Type and metaphor for computer programmers

Timothy Colburn
Department of Computer Science, University of Minnesota, Duluth
tcolburn@d.umn.edu

Gary Shute
Department of Computer Science, University of Minnesota, Duluth
gshute@d.umn.edu

Abstract

The duality of computer programs is characterized, on the one hand, by their physical implementations on physical devices, and, on the other, by the conceptual implementations in programmers' minds of the objects making up the computational processes they conceive. We contend that central to programmers' conceptual implementations are (i) the concept of type, at both the programming and the design level, and (ii) metaphors created to facilitate their use.

Keywords: type, metaphor, programming, philosophy of computer science

1 Introduction

Philosophy has a long history of investigating how linguistic forms work. For example, how do words refer? How do sentences mean? With the advent of the digital age, an additional and ubiquitous linguistic form is the computer program. Like sentences, programs have parts that are related by a grammar. Like sentences, programs convey information to human readers. Unlike sentences, programs are also intended to process information through computation. Programmers cause computations to occur by coding these processes through formal languages. So, how do programmers understand the processes they create?

Our objective in considering this question is not to contribute to the established body of work that makes up the philosophy of mathematics, or the philosophy of logic, or the philosophy of language. Instead, we hope to contribute to the emerging discipline of *philosophy of computer science*, from the perspective of those who actually program.

Turner (2014) has documented the breadth and depth of the philosophy of computer science around central concepts surrounding computation, including *programs*, their *specification*, and their *implementation*. Of particular interest is the *ontology* of programs. As Turner puts it, "they appear to be abstract, even mathematical objects with a complex structure, and yet they are aimed at physical devices." This "dual" nature of programs, as formally or mathematically specified objects having physical implications in running computational systems, has interested philosophers for decades (see for example Moor 1978).

Elsewhere, Turner (2013) tries to explain the duality of programs by appealing to the notion of a *technical artifact* (Franssen et al 2013). Technical artifacts are distinguished from natural objects in that "they are intentionally produced things" (Turner 2013, p. 379) that have a *function*. The function of a technical artifact pertains to the intentions of its producer and thus is not a property of the physical object. The object itself only has *structural* properties that describe it physically. Functional properties pertain to the technical artifact, and not the physical object, so we have a duality: "technical artifacts are individuated by the two descriptions: the functional and structural" (p. 379).

Turner believes that the duality inherent in technical artifacts accounts for the duality we perceive in programs. The function of a program is imparted by its specification, which "informs us how input and output are to be related" (p. 380). However, the structure of a program relates it to the environment in which it executes. "A structural description of the language ... must spell out how the constructs of the language are to be physically instantiated" (p. 385). While Turner seems to be most concerned with characterizing programming languages in the whole, he also claims that "the same conceptual questions arise for both languages and programs" (p. 379). We are concerned here with programs, because they are the things that programmers create.

1.1 Programs As Virtual Artifacts

We believe Turner's account is correct as far as it goes. A program, like a copper kettle, does indeed exhibit a duality between its function and structure. But this duality does not capture a fundamental difference between the ways kettles and programs are made. While the maker of a kettle must at all times be aware of and constrained by the physical medium in which she works, a programmer feels constraints of a far different sort. A program's structure itself harbors the interesting duality we observe in software: that between the physical environment in which it runs and the *virtual* environment in which the programmer intends for her computations to take place.

This is an environment not of electrons, or logic gates, or registers, or processors, but of stacks, trees, shopping carts, and chat rooms. Not physical stacks, trees, etc., but virtual ones that somehow must be understood and controlled. The virtual objects programmers manipulate are not subject to physical constraints. Stacks and trees are not subject to gravity. Shopping carts and chat rooms do not have volume. While it is correct to characterize programs as technical artifacts, it is perhaps more salient to call them *virtual artifacts*.

Regarded as a mere technical artifact, a program's function can be described, as Turner does, through an *abstract semantics*, either operational or denotational, of the language. But regarded as a virtual artifact, a program's function is to manage virtual objects and the relations among them. It is as a virtual artifact that a program's relation to a programmer's intent becomes clear: programs describe interactions among objects—computational and/or informational objects—in a space whose constraints are not physical but self-imposed by the programmers.

The dual nature of software is a direct result of computer science's unique status as, on the one hand, being the creator of its own subject matter, which it shares with a kettle artisan, and, on the other hand, operating in an abstract domain with relaxed physical constraints, which it shares with mathematicians.

However, mathematicians don't have to deal with physical implementations at all; theirs

is a world of purely formal abstractions. Programmers must deal with implementations, but the nature of the implementation depends on the programmer's concern. A programmer's *end* concern is to cause computational processes to take place in a physical medium—a physical implementation. That final task is largely automated and relies on a vast infrastructure of support to bring it about: language translators, operating systems, virtual machines, etc.

A programmer's *initial* concern is how to specify, design, and code the program in the first place. This process encompasses the bulk of the software development life cycle and constitutes programming's primary concern: the *conceptual implementation* in programmers' minds of the objects making up the computational processes they conceive.

When one observes a physical implementation of a program, one sees glowing pixels on a screen, hears the whirring of a fan, feels the heat of the processor, etc. One does not, and cannot, observe the runtime stacks, network sockets, and other computational objects that it is the job of a programmer to create and control. These objects are conceived by the programmer and described through abstract and formal languages—they make up the conceptual implementation of what will become a physical program. So we ask, how do programmers create conceptual implementations?

Conceptual implementations require conceptual tools that aid programmers' understanding of computational objects and their interactions. These tools are necessary because programmers must both create and abide by the constraints of the virtual environments in which they work. They must build complex layerings of abstractions to bridge the gap between what they intend to do and how they accomplish their intentions.

We contend that programmers' conceptual tools include, but are not limited to, modern programming *type systems* and *metaphors* created to facilitate their use. In particular, (i) A programmer's concept of type goes beyond philosophy's traditional understanding of such through its type/token distinction. (ii) Metaphors help programmers conceptually implement their intentions and help others to understand those implementations, and (iii) Programs as virtual artifacts benefit, as other engineering disciplines do, from the conceptual implementation of *design patterns*, and such patterns, which are types at a design level, are also best understood through metaphor.

1.2 Programming's Foundations In Type Theory

Those of us who have been programming or teaching programming for over thirty years have seen the introduction of many programming languages and several programming paradigms. From a programmer's point of view, a language is useful if it is both expressive and helps avoid programming errors. The concept of a programming language *type* is indispensable on both of these counts.

Programming language entities (variables, functions, and procedures) are typed according to what you can do with them. Doing something with an entity that it wasn't designed to handle results in an error. But languages differ with regard to when the error is detected.

Typed languages require that entities either be typed by the programmer or that their types can be inferred by compilers (the programs that translate high-level code written by programmers to machine code executed by computers). For example, in most languages a programmer can use named variables for holding data that can change during the execution of a program. Some languages require that a type is specified when a variable is declared, facilitating the *static* checking for type errors by compilers before the program is run. Other

languages make no such requirement, so that type errors can only be found *dynamically* at run time. The distinction between static and dynamic checking for type errors is related to another distinction between *strongly* and *weakly* typed languages, although there often is some disagreement about how specific languages are classified.

Modern typed languages relax typing requirements by using type inference. This is necessary to avoid requiring typing for parts of an expression. In the expression `x * (y + z)`, for example, programmers usually do not have to specify the type of `y + z`. Possible types can be inferred from the types of `y` and `z`. Then a type for the entire expression can be chosen according to the type of `x` and possibly the use of the expression. Some languages, such as ML, use a complex type inference system so that types do not need to be specified for many programming entities.

Computer scientists working in the area of *type theory* attempt to create formal language semantics that can be used by compilers to automate the process of catching type errors before a program is run. (See Reynolds 1983 and Cardelli & Wegner 1985 for seminal approaches to modern type theory.) The assistance programmers gain from type theory and type checking is in the creation of programs that are more likely to run error-free in their physical implementation, i.e. as technical artifacts.

While type checking to catch errors is useful for a programmer, language types are just as important for the expressiveness they offer programmers in thinking about and coding the computational interactions they wish to bring about. Here, the assistance programmers gain is in the creation of their conceptual implementations of programs as virtual artifacts. The ability of languages to provide this expressiveness is also afforded by type theory. As Cardelli and Wegner (1985) put it, "The objective of a [formal] language for talking about types is to allow the programmer to name those types that correspond to interesting kinds of behavior" (p. 490).

Our purpose in this paper is not to describe the formal languages that provide such behavior (typed lambda calculus with universal quantification, for example), nor to examine all programming languages. Instead, we look at how that work benefits programmers in their conceptual implementations. We examine the evolution from early languages to modern object-oriented languages that allow programmers to think directly about the real-world entities they wish to model in their programs, and to create software that adapts and grows like the real-world environments in which they are situated. We will describe how the description and manipulation of computational objects by modern programmers relies on an interplay between type and metaphor as a mechanism for understanding, designing, and coding computational processes.

