Programming in the 21^{st} Century:
Introducing Mathematical Methods

ISO8583 Validation

TALK NOTES

Abstract. In this notes I'm showing a realistic example of the effectiveness of maths in software engineering by tackling the validation of ISO8583 messages both using object-orientation and algebraic modelling. The material here can be used to give a 20 minute tech talk in the context of the Mathematical Methods presentation—i.e. it can serve for the in-depth tech discussion mentioned in the talk preview. This is more advanced than the "string reverse" example and is only suitable for a "smart" technical audience, but it does give a good idea of the kind of thinking behind the parsing in Eezy8583 and so should convey a better appreciation of how those seemingly impossible numbers shown in the comparison chart are actually attainable.

Andrea Falconi
andrea.falconi@gmail.com

June, 2012

Problem Statement

Let's talk about validation. Why validation? Because it's something programmers have to deal with on a daily basis and has a deep impact on the system architecture, so it affects both the quality and the effort required to deliver the product.

In particular, we're going to talk about validation of ISO8583 messages. ISO8583 is a protocol for the specification of electronic transactions within a payments network; financial institutions around the world carry out electronic transactions by exchanging messages as dictated by the ISO8583 protocol. Each ISO8583 message carries a sequence of fields and each field is specified by means of

Content Indicators:
What kind of content is allowed in a field: A (alphabetic), N (numeric), S (special), P (padding); they can be combined to specify mixed content.

Length Indicator:
How many characters to expect: either the field has a fixed length of k characters or has a variable length, in which case at most k characters are allowed—k being a given positive integer in both cases.

Examples

- $A5$: A fixed alphabetic field of exactly 5 characters.
- $N.9$: A variable-length numeric field (the dot stands for variable) of at most 9 characters.
- $AN.7$: A variable-length alphanumeric field of at most 7 characters.

In practice fields are slightly more complicated than this and messages also have additional structure; however, for the sake of this discussion we're going to run with the above and also pretend that each field is just a sequence of characters—i.e. a string. Indeed, what we're going to see would work in the "real world" too with some minor modifications. (You can have a look at the Eezy 8583 code if you're curious to see the "real thing".)

Our task is to implement a validation framework which allows to validate any message field to its field specification as given by its content and length indicators.

Mainstream Engineering Approach

If we're thinking of the problem in object-oriented terms, a possible design would be the following.

We can confine all the validation code in a single component which exposes a *validation* interface to abstract out the process of validating a field. This process is encapsulated by a *validate* method, which takes a field and returns true or false depending on whether the field is valid. Concrete validation classes implement this method to provide validation for each possible field spec. So we have a *FixedA* class to validate fixed alphabetic fields; *VarA* to validate variable alphabetic fields;

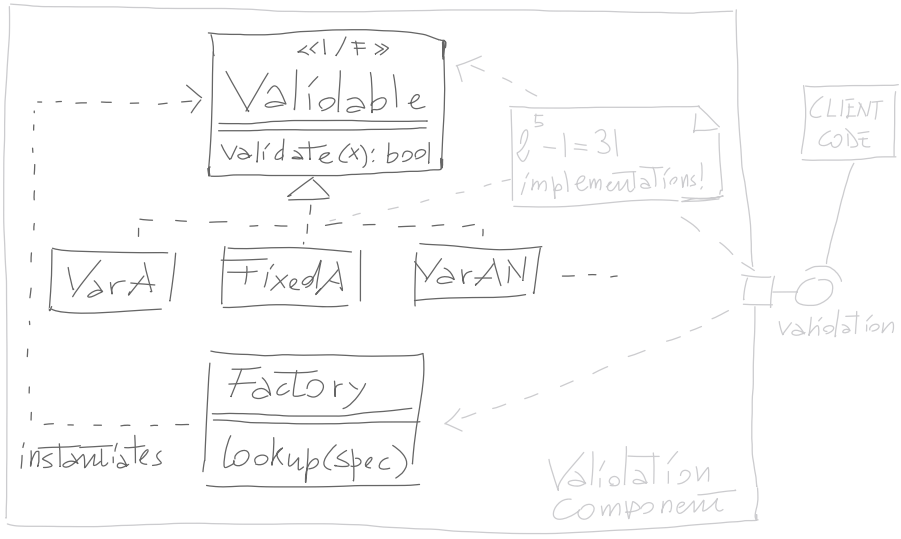

FIGURE 1. Validation component overview.

