The
CHECKS
Pattern Language of Information Integrity

by
Ward Cunningham
Cunningham & Cunningham, Inc.
Portland, Oregon

Copyright (c) 1994, Ward Cunningham
All Rights Reserved

Any program that accepts user input will need to separate good input from bad, and to make sure little of the latter gets recorded. This pattern language tells how to make these checks without complicating the program and compromising future flexibility.

The language has eleven patterns presented in three sections. The first section describes values as they should be captured by the user-interface and used within the domain model. The second and third sections discuss detecting and correcting mistakes, first during data entry and then after posting or publication. The patterns draw from the author's experience developing financial software in Smalltalk. They are written as if part of a larger language and therefor may seem sketchy or incomplete. This paper is as much an experiment in the selection and linking of patterns as an attempt to communicate practical knowledge.

Section 1. First consider quantities used by the domain model. Your domain code must express the "logic" of the business in its richest and often illogical detail. Every clause of every statement should be motivated by some business fact of life. Other concerns will be pushed into the specialized values of this section or pulled out into objects described later. The patterns are:

1. Whole Value
2. Exceptional Value
3. Meaningless Behavior

---

1. Whole Value

Beside using the handful of literal values offered by the language (numbers, strings, true and false), and an even smaller complement of objects normally used as values (date, time, point), you will make and use new objects that represent the meaningful quantities of your business. These values (values like currency, calendar period or telephone number) will carry whole, useful chunks of information from the user-interface to the domain model.

When parameterizing or otherwise quantifying a business (domain) model there remains an overwhelming desire to express these parameters in the most fundamental units of computation. Not only is this no longer necessary (it was standard practice in languages with weak or no abstraction), it actually interferes with smooth and proper communication between the parts of your program and with its users. Because bits, strings and numbers can be used to represent almost anything, any one in isolation means almost nothing. Therefor:

Construct specialized values to quantify your domain model and use these values as the arguments of their messages and as the units of input and output. Make sure these objects capture the whole quantity with all its implications beyond merely magnitude, but, keep them independent of any particular domain. (The word value here implies that these objects do not have an identity of importance.) Include format converters in your user-interface (or better yet, in your field and cell widgets) that can correctly and reliably construct these objects on input and print them on output. Do not expect your domain model to handle string or numeric representations of the same information. Consider:

```
the message: contractDurationInDays
answers: 21

the message: contractDurationInDaysAsString
answers: '21'
```

Both of these messages use wordy protocol and answer quantities devoid of meaning once returned. The following simpler message returns a whole value with meaning in isolation.

```
the message: contractDuration
answers: Weeks(3)
```

You will find that these objects will capture some of the irregularity and (possibly) ambiguity of the domain model. Expect particular classes to grow into hierarchies over time. But, do not extend whole values to include non-applicable or exceptional quantities better represented by an Exceptional Value (2). Also, avoid undue reasoning regarding inappropriate combinations of values so long as Meaningless Behavior (3) will eventually result.

2. Exceptional Value

A business model will normally be composed of a basic case or abstraction that is specialized and/or refined to capture the diversity present in the business. However, there will often be circumstances where the inclusion of all business possibilities in the class hierarchy would be confusing, difficult or otherwise inappropriate. You will, at times need to extend the range of an attribute beyond that offered by a WholeValue (1).

Consider the poll taker who collects answers of the form "agree", "strongly agree", etc. Answers that defy quantification, like "illegible" or "refused" are better represented outside the range of values, no matter how fuzzy it may be. However, the structure of a domain model had best leave a place for this sort of missing data for it may appear later. In fact, missing values are impossible to avoid during the creation (data entry) of all but the most trivial domain models. Therefor:

Use one or more distinguished values to represent the exceptional circumstances. Exceptional values should either accept all messages, answering most with another exceptional value, or reject all messages (via does not understand) with the possible exception of identifying protocol like isNil or printOn:. Interface widgets should produce nil on blank input and produce blank given nil output. Domain models should accept nil or other exceptional values as legal input, at least temporarily. It is possible to make refinements of UndefinedObject that can carry an explanation. If you do, note that aValue == nil no longer means the same thing as aValue isNil.

```
purchaseDate
	^ (buys := self trades select: [:each | each isPurchase]) size <= 1
		ifTrue: [buys first settleDate]
		ifFalse: [ExceptionalValue reporting: 'various']
```

Exceptional values will simplify the domain model hierarchy and method structure. It should not be necessary to explicitly test for exceptional values in methods because they will either absorb messages or produce Meaningless Behavior (3). The little exceptional value handling that is required can be concentrated extremely close to the user interface. For example, the report writer needs to detect exceptional values to correctly compute a weighted average. Properly objectified, the WeightedAverageColumn object can perform this computation on behalf of any domain object and thereby separate concerns. Deferred Validation (6) is responsible for testing completeness, recognizing that Hypothetical Publication (7) may not require completeness.

