

## Input Validation

AAUP Conference 2008

Text to accompany slides presented by Chris Cosner in the "How Meta is Your Data?" panel.

### [Slide 1]

I've been thinking about input validation a lot lately as we've cleaned up our data for ONIX. There are certain things you find you just cannot fix at the time of export. They need to be resolved up front, when the user is entering data.

The 'concrete' thing I would like to convey in this conversational format are some patterns of system design that should make data validation more intuitive and easy to accomplish in a practical time frame with limited resources. If you're not a techie but need to communicate with consultants or your IT people, then reviewing and better understanding these concepts will improve your feature requests and communication with your programmers. I hope to also convey some idea of the complexity of input validation so that if you are not the programmer implementing it, you will at least appreciate how difficult it is to get right.

### [Slide 2]

So why validate input? We as publishers can point to title information, specifically in ONIX format, as a data set that needs to be as clean and accurate as possible. I'm not going to talk about security in the context of input validation as that is a completely different topic.

### [Slide 3]

The most basic form of validation is to make the input conform to a specific type. Most of you are familiar with basic data types. These exist to optimize a program's treatment of information. I'm not really going to talk about that either.

### [Slide 4]

The great Russian novelist Leo Tolstoy wrote that all happy families are the same, but that unhappy families are each unhappy in their own way. Something similar can be said about data--it's easy to define what is good, and difficult to enumerate or predict the many ways it can be destructive.

What are some ways bad data gets into a system?

- Operator Error
- Importing data
- Batch changes by admin
- copy paste
- privileges (power user supposed to know better)
- invisibility (allowing NULL / empty when you shouldn't)
- buggy programming ("fail-safe systems fail by failing to fail-safe" -- John Gall.) sounds like a joke, but realistically, if you install/write fail-safe mechanisms, you've pushed the point of failure to those mechanisms, whether it truly originates with them or not.
- Legacy data grandfathered in -- too much trouble to fix, or we'll get to it later

### [Slide 5]

We sometimes speak of programs as having architectures, and one architect in particular has in fact inspired programmers. In 1977 an architect by the name of Christopher Alexander published a book called ***A Pattern Language: Towns, Buildings, Construction*** in which he put forth the idea of a pattern language. One might say that Alexander came up with the most pedantic way to write architectural advice. However, as dry as some of the patterns are, they can be quite insightful. In effect he came up with a way to write logical

and effective rules for best practices that are loose enough to allow the implementer to adapt easily to circumstances.

#### **[Slide 6]**

Inspired by Alexander's idea of pattern languages, a programmer named Ward Cunningham came up with the CHECKS pattern language of data integrity, which is essentially advice on input validation. I've found his patterns to be useful, if only insofar as they really do represent how broad and difficult the problem is.

#### **[Slide 7]**

There was a time when programmers conserved every possible bit. Abbreviations, codes, and shortened versions of words abounded in order to save disk space and speed up queries. Cunningham's first pattern basically says "Welcome to the 1990s" We have disk space and CPU to spare, so use the whole value if possible. Not an abbreviation, code, or acronym.

For example, Stanford uses full-length descriptions for subject, series, and imprint, paired with the codes, and separate the code and description behind the scenes for export. However, we still use acronyms for our discount codes, which goes against this rule.

#### **[Slide 8]**

Another name for a bad piece of data is an "exceptional value". In spite of your best efforts bad data will get into your system. So your system should explicitly handle these "exceptional values" in an appropriate manner.

Possibilities include:

- Removing the value
- Adding a message to the record
- Flagging the record as incomplete
- Replacing the value instantly

Exceptional value handling is a safety valve. Putting safety valves everywhere is somewhat impractical, so you may have to limit it to the fields that really count.

#### **[Slide 9]**

Meaningless Behavior is the flip side of exceptional value handling. Rather than go to the trouble of guessing the nature of likely exceptions, do nothing if the input doesn't match the criteria for being "good".

The result is a field that won't fill, a screen that won't update. No error message or explanation, just flat refusal.

This has a certain elegance to it, but cannot be applied across the board.