After providing some background context in section 2, we introduce in section 3 the significance of philosophy's type/token distinction for the present discussion. Section 4 considers the relations between processor types, language types, and programmer-created types, and the role metaphor plays in their understanding. Finally, section 5 describes how type and metaphor are involved in the handling of software complexity through effective design.

2 Background

In this section we will set the context by further characterizing computational objects and how their software environment has changed. We will also provide some background on metaphor in general and its pervasive role in computing.

2.1 Computational Objects

As we have discussed elsewhere (Colburn and Shute 2010), computer science is distinct from both natural and social science in that *it creates its own subject matter*. It at once creates and studies abstractions in the form of algorithms, data structures, specifications, programs, correctness proofs, etc. It shares the distinction of creating and studying abstractions with mathematics, but as we argue in Colburn and Shute (2007), the primary objective of mathematics is the creation and manipulation of *inference structures*, while the primary objective of computer science is the creation and manipulation of *interaction patterns*. The objects taking part in these interactions are the "computational objects" to which we refer. We have already given a few examples, such as stacks, trees, and shopping carts.

In this sense, we may (loosely) regard computational objects as the virtual counterparts of the electronic objects that actually carry out a computational process on a physical machine. The precise nature of this abstract/concrete relationship is complex, but it mirrors the essential duality of software we mentioned above. Another way to understand this duality is to recognize that software has both a *medium of description* in the abstract world of algorithms and textual programs, and a *medium of execution* in the concrete world of silicon structures and their physical states (Colburn 1999). A program's conceptual implementation occurs in a medium of description, while its physical implementation occurs in a medium of execution.

Computer scientists, unlike natural scientists, neither discover nor exploit the natural laws underlying their physical subject matter, i.e., software's medium of execution. (However, some computer scientists may discover laws that exist in information, as in data mining.) Instead, they use abstractions in the form of algorithms and data structures available through software's medium of description to *create* laws that must be followed if a computational process is to achieve its objective. Such laws often take the form of *invariants*, for example, "the parity bit must equal the exclusive-or of the remaining bits," or "The array index must not be outside the range [0..size-1]." Invariants prescribe constraints that computational objects must obey, so that the concrete processes that ultimately run on physical machines can be controlled.

2.2 How Software's Execution and Description Have Changed

Software's medium of execution, or the actual machines on which it runs, has of course changed dramatically over the decades, witnessed by the steady march of "generations" of new processors in tandem with the celebrated "Moore's Law" that describes hardware's relentless miniaturization. But the basic ontology underlying software's medium of execution is essentially, and perhaps remarkably, unchanged from the early days.

From the development of the stored-program computer in the middle of the last century until today, a computer's high-level architecture has consisted of one or more central

processors, peripheral processors, primary memory, and secondary memory. And although processor and memory devices have undergone dizzying change during that time, such change is due more to miniaturization and parallelization than to fundamental changes in architecture or ontology. Modern multi-core processors, for example, are implemented by smaller versions of the same basic logic gates as their predecessors from generations back.

In contrast, software's medium of description, or the languages devised for programmers to specify computational processes, has changed markedly with respect to the basic ontology of objects available to a programmer. This ontology is tied directly to the concept of a *type* in computer science. Although computer science types were initially intimately related to machine processor architecture, they have evolved dramatically throughout the history of programming, consistently increasing software's semantic expressiveness in a way that, we argue, relies on metaphor.

We have described elsewhere (Colburn and Shute 2008) how the use of metaphor in computer science relates to prevailing philosophical theories about how metaphors work, and the relation between metaphor, ontology, and virtual entities in computer science. Here we focus specifically on how metaphor supports the understanding and use of types in computer science from the point of view of a programmer. Before we turn to that, we offer some background and terminology about metaphor in general.

2.3 Definition and Metaphor

A language user employs metaphor to describe reality by applying a familiar concept in an unconventional way. For example, in the Carl Sandburg poem, "The fog comes on little cat feet." The familiar concept of cat feet is applied in an unconventional way to a description of fog by trading on a similarity between the two pertaining to their silent creeping. Of course, to apply the metaphor one must first understand what cat feet and fog are; they must be *defined*. A metaphor is useless if the language constructs it uses are not already defined and understood. What the metaphor provides is an *understanding* of the poet's intent when creating this image.

Similarly, we are concerned with how metaphor provides programmers with an understanding of the language designer's intent when providing language constructs like types. For example, suppose a designer gives a precise description in code of a data structure composed of various integers and byte arrays. This description is a *definition* of a concept the designer is trying to implement. While this definition would be given using formal programming concepts (like those in the language C), it needs to be given a name so that it can be referred to in ordinary discourse. The designer could choose any term for this name, e.g. "frivet." Instead, she chooses "socket," which both names the definition and invokes a metaphor that provides some understanding for programmers about what the data structure is for. The metaphor suggests the idea of a pluggable connection for communication and provides, for the programmers who use it, an *interpretation* of the associated integers as port numbers and the byte arrays as data streams. Computer science abounds with other familiar, non-digital concepts being applied to describe objects or phenomena that populate only digital realms.

While poetic metaphors connect concepts by way of a single natural language, programming metaphors connect definitions in formal languages, say C, to concepts in natural language, say English. This is not a problem for programmers, who become proficient in multiple languages, and who learn to move easily among them. Good programmers also lace their

programs with English in the form of rigorous comments, which become part of the documentation of their code for others to understand. This understanding can be facilitated through metaphor. Still, metaphor is no "magic bullet' for learning to program; someone already struggling to learn programming may be confused by the introduction of a metaphor, just like it may take multiple readings to understand a poem.

The distinction between definition and metaphor for programmers can be understood by looking at modern philosophical treatments of metaphor that focus on the relationship between language and reality (Ortony 1993). This relationship can be viewed from the perspective of two opposing extremes.

In the "nonconstructivist" view, inspired by logical positivism, reality is describable only by language properly used, that is, through the reporting of direct experience or logic. Metaphorical use of language characterizes rhetoric, not science, and any knowledge we gain through metaphor could have been gained without it. Metaphors are a purely linguistic phenomenon and do not contribute to knowledge of reality. Cohen (1993), for example, proposes that metaphorical meaning can arise directly from literal meaning through a set of linguistic rules.

On this view, the choice by our language designer to use "socket" instead of "frivet" to name her data structure is of utterly no consequence; if we were to go through the entire library and universally substitute "frivet" for "socket," no information would be lost and all programs written using these terms would run the same as before. On this we totally agree, but as teachers of programming we would regret such a language choice for making it more difficult to *learn* a fundamental concept in network programming.

Our point of view is the programmer's point of view, and it is more in keeping with the "constructivist" view of metaphorical meaning. This view holds that language actually requires metaphor in the creative construction of new knowledge, because reality reveals itself only through the interaction of direct experience with preexisting and contextual knowledge. Under this view the distinction between literal and metaphorical meaning is blurred. Under constructivism, metaphors are involved in the general phenomenon of thought and mental representation (Lakoff 1993), including scientific theory transmission (Boyd 1993 and Kuhn 1993). Metaphors such as the orbital model of the atom or string theory as a model of fundamental physics perform functions that cannot be fulfilled by literal language.

We contend that computer science metaphors are paradigm examples of the constructivist relationship of language to our knowledge of reality. They have worked their way into our programming languages, the vocabulary of our algorithms, and our design models. They expand the epistemic framework of our language for talking about computational processes and thereby facilitate the programmer's task of conceptual implementations of programs as virtual artifacts.

This use of metaphor has been recognized by other authors. Lakoff and Johnson (2003) describe many uses of metaphor, including how it is used to describe (they actually use the term "define") difficult concepts such as time and love. But they also recognize the ubiquity of metaphors in computing: "Conceptual metaphors even lie behind the building of computer interfaces (e.g the desktop metaphor) and the structuring of the Internet into 'information highways', 'department store', 'chat rooms', 'auction houses', 'amusement houses', and so on." (p. 244) In this usage metaphors serve an informational or educational purpose.

Programmers benefit in a similar way from metaphors in programming languages, but software metaphors often go beyond that. We will give examples of how these metaphors

serve as a specification for how code is designed. In this sense the metaphor is "engineered," revealing aspects of language or code design that aids a programmer's understanding. These are, admittedly, specialized uses of metaphor compared to how metaphors appear in natural language. But this is in keeping with programming languages' being rather specialized forms of language in general. For this reason, programming metaphors do not suffer from the "slippage" that can occur with natural language metaphors over time.

