1

# 1 Introduction

A central theme of the following chapters will be computer programming languages, seen as practical technologies which have their own theoretical orientations and engineering requirements. One topic will be the relation between these artificial languages and natural (human) language. Another will be the relation between computer languages and various facets of "information representation" — for data modeling, designing data-sharing protocols, or database engineering. There is a connection between these themes insofar as data models constrain data structures, which supply the values on which programming languages operate. In other words, data models are an essential aspect of programming-language "semantics".

Preliminary to further discussion of these subjects, I will start with a review of computer languages as technical artifacts. Perhaps it is not quite accurate, in fact, to describe computer programming languages as "languages". This is partly because natural language evolves organically, whereas programming languages are *created* (usually "by committee"). They evolve in response to engineering challenges — in particular, those concerned with getting computers to "understand" high-level source code in lieu of physically realizable "machine" code. There are still some processing stages or semantic/syntactic formulations where one can make comparisons or even borrow formal models from one domain to the other, but in general, we should be careful about applying intuitions about natural language to programming languages too broadly.

For example, people do not use source code to "talk to" or even generically "communicate with" computers. Rather, computer code acts as the material through which programmers implement applications, which are tools pressed into service to manipulate computational states. More specifically, applications in general do not have default behaviors — instead, they are essentially computational environments, which do not initiate actions until guided by a human user.[1] Once the user *interacts* with an application, the application should have functionality to respond to that user's input. Typically this response will be processed and then the application will return to a passive state, awaiting further user actions. The passive-active-passive cycle is often called an "event loop".

Users typically interact with applications because they want to change something. At a minimum, they want to change what they *see* while the program is running, and/or to modify some data somewhere. They might, for example, seek to save, upload, or download a file. For conventional computer systems (personal desktop/laptops, tablets, smart phones, and even *en situ* touchscreens built into industrial hardware and appliances) almost all information which a user acquires from applications is *visual* and *two-dimensional* (although a 2D screen might show a simulation of 3D graphics).[2] In this sense the behavior which a user expects from their application is intimately connected with what they "see".

Fundamentally, then, the goal of an application's code base when responding to user input is to modify the on-screen display that users consume visually — even when the primary response is some side-effect, such as saving a file, the display should have a change indicating that the operation was successful (or, if not, alerting the user that there was a problem).

These points might seem obvious, but they are worth emphasizing for any discussion where we examine the "semantics" of computer code and application data. Since the emergence of the modern internet and World Wide Web, a lot of emphasis in computer science has concentrated on data *semantics*, or on how digital artifacts can model/represent empirical things in the world. The goal of viewing and/or analyzing data structures as a proxy for manipulating actual "things" underlies diverse branches of Information Technology, from Object-Oriented programming to database engineering to the Semantic Web. But digital resources only model real-world phenomena at one step removed: when we talk about computational representations of external (either concrete or abstract) things, we are actually referring to capabilities for creating *visual* displays which convey information about those objects to human users, according to signifying conventions that users understand. If, say, a user wishes to learn the current score of a World Cup match, the application will perform correctly if it changes its visible state to show

[1] Granted, there is a class of programs which run behind-the-scenes following a preprogrammed sequence, constructed with the expectation that human users will not interact with the program during its execution. Even here, though, we can argue that the choice to initiate the program in the first place constitutes a human action, so such programs do have at least *one* point where they are dependent on users' input. The choice of *when* to run a program will (potentially) influence its behavior, so this choice (which depends on a user's discretion) acts as a kind of parameter which users put in place to guide the program's actions. This holds even (or, indeed, especially) when users arrange for a program to run automatically (typically when the computer first starts up), such that they do not manually interact with that program *each time* it executes. Instructing the operating system to launch a programming automatically is merely a convenient way for users to ensure that the program runs at their desired moments. Given these points it is reasonable to say that *all* programs depend on *some* input from users, even if such input is restricted to a decision about when the program should (or should not) execute.

[2] To this we can occasionally add audio feedback, though — setting aside interfaces explicitly designed for the hearing-impaired — audio is rarely an important data-source except when playing an audio or audio/video file.

this data in a standardized pattern, such as a number (indicating goals scored) next to an Argentina logo (perhaps an icon of the Argentine flag) and an analogous number next to a France logo/flag.

There is nothing in the application's state in this example which actually "means" anything about Argentina, France, football, goals, or anything else as users understand it. Any "semantics" here is merely the result of adequate engineering. If the application behaves properly, the number next to the France logo will match the number of goals France has scored so far. We cannot say that the number visible on-screen is actually a "signifier" for the France goal-total, because even a well-designed application could be incorrect (consider internet latency, making it impossible for the software to update the displayed goal total in exactly "real-time"). We can only say that a properly implemented application will be a useful source of information about the soccer game because it is engineered to satisfy the requirement that, on most occasions, its visual interface will match the state we associate with correct information about the game. The "meaning" of data displayed by an application depends on an engineered correlation between a data source and "ground truth" facts; but such a correlation is something that has to be artifactually constructed by programmers and database administrators. This kind of semantics is very different from natural language word-object associations (that evolve organically via entrenchment among a given dialect community).

As a result, when considering how to formalize and evaluate programing language "semantics" we should not work in a paradigm where data structures are innate proxies for real-world concepts, the way that linguistic signifiers are communally-recognized expressions of their correlated signifieds. Instead, data structures offer *tools* which engineers leverage to assemble pieces of information that on aggregate simulate or describe external objects with varying degrees of detail. Different contexts allow for different data structures which are molded to empirical phenomena, more or less selectively. To provide the score for a cricket match, for example, unlike a football score, one would need to represent the wickets taken for each side as well as their runs scored; thus a two-valued structure (for football) would be replaced by a four-valued one (or, equivalently, two number-pairs). Of course, much more info could be modeled for any sporting event (time elapsed, for many sports, or the number of remaining balls in a T20 innings; where the game takes place; the stats on each player; and so on).

How much of this data is actually curated by any given piece of software depends on decisions made while the software is designed. Data structures can indirectly represent empirical concepts because they provide programmers with a tableau of representational tactics — data types and models, numerical encodings, units of measurement, valid ranges and other quantitative constraints, etc. — which allow data structures to track real-world objects by selectively sampling some of their properties. Even then, these structures are not intrinsically representations *of* their intended targets. Rather, applications may convert data structures to visual displays which stand in for objects by representing their properties textually or graphically (the former being, for instance, a printed number asserting goals/runs scored; the latter being a visually-conveyed magnitudes, such as a bar's height or color's hue to present weather temperatures).

Applications, in short, *simulate* semantics by curating data structures which are engineered to track real-world concepts *and* subsequently translating these structures into user-visible displays. A competent programmer must ensure that software performs these tasks correctly. Computer code manipulates applications' internal data structures and external displays. In this sense computers do not "understand" code the way that people understand language. An application does not reach a desired state given user input by treating code as a kind of message or communiqué that it has to interpret. Instead, computers are engineered (via compilers and runtimes) so that source code can be employed as an indirect tool to predictably reconfigure applications' internal and external state such that the latter correctly conveys certain kinds of information.

Needless to say, the machinery for engineering the proper synergy between external facts, internal data structures, and visible displays is vastly different than the cognitive architectonics of people hearing (or reading) language. This is one reason why I implied that the expression "programming *language*" is a misnomer. Nonetheless, I believe that in some sense (natural) linguistics can benefit from analyzing programming languages, precisely because of their mechanical and artificial nature.

Specifically, much of the subtlety of natural language derives from interpretation and disambiguation. Almost all words have multiple meanings and almost all sentences can be parsed in different ways. Addressees must therefore infer which construal of parse and word-use is most likely to match the speaker's intent. There is nothing analogous to such hermeneutics in programming: software-language

implementations do not "guess" at meanings; they can only rely on precisely formulated rules to deduce which operations are designated by specific segments of source-code.

Consequently, programming languages are much blunter and less expressive than human languages. But by discarding issues of interpretation and "most likely" meanings, computer code enables us to study how structural manipulations can be encoded in symbol-systems which at least superficially *resemble* human language.[3] The chain of computations which extend from source code as written to visible and information-carrying display-changes in an applications' user interface are guided by formally describable systems that potentially have some correlates to how human "process" language — not in the sense of how we interpret the meaning of sentences initially, but rather in terms of how we follow up. According to the *pragmatics* of speech-acts, the "meaning" of a sentence lies in how cooperative listeners should respond (not just linguistically, but also potentially via concrete actions). This is analogous, at some level, to how software applications' semantics are determined by how they respond to user actions, making "correct" changes (i.e., those users anticipated and intended) to application state.

