Reflections on Reflection

James O. Coplien

Gertrud & Cope
cope@gertrudandcope.com

Abstract

Though it usually makes its appearance only as a footnote in the broader discourse of object design, reflection is a recurring and sometimes noisily divisive topic in object-orientation. Glimmers of reflection pervade even the darkest corners of the tapestry of object orientation's history. In fact, the broader notion of code's self-knowledge, such as run-time method dispatch, goes to the heart of what differentiates objects from other paradigms.

Object orientation, at its roots, was about people and human mental models. It is impossible to make serious headway in these models without reasoning about the system outside of its simple imperative expression. By analogy, the silent movie era of film held that by removing speech, the media of film could both appeal to broader audiences and to tap into the broader human universals that speech obfuscates.

Programs are the silent films that connect much of humanity today. The silent experience plays out at the screen; the Internet is the deep hardware on which it runs, and our software illuminates and articulates the connections between them. To make software fulfil any social agenda of human problem-solving requires a link between the reflections of the individual and those of the software; to rise to social phenomena requires a computational model that accommodates reflection at the social and societal layers.

The DCI (Data, Context, and Interaction) paradigm provides a world model whose reflection allows program structure to shift with the dynamics in the context of application while featuring new ways to clearly present program structure to faithfully capture end-user mental models of uses cases and data. DCI and other recent post-modern approaches offer breakthroughs that raise reflection to its proper place as a first-class programming concern.

Categories and Subject Descriptors C.0 [**System Application Architecture**]

General Terms Algorithms, Design, Human Factors, Languages, Theory, Verification.

SPLASH'12, October 19–26, 2012, Tucson, Arizona, USA.
ACM 978-1-4503-1563-0/12/10.

Keywords DCI; aspect-oriented programming; reflection; object-oriented programming; use case; silent movies; social discourse.

1. The Vision

For me, the high point of the past 27 years of OOPSLA was Dave Thomas' (originally Jerry Archibald's) tutorial on "The behavior of behavior" at OOPSLA '91. George Bosworth, Adele Goldberg and other notables interjected key technical and historic insights during the talk. The talk was an example of the potential of reflection to accomplish great work with little effort, but the very "gee-whiz" timbre of the talk revealed its arcane nature. In fact reflection has had difficulty gaining a footing over the years, due in large part to a lack of understanding of how to constrain it.

We can better appreciate the need for through its namesake in the human domain, and use that understanding to create better apologies for reflection and better-tuned computational models for its application. In this talk I will show that reflection is crucially fundamental to the success of any program that enjoys use in a human context, and will suggest ways in which technology can smooth the way to better reflection in design and programming.

2. Meta is fundamental

The object vision of programming is rooted in a belief that we can go beyond formal logic to tap into human cognition, while the trappings of polymorphism and encapsulation are largely derived afterthoughts from software engineering. If we want programming languages to support human endeavour, we must revisit and reflect on how we think. Human discourse unfolds along ever deepening, alternating layers of extemporization and reflection. To communicate literally, while excising the contextual underpinnings, is not to communicate effectively. The silent movie era of film was in fact rooted in a belief that pictures opened a rich interchange that could tap into the broader human contexts that speech obfuscates, with no lesser aspiration than to head off what ultimately became World War I [7]. Written scripts underpinned the unspoken scenes; the scripts built on timeless human concerns.

Programs are the silent films that connect much of humanity today. The silent experience plays out at the screen; the Internet is the deep hardware on which it runs, and our software illuminates and articulates the connections between them. To make software fulfil any social agenda of human problem solving requires a link between the reflections of the individual and those of the software; to rise to social phenomena requires a computational model that accommodates reflection at the social and societal layers.

Hofstadter's *Gödel-Escher-Bach* [1] playfully makes a strong argument that having a concept of self is the essence of intelligence. Individual objects can represent information as data, and can interpret that data in their methods. But intelligence is more than knowledge. We need a computational model that lifts us above semantics and epistemology into hermeneutics: the ability to reason. That means reflection at the level of network computation: connections *between* objects. This doesn't mean the kind of unconstrained reflection that gave it a reputation as a "dangerous" technique, or the kinds of surprises one finds with Aspects [2]. We need a computational model whose code self-organizes around mental models. That's roughly how objects started. [3]

Human beings of all sorts figure into this issue. I as a programmer must reflect more about the interaction between users and their program and less about my interaction with the program or about the end user's interaction with me during requirements. Agile is about the latter and user experience (UX) work is about the former. Reflection is about all three together, but we have matured to deal with these issues only pair-wise.

This view of system construction implies that "going meta" is not an option, or a distraction, or a deferrable phase. Meta is where it starts. Meta provides the foundation on which non-meta stands. We need to move beyond program semantics to epistemology — a theory of knowledge — and interpretive hermeneutics. We need the meaning of meaning; the behaviour of behaviour.