We do not argue that every computer science concept depends on metaphor, only that metaphor often helps to illuminate dense and labyrinthine concepts for those designing code or learning the discipline. For example, "von Neumann architecture" is a term that has a complex definition familiar to students of machine organization, and metaphor is not particularly helpful in understanding the concept simply by grasping the term. However, "pipeline architecture" uses a metaphor that offers a headstart in understanding how instructions are processed. Similarly, "von Neumann bottleneck" uses a metaphor to summarize the drawback to von Neumann architecture.

As Indurkhya (1992) points out, general theories about how metaphors work divide into two groups, comparative theories and interaction theories. We will not attempt to characterize computer science metaphors as falling into one or the other of these groups in general. Our concern is with showing how the understanding of type systems available to programmers is facilitated through metaphor, and for that it is helpful to look more closely at the structure of metaphorical attribution.

We will follow Indurkhya in calling the object or event being described through metaphor as the *target* and the concept(s) being unconventionally applied in the metaphor the *source*. We assume that the target and source already have definitions, and, as pointed out previously, that the target and source may be described in formal or natural languages.

A metaphor works through the target concept being given a descriptive *representation* through a source concept that must be *interpreted* unconventionally in its relation to the target. In the Sandburg poem the target concept *fog* is descriptively represented through the source concept *cat feet*, whose metaphorical interpretation requires a recognition of similarity between them. Figure 1 shows the general relationship between a metaphor's source concept and its corresponding target concept.

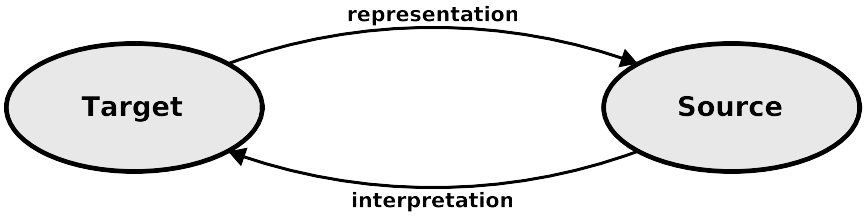

Figure 1: The Structure of Metaphor.

The language of computing is laced with examples of familiar, non-digital concepts providing interpretations of objects or activities in unconventional, digital, settings. In addition to the shopping cart example mentioned above, computer users, dealing only with glowing pixels on a screen, have incorporated *folders*, *directories*, *files*, *registries*, and *pages* into their language to describe computational abstractions. Operating systems, which are just

(albeit complex) programs, provide *servers*, *clients*, and *daemons*. Application programs execute *threads*, occasionally experience *memory leaks*, and undergo *garbage collection*.

Even the concept of a machine itself is used unconventionally to describe some programs that act as *virtual machines* and that, as programs, are not machine-like at all. Metaphors are even ubiquitous in computer science marketing, with software, data, and support all hosted in the *cloud*.

2.4 Metaphors in Programming

Metaphors also abound in programming and are essential for even a basic understanding of how programs work. Consider the procedure in Figure 2, written in the language C, for computing the factorial of a non-negative integer. The procedure consists of *statements*, for

```
int factorial(int n) {
   int p = 1;
   while ( n > 0 ) {
      p = p * n;
      n = n - 1;
   }
   return p;
}
```

Figure 2: A Procedure for Computing Factorial.

example "`p = p * n`" (note that "`=`" indicates a variable assignment, not an equality test). Before a procedure is stored in memory for execution, its statements, written in a high-level language, are translated by a compiler into *machine instructions*, which are represented in a binary language of zeros and ones.

This description depends heavily on metaphor. *Program statement* is a metaphor, because the conventional interpretation of "statement" is something *declarative* — true or false and checkable against reality — while program statements are abstractions for compiler-produced sets of instructions, which are *imperative*. *Machine instruction* is a metaphor, because the conventional interpretation of "to instruct" assumes the existence of a sentient instructee cognitively capable of receiving meaning. This interpretation also regards an instruction as a "speech act" in the sense of Austin (1962), a notion far removed from that of a formatted array of bits in a processor's register.

Beyond this basic understanding, programmers must be acutely aware of their procedures' properties, including their *size* and *run-time*. A procedure's size, or amount of space it takes in computer memory, depends on the number of machine instructions a compiler creates when it translates the procedure's statements. A procedure's run-time, as the term suggests, is the time it takes to run on a particular architecture. A procedure's run-time is determined by how often its instructions are executed. In `factorial` the instruction for "`p = p * n`" is located in the context of a `while` statement, which iterates n times, where n is given as a procedure parameter.

When teaching introductory programming, one has a choice when explaining the concept of iteration. One approach is to start with a definition, such as "iteration is a sequence of

program statements that get executed repeatedly as long as some continuation condition is met." Every student must at some time or other internalize this definition. However, from a pedagogical point of view it might be more useful to start the explanation with a metaphor, such as, "iteration is like looping through laps in a pool until it is time to get out." The concept of a *program loop* trades on this metaphor. Veteran programmers may not invoke metaphors when writing programs that iterate, but they all had to learn programming some time.

One of the thornier concepts to teach in programming is that of the "dynamic nonlocal exit" from a procedure. Such an exit is brought about by an exceptional event (such as a system error or user abort) that requires non-normal processing. The programmer must know a precise location on a runtime structure known as the "control stack", and the code produced needs to ignore a number of pending actions represented on this stack and give control to the precise location in question. Understanding the action required, let alone writing code to do it, is difficult. Language designers realized that a "throw/catch" metaphor was ideal for representing language constructs supporting dynamic nonlocal exits. Not only does this metaphor help language design, but it helps programmers understand how to control their programs. While the language could have been equivalently designed using "foo/bar" terminology instead of "throw/catch," it would have been more difficult to learn.

It is worth pointing out that even untyped languages benefit from metaphors for their understanding. Javascript has the *throw/catch* construct in its language, and its memory management approach is best described by *garbage collection*. Scheme and Lisp have both garbage collection and independent program units understood as *threads*. As with typed languages, these concepts have precise definitions in their language specifications, but the choice of names aids their understanding through the effect of metaphor.

The use of metaphor is thorough-going in computer science and its attendant activities, including the activity of programmers. Later we look more closely at the role metaphor plays in relation to the programmer's concept of "type." But first, it is useful to consider the relevance of a traditional philosophical treatment of this concept.

3 Programming and the Type/Token Distinction

C. S. Peirce introduced the distinction between a word as a *sinsign*, say, ink on paper, and a word as a *legisign*, or the general abstraction that groups like sinsigns (Peirce 1955). Known as the *type/token* distinction, Peirce gave this example: "Thus, the word 'the' will usually occur from fifteen to twenty-five times on a page. It is in these occurrences one and the same word, the same legisign [type]. Each single instance of it is a replica. The replica is a sinsign [token]" (p. 102).

Loosely, the type/token distinction is a general/particular distinction, and as such it has been associated with many issues in philosophy having to do with the ontological status of abstract entities such as universals, properties, and sets. But this distinction has a unique significance for programming with respect to both (i) measuring a program's size and run-time, and (ii) understanding equality in programs.

3.1 Measuring a Program's Size and Run-Time

While the distinction was conceived to apply to words, it is useful in understanding the difference between a program's size and its run-time. When counting instructions to determine a procedure's size, we count instructions in the *type* sense. For example, there would be one instruction for the statement "`p = p * n`" in the `factorial` procedure of Figure 2.

When determining a procedure's run-time, we count instructions in the *token* sense. Each time the statement "`p = p * n`" is executed, a *copy* (token) of it is created and moved into the processor's instruction register for decoding and execution. Since the statement appears in a loop, n tokens of the instruction exist during the procedure's execution.

Philosophical treatments of the type/token distinction usually place it in an abstract/concrete context. Wetzel (2014), for example, asserts that "Tokens are concrete particulars; whether objects or events they have a unique spatio-temporal location." However, it may be argued that the type/token distinction for a program statement given above is not an abstract/concrete distinction, since program statement tokens are themselves abstractions of electronic state inherent in semiconductor circuitry. It is the case that each token of "`p = p * n`", though unique in time, has, by virtue of its representation in the instruction register, exactly the same spatial location as any other instruction.

The context of the type/token distinction for programming is richer than it is for words. As we've just seen, the ontological status of tokens can vary depending on whether we consider program statements as text or bit patterns. This flexible use of types is common, but especially so with regard to our own creations. In writing software, a programmer often draws custom type/token distinctions appropriate for an application. We will give an example later when considering the concept of equality in object-oriented programming.