I have said that an application's external state is primarily manifest in its visual displays. This is true for conventional software, although we can envision technology evolving so that the 2D digital screen is not the only medium through which programs can show information. Perhaps 3D printing or even holograms will break through the limits of two dimensions. Virtual Reality environments might increasingly incorporate kinaesthetic, haptic/tactile, audio, and even olfactory qualia to complement visible scenes [**?**], [**?**], [21], [32], [25], [24], [27]. And, of course, robotics introduces an entirely different set of dimensions for external configurations: a robot's external state is not only defined by what is visible on a display screen (if it has one), but on the position of the robot relative to its surrounding environment and all the degrees-of-freedom for its gears, joints, wheels, and other moving parts. In this way robots can be programmed to adopt postures that physically alter their surroundings, by lifting and carrying objects for example. Such mechanical configurations are analogous to the purely visual configurations evinced by software whose effects are mostly localized to a two-dimensional touch-screen or monitor. Nonetheless, the same internal architecture is in effect: robots are *caused* to take on their useful positions because internal data structures are engineered to guide external configurations according to desired outcomes, which is the same principle as old-school software wherein the "desired" outcome can be fully expressed through the visual content present at any moment on a computer screen.

The engineered causation between internal data and external configuration, then — be this visual (for most computational environments) or mechanical (for robots) or something more multi-sensory (for Virtual Reality) — is in any case the essential detail for analyzing computer technology: that is, for improving the science of Information Technology in ways that could make software more reliable. Although a lot of computer programming is done in an experimental, trial-and-error fashion — consider the ecosystem of bug reports, bug fixes, security patches, and other incremental improvements which are part of any application's life-cycle — there is still room for examining software development (and programming languages) from a "theoretical" perspective. Note that ideas about effective computer code and application development have evolved noticeably during the "personal computing" era: conventions for conveying information through interactive visible displays are refined over time, yielding software that is progressively more intuitive and informative. Meanwhile, technologies for implementing computer languages themselves — compilers to translate source code into machine language and runtime environments to serve as containers where programs execute — have become more sophisticated, allow programming languages themselves to change over time, which in turn (in principle) makes software both less costly to implement and more trustworthy and secure.

Although physical improvements in the underlying hardware — ever-more-powerful silicon chips, say — have obviously fueled the power of modern computers (even quotidean ones owned by the general public) the evolution of programming-language and software-development methodology has not been driven by hard science *per se*, unlike advances in chip design, solid-state hard drives, or multi-core processors, but rather by relatively abstract considerations around the nature of parsers, compilers, data structures, Virtual Machines, and other accoutrements in the ecosystem of language implementation. In effect, Software Language Engineering is almost philosophical, in that progress can be made through relatively abstract investigations (without empirical or physical substrata), but not in the sense of academic speculation without concrete payoffs: better computer languages means better software.

[3] In the sense that source code is composed with the same letters and punctuation marks that are the basis of natural writing systems; moreover, many symbol-names and keywords in computer code are chosen to suggest words in natural languages (especially English).

This is therefore the practical backstop surrounding chapters in Section 5 of this book, which will focus on issues of language implementation and Virtual Machines (VMs). Almost all modern programming languages invoke VMs at some stage in code-processing, even if this is done prior to an application being executed (one way to gather compiler-related data about an application is to "virtually" simulate running code, so virtual machines can come into play during compilation even if the compiler's target is low-level machine code).[4] All computer code is eventually translated to "machine language"; i.e., to instructions that can be *physically* realized within a computer's Central Processing Unit (CPU). A "virtual" machine, by contrast, is not a physical device which performs calculations, but rather a software system that emulates the behavior of such devices, to some approximation.[5] As a result, VMs are not limited to the data structures and instruction sets that can be physically realized in a CPU (or analogous settings, such as a Graphics Processing Unit or a Quantum Processor): because VMs are software, not hardware, they can take on almost any set of kernal operations and built-in data types that can be programmed within the software through which the VMs are themselves implemented. In this sense, VMs are a valuable environment for analyzing programming language design and implementation, because they are a flexible and adaptable environment for parsing, compiling, and executing high-level computer code.

For reasons sketched in this introduction, here I consider VMs especially in the context of how applications link internal data structures to external (mostly visual-display) configurations — which I have argued is *the* central concern of software engineering in general. That is, we can use VMs to analyze systematically the processes through which *source code* formalizes digital machinery such that *user actions* are responded to by manipulating *internal data* resulting in changes to *visible displays*. Each of the scaffoldings connecting these four principal elements (source code, user actions, internal data structures, and visible displays) is governed by overlapping sets of conventions, specifications, and computational requirements; modeling the full space of their interoperation via Virtual Machines is a good way to make theoretical sense of the complex interworkings that come into play once an application is up and running.

[4] I'm thinking of something like (Low Level Virtual Machine) as a foundation for the C++ Clang compiler [18]. Even if it is not entirely accurate to speak of "virtually executing" C++ code during compilation, this is not a bad way to visualize compiler details in a relatively informal context. For example, type-checking the outputs of a procedure once specific types are assigned to its input parameters can be construed as an approximate "running" of the procedure where we abstract from the *specific* input values and consider only their types.

[5] Virtual machines typically model a wider range of operations than can be computed *in silico* directly; but such higher-level operations themselves are then destructured into sequences of lower-level machine-language steps.

## 2 Hypergraph Data Modeling

I claimed in the introduction that a software application's primary role is to await user actions and then, in response, produce some visual change (in the on-screen display) which presents information to the user (presumably information relevant to their original action), along with possible side effects (like saving a file). On this account, studies of applications' behavior would reasonably focus on the starting and ending points of these action/reaction cycles: how do we articulate the full space of possibilities for users to initiate actions within the application? For mouse-and-keyboard desktop setups, an action like "left click" on the mouse can be performed at any point in an application's window (or windows), and mouse-events could click one of several buttons, or turn the scroll wheel, and they can be modified by pressing certain keys. In some contexts, likewise, users enter data by typing on a keyboard.

Applications have to systematically model each of these possibilities to create data structures noting all relevant details about user actions, which in turn is needed for responding correctly (left-click and right-click usually have different "meanings", for example). At the other end of the user-action/application-response cycle, applications need to subdivide their display window into parcels of screen-space so that data updates are located in useful visual contexts (continuing the above example, the France score should be placed near the French flag/logo, and likewise for Argentina). This, of course, means that the computer screen is not just a "picture". It is an organized system analyzed through data models and types. And such analyses may potentially be extended to robots and/or Virtual Reality, which are not just 2D screens but possess 3D physical or visual-immersive (and maybe synaesthetic) configurations.

Well-designed applications have multiple features and are easy or intuitive to use: this means that users have a variety of options for leveraging applications' capabilities, and that it is not difficult for users to learn or infer the steps needed to prompt their application toward their desired course of action. Applications can generally become "better" — more featureful and intuitive — by being flexible and subject to continuous improvement: acquiring more functionality over time, and updating their interface based users' experience (they may find different features easier or harder to access). For example, specific oper-

ations might be exposed in different ways, in that the user has multiple options for initiating certain responses (e.g., the effects of clicking on a certain button might be duplicated by pressing a specific key on the keyboard, or by activating the mouse's scroll wheel — consider scrolling down, hitting an on-screen "down" button, or pressing the down-arrow key).

Many applications also allow plugins or extensions to adjust the visual displays in ways not envisioned by the program's original designers.[6] This flexibility helps ensure they can improve and become more user-friendly, but more specifically these are engineering goals: there are steps which developers can take to maximize applications' ability to adapt and evolve.

Aside from a generic desire to build "quality" software, engineering applications for future adaptation also prompts developpers to design software in rigorous, well-documented ways. Applications can be thought of, rather metaphorically, as virtual/digital "piping" linking a tableau of potential user actions (mouse, keyboard, etc.) to a space of possible screen-configurations. A software engineer's goal is to build up this intermediate "plumbing" in an orderly rather than haphazard manner, so that future programmers have a clear picture of how the application's functionality can be augmented (without affecting its current behavior, unless a conscious decision is made to alter the application's pragmatics based on user feedback). Users initiate actions so as to invoke application features: to organize responses systematically, it is important to have an efficient structural breakdown of all the functionality which the application makes available, by analogy to how books might be organized on the shelves of a library. But this structuring principle also promotes extensibilty — once a system is in place to enumerate functionality, it is easier to add new features because there is greater transparency in how new features can "slot in" alongside their predecessors. Similar points apply to the visual display of information: the more systematically we model how data types recognized by an application are translated to visual form, the more readily we can build new display units — such as isolated windows or dialog boxes — to display *new* data types, formulated in the course of providing new functionality.

Planning for continuous-adaptation therefore implies that applications should deliberatively model the functionality they *expose* as features available to users: improvements can either provide alternative (perhaps more user-friendly) ways to invoke these same features, or to create new features that fit comfortably with the existing application by honoring its protocols and user-pragmatic conventions. As such, an important dimension of software design is formulating the protocols and concrete implementations through which applications "expose" functionality.

