Vrije Universiteit Brussel

How simple analysis patterns can be of benefit to software developers

Software Architecture

Ward Van Heddeghem

19 March 2009

Analysis patterns are less well known than design patterns. This is a pity, since they can provide some convenient solutions to everyday software design problems. By looking at a number of real-life issues in an example program (a water tower simulator/controller), we show how these can be alleviated by making use of a few simple analysis patterns. This provides a gentle introduction to using analysis patterns; hopefully thereby stimulating the reader to explore and use more involved patterns.

1 Introduction

Software that simulates or controls a physical process generally deals with **dimensioned quantities**: values such as 15 cm or 5 s. Usually these numbers are represented internally as bare numbers, because that is the best we can do in the limited type system that programming languages give us. But what does it mean if the program states that the height of some object is 15, or that the time sample it is 5? To make sense of these numbers, we need units.[1]

Moreover, the most appropriate unit to dimension a number might differ depending on where the quantity is used in the program. For example, a water level sensor might feed its reading to the software as an 8-bit value (from 0 to 255), but the software internally represents and handles it in SI units (millimeters in this case). And, it could be that the quantity is visualized to the user in yet another unit for convenience, for example in centimeters. This 'unit juggling' requires great care and attention on behalf of the software developer to use the correct representation of the value and correct **conversion** values.

It is also common in simulation and control software that quantities and values are limited to a certain **range**, i.e. they are bounded to an upper and/or a lower value. For example, for a water level measurement to be valid its should always be between, say, 100 mm and 350 mm. Any value outside the upper and lower bound of this range has to be dealt with appropriately. Internally, ranges are usually handled by a pair of values and you check against both of them. To avoid passing each time two values as function arguments, and to avoid such bugs where a less-than symbol (<) is entered as a greater-then symbol (>)[*], it would be convenient if a more simple construct was available to deal with, and check against ranges.

In this paper, we show that the use of some simple *analysis patterns* can alleviate the problems and inconveniences described above. More specifically, the ***Quantity***, ***Conversion Ratio*** and ***Range*** analysis pattern are (theoretically) applied to a real-life software project: the design of a water tower controller/simulator in Java [2]. These patterns offer solutions to problems that have been experienced in real-life while developing the water tower software.

2 What are analysis patterns?

Within software design, reuse has been recognized as an important factor to improve the quality of software products, while also reducing the cost to build and maintain it. [3] Patterns are an important element to exploite reuse. Fowler [4] defines a pattern as "an idea that has been useful in one practical context and will probably be useful in others". While *design patterns* are typically

[*] This is not as far-fetched as one might think. Here is an example from a C source file in the Contiki operation system (2008-12-06). Spot the bug:

```
        #define MAX(a,b) ((a)>(b)?(a):(b))
        #define MIN(a,b) ((a)>(b)?(a):(b))
```

close to implementation and focus on design aspects like user interfaces, creation of objects and basic structural properties[5], *analysis patterns* are more high-level and "reflect conceptual structures of business processes rather than actual software implementations"[4].