A similar distinction arises at a higher programmatic level. The term "program" can refer to either the source code or the compiled code or to an instance of the program's execution. The latter usage is important for operating systems, where each execution instance must be treated separately. To avoid ambiguity, people who deal with operating systems refer to an execution instance as a "process." When the operating system loads a program to create a new process, it is making a token in memory of a type on disk. Just as a type may have multiple tokens, the same program can run in several different processes.

3.2 Understanding Equality in Programs

One of the most important aspects of computer programming involves testing computational objects for equality. Such tests are required for making decisions of all kinds, including when to terminate loops, how to search data structures, etc. So programmers must learn how to use equality operations, one variety of which is often indicated with the "`==`" operator. A programmer learns, for example, that if the following variables are declared and initialized in Java,

```
int a = 5;
int b = 5;
```

then the value of the expression "`a == b`" is true, whereas for the following:

```
Integer x = new Integer(5);
Integer y = new Integer(5);
```

the value of "`x == y`" is false.

The reason for this apparent inconsistency is that "`5`," an expression for the value of variables of type `int`, is the value of a "simple" type in Java, while "`new Integer(5)`," an expression for the value of variables of type `Integer`, is the value of an "object" type.

The sense of "object" here is a technical sense. Objects are structured, and similarly structured objects are members of the same "class." Two identical tokens for a simple value are references to the same entity, while two identical tokens for class object creation are not. While programmers are taught this distinction in class/member terms, rather than type/token terms, a similar distinction is at work.

We will return to the role objects play in modern programming later in the next section. We will also see, in section 5, how their relationships necessitate an enhancement of the type/token distinction. But next we turn to types from a programmer's point of view.

4 Types from a Programmer's Point of View

The types of computational objects taking part in the computational processes created by programmers are determined by the programming languages used. Programmers are interested in types for their expressiveness and how they support good programming practice. Programming language designers are concerned with types for those reasons, but also to base their languages on solid mathematical foundations. (See for example Pierce 2002.)

Most programmers come to learn what a statement in a programming language means by learning its syntax and grasping various concepts — branching, selecting, catching, throwing, and threading, to name just a few — that rely on metaphors to help create an abstract mental model of what is occurring physically in a computer when a program runs. Those with an interest in the theory of programming languages, on the other hand, including language designers, mathematicians, and logicians, are concerned with the very meaning of computation in programming languages, or *programming language semantics*.

For theorists, the meaning of a program is given by a mathematical formalism rather than by a mental model of program execution. Such formalisms range from ones based on sets or categories to ones based on abstract machines or lambda calculus. (See Turner 2014 on operational vs. denotational semantics). In either case, program semantics for theorists is mathematical and reductivist in nature, while program semantics for work-a-day programmers involves creating mental models of program execution.

For example, what a programmer conceives as a *shopping cart*, a typed data structure, may, on a denotational semantics account, have a *formal definition* as a certain meticulously constructed subset of all possible computational values — not what a programmer has in mind when thinking about and coding computational processes that manipulate abstractions in an e-commerce application. Instead of naming such an abstraction arbitrarily as, say, a "maflig," a programmer can invoke the "shopping cart" metaphor as an aid to understanding.

The expressiveness of a programmer's mental model of what he or she is coding about is directly dependent on the sophistication of the underlying programming language semantics. In particular, if the semantics allows the introduction of new types by the programmer through type constructors, then the expressiveness of the language for the programmer is limitless. As Cardelli and Wegner (1985) put it, "These constructors allow an unbounded number of interesting types to be constructed from a finite set of primitive types" (p. 490).

The ability to create types that match a programmer's mental model of his or her computational domain is a hallmark of object-oriented programming.

Long before object-oriented programming, however, types in programming languages originated due to the fact that different logic circuitry is required for computing arithmetic operations on integer numeric values than is required for real values (numbers that include fractional parts), and also because integer and real value representations have different binary formats. Types that are based on circuitry and binary representation are called *processor types*. Modern application programmers, however, need rarely concern themselves with processor types, focusing instead on *language types*.

Interestingly, the evolution of high-level programming languages has seen an enrichment of language types, while processor types have become relatively impoverished. In the rest of this section we consider the relations between processor types, language types, and programmer-created types, and the role metaphor plays in their understanding.

4.1 Processor Types

All processors have instructions for moving uninterpreted data, that is, bunches of zeroes and ones. When we say a processor has support for a more complex type we are indicating that the processor can perform operations on the data. We place an abstraction on the zeroes and ones and come up with a *conceptual type* for which we would like processor operations that are tailor-made for it. These instructions provide a *representation* (or perhaps also, an *implementation*), in hardware, of our conceptual type, and the hardware provides an *interpretation* of the zeroes and ones that is in keeping with our conception. Figure 3 shows this relationship. Note that this relationship is identical to a metaphor's target/source

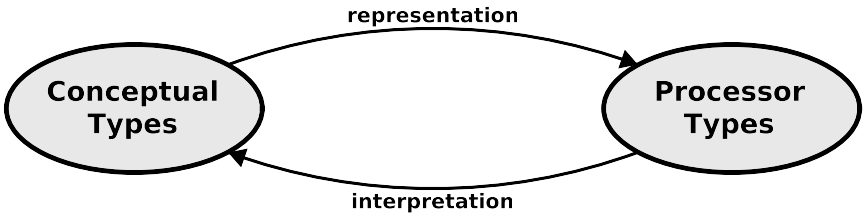

Figure 3: Conceptual and Processor Types.

relationship in Figure 1. Understanding a conceptual type for a programmer is analogous to understanding a metaphorical description in everyday language, because the programmer knows that the conceptual type has a representation (or implementation) on the processor that may require a particular interpretation to be useful.

An example of a conceptual type is *integer* and its associated instructions (Figure 4). For our purposes we consider a programmer using a standard 32-bit integer type such as in C or Java. Saying that a processor supports an integer type means that for two conceptual integers x and y, there are corresponding bit representations x_b and y_b such that applying the processor's "add" instruction to x_b and y_b results in a bit representation corresponding to $x + y$. Figure 5 depicts the representation and interpretation involved in a specific example.

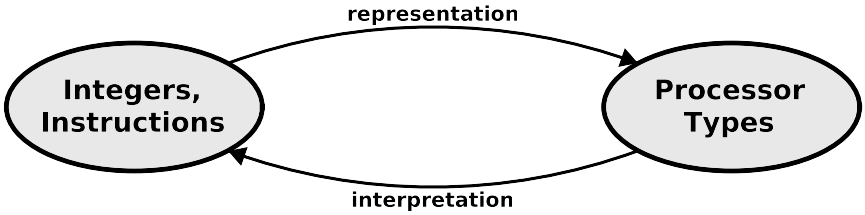

Figure 4: Integer as a Conceptual Type (1).

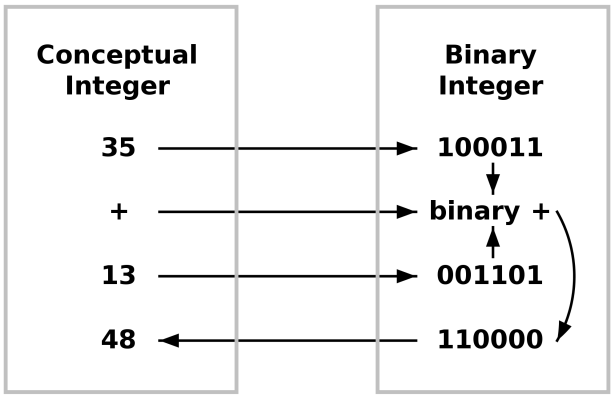

Figure 5: Integer as a Conceptual Type (2).

In natural language, a target concept (like fog) can exist long before a source concept (like cat feet) is used to metaphorically describe it. In the case of processor types and integer types, however, these concepts are developed together. This is in keeping with computer science being a discipline that creates its own subject matter. Computer science is not unique in this regard, but software development has a unique combination of other qualities that focus our philosophical attention on typing and metaphors:

1. It is relatively free from physical constraints, with its objects only being subject to laws (invariants) of developers' own creation. This it shares with mathematics, but not with other engineering disciplines.

2. It must model many types of concepts created by human stakeholders. This it shares much with other engineering disciplines (as well as artisans and artists), but not as much with mathematics.

3. It requires great precision, which it shares with mathematics.

Because software concepts range widely over those imposed by external stakeholders and developers themselves, developers are forced to carefully analyse, distinguish, and control computational objects by understanding their types.

To return to processor types, the fact that typing "35 + 13" at the keyboard while running a calculator program yields "48" on the display is a result of keyboard design,

operating system design, and software design that translates (represents) the text "35" and "13" as processor integers and represents "+" as a processor operation. It is a result of processor design that the operation applied to the processor integers results in a processor integer that is the same as the one we would get if we typed "48" at the keyboard. More software design translates the processor's result back into text and display design makes the text appear in a way that humans recognise as a representation for the integer 48.