Consider the case of image-processing (which will be the focus of Chapter 22). Computer Vision software will typically have many built-in functions for transforming and/or extracting data from images (or videos). Exposing these capabilities means allowing users to initiate image-processing workflows. The result of any workflow might be a modified version of the original image, or some extracted data structure that can be outlined or visualized. Because workflows are often chained together, "exposing" functions also involves software invoking procedures dynamically, so that although a user will manually initiate an overarching functionality there may be many behind-the-scenes intermediary stages which users would not directly perceive. In this kind of environment applications will typically offer an API (Application Programmable Interface) and/or an ABI (Application Binary Interface, which is more low-level) such that important features can be accessed both directly through user actions *and* indirectly as intermediate processes launched in response to some higher-level user action.[7] Moreover, users need some way to view or study the results of Computer Vision workflows, so applications should expose functionality for altering their visual display in accord with the details of specific pipelines (e.g., presenting a new image in a new window). In short, applications should expose both analytic/processing functionality and display/visualization functionality.

To "expose" functionality is to offer multiple ways that operations may be accessed: by users directly; by engines which step through workflows; by application "macros", which are (for some programs) programmable sequences of user-actions that typically are performed together (linking multiple actions into one group, initiated with one single action); by "updates" which can add new features without software needing to be re-installed; by "personalization" features which allow programs to enter different states depending on the identity of their current user; by configuration files; and by "scripting" environments where code written in a programming language (typically one

[6] I adopt the convention that a "program" is a self-contained software component, something that can be executed, and an "application" is one kind of program: one with a visible user interface. Most programs are applications, on these definitions (when discussing technical VM matters sometimes however we can use the word "program" in a different sense, as a self-contained sequence of VM operations).

[7] I mention ABIs here in the sense that application-procedures might be exposed to be called directly with raw data values, in contrast to what I will later call "meta-procedures" whose inputs and outputs are padded with extra layers of indirection.

with an interpreter that can be embedded in standalone software, such as Lisp or Python) can modify or fine-tune application state and performance. Supporting such personalization and fine-tuning is itself a feature that makes software more user-friendly, but (for reasons I mentioned vis-à-vis rigorous design) also prompts developers to architect software in well-organized ways, with a logical model of user actions, visual displays, and all the coding that connects the one to the other.

An important dimension of application development, accordingly, is that of modeling and formalizing how applications expose functionality and User Interface details. Virtual Machines, in turn, are a useful tool for analyzing the features exposed by applications (and how they are implementationally tied to the application as a whole). This is the primary scenario where I will consider VM designs as theoretical artifacts in their own right.

### 2.1 Hypergraphs as General-Purpose Data Models

Ultimately, all of a program's functionality is achieved via procedures, which are provided by the software code base (perhaps along with third-party libraries that get re-used, sometimes as high-level source code and sometimes as compiled assets). As such, one way to "expose" functionality is simply to allow external code (that is, code which is not explicitly developed as part of an application's core code base) to call procedures which *are* in the code base. The vast majority of such procedures, however, are "internal," intermediate computations that would not be meaningful or appropriate for external access. Accordingly, applications need a more overarching model of exposed features to ensure that external code utilizes internal capabilities in an orderly fashion.

In general, functionality is exposed to outside components by constructing a specific procedure whose role is to be an "entry point" for externally-invoked features. That is, we assume that any exposed "functionality" requires multiple procedures to be realized, and one entry-point is necessary to invoke a larger package of procedures which collectively implement an operation that we can *conceptually* regard as "one" function. For example, if an exposed capability (in a Computer Vision program) is running a superpixel segmentation on a 2D image via color-watershed, there will be many intermediate procedures necessary to complete that computation, but we can reason about the segmentation as one identifiable step in an image-processing pipeline. It is a *conceptual* functional unit even if it is not "one" function, in the sense of one source-code procedure. I propose to use the term "meta-procedure" to describe functions that are, in this sense, conceptually singular but implementationally multiple.

Application-level capabilities often require certain details to be specified — continuing the superpixel example, such an operation would need to know which image to target, and potentially would require certain threshold parameters (superpixels are relatively small image-regions of similar color, but "how" small and similar depends on input values that might vary from one run to another). The "output" of such a segmentation would be a data structure (typically a "labeling" image, which adds an extra number to each pixel — alongside its existing color channels, such as red, green, and blue — identifying the superpixel to which it belongs); for visualization, sometimes superpixel segmentations will also be displayed by outlining the superpixels with a colored boundary (visually distinct from the underlying image). Thus a superpixel algorithm has inputs and outputs analogous to a single procedure. However, insofar as such an algorithm is not one procedure but rather (in my terminology) a "meta" procedure, these inputs and outputs are not delivered simply as binary data values (memory addresses or CPU registers) but rather must be marshaled and encoded between the software component which invokes the metaprocedure and the application which exposes it.

Unlike the execution-sequence *within* one component, where one procedure can call other procedures directly, for such "meta" procedures there is a series of steps wherein input parameters are built up via a common protocol which both components can understand, and output results likewise encoded in reverse. The procedure/metaprocedure contrast is analogous to the difference between a face-to-face conversation between two parties and a long-distance negotiation where diplomats (or lawyers, etc.) exchange offers and proposals and counter-offers according to a fixed set of rules. Unlike in-person dialog, such indirect communication is less open-ended, and relies on trustworthy intermediaries to convey messages and information without distorting their meaning.

What diplomats, lawyers, and mediators may be to human communication, their analogues in Information Technology would be data-sharing protocols and APIs. Usually, software components which interact via these indirect, carefully mediated pathways can potentially have divergent implementations — they may be written in different languages, adhere to different coding paradigms, and so forth. Therefore, data has to be transferred from

one component to another via neutral formats which are amenable to both sides: data in one context should be encapsulated in a neutral package and decoded (without distortion) on the other environment, and vice-versa. External-programming interfaces therefore depend on data representation systems which allow for information to be restructured according to the needs of different computing environments, while maintaining structural integrity (i.e., decoding the data back to its original form should produce an exact replica of the data prior to its originally being encoded).

Ensuring data integrity across divergent programming environments is a complex task. When being passed between components data is necessarily *serialized*, or converted from its innate binary/in-memory form to a textual or numeric encoding, and then reconstituted into a new binary package which should be "equivalent to" the original (we can define "equivalence" here in terms of bi-directionality: re-encoding the data and sending back from target to source, then re-decoding, should yield a copy of the original; or, more rigorously, modifying a smaller part of the received data and then back-sending should yield a duplicate of the original *except for* the specific change).

Data-serialization formats must be designed to ensure that information is not lost in the encoding/decoding process. For example, a quantity which originally has floating-point type (so it *could* have non-integer values) might get encoded as an integer if, in fact, on a specific occasion it has no fractional part. Then, on decoding, imagine a value interpreted *as* an integer (rather than a **float** which happens to be equal to an integer). Future calculations could then become corrupted (in some contexts arithmetic operations performed on an integer paired with a **float** will cause the floating-point value to be truncated to an integer, yielding incorrect results for algorithms expecting two **float**s), and/or erroneous data sent back to its source. Similar incompatibilities may exist between expected units of measurement (metric versus Imperial, for instance) or value-ranges (it is not obvious from a single number what are the sensible range of values which the number could take on, if modified: magnitudes for individual pixel-colors, for instance, in most image-formats are restricted to the range 0-255; percentages are often limited to 0-100; angles in degree to 0-359, and so forth). Failure to anticipate the full set of metadata conventions and standards which are prerequisite for multiple software components to act on shared data is a common source of data-corruption (in the terms proposed by [30, page 4], "globally inconsistent state").

For these reasons, data-sharing protocols have to be designed around detailed and "expressive" representations, meaning that it is possible to systematically describe all facets of data profiles (types, ranges, units, scales, metadata) that could potentially be sources of ambiguity and encoding/decoding errors. To be sure, a lot of data-sharing happens through relatively simple formats (such as JSON — JavaScript Object Notation — or CSV, Comma-Separated Values) which lack these higher-level guidelines, but such formats typically work in contexts where networking conventions are clearly defined — effectively relying on programmers' discipline and informal documentation to take the place of stipulations formally encoded in data protocols. In more "critical" contexts which should be less susceptible to programming errors — or for more open-ended protocols where networking between disparate end-points should not depend on programmers honoring informal specifications — developers would tend to prefer more rigorous representations, such as XML (with validators and predefined tag and attribute sets), or RDF (Resource Description Framework), associated with the Semantic Web (in conjunction with explicit Ontologies).