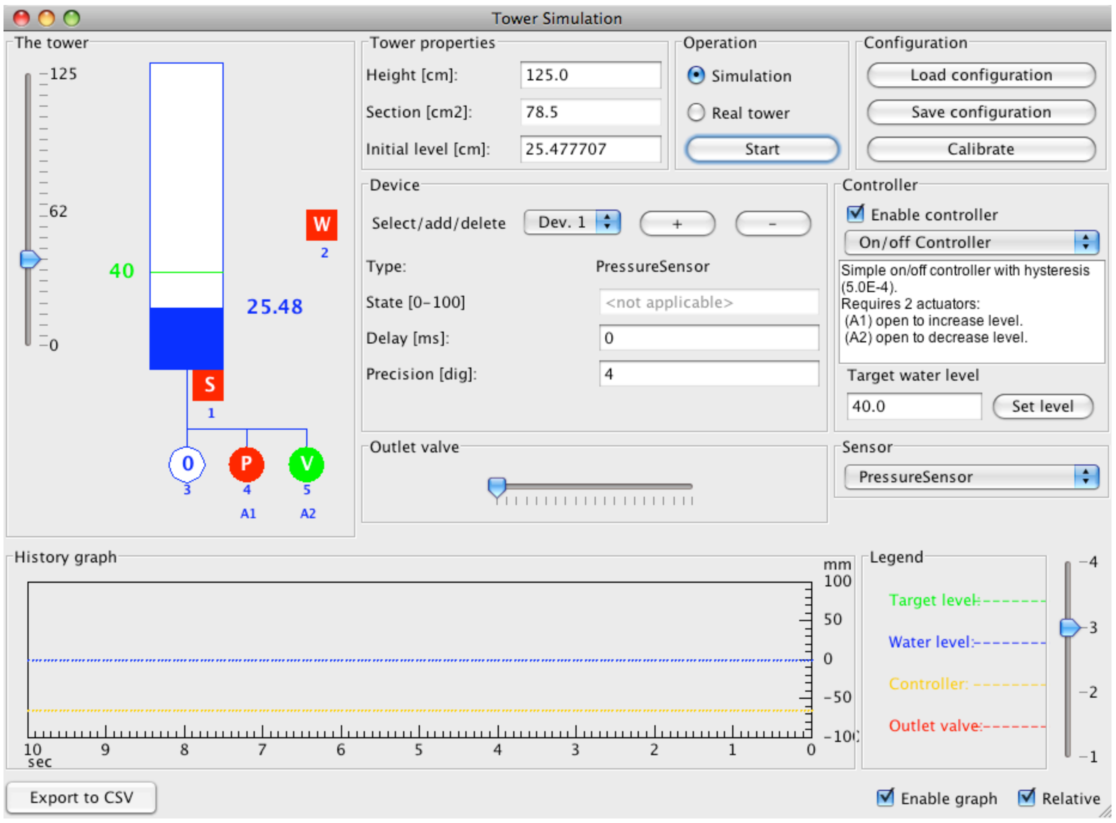
A great number of analysis patterns are discussed in Fowler's classic work 'Analysis Patterns, Reusable Object Models' [4]. Some of these are rather small, such as the Quantity, Conversion Ratio and Range pattern that we will describe in this paper, while others are more elaborate, such as the Accountability pattern which provides for a common construct in software that deals with parties (i.e. organizations or persons) and relationships between several of those parties.

We do not discuss any elaborate patterns in this paper, but instead try to show that even simple patterns can be of value.

Note that analysis patterns are not limited to software design. There is a large tradition of using patterns for example in architecture to construct bridges and buildings. To differentiate them from analysis patterns used in these other engineering domains, they are sometimes referred to as *software analysis patterns*.

3 Applying analysis patterns to the 'water tower' project

In the example 'water tower' project [2], the goal is to develop in Java a program to both simulate and control a laboratory water tower. The water tower consists of a number of pumps, valves and sensors. An externally controlled hardware valve simulates the usage of water by consumers, which unpredictably lowers the water level. The program should try to preserve a constant water level by controlling pumps and valves as appropriate. When not connected to the laboratory water tower, the program can also simulate the tower, and can be used to evaluate control strategies before testing them live on the tower. See Figure 1 for an example interface of such a water tower simulator/controller.



Tower properties
Height [cm]: 125.0
Section [cm2]: 78.5
Initial level [cm]: 25.477707

Operation
[✓ Simulation] Real tower

Device
Select/add/delete: Dev. 1
Type: PressureSensor
State [0–100]: <not applicable>
Delay [ms]: 0
Precision [dig]: 4

Controller
✓ Enable controller
On/off Controller
Simple on/off controller with hysteresis (5.0E-4).
Requires 2 actuators:
(A1) open to increase level.
(A2) open to decrease level.
Target water level: 40.0

Sensor: PressureSensor

✓ Enable graph ✓ Relative

Figure 1 Example interface of a water tower simulator/controller

3.1 Quantity pattern

In the water tower project, a number of dimensioned quantities are used. For example, the actual water level in the tower, the target water level and the section of the tower. A number of these quantities are visualized in the interface (Figure 1). While these quantities are internally handled in SI units (by project convention), the units represented on screen might use a different unit if this is more natural for the user. For example, the water level is shown in centimeters (whereas the internally used SI unit is millimeter), and the device update delay is depicted in milliseconds (whereas the internally used SI unit is second).

These numbers are represented internally as bare numbers, because that is the best we can do in the limited type system that programming languages give us. For the programmer to make sense of these numbers, we need units. Documenting the relevant code with comments about the units used is a typical way to keep track of which unit is applicable where. However, it is more natural to represent these dimensioned quantities as object, which store both the value and unit of the quantity. This is exactly what the ***Quantity*** pattern does.