What does such an elaborately orchestrated arrangement have to do with metaphor? Again, it is about programmer understanding. When a programmer fully understands *integer* as a conceptual type, she knows that, because the source concept is a processor type, the conceptual type will have some limitations compared to how she may understand integers in elementary arithmetic. For example, when she writes a (C or Java) program to divide the integer 3 by the integer 2, the result will not be 3/2 as a math student may expect. The programmer knows that the processor, owing to how it handles binary representations of numbers, loses the remainder when dividing integers, so the result of dividing 3 by 2 is 1. In a sense, she understands that her conceptual type regarding integers has advantages over thinking about binary digits in a processor, and her understanding relies on the same representation/interpretation relationship that metaphor relies on.

Perhaps counterintuitively, processor types have evolved in the direction of simplicity in the past 35 years. Modern processors, called RISC (reduced instruction set computer) processors, have minimal support for fundamental aspects of programming related to types.

For example, we have mentioned how programmers rely on error checking to create correct programs. Processors themselves recognize a few kinds of errors, including trying to execute illegal instructions, giving invalid arguments to instructions (e.g. divide by zero), and referencing invalid memory addresses. However, it is easy to make type mistakes with processor instructions. An integer "divide" instruction can be applied to data that is intended to represent character strings, for example. The behavior that results is not an error in the hardware sense. To use software engineering terminology, it is a *failure* of the program, caused by a *defect* introduced by a human through a programming mistake. (Lethbridge and Laganière, p. 372) Such "errors" are typically not even detected by a processor. The only error indication is incorrect program results.

As a second example, as we have seen, programmers need to be able to perform tests for object *equality* that are specific to their conceptual types. Processors, which offer only checking for the same bit representation and the same location in memory, are impoverished in their support for this.

4.2 Language Types

While processor types support all software written today, they do so as a foundation for more expressive types available in modern programming languages. These include built-in language types such as structures, higher-order functions, and objects in the technical sense of object-oriented programming. These constructs provide a much better match to a programmer's conceptualization of the world than is provided by processor types. In Figure 6, processor types have been replaced by language types. Under this model, when a programmer writes a program she is less hindered by the limitations of the processor's type system. This is because programming languages have been developed whose type systems are expressive and flexible enough to produce representations of computational processes allowing programmers

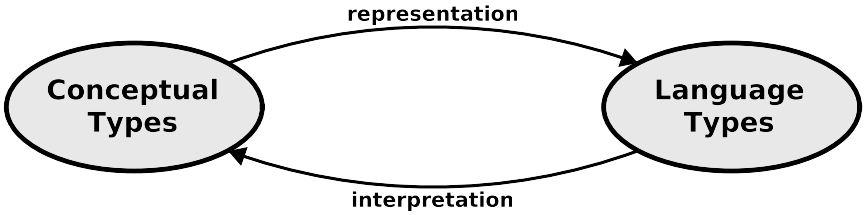

Figure 6: Conceptual Types and Language Types.

to interpret their representations through useful metaphors. As metaphors, these concepts are not fool-proof; consider the over-flowing of a 32-bit integer into negative range. These concepts include not just integers but real numbers, characters, and booleans (truth values), as well as structured types like arrays, character strings, and records or structs. They each rest upon an implementation in an underlying binary representation that is impoverished compared to what the programmer thinks about while programming.

4.3 Abstract Data Types

While language types offer more than processor types, they are still often not enough to satisfy a programmer's desire for expressiveness. As every beginning computer science student learns, the language types provided by even high-level languages do not always give direct support for common programming tasks, such as organizing data in ordered collections called *lists*, or processing data according to policies enforced by *data dispensers* such as *stacks* and *queues*. Understanding these data types depends heavily on the metaphors their names suggest. With the exception of certain specialized languages, commonly used languages do not include these types in their language models. Instead, they must be implemented "on top of" the language with their operations (e.g. removing from a list, pushing onto a stack) provided as a service by procedures or functions that hide the details of their implementation. Such types are supported by neither the processor nor the programming language and are called *abstract data types* (ADTs).

ADTs are ubiquitous in programming and often provided by code *libraries*, another metaphor. The sorts of abstractions they provide beyond elementary data structures like lists, stacks, and queues, is also invariably metaphorical. Consider the *dictionary operations* (inserting, removing, and searching) on ordered data collections represented by *trees* and *tables* (excluding *hash* tables), for example. These concepts are so indispensable to programmers that from their point of view ADTs provided by libraries are as important as types defined by the language.

4.4 Programmer-Defined Types

But even a rich set of ADTs provided by a code library does not match the modern programmer's conceptual model, which might include shopping carts, avatars in a game, objects in a simulation, etc. While libraries can provide the foundational abstractions on which to build, they cannot anticipate all the types of entities programmers will think about when

faced with developing software.

The key to high-level language expressiveness for programmers is therefore a mechanism allowing them to define their own types. While ADTs can be considered programmer-created types, unless a language includes specific constructs for defining new types, a programmer-created type abstraction can be clumsy to use. When a language allows programmers to define types that are first-class types like those already offered by the language, a programmer's conceptual types can more closely match the entities that populate the application domain. This capability is offered by, among other programming paradigms, object-oriented programming.

4.5 Object Types

The introduction of classes of objects as a conceptual type significantly changed how programmers could think about the computational processes they wanted to bring about, because when they defined a new class of objects they were adding a new first-class type to their conceptual model.

Understanding the essence of the object type is facilitated through metaphor. Before object types were available to programmers, the data that were manipulated by programs were best understood, metaphorically, as being *passive* — their only purpose was to be accessible and acted upon but not otherwise act on their own behalf. Object types, on the other hand, can be viewed as *active* data in the sense that besides providing access to their data, they possess a messaging mechanism allowing them to request information and changes of each other. That is, objects not only have *state* associated with them, but also *behavior* specified by programmer-supplied operations.

In the object-oriented programming paradigm, data objects can be viewed metaphorically as participating collaboratively in computation like processing nodes in a network. This gives programmers a medium of description allowing them to bring about computational processes whose objects behave in ways closer to how objects behave in the "real" world.

To return to the notion of equality for programmers, most object-oriented languages provide a default conception of equality based on *identity*: two objects are the same if they occupy the same memory location. However, equality can be overridden to allow programmers to specify their own criteria for equality. When `x` and `y` are defined in Java as the `Integer` object variables as shown in section 3.2, the expression "`x == y`" is false by virtue of the fact that `x` and `y` occupy different locations in memory — they must because there are two of them. However, the expression "`x.equals(y)`" is true because the value represented within those locations is the same, that is, `x` and `y` possess the same state (the integer 5). The designers of Java decided that their object model should include an operation for testing the equality of objects, and that this operation should be definable by programmers.

Object-oriented programming (as well as other programming paradigms) thus draws a clear distinction between identity (testing for "`==`") and equality (testing for "`equals`"). Programmers have the freedom to define "`equals`" in any way they wish for a given class of objects. This freedom in effect allows them to define custom type/token distinctions; the criteria for what counts as two separate tokens (objects) of the same type (class) can be made as strong or as weak as is called for by the application. The flexibility of equality for programmers is another example of computer science creating its own subject matter.

The object-oriented paradigm, which gained hold in the 1980s with languages like Smalltalk

and C++, and flourished in the 1990s with the introduction of Java, required a radical change in how programmers thought about designing code. Data structures and algorithms are still central in programmers' thinking, but instead of being controlled by functions and procedures, they emerge as objects and their operations. Objects, rather than procedures, are in control, and objects can be made to model anything: physical objects, events, relationships, systems, users, etc.

Since programmers can create types for anything they can think of, they can code in essentially the same language in which they think. While this is a significant advantage, it does not lessen modern software's complexity. In the next section we describe how type and metaphor are involved in the handling of complexity through effective software design. We focus on the object-oriented paradigm, but a similar case can be made for any modern language or paradigm allowing for the creation of programmer-defined types. Standard ML, a functional language, is just one example. We spotlight object-oriented design and programming because it predominates in modern software development.

5 Type and Metaphor in Object-Oriented Design

The size and complexity of modern software systems preclude "starting from scratch" for each project. Ideally, software would be constructed out of components previously written in other contexts by putting them together to create an entirely new application, much like a new piece of electronics hardware can be built from modules "off the shelf." While this ideal has not been entirely realized, software reusability is a major objective of object-oriented development. The "object" language construct lets a programmer code at a high level of abstraction. The higher a level of code abstraction, the more reusable that code becomes if the programmer designs those abstractions well. As we show in this section, effective object-oriented design is closely connected to types, and understanding design is facilitated through metaphor.