In this chapter I will discuss options for *hypergraph* representations (which in some sense generalize RDF, itself considered to be "graph-oriented", but, unlike hypergraphs, not internally multi-scale). I claim that hypergraphs are more effective at permitting rigorous data profiles to be described and confirmed (i.e., we can formulate hypergraph data paradigms that are more expressive still than, for instance, XML and RDF) while retaining the necessary attributes of formal verifiability and transparency. Indeed, hypergraph models in various forms have been proposed as extensions to both RDF (for example, in the transition from graph database engines to hypergraph engines such as HyperGraphDB, Grakn, or AtomSpace) and XML (in the context of "concurrent" markup and related supersets of XML trees, which tend to focus on allowing tags to overlap one another instead of being strictly nested, as they are in XML and its variants, notably HTML).

Although data (meta-) models are an important theoretical topic (particularly when the discussion turns to optimization [1], [15], [13] so that we are considering not only *whether* queries or validations can be completed, but how to do so efficiently), here I am equally concerned with practical software concerns: in particular, the coupling between *representing* information and *exposing* application functionality. The most common reason we seek to (digitally) encode data is to send it between separate components (including ones designed with little or no

explicit inter-connections, except for joint participation in data-sharing capabilities which have to be implemented *ex post facto*). Metamodels are most valuable when they permit inter-operative capabilities in an additive manner — i.e., applications should be able to expand the scope of other applications with which they might network. Representational systems that engender computational artifacts (parsers, for syntax, and validators, for semantics, let's say) which can be implemented, embedded, and leveraged by applications — as their capabilities are refined to support new protocols — are most effective when the amount of effort consumed by the requisite programming is minimized. Data-sharing protocols should thus be engineered to pass through application-integration phases as quickly as possible, but likewise applications should be architected to accommodate new data-sharing initiatives without substantial re-coding.

### 2.2 Examples: Building Information Management and Medical Imaging

Generically, we can describe a metamodel as relatively "expressive" to the degree that data structures and their concomitant semantic paradigms can be transparently encoded according to the modeling system's rules. Expressivity might imply a level of redundancy, or at least superficial redundancy when viewed from the sole perspective of data encoding. For example, so-called "Industry Foundation Classes" — widely used in AEC (Architecture, Engineering, and Construction) technology and BIM (Building Information Management) — employ two different sorts of data annotations (called "attributes" and "properties") which both are roughly analogous to XML "attributes". The IFC attribute/property distinction might seem locally superfluous in that asserting a given datapoint via a property rather than an attribute (or vice-versa) does not appear to convey greater information — the information is borne by the field's value, not by its property-or-attribute classification. However, this distinction is not semantically vacuous; its significance emerges in the larger scale of schema-standardization and validation. Data fields asserting properties of objects (in the sense of real-world physical materials incorporated into building designs) are classified as either *attributes* or *properties*, with the distinction being that attributes are fixed within schemata for objects of any given kind, whereas property-sets can be introduced in objects' context more flexibly. In type-theoretic terms, attributes are immutable parameters in types' schema, such that objects of the same type have the same attributes, whereas properties are not bound to types with equal force (usage patterns might dictate, for example, that common type-instances share property-sets *in some context*, but these are not essential maxims of the type itself).

In the realm of bioinformatics, a similar distinction is made in the context of DICOM (Digital Imaging and Communications in Medicine) wherein "tags" (which have descriptive labels but also two-part numeric codes) are encoded using a system that distinguishes "Standard Data Elements" from "Private Data Elements". The latter are assigned a numeric pair whose "Group" (first) Number is odd. PACS (Picture Archiving System) viewers (software that recognizes the DICOM format) will ignore Private Data Elements by default, so the Standard/Private distinction is intrinsic to protocols through which DICOM data is processed. This distinction is not functionally identical to IFC attributes/properties, but reveals similar motivations insofar as data specifications are also guidelines for implementing software which manages data structures conformant to a given standard. Standardization projects' tasks include defining requirements on software as well as data representation, and it is logistically significant to distinguish standardized parameters that software *should* recognize as a compliance-criterion from non-standard kinds of values that particular users or groups of users working within a given protocol may want to introduce internally (but software may legitimately ignore).

The BIM attribute/property and DICOM Private/Standard distinctions are suggestive illustrations of challenges encountered when applying generic data-modeling principles to specific domains. Most general-purpose representation frameworks (consider XML and RDF, for instance) lack a mechanism to directly express disjunction in parameters' level of standardization, as concretely found in these examples (not to imply that such details could not be represented indirectly, e.g. via meta-attributes or DTD stipulations, but the point is that one is thereby leveraging the affordances of a given representation scheme to accommodate semantic distinctions not specifically anticipated by that scheme, rather than organically encoding the semantics to begin with). Also, note that — for fields indexed by character-strings — descriptive labels are designed to be meaningful for human users/readers, but in most modeling approaches labels do not have an "internal structure" recognized by the framework (at least if we consider, say, XML namespaces as separate labels from element names). The DICOM pattern wherein two-part numeric codes are employed for something akin to namespace/element separation, and then the use of an

odd/even to signal Private/Standard tag rules, is also an idiosyncratic formulation which does not have a natural correlate in multi-domain modeling protocols.

Data-representation theories which are grounded solely in "logical" reasoning, or mathematical representation, can miss such real-world complications. If we start from symbolic logic (or from a mathematical picture of graphs or trees as formal systems) we might be inclined to recognize *properties* (essentially metadata on graph sites, which is how properties work in conventional property-graph database engines), or *attributes* in the XML sense (metadata on tree-nodes), but our theoretical framework could neglect to consider the possibility that a data-sharing protocol might need two *different* property/attribute mechanisms. The semantics of a property/attribute distinction (in contexts such as IFC, or DICOM public/private) derives not from logical models but from real-world implementational concerns. A metamodel which supports this larger semantic context is therefore sufficiently expressive to properly encode IFC or DICOM data; a less expressive model (one which collapses attributes and properties into a generic notion of "fields", say), would fall short, at least without compensating by leveraging its own formal resources (e.g., classifying fields as attributes or properties via meta-data). Here we can appeal again to application-level concerns: the paucity of less-expressive systems would become evident insofar as reifications, indirections, and other ad-hoc solutions to representational blind-spots can render application-integration more complex and time-consuming.

Metamodels are more expressive to the degree that they offer a greater range of representational parameters with which structural and semantic conventions might be communicated. For example, JSON can be deemed more expressive than primitive CSV-style records because JSON distinguishes arrays (which are structurally akin to lists) from "objects" (i.e., associative arrays, lists indexed by field-names rather than numbers). Likewise, XML is more expressive than JSON insofar as elements' data can be asserted either via attributes or via nested elements (for sake of argument, treating XML as a *de facto* superset of JSON, albeit with different syntax, wherein arrays correspond to sequences of similarly-tagged child nodes). Associative arrays in XML might be coded *either* as attribute key-value pairs on *one* node *or* as sibling nodes with unique tag-names. This "redundancy" allows for conventions to cohere (in a given data-sharing protocol) stipulating when one or another of these alternatives should be adopted, thereby supplying an extra layer of semantic detail which is not present in JSON.

In the case of property graphs and/or hypergraphs, data fields can be associated with an "object" (in the sense of an integral data structure) via properties (attributes on a node) or via node-to-hypernode relations (sometimes called "projections"), a duality reminiscent of property/attribute in IFC. Graphs, of course, have the further stipulation that any two nodes (or hypernodes) may be linked by edges (themselves equipped with labels, label-namespaces, controlled vocabularies, and potentially constraints/axioms enforced by "Ontologies"). Formats such as XML and JSON which are more "syntactically" oriented tend to be hierarchical, in that any specific value in a JSON object or array may itself be another object/array (rather than atomic value) and likewise XML elements may have other elements as children. By contrast, formats such as RDF and property graphs which are more "semantically" focused encode relations via edges across nodes (rather than hierarchical nesting). Hypergraphs combine both styles of representation, with hypernodes containing nodes hierarchically *and also* edges between two (or two-or-more) nodes and/or hypernodes (the precise rules as to which constructions are possible will vary from one system to another, but these are reasonable approximations).

Any representational system, as these examples point out, provides a range of parameters that could be pressed into service when formalizing a protocol for encoding specific kinds of data via structures conformant to the specific system. More expressive systems have a wider arsenal of parameters; for instance, along the lines of the above gloss, property hypergraphs (models which either add hyperedges to property graphs or, equivalently, add properties to hypergraphs) are more expressive than either property graphs or document-style hierarchical trees alone, because properties-on-nodes (as in XML attributes), nested hierarchies (hypernodes containing child nodes akin to child XML elements), and inter-node connections (as in Semantic Web labeled edges) are all potential representational devices in the property-hypergraph context. One conclusion to be made here is that property-hypergraphs form a flexible general-purpose metamodel, but the relevant point for the moment is that expressivity can be "measured" via the range of distinct representational parameters afforded by the system, at least intuitively.

