Objects of the People, By the People, and For the People

James O. Coplien

Gertrud & Cope
cope@gertrudandcope.com

Abstract

Computers were invented largely as mental aids. In inventing object-orientation, Alan Kay renewed that vision on a metaphorical level that no longer subordinated computers to human minds. Trygve Reenskaug tried to link the worlds of the human and the computer together with Model-View-Controller, but at the time he got only half the job done. Today we're stuck in this Kantian object world where individual objects act alone and programmers live inside of classes looking out: there is rarely any sense of collective behaviour in object-oriented systems, and there is rarely any degree of behavioural (self-)organization.

I have been working with Trygve on a paradigm called DCI (*D*ata, *C*ontext and *I*nteraction) that places the human experiences of design and use of programs equally at centre stage. It balances the object interaction view with the traditional data conceptualization of class-oriented programming. DCI offers a vision of computers and people being mutually alive in Christopher Alexander's sense of great design. It serves Kay's original vision of object-orientation powering computers as mental adjuncts, as well as his vision of objects as a recursion on the concept of a computer. In this world with a rapidly growing number of increasingly connected human minds, DCI opens up a playful dialogue contrasting metaphors of collective human reasoning and Kay's vision of object computation.

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

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

Keywords DCI; aspect-oriented programming; reflection; object-oriented programming; use case; domain modelling.

1. The Vision

Software has no value as a product. It generates value only when running on a host machine to provide a service. Many design approaches and most first- and second-generation programming languages focused on these services, on the behaviours, captured as procedures or functions. Today's programming languages, most of which relate to object orientation, tend to focus on data as the primary organizing structure. Classes featured heavily as the main building blocks of these programs.

Object orientation would evolve to the point where classes were thought of as collections of related behaviours aggregated around the data that related them, though the data became increasingly encapsulated and hidden. Though this refocusing reduced the preoccupation with data, its behavioural focus was more related to the programmer organization of code than to the business function or the end user benefits from the code.

2. Past Glimmers

Object orientation started with Simula 67, which was quickly followed by Smalltalk—a language dedicated to "computer-human symbiosis." [1] Smalltalk closely held to the ideals of modelling real-world entities and of linking to our dynamic mental processes, using objects as its basic building block. Syntactically, most of what the programmer focused on, while entering code, was the class. It in fact amplified a style of programming called class-based programming, introduced by Simula, which would be followed by C++ and Java.

Many researchers and innovators would strive to break the rather strict Cartesian classification scheme of classes over the ensuing 40 years. Howard Cannon's Flavors [2] was an attempt to move beyond a strict classification that forced every object to be of one class at a time, to one that permitted the class itself to be a composition of multiple class-like things. However, it was still class-oriented and failed to capture behavioural wholes. Multiple dispatch [3] was an attempt to stop classifying methods in terms of their method selector and the single type of a single object, but instead to classify each method as potentially belonging partly to several classes at once. It focused on only one method at a time—again, instead of overall behaviour. The **self** language [4] tried to destroy the very notion of classification as found in a traditional class, and to return to the object foundations that drew objects from the end-user mental model. Dependency injection [5] strove to blend the functionality of two objects into one. Multi-paradigm design [6], [7] refused to view the world according to a single classification scheme, making it possible to carve up different parts of the system in different ways.

Gregor Kiczales [8] considered layered or hierarchical tree structures are too simplistic for the complexity of the software world. He said that a clean separation of function and structure didn't recognize the essential, complex cross-cutting nature of their relationship. We should instead be using reflection to inject this functionality into the objects at finer levels of granularity. He called this the *aspect technique*: a way to do gene splicing at fine levels of granularity.

SIGPLAN'05 June 12–15, 2005, Location, State, Country.

The goal of Aspect-Oriented Programming (AOP) is similar to that of mixins, except its crosscutting units are more invasive at a finer level of granularity. They are like multi-paradigm design in that they allow a degree of separation of function and structure, but aspects' functional structure is much richer. It is more like having multiple knives carving up the same part of the system at the same time, whereas multi-paradigm design ensured that the knives didn't cross. However, AOP again is about thinking in classes rather than thinking in objects: it is a very static way to attach a kit of adjustments to a program at compile time, even though it uses reflection to achieve its end. Aspects are remarkably static: the injected logic (in mixins, called *advice*) is bound to join points at compile time, according to the rules of the pointcut. Last, Aspects suffer from a one-sided focus on the imperative side of design—the model of locally aggregated behaviour.