5.1 Classes and Inheritance

In languages like C++ and Java, like objects are grouped into a *class*, which serves as the objects' type. Class terminology provides language constructs that gather together coded descriptions of the grouped objects' state and operations (also called "methods"). As such, classes are essentially static abstractions used by programmers to describe the *instantiation* and behavior of objects at run-time.

In a sense, classes are the *medium of description* for programmers to describe computational interactions, while objects populate the *medium of execution*. Programs create instances of classes as they need them, like factories stamp out instances of their products. In fact, the *factory* metaphor is used to name and describe a common design pattern (see section 5.4) for object-oriented programming.

When a programmer defines a new class, a new type is introduced, and the class definition serves as the *implementation*, i.e. the internal code representation of the state and methods, of every object in the class. A class definition is a precise description of a new type, and instantiations of the class, its objects, are tokens of that type.

Wetzel observes that humans use language in ways that reflect multiple type/token distinctions involving the same entities. The particular distinction in a given usage is typically inferred from the context by a largely unconscious process. Borrowing Wetzel's example, we know that someone who claims their child knows 20 words is not basing that claim on 20 repetitions of the word "cat"; the child knows 20 types of words, not 20 word tokens.

In software development, we must make type/token distinctions precise. We have not yet developed compilers that can take context into account the way humans do. Consequently, software designers and programmers must tease out the different type/token distinctions and specify different types accordingly. This is what programmers do when they define new classes.

As new classes are introduced, programs that use them become more complex. As class-based object orientation became more popular, it was recognized that graphically-based models of classes and their relationships were useful in understanding the structure of programs. The Unified Modeling Language (UML, see Fowler and Scott 1997) became the standard for modeling both static class relationships and dynamic object interactions. UML diagrams often rely on metaphors to make their intentions known. Figure 7 models the relationship between a class and its *subclass* using UML. Subclassing is built into many object-oriented languages. The diagram employs an *ancestry* metaphor inherent in the par-

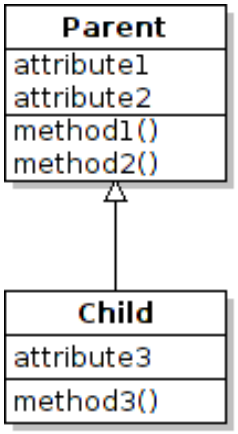

Figure 7: Class Inheritance.

ent/child relationship to show the class relationship.

The `Child` subclass defines its own state (also called "attributes") and methods but also *inherits* the state and methods of the `Parent` superclass. An example from geometry is shown in Figure 8 where the superclass is `Rectangle` and the subclass is `Square`. `Square` inherits its attributes and `area` method from `Rectangle`, but it also defines its own `inscribedArea` method, which computes the area of a circle inscribed in the square, an operation that does not make sense for non-square rectangles.

The inheritance metaphor is helpful in understanding the subclass/superclass relationship, because it establishes the direction of code reuse: descendant classes (subclasses) can reuse code from ancestor classes (superclasses), but not vice versa. When properly used by a programmer, the subclass/superclass relationship is also an "is a" relationship. That is, it should make sense to say that any instance of the subclass "is" an instance of the superclass,

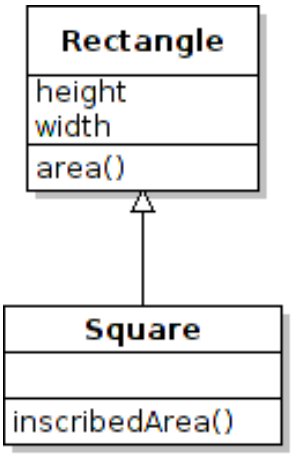

Figure 8: Class Inheritance Example.

just like it makes sense to say that the child of Johnson "is" a Johnson. In our example, any square is a rectangle, but the converse is not necessarily the case.

The subclass/superclass relationship among classes imposes a corresponding subtype/supertype relationship among the objects that are instances of those classes. Programmers are taught to understand class inheritance through supertype and subtype relationships among the objects they create in order to achieve code reuse. Class inheritance, built into the language of C++ and Java, lets a programmer define a new kind of object rapidly in terms of an old one, and get a new implementation almost for free, inheriting much of what he or she needs from existing classes.

The concept of a subtype may not have occurred to Peirce when conceiving the type/token distinction, but it is essential for a class-based object-oriented programmer. The supertype/subtype relationship and the accompanying concept of inheritance allows a programmer to create a type hierarchy that matches his or her conceptual model of the entities in the application domain.

5.2 Abstract Classes and Types

When a programmer defines a class, each method is given a *declaration*, including the method name, the order and types of data the method requires as arguments, and the type of data that can be communicated as a result. A declaration can be understood through a *signature* metaphor, because it uniquely identifies the method. While a signature identifies a method, it does not provide an implementation, which is a description of the actual code that is run when the method is invoked.

Many classes are defined by giving each method both a signature and an implementation. This is the case with both `Rectangle` and `Square`. Such classes are called *concrete*, a metaphor that trades on the fact that instances of them can actually exist. However, language designers have found it useful to allow the definition of classes whose implementation is only partially given. In such a class, all methods have signatures, but not all methods have implementations. Methods with no implementations are called *abstract*, a metaphor that trades on the fact that such methods have omitted the details of their implementation. If

an object were an instance of a class with an abstract method, to invoke the method would cause an error, because there would be no code to run. Therefore, classes with at least one abstract method are also called *abstract*, and though they may be defined, they may not be instantiated.

For an example of an abstract class, consider polygons. Rectangles are polygons, but so are triangles. So we can conceive a `Polygon` class that has both `Rectangle` and `Triangle` as subclasses (see Figure 9). A polygon has a location (say, the upper left corner of its bounding

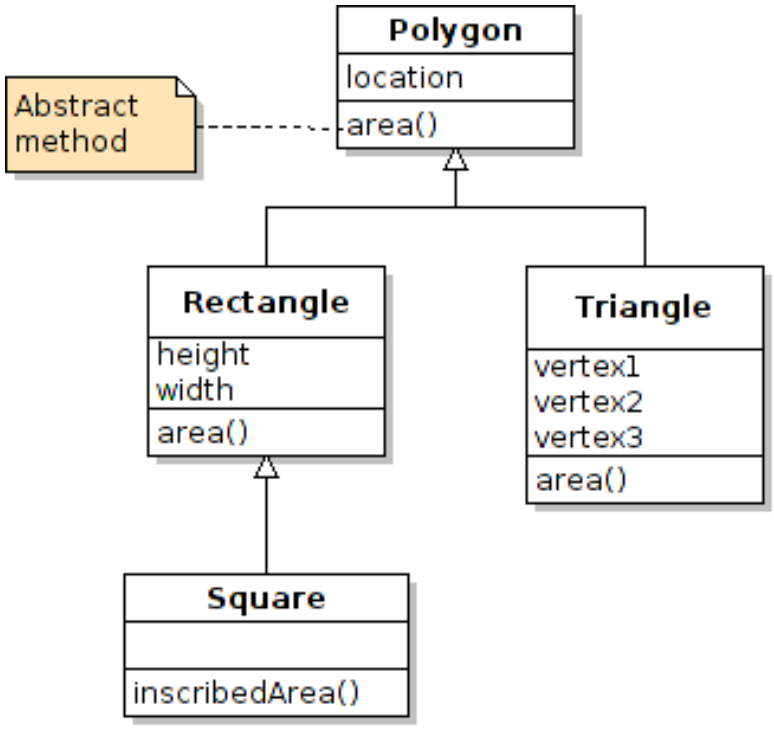

Figure 9: Abstract Class.

rectangle), which makes sense as an attribute for either rectangles or triangles. A polygon also has area, but how to compute it depends on whether it is a rectangle or triangle. A `Polygon` class could therefore give a signature for the `area` method, but it could not provide an implementation. It is an abstract method, and `Polygon` is an abstract class, although it has a concrete aspect in the `location` attribute. Abstract classes are therefore intended to be subclassed by other classes that provide the needed implementation and then themselves can become instantiated.

An abstract class can have many subclasses, and when it does the kinds of objects that can function as tokens of an abstract type can be wide and diverse. For example, a wider variety of objects can be polygons than can be squares. In object-oriented languages like Java, class variables are statically typed and contain not objects but references to objects. For a programmer, the *wideness* metaphor helps to describe the scope of objects to which variables of abstract class type can refer. Since this scope can include different kinds of objects that implement abstract methods in different ways, code that uses such variables can exhibit diverse and flexible behavior. The ability to elicit different behavior from the same code through class inheritance is an example of *polymorphism*, and it lies at the heart

of much of the success of modern object-oriented programming.