This intuition could be made more precise — insofar as I have deferred any rigorous definition for "representational parameters". That is to say, for a reasonably systematic analysis of metamodels we should specify building-blocks of constructions recognized through any metamodel. The overall concept may be clear enough — in general, the pa-

rameters of a modeling system are the full set of structural elements that might potentially be employed in fully describing any given structure covered by the system — but one would like a still tighter definition. For this chapter, I approach this problem from the perspective of Virtual Machines.

## 2.3 Virtual Machines in the Context of Data Metamodels and Database Engineering

The correlation between Virtual Machines and representational paradigms should be clear: suppose we take any structure instantiating a particular metamodel. Presumably, such a structure may be assembled over multiple stages. Insofar as node-hypernode inclusion is a constructional parameter, for instance, then a structure can be modified by including a node within the scope of a hypernode. Similarly, insofar as inter-node relations (via directed edges or hyperedges) are significant constructions, then a structure could be modified by adding an edge between existing nodes. In short, any structure can be derived from modifications on precursor structures. The full set of structure-modifying operations available for a given representational paradigm might then be enumerated as (at least one part of) the opset of a hypothetical (or realized) Virtual Machine. As such, VMs provide a potential formalizing environment for analyzing data metamodels.

Conversely, data-models can serve as a prompt for motivating the proper scope of a Virtual Machine. That is, VMs may be designed subject to requirements that they permit the accumulation of any data structure conformant to a given metamodel. Similarly, VMs could be characterized in terms of how they support various "calling conventions", in the sense of protocols through which computational procedures delegate to other procedures (supplying inputs, reading outputs, spawning concurrent execution paths, and so forth). This section's chapters will focus on VMs based on hypergraph data models, employing such structures both from the perspective of data-representations and interlocking procedures (i.e., describing procedures in terms of sequences of calls to other procedures).

Virtual Machines are useful tools for studying software-interoperability insofar as both data-representations and communications protocols could be modeled in terms of VMs (at least as abstract summaries of working code; or, more ambitiously, application-networking frameworks might provide actual VMs through which applications can route their networking logic, analogous to query languages as host-language-agnostic conduits for database access). As emphasized earlier, applications typically seek to "expose" functionality to external components. In order to do so rigorously, it is necessary to stipulate preconditions on how functionality being externally invoked should be designated — for instance, if a Computer Vision application has multiple algorithms available for many processing tasks, external code needs to specify which implementation is being requested — and how input parameters (and then output results) should be encoded. Applications can use VMs as a kind of neutral environment where third-party code build up representations of exposed-functionality invocations incrementally. Indeed, many programs support scripting via languages such as Python; it is plausible to generalize this idea to VM platforms amenable to multiple scripting languages, so long as they are compiled to the relevant "byte" code. Alternatively, even if applications do not expose VMs to third parties directly, they could use VM-backed processes to process API requests: since VM-targeted code may be updated without recompiling the application, the API could then adapt to new networking situations.

The prior discussion centered on data-sharing between disparate *software* components in general, but in this context VM models overlap with constructions pertaining to interoperability between applications and *database engines* in particular. Consider the general setup of database systems: applications store data structures for future reuse by passing some (suitably encoded) serialization of the relevant information to a database, which arranges the data into a layout optimized for storage and retrieval/querying. Typically the database will attempt not merely to preserve the presented data structure for future reconstruction, but will index or destructure it in such a manner that such specific data can be selected as matching a future query.

At any moment in time, an application will be working with one or more data structure that might be called "live"; they are directly implicated in the state of the application at that moment. The current user (or a different user) might, accordingly, be interested in reconstructing that application-state (or some relevant subset thereof) at a future point in time — at which point database queries are typically necessary, because the relevant information is no longer in live memory. To be sure, a database does not necessarily store "application state" as such — although developers can potentially create "application state objects" and persist those so that users may resume prior sessions — but the typical rationale for persistent data storage is supporting users' desire to resume prior work, return to previously-viewed files, and so forth.

As a concrete example, suppose we are considering an application which (at least as one of its features, and in the context of particular GUI windows or components accessed through the software) supports photo tagging and annotation. Imagine an image database allocated for documenting ecological and/or infrastructure damage from natural or man-made events, such as the 2022 Russian invasion of Ukraine, and perhaps setting the stage for reconstruction plans. A typical image for such an application might be a photograph of damaged building or facilities, or plots of land on which real estate will be rebuilt. Potentially, images would depict remnants of prior buildings that were destroyed, and/or could be annotated with data relevant to reconstruction designs (e.g., in recreating damaged neighborhoods a certain number of residential units might be targeted for each parcel of land or each block subject to redevelopment, perhaps optimized by models measuring the ideal urban density for that specific neighborhood based on environmental criteria, utility grids, public transit, and so forth).

Let's explore this case-study. A characteristic session for such a software component would involve users viewing individual images, previewing image-series depicted via thumbnails, searching for images (based on criteria such as street address, city/district name, or perhaps GIS coordinates), switching between 2D images and 3D street views, and — once in the context of a specific picture — viewing annotations and other associated data in tabular or otherwise structured forms (e.g., tables or key-value pairs displayed through independent GUI windows floating above the graphics viewport).

Assuming such an application works with relatively large image-collections (enough to be impractical for users simply to browse images in a filesystem folder, say), functionality for finding and tracking images would need to be based on systematic query capabilities, i.e., some form of image database: image file paths, feature sets for retrieval/similarity searches, metadata and formatting details, etc., hosted in a database so that images can be selected inside large series by variegated query strategies.[8]

Suppose an image depicts a city block where damaged buildings have to be replaced. Data associated with that image could include estimates of the number of people who lived in this location prior to the Russia/Ukraine war; the number of residential units slated for redevelopment; cost estimates; links to public transit info, street views, data concerning utilities grid, etc. Plausibly, such data would be held in a database and loaded alongside the image, or in response to user actions signaling an interest in the relevant data-points. The database, in effect, is a means to an end, whereas from a user's perspective the important detail is that the application can enter a state where multiple data-points are visible side-by-side: users might view a photograph in one window juxtaposed with a window or windows showing civic/residential data.

Continuing this specific (hypothetical) example, the envisaged scenario has image-viewport components serving as entry-points for a diversity of civic/architectural information. Presumably the specific kinds of data available will vary from one image to another, and users would signal through interactive GUI features their interest in accessing certain branches of the available data over others. That is, assume there is not a fixed metadata/associated information package that automatically accompanies each image, but instead that data and images are linked on a dynamically changing case-by-case basis. That setup would call for coding strategies which work to organize the available information and user-interaction pragmatics coherently. The resulting software-design choices would, moreover, propagate to GUI and database designs as well. Once some aggregate of data is identified as a coherent unit, this structural decision must be accounted for at the GUI level (insofar as users request *that specific* data from application-states where they are viewing an image carrying the appropriate information) and the database-integration level ("that specific" data has to be queried from the larger database context when needed). In other words, the interaction between GUI, database, and application logic is more complex than if each image were given a fixed set of data fields isolated from user pragmatics. This shows how database theory intersects with disciplines (e.g., Software Language Engineering) with practical benefits vis-à-vis efficient application design/deployment.

### 2.4 Database Engineering and Type Theory

Suppose again that one information-bundle that could be associated with (some) photographs outlines redevelopment plans: data points such as the number of residential units targeted, renderings of building designs, contractors, contact information, and so forth. Presumably, analogous data would only be applicable to images showing blocks or plots where relevant plans are in the works; and moreover such images would link to other forms of data as well (about, say, utilities grids). As part of the empirical background, in effect, one might conclude that various facts

[8] Images themselves (i.e., their pixel-data) might also stored as "blobs" — binary large objects — but the term "image database" generically covers cases as well where images files are hosted on a filesystem with only their metadata stored in the database directly.

pertaining to individual reconstruction projects/contracts can both be aggregated (as interconnected data-points) and isolated from other information potentially relevant to an viewed image. In general, software design depends on sensitivity to how information spaces, practically speaking, might be carved and organized. The decision to isolate (say) construction-project data as integral units would be made against that kind of design/decision context. Having this topic reified as a specific "category" of data affects GUI programming and event-handling (because user-visible components must be implemented enabling users to access information in that category, which in turn yields such signals as context-menu activations and the need for appropriate event-handlers) as well as database interop (insofar as data must be packaged in persistable forms). Subsequently, representations of integral construction-project data (in the form of, e.g., a cluster of interrelated datatypes) would be manifest as software artifacts threaded through a code base.