This pattern rightfully goes after nagging on-screen warnings, as well as systems that do too much to help the user.

#### **[Slide 10]**

The idea here is to defer validation when possible.

Stay out of the way of the user and separate the task of data input from proofreading, as it were.

Personally I think the instances when one would use deferred validation to good advantage are fewer than those in which you validate field by field.

A brand new record is a good point to defer validation, because the user needs to fill multiple fields and should not be interrupted until the record needs to be committed.

**[Slide 11]**

One of the goals of input validation is also to educate and train users. Showing them the implications of actions gives them insight into the how the various parts of the system tie together.

Instant projection is very common these days. All it means is that when the user updates one figure, you see the results immediately in other calculated values.

Hypothetical publication is a type of deferred validation. In a complex system where you need to be very certain of the results of committing the data in a particular record beforehand, hypothetical publication pretends to commit the record and reports on what the effects would be in other parts of the system.

One way of showing implications that we have used to good effect is the sandbox record. A user can spin off a sandbox record from the budget whenever they like. This avoids a lot of trail and error inputs, which are essentially bogus.

**[Slide 12]**

A diagnostic query takes you out of the cozy, mediated user interface and shows you all the relevant fields for what you are working on.

Not only does this give power users information they want, it can speed troubleshooting.

I've tried to set up such 'advanced' queries for users in the past and only succeeded in confusing them. They generally do not want to see behind the curtain.

**[Slide 13]**

This one can be dangerous.

The user types one thing, and the system transforms it.

A good example is a phone number converted into a regular format.

When we first set up our ONIX feed I had quite a time getting the filter right on our promotional copy, text that includes reviews, description, and author biographies. People paste in text from word documents that contains hidden characters, as well as curly quotes, and sometimes characters that are simply outside of the character set accepted by Eloquence.

**[Slide 14]**

Unless you really have a lot of time on your hands, you probably don't want to write your own pattern language.

You can however document your data structures. At the very least, a data dictionary can be extremely helpful. It serves as a glossary of the fields in the system. In getting started, you can limit yourself to sets of crucial fields and build out from there.

Say you have your data dictionary in a spreadsheet. You know there are 100 fields. Once a

week, go to a coworker and ask them to pick a number between 1 and 100. Then go to the corresponding field and ask yourself, "What if there was a problem with the data in this field?" This gives you exercise, interpersonal interaction, and peace of mind.

### **[Bonus - no time in presentation :)]**

#### **Choosing a validation method**

All corrective actions have their faults, but you can determine a best action. Given the context of the input, and the expected usage of the data, prioritize the traits below and choose a method that best fits. Obviously, you may describe your priorities differently.

- Does not interrupt natural workflow
  - There are natural pauses in data entry, such as at the end of a line item in a list, or when completing a form, at which point it is optimal to interrupt.
- Is consistent with the interface
  - Users prefer the familiar, so limit the types of validation you use, especially within discrete contexts.
- Is difficult to circumvent
  - If an input validation is easy to circumvent or 'optional' it is not validation.
- Uses positive reinforcement
  - A single "Success" or " Completed" message can be more effective than multiple error messages that interrupt the workflow.
- Increases portability of data
  - If the data is consistent, it at least can be transformed as needed.
  - However, if data actually conforms to a common standard, it is far more likely to be capable of unanticipated transformations. It is more future-proof.

#### **Links:**

Validation in general:

[http://en.wikipedia.org/wiki/Data\\_validation](http://en.wikipedia.org/wiki/Data_validation)

The CHECKS Pattern Language of Information Integrity:

<http://c2.com/ppr/checks.html>

Character Encodings: <http://www.sitepoint.com/blogs/2006/03/15/do-you-know-your-character-encodings/>

Cory Doctorow 2001 essay on "Metacrap"

<http://www.well.com/~doctorow/metacrap.htm>

FileMaker plug-ins:

Regular Expressions: <http://jensteich.de/regex-plugin/>

PHP functionality, including regexes: <http://www.scodigo.com/products/smartpill-php>

MS Excel validation tips:

<http://www.contextures.com/xlDataVal01.html>