TOWARDS AN ONTOLOGY OF SOFTWARE DESIGN: THE INTENSION/LOCALITY HYPOTHESIS

Amnon H Eden   Raymond Turner

Department of Computer Science
University of Essex, UK
and
Center For Inquiry, USA

Department of Computer Science
University of Essex, UK

Key terms: Philosophy of computer science, software ontology, software design theory, software architecture.

Ontology and Software Design

The history of programming languages is largely a history of the interaction between ontology and design. Recursion, abstract data types, objects and classes, were conceived as the direct result of the necessity to provide conceptual tools for problem solving and design. Without them programming would probably have got stuck on translating polynomial equations into machine instructions. Fortunately, matters have moved on in the last 50 or so years: we now write programs that operate on a rich variety of data types with a correspondingly enriched collection of control constructs. Indeed, every programming and specification language determines an ontology of types and operations. Moreover, their styles and strategies for design are partly fixed by this underlying ontology. For example, Java is an object-oriented programming language with objects, classes and methods as the central notions. Design in Java is principally concerned with the definition of clusters of classes and their included methods. In contrast, in a functional programming language such as Miranda, inductive data types and recursion are central concepts, and system design is concerned with the definition of inductive types and the recursive functions that operate over them.

While we have some understanding of the ontological commitments made by programming paradigms and how they impact upon the design of simple programs, the same cannot be said of large scale software design tools — the bread and butter of software engineering. Here the tower of Babel effect is still in full flow. Not only are there many different levels of design (e.g., architectural design vs. detailed design), but also a seemingly incommensurable range of different kinds of design statements, the spectrum of such is vast, intricate and largely ill-defined. The area suffers from a form of ontological gluttony: there is little common vocabulary, no guiding theory, no overall conceptual perspective and no uniform formalism.

We take a first step in the larger project of charting an ontology of software design. Specifically, the purpose of this paper is to provide an ontological taxonomy of design statements. We take this to be a necessary prerequisite for the larger and more ambitious project.

Towards a Conceptual Framework for Software Design

We observe three abstraction strata in the vernacular of software design:

- **Strategic statements** ("architectural design") expressing global design decisions, such as architectural styles [6], programming paradigms, component-based software engineering (CBSE) standards [8], design principles, and application frameworks [2], as well as assumptions that may lead to architectural mismatch [5] and law-governed regularities [7];
- **Tactical statements** ("detailed design") expressing localized design decisions, such as design patterns [4], refactorings[3], and programming idioms [1];
- **Implementation statements** expressing concrete descriptions such as class and object diagrams and program documentation.

We further observe that this informal hierarchy is based upon a dual distinction between local versus global statements and between intensional versus extensional statements. Seeking to formalize this distinction, we offer two model-theoretic criteria, respectively:

The Locality criterion. *A statement is local if and only if it is preserved under expansion.*

Informally, if a local statement holds in a (part of a) program then it will also hold in any expansion of that program.

The Intension criterion. *A statement is extensional if and only if it is preserved both under expansion and under reduction.*
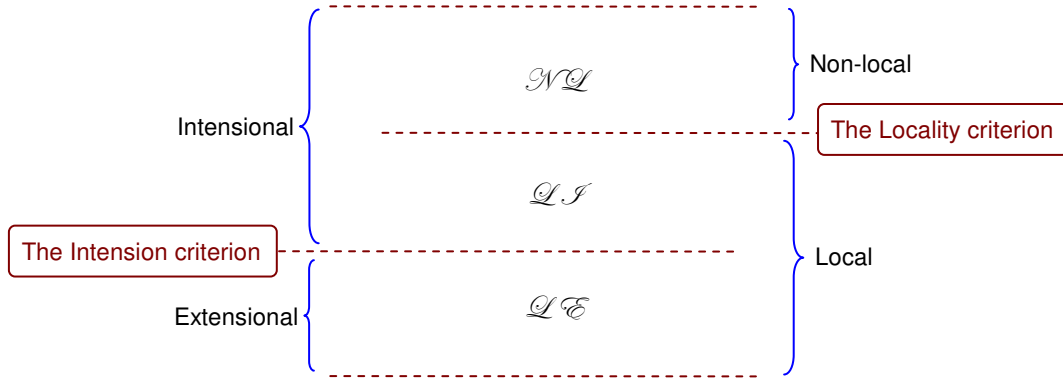
Figure 1. The Intension/Locality hierarchy

We designate the class of local statements $\mathscr{L}$, non-local statements $\mathscr{NL}$. By definition, extensional statements are local. Thus, the Intension criterion divides $\mathscr{L}$ into two abstraction classes, which we designate $\mathscr{LI}$ (local and intensional statements) and $\mathscr{LE}$ (local and extensional statements). Since "local and extensional" is a redundancy, we refer to statements in $\mathscr{LE}$ as extensional.