The specific patterns of data organization and programming-language types engineered for an application reflect practical concerns and aspirations to optimize User Experience; these patterns do not necessarily map neatly to native database architectures. Neither relational databases (built up from tables with fixed tuples of single-value columns) nor conventional graph databases (whose representations are confined to one layer of labeled edges) generically match the multivariate and multi-level structure of real-world information typically managed at the application level. This is why application-level data types generally need to be restructured and transformed when routed between applications and database back-ends.

Software engineers are responsible for ensuring a proper synchronicity between application and database state, although (as intimated above) this does not (by and large) entail application-state being directly persisted in a back-end. Instead, back-end updates are a property of application-state at certain moments; insofar as users have performed edits or in general made changes resulting a mismatch between the data as currently seen by the user and what is stored in the database, the latter has to commit such changes for perpetuity. Update-worthy application-state then needs to be distributed over (potentially) multiple database "sites" which are collectively implicated in an update.

For the sake of argument, consider an update (or part thereof) consolidated into a single (application-level) datatype instance; e.g., one object of a given C++ class. Quite possibly, there is no one-to-one correspondence between application objects and database values or records, particularly if the classes in questions contain many data fields and/or multiple one-to-many relationships and values which are more involved than simple strings or numbers (e.g., pointers to other objects). The structural mismatch between application data and database records is evident in technologies like Object-Relation Mapping (ORM) — or "Object Triple Mapping" for the Semantic Web — marshaling data for relational databases or triple-stores, respectively [7], [2], [23], [17], [20]. Engines with more flexible representation paradigms, needing less reconstruction of application data exported to a back-end, can be advantageous precisely because data-persistence capabilities end up consuming less "boilerplate" code [31], [11], [5], [3]. Hypergraph databases are a case in point: the kind of complexity in datatypes' internal organization (multiple multivariate fields for a single object, for instance) which give rise to ORM/OTM-style transforms map organically to hypernode or hyperedge constructions.

Modifying a database entails conveying a package of updates between applications and the database engine. The structure of this communiqué in turn reflects database architecture. An SQL INSERT statement, for example, derives its form from the layout of table-base data models: adding a record entails naming the table to which it will belong, which brings on board the specific list of fields (columns) whose values have to be accounted for in the query. Our impression that relational algebra is a relatively crude or inflexible meta-model derives, it would seem, in large part from the quantity of bridge code needed to implement database updates through query commands such as INSERT that are restricted to the relational architecture. To the extent that hypergraph engines (for example) are more flexible in principle, this advantage only becomes concretely evident to the degree that hypergraph databases support a query system such that updates (and analogous modifications to a database instance) are initiated with relatively less boilerplate code.

As I alluded to earlier, hypergraph data models have comparatively greater parameters available for representing information. In particular, this implies that we have greater flexibility in formulating queries to modify database instances. It is worth mentioning at this point that "queries" need not involve instructions passed to a database engine in the form of character strings (e.g., SQL code). Indeed, forcing applications to build query code on the fly is a often an antipattern (spurring boilerplate code-bloat) — better solutions involve query "factories" that assemble queries via procedure/method calls in a host programming language (perhaps through an embedded

domain-specific language, as one finds with LINQ vis-à-vis C#). On the other hand, in the best case scenario a database would *also* support a query language that could be executed as text strings outside of an application context (and without relying on a host programming language), for examining the contents of a database in situations removed from application-based access (debugging, analytics, general admin functionality, and so forth). In other words, ideally engines will encompass a query engine that works with query-structures compiled *either* from application-code factories *or* standalone query code, which is one rationale for embracing a query-evaluation Virtual Machine.

The specifics of database updates — sticking again for sake of exposition to single type-instances — depends of course on types' internal organization. For typically atomic types (like 1, 2, 4, or 8-byte integers) updates might only entail replacing one value with another, but more complex values ("objects", in effect[9]) with multiple and/or multi-value data-fields updates may include adding or removing a value from a collections type-instance as well as altering such a value, and so on. For many compound types the collection of fields will span more than one "collections" value (e.g., vectors, stacks, queues, deques, and unordered sets/multi-sets, plus map-arrays that are pair-lists with possible nonduplication restrictions on the first element) subject to add/remove operations, in contrast to full-on value-to-value replacement. Some collections have modification-restrictions (e.g., values can only be added or removed from one or both ends of a list, or, as in typical "sets", all added values must be unique) or enforce constraints such as monotone increase and decrease.[10] All of these are potential paths toward legal mappings of type-instances between states which initialized typed values can take on, given their internal organization.[11]

The mechanisms for aggregating multiple values into individual type-instances tend, accordingly, to be more complex for general-purpose programming languages such as C++ (see unions, pointers, arrays, multiple inheritance, enumerations and their base types, etc.) compared to databases (see SQL or RDF types); this is a proximate cause of complications in persisting application-level data. Conversely, databases with more refined type systems can (at least potentially) absorb application data more conveniently. For this to work in practice, however, the engine needs a query system which duly leverages type-system expressivity. Properly recognizing application-level intra-type organization depends on representing (and then exposing to a query interface) the full spectrum of morphisms through which a single type-instance might be updated.

For complex types, an effective query-system would have update operations that are more targeted than just replacing one value with another *tout court*; instead, updates may only involve one or some subset of all data-fields, and (for multi-value fields) could involve insertions/deletes in a collections context rather than a direct value-change. The degree to which a database engine seamlessly interoperates with applications depends on the latter constructing data-packages (without undo effort) that signal the *kinds* of updates requested alongside the relevant new values (notating, e.g., collections-context changes as distinct from single-value morphisms). Intuitively, flexible architectures (e.g., hypergraphs) accelerate the requisite implementations, although actually supporting a query-interface satisfying such ambitions depends on a confluence of factors (e.g., underlying database architecture but also query-representation and query-evaluation protocols and the tools to compile code/text or procedurally-generated queries to internally-represented structures suitable for evaluation).

Although this section's discussion has focused on updating a single type-instance, complex intra-type layout implies a diverse set of query-based update options. An UPDATE statement in SQL, say, only (directly) supports one particular "genre" of updates (value-to-value edits in one or more discrete columns). Given application state which can be expressed in terms of modifications to existing persisted values, keeping the back-end in sync entails accounting for the changes embodies in the new application-state by presenting the back-end engine with (in general) a series of multiple updates. The more flexibly updates can be encoded, the less development time need be expended figuring out how to translate application-state changes to updates the engine can process. This is one reason why a diversity of data-modeling parameters can streamline application integration: a flexible tableau of recognized constructions implies that applica-

[9] Taking the perspective that in some systems objects are formally defined as aggregate structures (rather than atomic data-points) and, even if not precisely stipulated, object/atomic value distinctions tend to align *de facto* with object-classes against, say, built-in types (cf. C++)

[10] Consider a collections type which has only one insertion operation, but will automatically place new values either at the beginning or end of the list to preserve increase-direction; here, new values have to be either greater or less than all prior values. Or, automatically sorted lists need only one insertion operation because the insert procedure would deduce the proper insertion-point.

[11] Nor is this discussion complete; we could also mention reclassifying "union"-type values (whose instances can be one of several types) or various types depending on binary arithmetic (cf. tagged/"decorated" pointers, or enumerations whose value can be members of nominal-value lists *or* bitwise combinations thereof, or unions where multiple type-tags are valid by virtue of shared binary encoding, in effect using one tagged value to update another member of the union — a simple case being integer/bitset unions where setting the integer automatically sets or clears corresponding fields in the bitset).

tions may describe updates with relatively less boilerplate destructuring. Consider the multitude of "sites" where data associated with a hypernode could be asserted. In the context of property-hypergraphs, at least (and some systems may have more elaborate constructions as well), properties *on* a hypernode, nodes *in* the hypernode, and edges *from* a hypernode to its peers. This articulation of site-varieties is, self-evidently, also a list of update-forms. Having multiple protocols for describing updates allows different forms of updates to be recognized with distinct semantics. For example — consider again the attribute/property distinction in IFC — updates to *properties* (which would in general be ad-hoc annotations on a hypernode less strictly regulated than nodes encompassed *in* hypernodes) might be subject to different validators than updates performed via node-insertion or (potentially Ontology-constrained) edge-insertion.

In effect, multi-parametric modeling tableaus in the database *architecture* propagate to multi-dimensional options for encoding updates, which in turn allows for multiple co-existing update protocols each with their own semantics. Applications can then choose which protocol most efficiently describes any particular change in application-state.

Of course, these points vis-à-vis changing *existing* database values have analogs in the context of inserting *new* values. Ideally, applications will have flexibility in how to encode data structure for insertion into a back-end via database queries. This is not only a matter of notating all information which should be persisted, but also providing cues to the engine about how the new data should interact with other records (e.g., using primary keys or globally-unique identifiers to secure inter-record links analogous to — perhaps translating — live-memory pointers) and subsequent find/select queries. What are the criteria through which a database record, once deposited, should be retrieved again in the future? Most database systems will give nodes/records unique global identifiers, but the whole point of typical search queries is that records should be located based on known data in contexts where an application does *not* know the requisite **gid**.