Aspects were one noble attempt to open the dialog on reflection and to move away from the imperative expression of program logic to more conceptual building blocks. As has been described elsewhere, this reflects a shift from the modern school of thought to a more post-modern framework. However, Aspects were based on overly find units of behaviour, focusing on class-granularity features instead of use cases. And they create serious challenges for comprehensibility of the program flow at that.

3. Reflection and the Programming Model

Reflection is to program dynamics as architecture is to program statics. There has classically been strong focus on the design of the source structure of an object-oriented program: choosing the right classes (CRC cards, designing class hierarchies) and capturing them in the code. Most object-oriented programming languages focus on classes as their primary building block. Most object testing regimens claim to focus on testing individual classes (though the actually just test class methods).

Object-oriented programmers have long held similar models, knowing that there are general recurring properties that recur place-wise in multiple enactments of business and social transactions. Base classes have traditionally served as the home for such recurring business logic. Programmers also know that no two enactments ensue in exactly the same way. While the general form of the use case may be formalizable in closed form, its details cannot. Designers and programmers represent those details as different values in the objects involved in the computation. Some variants can best be understood by structuring the data different or by adding data values, as an investment account is different than a deposit account in the presence of an interest value. Programmers organize these data differences using classes, and organize them as hierarchies. Method specialization has followed, happily building on the basic programming language facilities for structuring data variations through inheritance, to express method variations using the same mechanism.

No one talks about the dynamics of object relationships. No one designs objects. Use cases rarely survive beyond analysis: their identity disappears, scattered across the network of interacting objects. If they are not explicit in the code you cannot understand them from the code. Class method design is difficult because the code of one class's methods cannot be cognisant of the method it may invoke in another class: Dynamic object bindings and polymorphism, which try to compensate for an overly complex source code structure, make this impossible.

The polymorphism of Smalltalk, C++, .Net, and Objective-C today implement an impoverished kind of reflection that is more general than is necessary. It is reminiscent of the early visions of reflection as "a dangerous technique" in the spirit of changing class Behavior or re-wiring the virtual machine of a symbolic language. This leads to a paradox between accidental complexity of unconstrained rebindings and overly restrictive coupling between program use cases and data. We want the program to be able to reason about itself in a way that reflects how end users think about the program. Today's code reflects a static worldview that strips out on much of the end user model of the workflow, and attempts to compensate for it by dynamically dispatching methods on classes that the language forces to be designed in isolation, without regard for the interactions between them.

3.1 DCI

DCI, which Trygve Reenskaug and myself have been developing over the past ten years, provides a model of reflection that allows the program to create a new set of program structures for each new use case enactment, while capturing the common recurring rhythms of both system behaviours and of recurring data configurations. Regarding

the latter, DCI still has classes, but they are reduced to managing the way the computer represents information in storage. DCI gives system activities full first-class standing as algorithms, expressed in terms of the roles [5] involved in a use case. Each use case lives within another programming construct called a Context. Program interactions always take place in the context of some configuration of social actors, and the Context is the locus of that aspect of the human mental model.

The Context is the main unit of reflection. A Context corresponds to a use case, whose roles it encapsulates. For each use case enactment it changes the program structure to create a network of objects suitable to carry out the use case. All the same, those aspects of structure that are static in the end user mental model — such as the use case itself — remain static in DCI code and statically can be reasoned about in during program construction.

By adding this form of structured reflection to the computational model, we slice the program dynamics so the code clearly expresses the use case structure. Paradoxically, this selective reflection greatly aids code comprehensibility. The use case becomes a primary structure in its own right, with the data class structure another. Compile-time inheritance forces use cases to be split across class layers whose run-time dynamics cannot be reasoned about in the code. DCI's reflection gives the behavioural part its own expression apart from the data structure.

4. Conclusion

Reflection has had a checkered history as a technique in its own right, though by definition every OO program expresses some form of it. Reflection has stumbled and struggled for a lack of discipline and for a lack of concern for human mental models, and for the difficult task of tying together the triangle of relationships between the programmer, the end user, and the program.

Many thanks to Thore Bjørnvig and Trygve Reenskaug.

References

[1] Hofstadter, Douglas. Gödel-Escher-Bach: An eternal golden braid. Basic Books (1979).

[2] Kiczales, Gregor. Et al. "An overview of AspectJ." Proceedings of ECOOP (2001).

[3] Kay, Alan. "The Early History of Smalltalk." http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html (2007).

[4] Reenskaug, Trygve. http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf (1978)

[5] Reenskaug, Trygve. Working with objects: The OORAM Software Engineering Method. Prentice-Hall (1996).

[6] Coplien, James, and Bjørnvig, Gertrud. Lean Architecture for Agile Software Development. Wiley, 2010.

[7] Bjørnvig, Thore. The Holy Grail of Outer Space: Pluralism, Druidry, and the Religion of Cinema in *The Sky Ship*. In *Astrobiology*, October 2012, http://www.liebertpub.com/ast.