3. Meaningless Behavior

Given that the Whole Values (1) used to quantify your business logic will exhibit subtle variations in behavior and that Exceptional Values (2) may appear throughout the computations, it is possible that the methods you write will stumble in circumstances you cannot foresee.

Keep in mind that the rules of business apply only selectively, and that evolution of business practice wiggles around even those rules that "must" apply. In your domain models you are chartered to express business logic with no more complexity than originally conceived or currently expressed. Therefor:

Write methods without concern for possible failure. Expect the input/output widgits that initiate computation to recover from failure and continue processing. Output will remain blank because any other output would be an attempt to attach meaning to meaningless behavior. Users will interpret unexpected blanks to mean that inputs do not apply and/or outputs are unavailable.

Trying to be meaningful:

```
weightedAverageCost
	| total weight |
	total := self weightedTotalCost.
	total isCurrency ifFalse:
		[^ExceptionalValue reporting: 'N/A'].
	weight := self totalWeight.
	(weight isNumber and: [weight isZero not]) ifFalse:
		[^ExceptionalValue reporting: 'Empty'].
	^total / weight
```

Accepting possible meaninglessness:

```
weightedAverageCost
	^self weightedTotalCost / self totalWeight
```

Note: some readers have assumed this pattern to be about writing error handlers. It is not. It is about writing domain methods in the presence of diversity. It does assume a near trivial error handler to be in place in the input/output system which is not always the case.

You can view meaningless behavior as an alternate implementation of Exceptional Value (2) and in many cases the two are indistinguishable to your user. Choose meaningless behavior unless a condition can be anticipated and has domain meaning (as opposed to merely operational meaning such as not-yet-filled-out). At times there may be something very wrong inside the program so it is important that some clues surface. Echo Back (4) exposes failure by echoing blank. Input screens that report Visible Implications (5) can expect serious trouble to blank them too. Deferred Validation (6) should demand meaningful behavior where corruption of records is the alternative.

Section 2. A person reaches through a program's interface to manipulate the domain model. Although the interface is itself a program (an interface model and graphical machinery), its purpose is to enable the direct manipulation of the domain model as transparently as possible. The user interface is programmed to create the illusion of control in the mind of the user. To this end it must provide sufficient clues of the model's state so that sensible operation is the norm. These patterns offer the required feedback:

4. Echo Back
5. Visible Implication
6. Deferred Validation
7. Instant Projection
8. Hypothetical Publication

---

4. Echo Back

Field and cell widgits will be able to construct and deliver WholeValues (1) to the domain model. As is normal practice in object-oriented programming, the domain model is free to extract, interpret or reject any information it is presented. This pattern considers the domain model's modest obligation to explain such selection.

You can expect values to be entered in small batches followed by a quick review looking for transcription or typing errors. This cycle repeats but not always with the same batch boundaries. Since you will have no way of knowing exactly when a review takes place (it can be just a quick glance) you must inform users of their success entering values as each field or cell is processed and you must provide this information without disrupting the cycle when problems occur. Therefor:

Provide for the read back of any information written into a domain model. Expect field and cell values to be retrieved and echoed immediately after each entry. Answer reconstructed values using a protocol trivially derived from the original. Note: the usual getters & setters convention (i.e. attribute and attribute:) meets this requirement though the requirement applies to more than just attributes. The full turn around echo of entered values allows the user to observe any selection or interpretation of entered values.

Do not expect the domain model to explain its interpretation of marginal or incorrect values through notifiers. Such initiative on the part of the model is misplaced because it breaks the small batch entry behavior. For example, consider the entry of a pay date:

user types:	5/8/94	
echo back:	05/08/94	(the whole value May 8th, 1994, standard format)
user types:	5/5/94	
echo back:	05/08/94	(model has chosen nearest payday, always a Friday)

The leading zeros appear for the simple reason that the original six characters entered are discarded and replaced with the standard print representation of the date echoed back by the model. In the second case the actual numbers are different because the model has chosen to handle bad input by choosing to store the closest legal input in its place. The echo back makes this choice visible. Equally justifiable alternatives would be for the model to ignore the bad value or accept it unconditionally (possibly as an Exceptional Value (2)). Our point here is not that one choice is better than another but that the choice once made must be visible to the user without disruption. Following this point of view further, meaningless setters would be ignored and meaningless getters would print as blank.