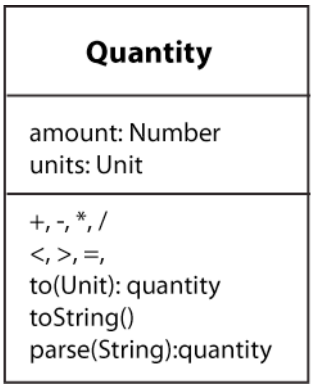


Figure 2 Quantity pattern

The basic idea of Quantity is a class that combines the amount (i.e. value) with the unit (Figure 2). This is a very simple representation, but its power lies in the behaviour that we can apply to it.

One kind of behaviour is arithmetic. We can add and subtract two quantities just as easily as we can add or subtract to numbers. We have the added benefit however, that our arithmetic methods can check if we are adding or subtracting quantities with corresponding units, thereby preventing obvious bugs.

Other useful behaviour is that we can add printing and parsing methods. This allows to easily produce strings from quantities which can then be visualized on screen. Or we could produce a quantity from the value and unit entered by the user in input boxes. Even if printing a quantity would be as simple as first printing the amount and then the unit, it would centralize the code to do so and make sure that the output is consistent across all output destination (for example, not “5 cm”, “5cm” or “5 CM” according to the concentration, or lack thereof, of the programmer).

Also, by using the Quantity pattern we have a foundation for the next pattern that we will use.

For illustrative purposes, some example code is shown below. Note that values are implemented here as integers. For more general application we could use a parameterized class.

```
public class TestQuantity {

   public static void main(String[] args) {
       Quantity waterLevel=new Quantity(15, Quantity.Unit.cm);
       Quantity userRaiseLevel=new Quantity(7, Quantity.Unit.cm);
       waterLevel.add(userRaiseLevel);
       System.out.println(waterLevel);
   }
}

public class Quantity {

   private int value; // getters and setters omitted for brevity
   private Unit unit;

   static public enum Unit {
       cm("cm", "centimeter"), mm("mm", "millimeter"),
       m("m", "meter"), l("l","litre");

       private final String abr;
       private final String descr;
```

```
        Unit(String abbreviation, String description) {
            this.abr = abbreviation;
            this.descr = description;
        }
        public String getDescription() {return this.descr;}
        public String toString() {return abr;}
    }

    public Quantity(int value, Unit unit) {
        this.value = value;
        this.unit = unit;
    }

    public void add(Quantity q) {
        if (this.unit == q.unit) {
            value += q.value;
        } else {
            // here could be error handling, or automatic conversion
        }
    }

    public Boolean equals(Quantity q) {
        if (this.unit == q.unit) {
            // If value is of Double
            return (this.value == q.value);
        } else {
            // here could be error handling, or automatic conversion
            return false;
        }
    }

    public String toString() {
        return value + " " + unit.toString();
    }
}
```

3.2 Conversion Ratio pattern

As has already been mentioned when discussing the Quantity pattern, the water tower example program represents the same quantities in different units, depending on whether the quantity is processed internally (millimeters), or displayed to the user (centimeters). Incorrect assumptions over the unit that corresponded with the value of an internal variable, have led to a few bugs in the water tower project.

One way to minimize the chance for this sort of confusion and bugs is to adhere to a coding convention that uses some kind of Hungarian notation: instead of prefixing variables with the basic data type (such as int, double, float, etc.), each variable and function has a prefix indicating an attribute that is actually relevant for the variable or function. For example, a variable that represents a water level could be named `cmWaterLevel` or `mmWaterLevel`, depending on what unit is used for the value it holds. Functions would have a prefix that indicates the relevant return type, for example `mmGetWaterLevelInput()` or `cmFromMM()`. Erroneous expressions like `mmWaterLevel= cmFromMM()` would stand out; i.e. it would make "wrong code look wrong""[6].

There is a nicer and more flexible solution though: the use of ***Conversion Ratio*** pattern. This pattern builds upon the Quantity pattern, and is a natural extension to it.