The combination of the Intension criterion and the Locality criterion divides the spectrum of design statements into a hierarchy of three abstraction classes, the Intension/Locality hierarchy, illustrated in Figure 1.

The vocabulary we use in defining the criteria is borrowed from mathematical logic. Both definitions are semantic rather than syntactic. That is, we classify design statements based on their meaning, not on their form. This choice allows us to apply the Intension/Locality criteria to statements articulated in a range of formal, semi-formal and informal, textual or visual languages, including first- and high-order predicate calculus, context-free languages, Z, LePUS and UML diagrams.

The Hypothesis

In the Intension/Locality hypothesis we postulate that the distinction observed in the vernacular is formalized (distilled and made explicit) by the Intension/Locality criteria, as follows:

♦ Strategic statements are in $\mathscr{NL}$
♦ Tactical statements are in $\mathscr{LI}$
♦ Implementation statements are in $\mathscr{LE}$

The hypothesis along with some of its implications is illustrated in Figure 2.
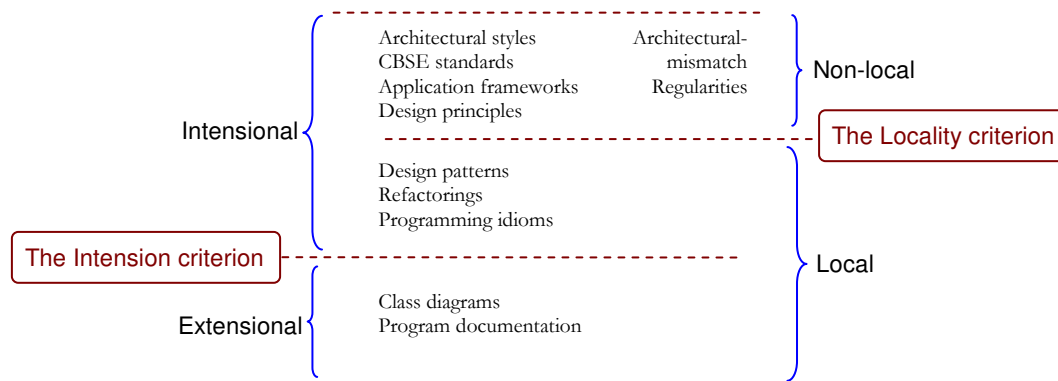
Figure 2. The Intension/Locality hypothesis.

The evidence corroborating the Intension/Locality hypothesis is largely empirical. Evidence was collected by examining a range of design statements from a variety of sources. In particular, these include architectural styles from Garlan and Shaw's catalogue [6], design patterns from the 'gang of four' catalogue [4], CBSE specifications from Szyperski's reference [8], refactorings from Fowler's catalogue [3], assumptions leading to architectural mismatch [5] and law-governed regularities from Minsky's work [7].

References

[1]   F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture*, Vol. 1. Hoboken: John Wiley & Sons, 1996.

[2]   M. Fayad, D. C. Schmidt. "Object-Oriented Application Frameworks." *Communications of the ACM,* Vol. 40, No. 10 (Oct. 1997).

[3]   M. Fowler. *Refactoring: Improving the Design of Existing Code.* Reading: Addison-Wesley, 2003.

[4]   E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Reading: Addison-Wesley, 1995.

[5]   D. Garlan, R. Allen and J. Ockerbloom. "Architectural Mismatch or Why It's Hard to Build Systems Out of Existing Parts." *IEEE Software*, Vol. 12, No. 6 (Nov. 1995), pp. 17–26.

[6]   D. Garlan, M. Shaw. "An Introduction to Software Architecture." In V. Ambriola, G. Tortora, eds., *Advances in Software Engineering and Knowledge Engineering*, Vol. 2, pp. 1–39. New Jersey: World Scientific Publishing Company, 1993.

[7]   N. Minsky. "Law-Governed Regularities in Object Systems; part 1: Principles." *Theory and Practice of Object Systems*, Vol. 2, No. 4 (1996).

[8]   C. Szyperski. *Component Software: Beyond Object-Oriented Programming*, 2nd Edition. Reading: Addison-Wesley Professional, 2003.