This pattern counters the common practice of ringing bells and flashing lights at the first sign of trouble. You will have plenty of opportunity to protest bad values in Deferred Validation (6).

5. Visible Implication

By combining the mechanisms of Echo Back (4) with methods that compute attributes we simplify some entries and improve the effectiveness of visual review for others.

People find many ways to quantify the things that matter. These measurements often duplicate each other with only a portion required to completely characterize the thing. Often there is a sense that some values are more fundamental than other, derived quantities. Other times the duplicate measurements are simply another way of looking at the thing. Therefor:

Compute derived or redundant quantities implied by those already entered. Display the computed values in fields or cells along side those that are changed. Where possible, allow a derived quantity to be entered and work backwards to compute the more fundamental measurements. Where the choice of fundamental measure is ambiguous, choose the unspecified over the specified or the more variable over the less variable.

given quantity: 12
and unit price: 6.50
compute total: 78.00

given quantity: 12
and total: 72
compute unit price: 6.00

given unit price: 7.00
and total: 77.00
compute quantity: 11

Perform this logic in the accessing methods of the domain model. Write getters that try to compute missing values from other inputs.

```
unitPrice
   ^ unitPrice isNil
      ifTrue: [total / quantity]
```

Or, write setters that transform redundant quantities into the chosen fundamental measurements.

```
total: aValue
   unitPrice isNil
      ifTrue: [unitPrice := total / quantity]
      ifFalse: [quantity := total / unitPrice]
```

Keep the domain model's implication logic simple. Also, do not try to encode field dependencies into the user interface. Instead simply refresh all fields when any one changes.

You can expect Meaningless Behavior (3) whenever a thing is incompletely specified. Expect these to show as blanks that will reinforce the incompleteness in your user's mind. Be sure all implications can be computed in a small fraction of a second. Longer calculations, or those unsafe to perform on partial specifications are best left to Instant Projection (7).

6. Deferred Validation

The Whole Values (1) that quantify a domain model have been checked to ensure that they are recognizable values, may have been further edited for suitability by the domain model and have been Echoed Back (4) to the user. All of these checks are immediate on entry. There is, however, a class of checking that should be deferred until the last possible moment.

As your user completes a series of entries there will come a point where more extensive action by the computer is expected. This may be a simple query, "how am I doing"; a pause in activity, "I'll finish this tomorrow" or a change in responsibility, "you take it from here". The exact integrity needs will not be known until the computers action is called for. Therefor:

Delay detailed validation of a domain model until an action is requested. Taylor the extent of the validation to the specific action. Saving incomplete work in a private location will not require as much validation as posting finished work in a public place. Validation checks may (but don't always) form a structure with more complex actions requiring all of the checks of simpler activities and then some. Write methods for your domain model that encode the anticipated use. Have these methods delegate to simpler validations before making their own checks. Checks should be made in passes so that the most specific problems are reported first. Check to make sure required quantities are present before checking that they and others are consistent.

```
validateForPublication: notificationHandler
   self validateForSave: notificationHandler.
   self validateForComputation: notificationHandler.
   self validateVariablesForPublication: notificationHandler.
   self validateRelationsForPublication: notificationHandler
```

Expect the individual validation methods to become complex and be subject to regular modification. As such, they may invoke systems designed specifically to validate business rules and restrictions. Check the obvious before delegating to these systems so that their rule bases are not polluted with trivial (but necessary) checks.

Deferred Validations are hurdles over which domain objects must pass on their way into the more public portions of a system. The system demands validation so that problems can be brought to the user's attention before publication. The user, on the other hand, may be aware of potential problems beyond those detectable by any strict validation. Instant Projection (7) and Hypothetical Publication (8) offer the user two related strategies for further assessment of entered information.

7. Instant Projection

As you collect information for future use, you have displayed the Visible Implications (5) of the entries in the hope that inconsistency will be noticed. Now, as your user's attention shifts from entry to publication, you will want to carry further your prediction of that publication's impact.