FixedAN for alphanumeric fields; and so on. A factory class allows clients to look up a suitable implementation of the validation interface given a field specification. (Implementation classes are hidden from clients.) The UML diagram in figure 1 summarises all this.

This design is modular—which is a good thing, obviously. The component's interface shields client code from the validation details which are completely encapsulated in each implementation class; also, the validation code is not scattered around the system, but sits in a single component.

The approach seems quite reasonable and indeed in practice, bearing slight variations on the theme, this is what I've seen in many code bases I worked on—for a concrete example, you can look at jPos. Like I said, some implementation details may be different (e.g. abstract base classes to factor out commonality across validators) but conceptually the approach remains the same: *Enumerate all possible validators and classify them.*

Let's try to gauge the implementation effort now. We're going to need to write validation code in correspondence of each combination of content indicators; each of those combinations will have to cater for fixed and variable length fields. All together we have $2^5 - 1 = 31$ validators to write then! We're in for at least a couple of weeks of development—assuming we call the task "done" only after we've tested the code thoroughly and QA have put it through their paces. Again, note how this is a consequence of object-oriented thinking which sort of lends itself to an *enumerate-and-classify* approach to problem solving.

Algebraic Approach

Can we find a design that requires sensibly less development effort, say hours instead of weeks, without compromising on quality and at the same time maintains modularity? The answer is yes if we stop thinking objects and study the problem mathematically instead.

A fresh look at validation. First of all, a way to dramatically reduce effort would be to avoid having to explicitly write any of those validators whatsoever. We'd like to come up with some sort of process that given a field spec produces the corresponding validator; in mathematical terms, we're looking for a function

$$\Gamma \ : \ \mathit{spec} \ \mapsto \ \mathit{validator}$$

where *validator* is a function $\mathit{String} \to \mathit{Bool}$ obviously. If you studied comp-sci, you could think of Γ as a compiler that input with a field spec (e.g. $A5$) outputs the code to validate to that spec (e.g. code to validate to $A5$). We'll be done as soon as we have a formula defining Γ.

At first glance, coming up with a definition for Γ seems quite a brain-teaser for we need to define a function that defines other functions! But this is where mathematical thinking really shines allowing us to see more clearly to the heart of difficult problems and to come up with simple solutions. If we are to define validators we need to understand what they're made of and how their components are structured; so an algebraic approach pays off in this case:

- ☐ What are the **essential building blocks** of the things we're looking at?
- ☐ How can they be **combined** to obtain larger structures?
- ☐ Can all the things we're considering be obtained in this way?
- ☐ Can those building blocks and ways of combining them be brought back to well-known algebraic systems?

Building up from examples. Let's have a look at some examples of validators to understand what their structure actually is. To validate the string content of a field s to the $A5$ spec, we have to check that each character x in s is alphabetic and that the length of s is 5. This "character check" is the function $a : \mathit{Char} \ \to \ \mathit{Bool}$ defined by

$$a \ x \ = \ \top \quad \Leftrightarrow \quad x \in \{\mathrm{a} \ldots \mathrm{zA} \ldots \mathrm{Z}\}$$

whereas the "length check" is the function $\mathit{String} \to \mathit{Bool}$ obtained by first taking the length of s and then equating this value to 5; therefore it's the composite function $(= 5) \circ \lambda$, where λ returns the length of a string. A similar argument goes for $N.9$ and $AN.7$, so in symbols we have

$$\begin{aligned} A5 \quad & (\forall x \in s.\ a\ x) \ \wedge \ (\lambda\ s = 5) \\ N.9 \quad & (\forall x \in s.\ n\ x) \ \wedge \ (\lambda\ s \leq 9) \\ AN.7 \quad & (\forall x \in s.\ a\ x \vee n\ x) \ \wedge \ (\lambda\ s \leq 9) \end{aligned}$$