While useful, class inheritance (also called implementation inheritance), whether from concrete or abstract classes, is just a mechanism for code and representation sharing understood through the "*is a*" metaphor mentioned above. Code reuse can also be accomplished if an object has, *as part of its state*, another object that can perform useful work. Such object relationships reflect "*has a*" relationships rather than the "is a" relationships among subclasses and superclasses. The "has a" relationship can be understood by programmers and designers through a *containing* metaphor according to which objects can contain other objects within their state.

"Has a" relationships offer opportunities for code reuse in the following way. If an object A has an associated object B as part of its state, and if B has a method (or methods) that accomplish something that A needs to accomplish, then A can *delegate* the work to B, thereby reusing B's code. The *delegation* metaphor is an offshoot of the *active data* metaphor that sees connected objects collaborating to accomplish computational tasks. To understand how this delegation is used to best effect in class-based languages, we must understand the concept of an *interface type*.

5.3 Interface Inheritance

We have seen how abstract classes contain abstract methods that are expected to be implemented in other classes. Variables of abstract class type are "wider" than variables of concrete class type, because they can refer to more than one kind of object. This idea is taken to its extreme with the concept of an *interface* type. An interface is just a named list of method signatures. Loosely speaking, an interface is like a class that has no state and *all* of whose methods are abstract. If a class provides code for all the methods listed in an interface, it is said to *implement* the interface.

The notion of an interface type for software can be understood metaphorically by comparison with hardware interfaces. Hardware interfaces are points of interaction where data is shared between various devices in a computing system. Details of these points of interaction must be rigorously specified by a hardware interface, but how a device creates data for the interface is irrelevant. For example, a computing system may receive mouse input through a USB (universal serial bus) interface, but how the mouse device generates data (whether by a trackball or optically, for example) is dependent on the device and of no interest to the interface. Similarly, variables of interface type in object-oriented programs are places where software objects of different kinds can "do duty" in a program's implementation. A variable of interface type may at one moment refer to an object of type A, and at another moment to a possibly very different object of type B, as long as both objects implement the interface's required methods. In this way, the same code can result in very different behavior, depending on which object an interface-typed variable refers to at the time.

Figure 10 shows the structure of interface inheritance. Neither class `Child1` nor `Child2` inherits code from `Parent`, since `Parent` is an interface type, and not a class type. However, since they each implement the `Parent` interface, they are obliged to define `method1` and `method2` in their own way. Note that they can (and in this case do) also define non-interface methods. Figure 11 shows an example with a `Shape` interface being implemented by both the abstract `Polygon` class (which has subclasses not shown—see Figure 9) and the concrete `Circle` class. Circles are not polygons, but they are shapes, and any (closed) shape has

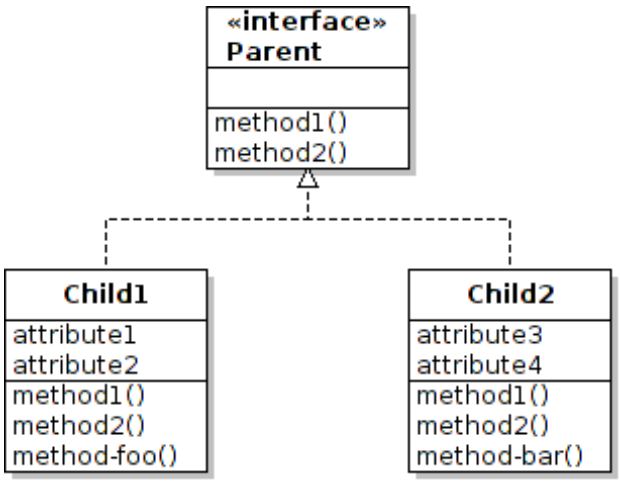

Figure 10: Interface Inheritance.

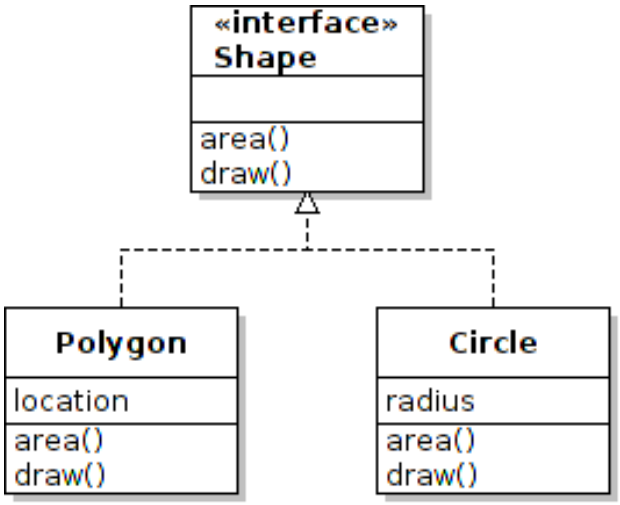

Figure 11: Interface Inheritance Example.

an area and can be drawn. Thus the `area` and `draw` methods are required by the `Shape` interface.

A class that implements an interface type is a subtype of the interface type, so an interface typed variable may refer to a wide and diverse set of objects. Code that uses interface types is even more flexible than code that uses abstract class types. For example, suppose a programmer defines a variable of type `Polygon`, an abstract class type. Code that uses it will only be able to draw rectangles, squares, and triangles. But if the variable is of type `Shape`, an interface type, the same code will be able to draw not only polygons but circles. The ability to elicit different behavior from the same code through interface typed variables is another example of polymorphism.

5.4 Design Patterns As Types

While class inheritance promotes code reuse at the expense of implementation dependencies between a class and its subclasses, interface inheritance and polymorphism can all but eliminate such dependencies. As a result, object-oriented designers are admonished to *program to an interface, not to an implementation*. In other words, whenever possible declare variables to be of interface or abstract class types, and not concrete class types.

Interface types are of central importance to class-based object-oriented programmers, because by separating method signatures from their implementation, they *decouple* code that uses interface typed variables from code associated with the objects to which the variables refer. We have argued elsewhere (Colburn and Shute 2011) that decoupling in a broad sense pervades both computer science as a discipline and the wider context of computing at large.

The decoupling concept can be understood metaphorically by considering its use in electronics, where a signal is eliminated between two circuits. In other physical contexts, decoupling usually means to allow previously linked systems to operate separately. For object-oriented programming, the concept is more subtle, and means eliminating dependencies between objects that result in changes to one object requiring changes in another.

Object-oriented programmers have developed a design discipline that emphasizes code decoupling by using *design patterns* (Gamma et al 1995) that focus on interface types. For example, consider again a programmer defining a variable of type `Shape`. If the variable refers to an object of type `Square` and code invokes the `draw` method on the variable, a square will be drawn. But if the variable refers to an object of type `Circle`, the same code will draw a circle. This simple arrangement uses a design pattern called *Strategy* (Gamma et al 1995, p. 315) to allow the code's behavior to be easily changed (Figure 12).

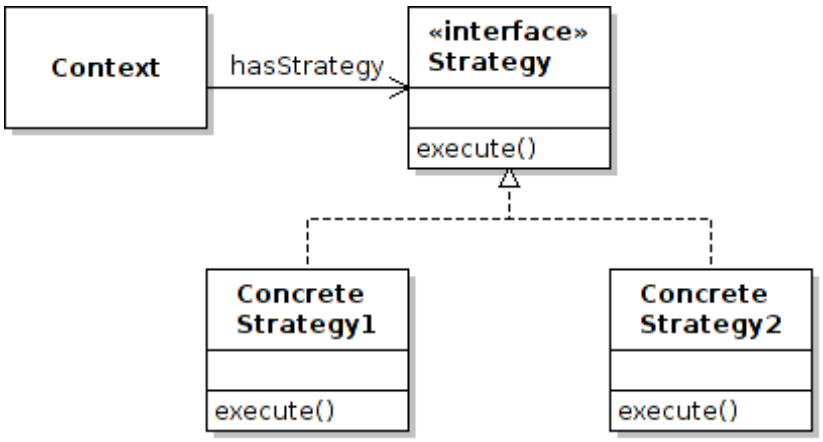

Figure 12: The Strategy Design Pattern.

Like many design patterns, Strategy employs interface typing to accomplish its objective, which is to delegate the execution of a strategy in some context to another object. Such delegation allows the replacement of that object by another that implements the same interface, in order to get different (possibly radically different) behavior. The strategy metaphor is most apt when considering how humans can "change" or "switch" strategies at will.

Note that the UML models shown in Figures 10 and 11 are "instantiations" of the general pattern shown in Figure 12. The general pattern makes use of general names of inter-

faces ("`Strategy`"), classes ("`ConcreteStrategy1`," "`ConcreteStrategy2`"), and methods ("`execute`") whose roles are played by particular actors in an implementation. In our simple example implementation, the role of the interface `Strategy` is played by `Shape`; the role of the classes `ConcreteStrategy1` and `ConcreteStrategy2` are played by `Square` and `Circle`, respectively; and the role of the method `execute` is played by `draw`. The Strategy design pattern, as an abstraction, is like a type, of which our particular implementation is a token, among possibly many others. As such, design patterns are for programmers "design types," and their mastery is every bit as crucial in the creation of effective conceptual implementations as is mastery of the programming language.

