Two-Faced Data

Vadim Zaytsev, vadim@grammarware.net
Raincode, Belgium

Intent

One data fragment has several alternative structural representations tailored toward specific data manipulation approaches.

Also Known As

- Concrete Syntax and Abstract Syntax
  - simplifying concrete syntax to abstract syntax [20, 58]
  - parsing [44], more than parsing [24, 30], parsing in a broad sense [61]
  - object grammars [32]
- Interparadigmatic Data Binding
  - COBOL — OO — Relational databases — XML [35]
  - OO — Relational [13]
  - CRUD — OO [49]
- XML Data Binding
  - XML to Java [37]
  - XML to Haskell [6]
  - XML to C# [38]
- GUI Data Binding
  - generic GUIs [1]
  - WebSocket-based data binding [23]
- Intermediate Representation
  - support imperative and declarative idioms [34]
  - multiple languages within one paradigm: FP [27], OO [11]
  - implementation-geared [5, 42]
  - validation-geared [9] and analysis-geared [22, 31]
- Views
  - integrated personalised views in databases [48]
  - model views [3, 4, 10]
  - view-based software engineering [7, 46]
  - architectural views [15, 51, 59]
Motivation

When modelling or programming, people tend to think in terms of conceptual constructs: “inheritance” (of classes), “conformance” (of models to metamodels), “conditional statement” (programming), “input” (data flow, side effects) and others. In practice these conceptual entities are represented as concrete elements: in textual form, in graphical diagrams, in memory blocks, etc. Since the actual solution has to be expressed in such elements, this notation exposed to the language end user, has great impact on the effectiveness of both the solution and the process of modelling or programming.

Results from ontological analysis tell us that a mapping between a modelling notation and an underlying domain model (in SE usually the Bunge-Wand-Weber ontology [57]) should be bijective [39,39,40] to avoid the following issues:

- **Construct deficit**: when something that exists in the ontology (i.e., in the mind of domain experts), has no counterpart in the modelling notation. Notations with construct deficit are called *incomplete* and have their place in environments that are deliberately limited for reasons of security or (sub)domain-specificity.

- **Construct redundancy**: when one conceptual entity can be modelled with more than one notational construct that are identical or subtly different from one another. Notations with construct overloaded are called *unclear* and are advocated by ontological analysts to be defective. Construct redundancy in programming languages often leads to discussions of taste and conventions being imposed on top of the language. For example, a functional language called Haskell [36] supports comprehensions and higher order functions equally well, so \( \text{map } (\lambda x \rightarrow x^2) \ 	ext{xs} \) is equally acceptable, equally performant and equally maintainable as \( [x^2 \mid x \leftarrow \text{xs}] \), and the choice is up to the particular programmer. Other functional languages like Rascal [29] have better support for comprehensions than for explicit mappings, so the choice there has farther going consequences, known only to programmers that reached certain affinity with the language at hand.

- **Construct overload**: when one notational entity represents several conceptual entities. Notations with this smell are a different kind of *unclear*: they are merely slightly counter-intuitive to domain experts but give wrong impressions to those who learn the domain through this notation. A famous example nowadays is the Git version control framework that bundles unrelated functionality: for instance, \texttt{git reset} is a command that, depending on parameters, can simply “unstage” code changes (which means they will not be included in the next commit) or undo several unpushed commits or even irrecoverably wipe any pending changes away.

- **Construct excess** is said to happen when a modelling notation have elements that do not have any correspondence in the domain model. Notations frequently have construct excesses as practical shortcuts and quick hacks that
solve the problem at hand but are totally alien to the uninvolved domain experts. Excessive constructs are never “designed” into a notation but find their way into it by the time of implementation, especially under deadline pressure.

Success stories from updatable views in databases [8], synchronised model views [4], data integration [43], serialisation [16] and structure editors [25] demonstrate how it can be useful to have several systematic representations of the same underlying constructs [14]. We argue that this pattern is universal to the entire software language engineering and thus can be used across technical spaces anywhere where a language has several user groups or application varieties.

**Applicability**

Use the Two-Faced Data pattern when

- You design a software language and must provide functionality in the entire spectrum from parsing the textual input to advanced semantic consistency validation like type checking.

- If you make your grammar too close to the desired conceptual representation, you risk making it ambiguous, inefficient for parsing and/or not user friendly for the language users. Projectional language workbenches deliberately choose this path due to their naturally powerful IDE support [25, 55, 56], other approaches are filled with perils, unless they adopt the same techniques [53].