Consider again the case-study of image-curation software that could be used in a redevelopment/urban planning context, such that photos include depictions of future building sites. As suggested earlier, data associated with each picture might reference construction-project data (e.g., the number of units slated to be built, or the identity of the firm contracted to oversee the project), alongside generic information (image format, dimensions, color-depth, GIS coordinates) not specifically endemic to AEC. One consideration when designing such an application would be how users would find images that they hadn't seen before, or had not revisited for an extended period of time (so that the presumptive image **gid** is not cached in recent history). In would make sense to track images by longitude and latitude, for example, assuming that users would know such data. Perhaps street address (accounting for the possibility that large-scale redevelopment might alter the street grid so that pre-war addresses, say, become obsolete; one might still maintain a mapping from such addresses to GIS coordinates so they remain useful for queries); or (less granularly) by district or city. We can similarly envisage scenarios where architectural details are mentioned ("find images of sites within a 10-kilometer radius featuring planned buildings over 4 stories tall").

For queries along these lines to work, back-ends need to structure type-instances such that (potentially large collections of) values can be filtered into subsets meeting specific criteria: GIS coordinates restricted to a given locale, housing matching a given contractor-name, and so forth. When exporting info to a back-end, the relevant details are not only specific values comprised by new data but also which fields may serve as eventual query-parameters, and how they should be indexed. Such "selectability" criteria should be encoded alongside persisted data structures themselves, and convenient application-integration entails supporting protocols for noting pathways for query-retrieval and doing so with minimal boilerplate code (for analogous reasons as with update queries).

Of course, selection/retrieval and update protocols tend to be defined on types rather than the type-instance level. To the degree that certain data-fields are indexed and queryable (for retrieving instances when their global id's are not at hand *a priori*) decisions as to which fields to thereby expose tend to be made for each type — as part of the type's design and contract — rather than negotiated on a case-by-case basis per instance (though note that properties in property-graphs are possible exceptions; indeed this is one rationale for property-graphs in the first place). Similarly, type-level modeling tends to define which update protocols to invoke for different state-changes. Accordingly, an expressive query interface may allow stipulations regarding updates and searches to be described as attributes of *types* as well as single instances (e.g., records and/or hypernodes) and to be deferred from instances to type-contracts (analogously, inferred in instance-contexts by virtue of type attributions). An obvious corollary here is that query systems

should recognize type descriptions as well as encodings of type-instances.[12] Formally, this results in a technical merger between database engineering and mathematical type theory [16, page 21], [6], [28], [14], [9], [8], [29].

## 3 GIS Databases and Digital Cartography

This discussion has just touched on themes related to flexibly-typed database systems: the idea being that expressive database engines support *strongly typed* data (in the sense that all records have type attributions that offer guarantees about their internal structure, such as a specific set of defined fields) but also allow new types to be introduced into an active database. A good case-study of where such design is apropos is that of maintaining geospatial databases that track information associated with geographical coordinates — for example, roads, bridges, buildings, waterways, transportation infrastructure, or land-plots, but also any kind of scientific data (environmental, sociodemographic, etc.) that can be overlaid on a digital map. GIS systems are typically divided into two distinct data categories: there are "basemaps" which represent the most common or generic types of GIS data (roads, buildings, and the like) that are then augmented with supplemental "data layers" that are narrower and more domain-specific (such as ecological data). Digital cartography renders *basemap* visuals by subdividing geographic areas into relatively small segments (within a 256×256-pixel area, for example) translating geospatial (typically latitude/longitude) coordinates to image-coordinates. Digital maps are composed at multiple scales of resolution; in a common format, so-called "XYZ" tiles, the lowest zoom level produces a "map" of the whole world (basically an outline of the continents) whereas the highest level (around 20, or slightly less depending on the environment) is sufficiently detailed that maps show the contours of individual buildings and street-intersections. In XYZ, each successive zoom level doubles the underlying magnification, so that each tile at one level splits into four tiles at the next level.

Software known as "tile renderers" and "tile servers" [22], [34] map latitude and longitude coordinates (or other GIS units, such as those of Marcatur projections) to tile-coordinates that include zoom level as one axis (the "z" in XYZ). An XYZ triple designates one geographic area at a specific zoom level; individual latitude/longitude and Mercatur points may then be identified via fractions of the tile width and height. Via such mapping, locations of objects in a GIS database are converted to pixel-coordinates, so that features such as roads and building can be marked visually. Tiles are rendered according to graphical rules called "styles", which stipulate details such as the colors that should be used to outline buildings or roadways, or the icons for points of interest (transit stops, historical sites, special-purpose buildings or areas such as schools, hospitals, parks, government offices, and so forth). Tile-rendering requires some computational algorithms because streets/highways and other noteworthy spots need to be named or labeled, and the positioning of these identifiers (unlike the fixed latitude-longitude of the labeled locations) is indeterminate, so cartographic engines try to calculate how to position street-names (for example) so that this text does not intersect with other map data (such as roadway lines).

The end result of a tile-rendering pipeline is a miniature map in the form of an *image*, often via photo formats such as PNG or JPEG. A GIS application will form maps that users see by composing together multiple tiles. This provides the "basemap" layer, representing a general-purpose view of a given geographic region. On top of these base maps, applications will usually superimpose domain-specific data points (sometimes called "attribute" data) relevant in specific contexts: data layers might show locations where Ukrainian buildings or infrastructure was damaged in the 2022 war; or distributions of COVID-19 cases; or environmental hazard sites; or tax lots and zoning codes for architecture, real estate, urban planning, and property management; or sightings of specific plant or animal species, and so on. In effect, whenever scientists or researchers have data sets indexed by geographic coordinates — epidemiological, ecological, sociodemographic — such information can potentially become a Data Layer which is visualized by superimposing special-purpose icons or colorations on a generic basemap. These extra layers can be added at different processing stages: for instance, they might be consumed by the tile servers themselves, as a post-processing step immediately after rendering basemap tiles; or they could by superimposed on basemaps by GIS front-ends, immediately prior to showing maps to users (in this case applications would typically download tiles and data layers separately, and merge them together at the last moment).

I provide this digital-cartography overview mostly to mention certain details concerning GIS data systems. Al-

[12] By implication, query languages would then require, as a subset, "type-expression" languages where types' attributes, internal organization, and back-end protocols are duly notated (so that type-information can be "loaded into" the system and consulted as the engine resolves how to accommodate insertions, updates, and filtering for specific instances).

though there is a core dimension of familiar mappable elements (roadways, buildings, and so forth) with common presentation guidelines (formalized via tile-rendering styles), the space of information that could be added onto base maps is open-ended. Any data set with latitude/longitude coordinates (or fields convertible to such coordinates, such as a street address) can be visualized against the backdrop of a GIS basemap.

If we consider GIS data to encompass common basemap info *together with* special-purpose supplemental layers, then GIS databases are a good example of "strong but flexible" typing — it is impossible for a GIS system to anticipate *a priori* the full range of data profiles evinced by objects visualized in a map context, within supplemental data layers. Every mappable object has latitude/longitude coordinates (or a collection thereof, giving a geometric outline), but in addition objects will encompass other data-points as well, relevant to their specific domain (civil engineering, etc.) but opaque to mapping applications themselves. In this sense GIS systems manage non-GIS data by tracking opaque objects without consuming them directly — it is up to individual applications to download data within supplemental layers and reconstruct the relevant domain-specific info, so that users can employ street maps as entry points for accessing information which is then visualized through other features in the application. For example, urban planners taking on the responsibility of rebuilding Ukrainian cities might start with street maps showing sites of destroyed buildings and infrastructure, and then (typically by clicking on a map) transition to windows showing structured data about individual locations, such as the number of families formerly living in a damages residential complex, or details about utilities and water/gas/electricity grids, or estimates of environmental hazards (chemical leakages, say, or unexploded munitions) resulting from the conflict.

The point here is that objects containing such domain-specific data need to be packaged and stored in a GIS database even if they are not GIS data proper, and then correctly reconstructed and presented to users in conjunction with digital-cartography front-ends. This data-management workflow is roughly analogous to the case of routing data packages across third-party networks discussed above (in ways I will summarize more completely at the end of this section).

### 3.1 Geospatial Data and GUI Events

Before that, though, a final topic relevant to the current discussion relates to front-end implementations for GIS software.

