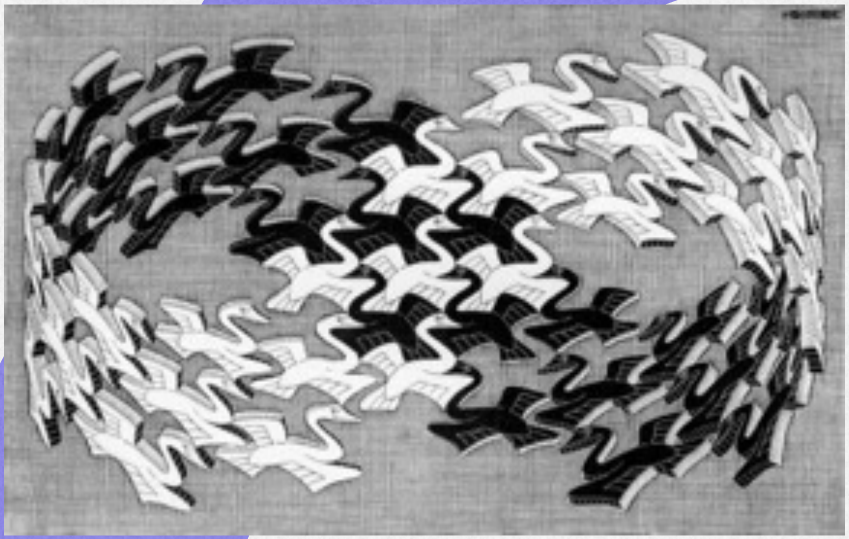
Two-Faced Data

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

Vadim Zaytsev



Foreword by Eugene Syriani
Richard Paige
Steffen Zschaler
Huseyin Ergin

Patterns in Model Engineering at STAF 2015

2015

Two-Faced Data

Vadim Zaytsev
Universiteit van Amsterdam, The Netherlands, vadim@grammarware.net

Intent

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

Also Known As

Concrete Syntax and Abstract Syntax; Data Binding; Model Views

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 [14]) should be bijective to avoid construct deficit, overload, excess or redundancy [11]. While the right for existence of ontologically *unclear* notations (with the latter three properties) is being disputed, ontologically *incomplete* notations (the ones with construct deficit) have their place in environments that are deliberately limited for reasons of security or domain-specificity.

Success stories from updatable views in databases [2], synchronised model views [1], data integration [12], serialisation [5] and structure editors [8] demonstrate how it can be useful to have several systematic representations of the same underlying constructs [4]. 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.

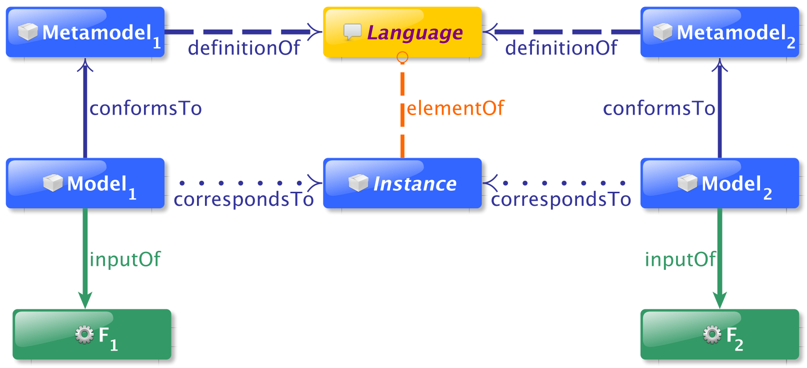
Applicability

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. 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.

– 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.
– Structured data that you are working with, needs to be serialised — for storage, communication or reserve. However, using the existing textual syntax would mean losing the structure and may imply future overhead and/or ambiguity in deserialising such data. Hence, you develop a yet another format which conceptually represents the same structure of the same data, but is more suitable for marshalling and unmarshalling.

Structure



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 [13].
– Some mapping need to bridge a semantic gap between "faces" that cannot be fully bridged — e.g., ADT vs OO [3].
– 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 [9] — that is, infer model-level mappings from metamodel-level ones. This has been done for various technical spaces: modelware [6], grammarware [15], databases [7], xmlware [10].

Related Patterns

Adapter; Bridge; Visitor; Interpreter.

References

1. M. Antkiewicz and K. Czarnecki. Design Space of Heterogeneous Synchronization. In *GTTSE'07*, volume 5235 of *LNCS*, pages 3–46. Springer, 2008.
2. F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM TODS*, 6(4):557–575, 1981.
3. W. R. Cook. On Understanding Data Abstraction, Revisited. In *OOPSLA*, pages 557–572. ACM, 2009.
4. K. Czarnecki, J. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *Theory and Practice of Model Transformations*, pages 260–283. Springer, 2009.
5. K. Fisher and R. Gruber. PADS: A Domain-specific Language for Processing ad hoc Data. In *PLDI*, pages 295–304. ACM, 2005.
6. T. Gîrba, J. Favre, and S. Ducasse. Using Meta-Model Transformation to Model Software Evolution. *ENTCS*, 137(3):57–64, 2005.
7. M. Gogolla and A. Lindow. Transforming Data Models with UML. In *Knowledge Transformation for the Semantic Web*, pages 18–33. IOS Press, 2003.
8. Z. Hu, S.-C. Mu, and M. Takeichi. A Programmable Editor for Developing Structured Documents Based on Bidirectional Transformations. *Higher-Order and Symbolic Computation*, 21(1–2):89–118, 2008.
9. R. Lämmel. Transformations Everywhere. *SCP*, 52:1–8, 2004.
10. R. Lämmel and W. Lohmann. Format Evolution. In *RETIS*, volume 155, pages 113–134. OCG, 2001.
11. D. L. Moody. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE TSE*, 35(6):756–779, 2009.
12. J. Oliveira. Transforming Data by Calculation. In *GTTSE'07*, volume 5235 of *LNCS*, pages 134–195. Springer, 2008.
13. E. Visser. Meta-programming with Concrete Object Syntax. In *GPCE*, volume 2487 of *LNCS*, pages 299–315. Springer, 2002.
14. Y. Wand and R. A. Weber. An Ontological Model of an Information System. *IEEE TSE*, 16(11):1282–1292, 1990.
15. V. Zaytsev. Coupled Transformations of Shared Packed Parse Forests. In D. Plump, editor, *GCM*, volume 1403 of *CEUR*, pages 2–17. CEUR-WS.org, 2015.