- If you make it too close to the desired way of writing and reading sentences in the language, you risk overburdening your traversals and rewritings with unnecessary details concerning a particular textual representation. Solutions without multiple “faces” usually include conventions that allow to use one representation to mean multiple things at once [28, 54] (e.g., using layout for pretty-printing but ignoring it for parsing/matching).

- You want your software language to have both textual and visual concrete syntax which are conceptually the same but technically get a different representation each. Due to the “natural” flow of the textual representation (usually left to right, character by character) and a much freer structure of the visual syntax, elements that correspond to the same entities may not only be represented differently individually, but also appear in different order.

- The need for several notations of one domain-specific language is widely known and acknowledged in practice [18, 36], but its foundations are lagging somewhat behind.

- In general textual information is perceived by humans to be more trustworthy [50] and is faster digestible [45], but with appropriate training visual notations can be more effective and maintainable [41, 47].

3
Graphical models of text that take physical distance between words into account [2] and consider the visual aspect of operations performed on text [12] are an emerging field of research without a readily available cookbook of practically mature recipes.

Structured data that you are working with, needs to be serialised — for storage, communication or backup.

Using the existing textual syntax would mean losing the structure and may imply future overhead and/or ambiguity in deserialising such data.

In practice people tend to develop a yet another format which conceptually represents the same structure of the same data, but is more suitable for marshalling and unmarshalling. Such a format can be a standalone project but usually is a sublanguage of XML or JSON.

**Structure**

![Image of a diagram]

**Participants and Collaborations**

The same language (yellow box on the megamodel) can be defined by different, possibly incomplete, metamodels, and thus the models that conform to them, correspond to the same language instances, but belong to different technological stacks and thus can be effectively used with different algorithms. Functions $F_k$ are used in a broad sense and can represent true functions like sorting or traversals, as well as other data manipulation activities such as editing or validation.

**Implementation**

Consider the following implementation issues:

- If the “faces” of the data allow interaction, you need some set of bidirectional update mappings; these imply overhead which might outweigh the advantages of using the faces.
One of the “faces” can be dominant within a domain for historical reasons and so advanced that over the time it developed all necessary algorithms usually associated with other faces — e.g., concrete syntax in metaprogramming [54].

Some mapping need to bridge a semantic gap between “faces” that cannot be fully bridged — e.g., ADT vs OO [13], even though many practically sufficing strategies exist [35].

In scenarios with more than two “faces” it gets too complex to develop direct mappings for each pair; in that case it is better to consider a star-shaped infrastructure with one canonic representation which is capable of synchronising with any of the other ones.

When metamodels are well-defined and their differences are explicitly expressed, we can do coupled transformations [32] — that is, infer model-level mappings from metamodel-level ones. This has been done for various technical spaces: modelware [21], grammarware [60], databases [19], xmlware [33].

**Sample Code**

Consider the following Rascal [29] code:

```rascal
data A = foo(bool)
    | bar(set[A] xs)
;
```

It defines a piece of very simple abstract data type with two constructors. The metaprogramming facilities provided by Rascal allow us to comfortably traverse instances structured in such a way and perform computations:

```rascal
visit(T)
{
    foo(True) : cx += 1
}
```

(even more concise, `len([1 | /foo(True) := T])`), and in place rewritings:

```rascal
visit(T)
{
    bar(_) => foo(False)
}
```

However, writing them to a file can only be done in one fixed notation, and reading back will not be smooth. For such actions, we need concrete syntax — for example, this one:

```rascal
syntax A = foo: "FOO"?
    | bar: "<" {A ":"}+ ">";
```
Parsing any textual input with this concrete syntax definition is trivial in Rascal with the use of \( \text{parse(#A, ...)} \) function. The resulting trees, however, are somewhat clunky, contain too much information (who cares that we used colons as a separator? should we really update the traversal code if the separator changes in the future?) and can only be traversed in their default term form. However, there is a built-in matching function called \( \text{implode} \) that can couple the two:

\[ T = \text{implode}(A, \text{parse(#A, input)}) \]

The \( \text{implode} \) function follows grammar production alternative labels and match them to the constructors of the data type. Then, it maps the presence of \texttt{FOO} text to a true value and the lack of it to a false value of the Boolean argument of the \texttt{foo} constructor. Parse-guiding anti-ambiguity angle brackets in the concrete syntax carry no structural meaning, so they are disregarded, and the collection of inner entries is mapped to a set because that is what the abstract data type expects (it could have been mapped to a list instead).

**Related Patterns**

Adapter; Bridge; Visitor; Interpreter [17].

**References**


12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


