in terms of `hours`, `minutes`, and `seconds`. With
suitable restrictions on the admissible values of hours, minutes, and seconds, it is also possible to express these in terms of totalSeconds. Therefore, let us first consider some simple invariants for the admissible values for the result of each of these four queries:

- I0: 0 <= hours && hours < 24
- I1: 0 <= minutes && minutes < 60
- I2: 0 <= seconds && seconds < 60
- I3: 0 <= totalSeconds && totalSeconds < SECONDSINDAY

where the constant SECONDSINDAY stands for the number of seconds in a day, i.e. 24 * 60 * 60.

The relation between the values of the four queries can be captured by the following invariant:

- I4: totalSeconds == hours * 60 * 60 + minutes * 60 + seconds

As validity conditions like those expressed in invariants I0 through I3 are likely to occur in more than one place, it is helpful to express them by means of some auxiliary validator functions as follows:

```java
// Validators
public boolean validHours(int h) {
  // ret: 0 <= h && h < 24
}

public boolean validMinutes(int m) {
  // ret: 0 <= m && m < 60
}

public boolean validSeconds(int s) {
  // ret: 0 <= s && s < 60
}
```

Using these validators, the whole group of public invariants can be restated as

```java
// Invariants
// I0: validHours(hours)
// I1: validMinutes(minutes)
// I2: validSeconds(seconds)
// I3: 0 <= totalSeconds < SECONDSINDAY
// I4: totalSeconds == hours * 60 * 60 + minutes * 60 + seconds
```

The query toString is not primitive. Its result can be specified in terms of hours, minutes and seconds as

```java
public String toString() {
  // ret: (hours, minutes, seconds) in string of the form hh:mm:ss
}
```

where it is understood that hh, mm, and ss stand for two-digit string representations of the yields of hours, minutes and seconds respectively.

Next, let us turn to the specifications of the constructor and procedures.

- The constructor TimeofDay() should create and initialize the TimeOfDay object. The informal requirements state that the current time should be initialized to 00:00:00. In terms of primitive queries, this can be expressed either in terms of hours, minutes and seconds, or in terms of totalSeconds. We choose for the latter possibility, resulting in the following specification for TimeofDay():

```java
// pre: true
// post: totalSeconds == 0
public TimeofDay() {...}
```

- The obvious post-condition for method setHMS(int h, int m, int s) is

```java
// post: hours == h, minutes == m, seconds == s
```
However, in order not to violate invariants I0 through I2, we should restrict the values of the parameter to valid ones by imposing a pre-condition. This results in the following specification for setHMS:

```
// pre: validHours(h), validMinutes(m), validSeconds(s)
// post: hours == h, minutes == m, seconds == s
public void setHMS(int h, int m, int s){...}
```

- Similarly, the specification of setTotalSeconds is:

```
// pre: 0 <= ts < SECONDSINDAY
// post: totalSeconds == ts
public void setTotalSeconds(int ts){...}
```

- The method advance should advance the current time by the number of seconds given as its parameter, modulo the length of the day (measured in seconds). This effect is most easily expressed in terms of the primitive query totalSeconds before and after the operation. To distinguish between these two values, we use the primed notations ‘totalSeconds’ and totalSeconds’ respectively. As pre-condition we stipulate that the parameter value is at least zero. All this results in the following specification:

```
// pre: 0 < s
// post: totalSeconds’ == (totalSeconds + s) % SECONDSINDAY
public void advance(int s){...}
```

Alternatively, method advance can be specified using a specification constant C resulting in the following specification:

```
// pre: 0 <= s && totalSeconds = C
// post: totalSeconds == (C + s) % SECONDSINDAY
public void advance(int s){...}
```

It expresses that, for any possible value of C, if primitive query totalSeconds yields value C before the operation then it will yield value (C + s) % SECONDSINDAYS after the operation. Note the implicit universal quantification over C in the above specification.

- Likewise, we obtain the specification for method tick:

```
// pre: true
// post: totalSeconds’ == (totalSeconds + 1) % SECONDSINDAY
public void tick(){...}
```

- Finally, the obvious specification for method reset is:

```
// pre: true
// post: totalSeconds == 0
public void reset(){...}
```

In summary, the complete class header, extended with contracts and validators, is as follows:

```java
public class TimeOfDay {
    // Constants —
    // SECONDSINDAY is the number of seconds in a day
    public static final int SECONDSINDAY = 24 * 60 * 60;

    // Constructor
    // pre: true
    // post: totalSeconds == 0
    public TimeOfDay(){...}

    // Queries —
    public int hours(){...}
}
```
Step 4: Design Internal Representation

In this section we design an internal representation suitable for implementing the class specified in Step 3. But before we do so, we note that even without knowledge about the internal representation it is already possible to implement some of the operations, because their effect can be achieved by using other operations. We can do so because we have completely specified each operation in Step 3.

• Consider the methods reset and setTotalSeconds. From their specifications in Step 3 it follows that reset can be implemented by using setTotalSeconds as follows:

```java
// pre: true
// post: totalSeconds == 0
public void reset()
{
    setTotalSeconds(0);
}
```
• Similarly, consider the methods tick and advance. From their specifications in Step 3 it follows that we can implement tick by using advance as follows:

```java
public void tick() {
    advance(1);
}
```

• Method toString can produce its required result by converting the yields of functions hours, minutes and seconds to a formatted string (for more information on the String.format() method and on format strings consult the Java documentation):

```java
public String toString() {
    return String.format("%02d:%02d:%02d", hours(), minutes(), seconds());
}
```

• The constructor TimeOfDay() can achieve its initialization task (setting the current time to 00:00:00) by using method reset:

```java
public TimeOfDay() {
    reset();
}
```

Notes w.r.t. this approach:

1. Although it is possible to define some of the operations in terms of others before a choice for an internal representation has been made, this should be done with care. Some choices may be premature and may have a negative effect on efficiency.