We collect information so that we can use it later on. How it gets used will depend on future events that we cannot predict. Even so, we have a notion of a likely chain of events and carry expectations as to how things will probably work out in the end. Therefor:

Offer to project the consequences of any publication before that publication is actually made. You may require the entry of additional assumptions or offer alternatives in forecasting technique. Expect the interpretation of a projection to take some effort. Consider opening a "projection window" with various tabulations and summaries. Obsolete tabulations will be useful for observing parameter sensitivities.

A projection anticipates a question likely asked of a soon to be published domain model. Asking the question has no impact on the system other than reporting the answer. In a situation where questions concern multiple publications, Hypothetical Publication (8) offers a versatile, though more cumbersome, alternative to projection.

8. Hypothetical Publication

A complicated domain model might pass all Deferred Validations (6) required for publication but still be in doubt if the risks of publication are high. The risks may come more from taking an action described by the model than from incorrectly describing an action already taken. This is all the more reason to consider a most elaborate mechanism for detecting mistakes.

When one publishes (or posts, or otherwise completes entry of) information, that information is expected to travel to many destinations. It may be difficult to access the full impact of any piece of information out of context and independently of other, also questionable, data. Therefor:

Allow your user to make any number of hypothetical publications that can be released into the system in a controlled way. Limit their distribution to subsystems on the users own workstation or clearly mark them as tentative for other users. Provide mechanisms to retract hypotheticals individually or en mass.

Hypothetical Publication can substitute for Instant Projection (7) when forecasting tools are available for published models. Hypotheticals are also useful when large quantities of historical information must be entered and checked.

Section 3. Now consider mechanisms that address the long term integrity of information. Both patterns address one form or another of questionable information. They recognize that many quantities in business are ambiguously specified or otherwise open to interpretation. These patterns lead up to the treatment of accounting integrity which, unfortunately, has been omitted from this language. The patterns are:

9. Forecast Confirmation
10. Diagnostic Query

---

9. Forecast Confirmation

Events in the world often run ahead of those modeled in a computer system. When events can be anticipated, it makes sense to mechanically generate their models and to publish them for public use. However, when the computer system does catch up with reality, it is important that reality is accurately modeled. Therefor:

Provide a mechanism for adjusting and confirming values associated with mechanically published events. Consider the sequence:

Thursday: we predict an automatic deposit of 187,655.47 for Friday
Friday: we mechanically post 187,655.47 to the cash account
Monday: bank records show 187,655.50 was deposited on Friday, we adjust accordingly
Later: records for the month are closed showing no unusual activity

What is important here is that the best information was available at every moment even though no one was technically accountable for the posting until after the fact. Forecast Confirmations look like original entries from the point of view of accounting integrity.

Forecast Confirmations only apply to mechanically generated models. Once confirmed, the model's values become subject to accounting integrity.

10. Diagnostic Query

Whole Values (1) and Exceptional Values (2) accumulate useful information that may not always be visible in the user-interface due to rounding or other simplification. That information and more must be retrievable from the point where values are displayed.

As the various records of business activity are cross-checked, disagreements are sure to arise. Much published information will be in doubt until the nature of the error is determined. Tracking down a recording error is substantially different than observing the operation of normal business. However, if we turn to completely new observation mechanisms, we loose track of the familiar making for tedious work. Therefor:

Incorporate mechanisms for the diagnostic tracing of every value in the system. Make every display that rounds or summarizes offer the unprocessed values for inspection.

Normal display:	67%
Diagnostic display:	66.6454329

Normal display:	652 MM USD EQV
Diagnostic display:	622,456,325.07 USD + 3,624,878,450 JPY + 23,549.54 FRF

Likewise, where rules and formulas have been applied, make these retrievable from the system itself and format them with variable names and the values bound in the particular calculation. Since the trace will ultimately lead to value entry, make sure you can report the date, time and identity of the source.

Normal display:	22%
Diagnostic display:	ROR = 22% = internal rate of return (P0, P1, CFi) P0 = 32,454.55 = market value (1/1/93) P1 = 36,537,39 = market value (12/31/93) CF1 = 354.00 = cash flow (3/15/93, #1000324) CF2 = -400.00 = cash flow (7/31/93, #1000378) CF3 = -100.00 = cash flow (8/30/93, #1000412)

The correction of input errors offers another source for diagnostic information. Prior values and the time and identity of all sources should be available to diagnosis.