5.5 Roles As Types

In the "real" world, humans often make classifications not of individual entities, but of *roles* that such entities play in complex interactions. For example, national constitutions and laws define complex protocols governing the interactions between various legislative, executive, and judicial agents. In democratic countries, at least, we expect these roles to be designed polymorphically. That is, within broad limits they do not restrict people acting in these roles to certain types of behavior such as liberal or conservative; they only govern how they interact to define new laws, enforce them, and interpret them.

We may even metaphorically describe the governmental positions as `Strategy` references. When citizens elect new individuals to serve in these positions through the election process, they can implement different political behavior. Analogously, changing a `Shape` variable's reference from an object of class `Square` to an object of class `Circle` changes the program's drawing behavior.

Experienced object-oriented programmers consider roles to be types just as much as they consider individual entities to be types. For example, when designing software for keeping track of information about people in a complex educational organization, a programmer could define a `Person` class with several subclasses such as `Student`, `Faculty`, and `Administrator`. While this might work in a simple program, it would require significant changes when the software grew to deal with more complex needs, including situations where people change roles, or have two or more roles at the same time. A better design is to create a type system where the tokens are abstractions (roles) instead of concretes (people).

To emphasize the importance of roles in programming, design patterns are often described in terms of *participants*, a metaphor that is again in keeping with conceiving of objects as *active* data. For example, the *Observer* design pattern (Gamma et al 1995, p. 293), shown in Figure 13, defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically. Saying of a software object that it "observes" another and waits for notification from it is, of course, a metaphorical attribution, but it is enormously helpful in understanding how to code complex interactions. It also describes a design that pervades modern programming, from interactions as basic as clicking a button in a graphical user interface, to creating and serving complex web pages backed by databases.

Two of the essential participants in Observer are a `Subject`, which provides a mechanism for registering and unregistering observers, and an `Observer`, which provides a mechanism for responding to change. In complex software these participants are not types of concrete entities. Instead they are types for roles in a certain kind of interaction between objects, and

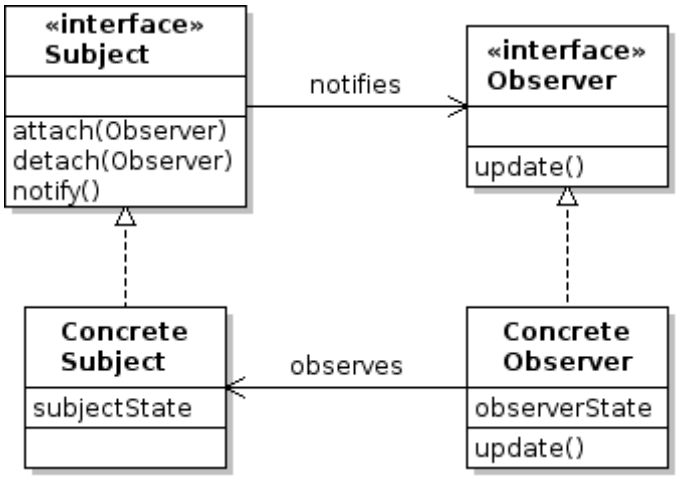

Figure 13: The Observer Design Pattern.

as such they are *abstract* participants in the pattern. Defining interfaces for these participants allows diverse objects with different *external* responsibilities, that is, responsibilities not inherent to the Observer design pattern, to play the roles in the pattern. These objects are *concrete* participants in the pattern. The `Concrete Subject` adds methods that observers use to determine the nature of the observed change, while the `Concrete Observer` uses those methods to respond to the change. When used in this way, the interface types `Subject` and `Observer` are therefore doubly abstract: (i) they type variables, not objects, and (ii) they type roles, not entities fulfilling those roles.

6 Conclusion

A programmer's objective is to create, in formalized text, a program that ultimately has a physical implementation in a running computational system. In order to do that, he must create a mental model of the computational objects interacting to perform the tasks required—a conceptual implementation.

Conceptual implementations are facilitated by the concept of type, beginning with the types offered at the fundamental level of the processor. On top of this, programmers must make use of increasingly rich type concepts, from language types, to abstract data types, to programmer-defined types such as those offered by the object oriented programming paradigm, among others. Effective mastery of these type concepts requires interpretations of deeply complex representations of programming language notions. We have shown that at all levels type concept interpretations are provided through metaphor.

Type systems can be partially understood through philosophy's type/token distinction, but we have seen how programming languages have enriched this distinction in various ways. Subtypes allow tokens to be of more than one type and to benefit from relationships of inheritance. Abstract class types accommodate tokens of widely different subtypes, so that programmers can write code that is polymorphic.

The exploitation of polymorphism reaches its furthest extent in object-oriented programming through the concept of an interface type, the basis for many successful design patterns

that are also best understood through metaphor. Design patterns are independent of any programming language but are best understood as types themselves and take their place alongside programming language types in the programmer's toolbox for creating successful conceptual implementations. By writing code that instantiates design patterns, programmers have a type language allowing them to code by thinking about not just the objects that are involved, but their roles.

The type systems offered by many modern programming languages, combined with the power of metaphor, provides a rich conceptual environment in which programmers can code with the objects of their thought.

References

Austin, J. (1962). *How to do things with words*, Oxford University Press, Oxford

Boyd, R. (1993). Metaphor and theory change: What is metaphor a metaphor for?, in Ortony, A. (ed), *Metaphor and thought*, Cambridge University Press, Cambridge, 481–532

Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys* 17:4, 471–522

Cohen, L. J. (1993). The semantics of metaphor, in Ortony, A. (ed), *Metaphor and thought*, Cambridge University Press, Cambridge, 58–70

Colburn, T. (1999). Software, abstraction, and ontology, *The monist* 82:1, 3–19

Colburn, T. and Shute, G. (2007). Abstraction in computer science, *Minds and machines*, 17:2, 169–184

Colburn, T. and Shute, G. (2008). Metaphor in computer science, *Journal of applied logic* 6:4, 526–533

Colburn, T. and Shute, G. (2010). Abstraction, law, and freedom in computer science, *Metaphilosophy* 41:3, 345–364

Colburn, T. and Shute, G. (2011). Decoupling as a fundamental value of computer science, *Minds and machines* 21, 241–259

Fowler, M. and Scott, K. (1997). *UML distilled: applying the standard object modeling language*, Addison-Wesley

Franssen, M., Lokhorst, G. J., van de Poel, I. (2013). Philosophy of technology, in: Zalta, E. N. (ed), The Stanford Encyclopedia of Philosophy, http://plato.stanford.edu

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of*

reusable object-oriented software, Addison-Wesley

Indurkhya, B. (1992). *Metaphor and cognition*, Kluwer Academic Publishers, Dordrecht, The Netherlands

Kuhn, T. (1993). Metaphor in science, in Ortony, A. (ed), *Metaphor and thought*, Cambridge University Press, Cambridge, 533–542

Lakoff, G. (1993). The contemporary theory of metaphor, in Ortony, A. (ed), *Metaphor and Thought*, Cambridge University Press, Cambridge, 202–251

Lakoff, G. and Johnson, M. (2003). *Metaphors we live by*, University of Chicago Press, Chicago, IL

Lethbridge, T. C. and Laganière, R. (2005). *Object-oriented software engineering*, 2nd ed, McGraw Hill

Moor, J. H. (1978). Three myths of computer science, *The British Journal for the philosophy of science* 29:3, 213–222

Ortony, A. (1993). Metaphor, language, and thought, in Ortony, A. (ed), *Metaphor and thought*, Cambridge University Press, Cambridge, 1–16

Peirce, C. S. (1955). Logic as semiotic: the theory of signs, in Buchler, J. (ed), *Philosophical writings of Peirce*, Dover, New York, 98–119

Pierce, B. C. (2002). *Types and programming languages*, MIT press

Reynolds, J. (1983). Types, abstraction and parametric polymorphism, *Information processing* 83, 513–523

Turner, R. (2013). Programming languages as technical artifacts, *Philosophy & technology* 27:3, 377-397

Turner, R. (2014) The philosophy of computer science, in Zalta, E. N. (ed), The Stanford Encyclopedia of Philosophy, http://plato.stanford.edu

Wetzel, L. (2014). Types and tokens, in Zalta, E. N. (ed), The Stanford Encyclopedia of Philosophy, http://plato.stanford.edu