3. The Realization

All these historic efforts strove to break free from parts and their classification, to wholes and how to express them. Most of the problem lies with behaviour. The broader questions of organization of data structure have long been well in hand through approaches such as domain analysis [9], [10]. The advent of object orientation tended to accord second-class status to behaviour.

A key concern in programming, and a longstanding foundation of object orientation, is the parallelism between mental constructs and programming constructs. Object-orientation owes a rich legacy to frame-based cognitive modelling and, in a more direct way, to Douglas Englebart's vision that extended the Whorfian hypothesis beyond language to the broader tools of computing. Computers were to augment *human* intellect.

This perspective had taken a step forward in 1978 when Trygve Reenskaug invented Model-View-Controller (MVC): a framework that linked the end-user cognitive model with the objects in computer memory [11]. However, much like the "improvements" to object orientation that would follow, it was overly focused on the static part of object orientation.

Trygve's vision started with the barely smart data of a system—the places where information was stored through simple APIs that maintained it in a consistent way. MVC was largely about viewing these data, independent of how the computer processed them. The key missing element was the interactions between the objects at run time, because that's where the value of software-as-service lies.

Trygve's early work [12] established role modelling as a good formalism for interactions between objects. Users and programmers used role names in natural language descriptions of system behaviour. A role is the name of an object according to its use. Roles were a subtle and yet important and radical departure from classes, because they provided an important stepping stone to what happens between the objects: the interactions. The structure of these *interactions* collaborate in some *context* of execution. Together with the data, the context and interaction provided the foundation of a new programming paradigm: Data, Context, and Interaction, or DCI.

DCI is a radical departure from class-oriented programming, which has gone under the heading "object-oriented programming" for some 40 years. Class programmers look at the world from inside of the class looking out (or, if we are lucky, from inside the object looking out)—a perspective from which it is impossible to understand system behaviour. DCI programmers have a longitudinal view across objects from which system operations and use cases can be understood and reasoned about.

The code exhibits the structure of these operations in the collaboration structure between roles. The roles are first-class programming entities whose methods combine into system operations. Roles can be dynamically bound to objects at run time. However, their mutual invocations are statically bound to each other in the code, making it possible to reason about behaviour even in the static structure.

This dynamic view of programming affords a better match to human mental models—first, for end users, and for programmers as well. It is much more firmly in the vein of the current Agile fashions of "individuals and interactions" and the first-class standing of algorithms in the code that can lead to "working software" that expresses the use cases that come from "customer collaboration." It separates the shear layers of slowly changing data structure from more rapidly changing business logic that increase our chances of "responding to change."

From the perspective of software architecture, DCI provides a unifying framework to consider both the function of form and the form of function. In this sense it touches on deep foundations of design related to analogous dichotomies through the ages. DCI is a paradigm that sheds light on the same kinds of "patterns of events" that Alexander holds as fundamental to his school. [13] We can conjecture that this view of design drives very deep into the nature of things: consider temporal symmetry-breaking and the place of time in contemporary cosmology. There lies much more here than just some engineering conventions.

In summary, DCI is a programming paradigm that exemplifies many longstanding aspirations for computing as a field.

Acknowledgments

A very special thanks to Trygve Reenskaug, originator of DCI.

References

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

[2] Cannon, Howard. Flavors: A non-hierarchical approach to object-oriented programming. http://www.softwarepreservation.org/projects/LISP/MIT/nnnfla1-20040122.pdf, (1979).

[3] Steele, Guy L. "chapter 28", Common LISP: The Language. Bedford, MA., USA: Digital Press (1990).

[4] Ungar, David, and Randy Smith. Self, the power of simplicity. http://labs.oracle.com/self/papers/self-power.html (1987).

[5] Fowler, Martin. Dependency Injection. http://martinfowler.com/articles/injection.html (2004).

[6] Budd, Tim. Multi-paradigm programming in Leda. Addison-Wesley (1994).

[7] Coplien, James. Multi-paradigm design for C++. Addison-Wesley (2000).

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

[9] Neighbors, J. M. "Software Construction Using Components." Tech Report 160. Department of Information and Computer Sciences, University of California. Irvine, CA. (1980).

[10] Evans, Eric. Domain-driven design. Addison-Wesley (2003).

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

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

[13] Alexander, Christopher. The Timeless Way of Building. Oxford (1979)..