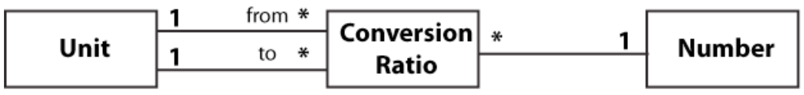


Figure 3 Conversion Ratio pattern

It allows to transparently convert from one unit to another, provided that the conversion makes sense of course. As shown in Figure 3, we can use conversion ratio objects between units, and then give Quantity an operation, `convertTo(Unit)`, which can return a new quantity in the given unit. This operation looks at the conversion ratios to see if a path can be traced from the receiving object's quantity to the desired quantity.[4] For example, we could define a conversion ratio from mm to cm (which would be a trivial multiplication by 10).

When the Quantity methods such as `add()` and `equal()` are called with different units as arguments, the method could automatically invoke the `convertTo(Unit)` function. Only when no conversion is possible would the method fail.

Thus, the Conversion Ratio pattern gives as a solid construction to convert values in between different representations.

3.3 Range pattern

The water tower example program makes extensive use of ranges: for example, the actual water level and the target water level should be between zero and the height of the tower, various controller classes use ranges to determine whether valves should be opened or closed, the state of proportional valves is represented by a value between 0 and 1. Throughout the program, values are constantly checked for being within these ranges.

Internally, ranges are usually handled by a pair of values and you check against both of them. To avoid passing each time two values as function arguments, and to avoid hard-to-spot typographic bugs (for example, where a less-than symbol (<) is entered as a greater-then symbol (>)), the ***Range*** pattern can be used. It provides a more natural representation of a range.

The basic class is very simple (Figure 4): two fields represent the start and end range. An `includes(value)` method allows to test if the supplied value falls within the range.[7] Additional comparison operations can be implemented as well, depending on the requirements.

Range is an obvious choice for a parameterized class, which is indicated in the right picture of Figure 4. More sophisticated ranges can have open ended ranges (for example, greater than 10), although there is no need for this in the water tower project.

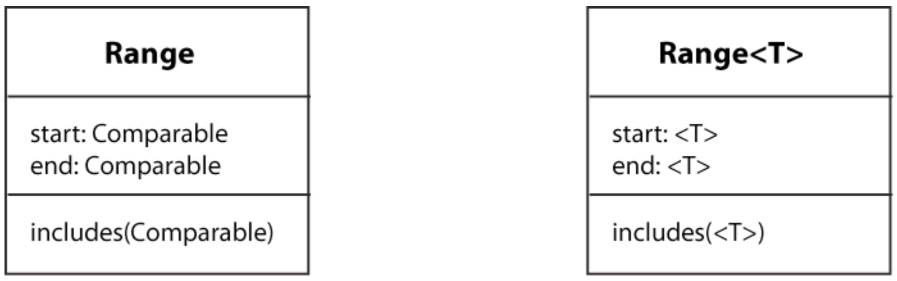


Figure 4 Range pattern

3.4 Performance

The use of these three patterns results in a couple of small objects (one for each range and dimension quantity variable; only very few different units are used in the example project). It is a valid concern whether this would lead to reduced performance. However, in the context of this paper, this has not been researched. Fowler[1] argues however that so far no reporting has been made of performance being an issue.

4 Conclusion

As we have tried to show in this paper, analysis patterns are not the high-level mumbo jumbo that is only of use to everyone but the developer of software. When the language provides object-oriented abstractions, analysis patterns can provide easy solutions to real developer problems or at least allow a more natural handling of everyday constructs.

We have used the Quantity, Conversion Ratio and Range pattern to illustrate how they can be of benefit in an example project. Since the patterns discussed in this paper were small, they provide a gentle entry point for developers to get to know analysis patterns, and explore the more elaborate ones at their own pace.

5 Literature

[1] Marting Fowler, "Analysis Patterns – Quantity", http://martinfowler.com/ap2/quantity.html, visited 2009-03-11

[2] Nico Deblauwe, Jacques Tiberghien and Alain Barel; "A water tower to help future engineers choosing their specialisation", 2005

[3] Haitham Hamza, Mohamed E. Fayad; "Model-based Software Reuse Using Stable Analysis Patterns"

[4] Marting Fowler; "Analysis Patterns, Reusable Object Models", 1997, Addison Wesley

[5] Eduardo B. Fernandez, Xiaohong Yuan; "Semantic Analysis Patterns"

"[6] Joel Spolsky, "Making Wrong Code Look Wrong", http://www.joelonsoftware.com/articles/Wrong.html, visited 2009-03-11

[7] Martin Fowler, "Analysis Patterns – Range", http://martinfowler.com/ap2/range.html, visited 2009-03-11