2. There are other possibilities. Consider for instance:

```java
public void advance(int s) {
    for (int i = 0; i != s; i++) {
        tick();
    }
}
```

```java
public void setTotalSeconds(int ts) {
    reset();
    advance(ts);
}
```

Do you see problems with these solutions?

3. From the invariants I0 through I4 we can obtain the following relations between hours, minutes, seconds and totalSeconds:

- totalSeconds == hours * 60 * 60 + minutes * 60 + seconds
- hours == totalSeconds / 3600
- minutes = (totalSeconds % 3600) / 60
- seconds == totalSeconds % 60

This can be used on the one hand to implement totalSeconds in terms of hours, minutes and seconds, and on the other hand to implement each of hours, minutes and seconds in terms of totalSeconds. But we should not do both, as this results in non-terminating recursion.
The examples above show that there is a lot of freedom in implementing the operations. We may use this freedom to advantage to choose an implementation that is optimal in some sense, e.g. in terms of computation time, ease of coding, or some other criterion. For our current case, two choices suggest themselves:

1. Use three private instance variables to represent hours, minutes and seconds, subject to private invariants:

   ```java
   private int fSeconds;
   private int fMinutes;
   private int fHours;
   // private invariants
   // Pri0: validHours(fHours)
   // Pri1: validMinutes(fMinutes)
   // Pri2: validSeconds(fSeconds)
   ```

2. Use one private field to represent the total seconds, subject to a private invariant:

   ```java
   private int fTotalSeconds;
   // private invariants
   // Pri0: 0 <= fTotalSeconds && fTotalSeconds < SECONDSINDAY
   ```

When we consider how the operations can be coded using each of these two alternatives, it turns out that most of them are straightforward for each of them, and that there are no significant differences in computation time. The only non-trivial case is procedure `advance()`, which is easy using alternative 2, and rather cumbersome using alternative 1 (try it!). This leads us to the choice of alternative 2.

**Step 5: Implement Methods**

As mentioned before, for alternative 2 the implementation is easy, and does not need much clarification any more. The only point deserving some attention is the handling of preconditions. Here we follow the principle that each method checks its own precondition (if not identically true) by means of an `assert` statement (see the Java documentation for details of the assert statement and associated mechanisms). The complete code of the `TimeOfDay` class is given in the Appendix.

**Appendix: Code**

```java
/**
 * @author Kees Hemerik
 *
 */

public class TimeOfDay {
    // Constants
    // SECONDSINDAY is the number of seconds in a day
    public static final int SECONDSINDAY = 24 * 60 * 60;

    // Private part
    // fTotalSeconds is the number of seconds elapsed since 00:00:00
    private int fTotalSeconds;

    // Private invariants
    // Pri0: 0 <= fTotalSeconds && fTotalSeconds < SECONDSINDAY

    // Constructors
    // pre: true
    // post: totalSeconds == 0
```
public TimeOfDay(){
    reset();
}

// Queries

public int hours(){
    return fTotalSeconds / 3600;
}

public int minutes(){
    return (fTotalSeconds % 3600) / 60;
}

public int seconds(){
    return fTotalSeconds % 60;
}

public int totalSeconds(){
    return fTotalSeconds;
}

// ret: (hours, minutes, seconds) in string of the form hh:mm:ss
public String toString(){
    return String.format("%02d:%02d:%02d", hours(), minutes(), seconds());
}

// Validators

// ret: 0 <= h && h < 24
public boolean validHours(int h){
    return 0 <= h && h < 24;
}

// ret: 0 <= m && m < 60
public boolean validMinutes(int m){
    return 0 <= m && m < 60;
}

// ret: 0 <= s && s < 60
public boolean validSeconds(int s){
    return 0 <= s && s < 60;
}

// Invariants
// 0: validHours(hours)
// 1: validMinutes(minutes)
// 2: validSeconds(seconds)
// 3: 0 <= totalSeconds < SECONDSINDAY
// 4: totalSeconds == hours * 60 * 60 + minutes * 60 + seconds

// Commands

// pre: validHours(h), validMinutes(m), validSeconds(s)
// post: hours == h, minutes == m, seconds == s
public void setHMS(int h, int m, int s){
    // check pre
    assert validHours(h) && validMinutes(m) && validSeconds(s) :
        String.format("setHMS.pre failed; h=%d, m=%d, s=%d", h, m, s);

    fTotalSeconds = h * 3600 + m * 60 + s;
}

// pre: 0 <= ts < SECONDSINDAY
// post: totalSeconds == ts
public void setTotalSeconds(int ts){
    // check pre
    assert 0 <= ts && ts < SECONDSINDAY :
        String.format("setTotalSeconds.pre failed; ts=%d", ts);

    fTotalSeconds = ts;
}
public void advance(int s)
    // check pre
    assert 0 <= s;
    fTotalSeconds = (fTotalSeconds + s) % SECONDSINDAY;

public void reset()
    // post: totalSeconds == 0
    setTotalSeconds(0);

public void tick()
    // post: totalSeconds == (totalSeconds + 1) % SECONDSINDAY
    advance(1);