In the above n stands for the "digit check", i.e. the function $Char \to Bool$ defined by

$$n\ x\ =\ \top \quad \Leftrightarrow \quad x \in \{0 \ldots 9\}$$

We see the above validators all have the same formal structure

$$f\ s\ \wedge\ g\ s$$

where f is a content check and g is a length check; both are functions $String \to Bool$. Moreover, each f is obtained by verifying that some "check" $Char \to Bool$ holds true for each character x in s

$$f\ s = \nabla\ \varphi\ s$$

with

$$\nabla\ \varphi\ s = \top \quad \Leftrightarrow \quad \forall x \in s.\ \varphi x$$

and obviously $\varphi x = ax$ for $A5$, $\varphi x = nx$ for $N.9$, $\varphi x = ax \vee nx$ for $AN.7$. Similarly, each g can be broken down into

$$g\ s = (\omega \circ \lambda)s$$

by taking ω to be the function that either equates the required field length k to the actual length of s (i.e. λs) or verifies that $k \leq \lambda s$, depending on whether the field has a fixed or variable length. So $\omega x = (= 5)x$ for $A5$, $\omega x = (\leq 9)x$ for $N.9$, and $\omega x = (\leq 7)x$ for $AN.7$.

Standing on the shoulders of Giants. At this point it should be clear that any other validator will yield the same formal structure, namely

$$(\nabla\ \varphi\ s)\ \wedge\ ((\omega \circ \lambda)s)$$

which is "assembled" using first-order logic operators ($\forall$, $\wedge$, $\vee$) and function composition from these simple building blocks:

- "character check" functions $Char \to Bool$; there are four in total: a, n, s, and p—one in correspondence of each content indicator.
- "length checks" $String \to Bool$; there are two in total: one for fixed and one for variable length.

Now we can use some basic facts from abstract algebra. First of all, the Booleans form a monoid under both $\wedge$ and $\vee$; secondly, if M is a monoid, its monoidal structure can be lifted into the function space $X \to M$, X being any given set. Specifically, we can "and" and "or" any two functions $f, g : X \to Bool$ by defining, respectively,

$$(f \mathbin{\owedge} g)x = f\ x\ \wedge\ g\ x$$

and

$$(f \mathbin{\owedge} g)x = f\ x\ \vee\ g\ x$$

All this gives us a concise way to define our Γ in a single, beautiful equation

$$\Gamma\ spec = (\nabla\ \varphi)\ \mathbin{\owedge}\ (\omega \circ \lambda)$$

where ω, as we've seen already, is defined by cases

$$\omega = \begin{cases} (= k) & \text{if spec is fixed } k \\ (\leq k) & \text{if spec is variable } k \end{cases}$$

and φ is the function obtained by mapping the content indicators $\langle t_1, t_2, t_3, \dots \rangle$ of the input spec to the corresponding character checks $\langle c_1, c_2, c_3, \dots \rangle$ and then taking their functional "or", i.e.

$$\varphi = c_1 \ \oslash \ c_2 \ \oslash \ c_3 \cdots$$

For example: $ANP \mapsto \langle a, n, p \rangle \mapsto a \oslash n \oslash p = \varphi$.

Conclusions

The entire design is captured by a single, beautiful equation

$$\Gamma \ spec = (\nabla \ \varphi) \ \oslash \ (\omega \circ \lambda)$$

and this shrinks down our development cycle to hours instead of weeks, while at the same time delivering the highest product quality. Compare this with the object-oriented approach and you can see immediately why maths really shines: it allowed us to see more clearly to the heart of this validation domain and to come up with a simple solution.

Indeed, developing software is an intellectually demanding enterprise which requires creativity, abstraction, and logical thinking. The intellectual "tool box" we use to formulate problems, devise solutions, and verify that what weve done is correct deeply affects the outcomes both in terms of development effort and quality of the product being delivered. In this regard, mathematical modelling is far more effective than mainstream engineering techniques—object-orientation, test-driven development, etc. Hope you have come to realise the effectiveness of maths is not even up for debate—or try to convince a physicist to use object-orientation to model the world...

Notes

This entire discussion can fit nicely on a whiteboard, as shown in figure 2.

Also, this is the same discussion we had in session 4 and 5 of Workshop IV of the functional programming course at ACI; see emails for the details. (Some of the text here is a copy & paste from those emails, some other text has been omitted and some added.)

FIGURE 2. Whiteboard presentation.