Most of the time, users consult GIS maps with the intent to find and get information about specific addresses or locations, i.e., to switch from a cartographic overview to a more detailed display of information specific to a given spot the map (perhaps a restaurant's menu, or departure times at a bus stop, or a business's hours of operation ...). Presumably their GIS software (which can include domain-specific applications that have digital-map windows as one front-end format among others) has the capacity to render such location-specific data in a format useful to the user. When accessing street maps via a web browser, for example, the simplest way to access non-GIS data is to open third-party websites in a separate browser tab. Consider users intending to search a library's card catalog: finding the library on a map, they might then rely on the map generating popups with location info, hopefully including a link to the library's website — following that link would allow the user to query the desired catalog.

Outside of a browser, on the other hand, we can imaging a city's overarching library system providing an app which includes cartographic displays as one window where users can seek individual library locations, giving then the option of clicking on a locations associated with a map icon, but then presenting a catalog-search window as a GUI component of the same application, rather than a separate web site. This would be an example of an application which implements both digital-map windows and other forms of displays side-by-side, routing the user interface between then based on user actions.

Whether via multiple intra-software GUIs or through external web sites, the common denominator in both scenarios is that users interact with GIS maps as a way of requesting information about specific places. Supporting GIS front-ends for these purposes requires, at a minimum, the ability to map geographic locations to site-specific information. For example, a library-system app would need to know which latitude/longitude coordinates refer to which library branches, so that users clicking on an icon at a given geographic coordinate would be routed to the appropriate card catalog. In terms of front-end development, it is likewise essential for the software to know how on-screen coordinates map to latitude/longitude pairs.

This last step may seen obvious, but most users probably do not notice the computational details involved in deciphering their actual GIS locations. When a user sees a map, they intuitively understand that each point in the display corresponds to a specific GIS position (with identifiable latitude and longitude coordinates). Clicking

a mouse when the cursor hovers over a given site — or tapping on a touch-screen — is thereby understood to be a gesture which signals the user's interest in selecting the location thereby singled out as a focus point.

From the application's point of view, however, the user-action is not intrinsically endowed with geospatial axes, but instead is localized on the screen (they have clicked on a given pixel-location relative the overall dimensions of their screen/monitor). There are a variety of mathematical quantities to be tracked when converting screen-coordinates to latitude/longitude, including the current zoom level, and the specific geographic rectangle which is visible in the map's view-port (following how the user has "panned", viz., orthogonally scrolled the display, so that the click-location is quantified as a certain distance right and down from the window's top-left corner, which corresponds to moving a certain distance west and south). Moreover, because map-tiles are acquired through coordinate-systems such as XYZ rather than through latitude/longitude rectangles directly, applications will typically need to map on-screen coordinates to (e.g.) XYZ triples as an intermediate step, and then on to latitude/longitude as desired (in the case of XYZ, there might actually be still a further coordinate-transform involving Mercatur projections).

Although the equations to effectuate the various inter-coordinate calculations just outlined are not prohibitively difficult to implement, an application's ability to properly respond to front-end user gestures depends on having complete information about the digital map which the user sees at any point in time, prior to his or her interactions (e.g., current pan and zoom state). Depending on the map-source used, such information may or may not be available. For example, a GIS font-end could be constructed through a web-view window, pointed at a street-map URL (e.g., GoogleMaps). Via this arrangement, however, there is no way for the host application to get detailed pan/zoom info from the embedded map display, so the host software has no way to translate user actions (such as mouse-clicks). The application could certainly register the fact that users have clicked on a given screen coordinate, but they will not be able to convert screen coordinates to latitude/longitude without knowing the current map-state, and this is not data that systems such as GoogleMaps make computationally available.

For these reasons, relatively full-featured GIS applications have to manage map-rendering displays on their own (rather than deferring these operations to an embedded service), which involves connecting to a raw map-tile service (a tile server) rather than to a higher-level component such as Google Maps, and internally managing issues such as local tile-storage and scheduling updates. This is why open-source tile servers such as Open Street Maps are an important contribution to the software-development ecosystem, because they remove many barriers that would otherwise exist to applications implementing fully functional GIS front-ends as one facet of their user interface.

On the other hand, even well-intentioned projects such as Open Street Maps have certain limitations (including data usage restrictions), so large-scale projects have an incentive to implement their own tile-server back-ends rather than rely on third parties. Consider the case-study of Ukraine reconstruction: assuming multiple organizations and international funders commit to significant reconstruction projects for Ukraine — and that groups agree to collaboratively engineer and utilize digital resources for sharing information and ideas that could be useful for reconstruction projects — it would make sense for the shared resources thereby deployed to include a dedicated tile-server specific to digital maps of Ukrainian territory, including both maps designed to show street/urban details and to diagram natural/geological features. Apart from giving software open-ended access to raw cartographic images and GIS data, a special-purposes tile server could choose rendering styles and conventions (for instance, duplicating tiles with street-names, place-descriptions, and other textual info in multiple languages and alphabets: Ukrainian, English, German, etc.) collectively designed to maximize maps' usefulness for the specific purpose of a multi-national rebuilding effort. This could ensure that (e.g.) users unfamiliar with the Cyrillic alphabet would be able to see versions with transliterated place-names; and that base-coloring for features such as roadways and building-outlines would not conflict with the display of non-GIS data (e.g., tracking patterns of refugee displacement and return).

When analyzing front-end event-processing, a further detail involves distinguishing between user gestures targeted at visible base-map features and those focused on supplemental data layers. With respect to base maps, although structured information such as details about individuals roads and buildings is implicit in tile-images — because GIS databases are consulted in order to render tiles in the first place — the actual tiles which GIS front-end components receive from tile servers are just images, which function from a software-development point of view more or less like photographs. Although a front-end event handler can (with suitable state-info) translate event coordinates (screen locations) to latitude/longitude,

the application would not (at least from map-tiles themselves) be able to ascertain whether the user's selected latitude/longitude point corresponds to a particular building, a roadway, a street intersection, transit point, or any other kind of GIS feature. This further information must obtained from a remote source (such as an address-lookup service) and/or a local database which applications maintain internally. In the case of Ukraine reconstruction, for example, a dedicated tile-server could include information packets co-extensive with tiles identifying street-map landmarks keyed to geospatial locations, so that client software would be able to translate screen coordinates to points-of-interest.

These comments apply to details visualized through base maps. The situation is different, however, with respect to supplemental data layers. In this case, typically, front-end applications acquire base tiles and non-GIS data separately, and then overlay the latter on the former prior to rendering map windows for users to see on-screen. Because the client application thereby *itself* correlates non-GIS data objects with map locations, the correspondence between these objects and latitude/longitude coordinates is already present in the system (it does not need to be externally queried from an address-locator, for instance). At the same time, there are still computational details which applications must reckon with in order to properly route user events. For example, implementations can produce the visual effect of non-GIS overlays visibly superimposed on base map background in different ways.

By was of illustration, consider the options for a GIS front-end coded via Qt (the most popular cross-platform GUI framework for C++ and, via bindings, for most scripting languages). There are at least two possibilities: in one case, data layers could be superimposed on base maps by overriding the functionality for "painting" base maps as images, such that icons, diagrammatic elements (arrows, lines, and so forth) or color-effects (e.g., semi-transparent colors to represent something like population density) are "painted" on top of base-map images. In other words, non-GIS layers are painted onto GIS tiles via techniques similar to how those tiles are rendered with GIS info proper (roadway/building outlines, etc.) in the first place. The result of that operation would be a composite image which, perhaps via a **QImage** subclass, gets treated by the Qt displays as equivalent to ordinary images (with no internal structure). In this case all user events would share the same profile, as signals relative to image-coordinates, and the application must determine whether a mouse-click (or similar gesture) is targeting a point in the base-map image or an overlay painted over it.

On the other hand, an alternative technique would render the composite of base-maps and overlays through **QGraphicsScene**, where the base-map is treated as a raw image but overlay data is introduced as scene-objects (for example, icons representing specific points where non-GIS data is accessible), that the application tracks and holds in memory as data structures fully separate from base-map images. In this case, clicks on scene-object icons would be registered via different event-signals and a different application-response pipeline than clicks on base-map locations, so programmers have to engineer two different event-handing systems, one for base maps and one for supplemental layers. These two options ("painting" non-GIS layers versus introducing them as graphics-scene objects) have respective trade-offs: different functionality is more or less complex in one scenario versus the other, and vice-versa.

This specific example — implementing GIS displays with supplemental data via Qt windows — is a good case-study in how applications should systematically prepare and document their event-handling logic. Failure to clarify how a Qt application is approaching the task of rendering layers on GIS base maps, and on how these design choices propagate to event-handling requirements, could make such applications difficult to maintain once the original developers move on to other projects.

### 3.2 Representing Functional Organization

Supplemental GIS data layers are good examples of a computational theme which recurs in multiple contexts: the co-existence of spatial and function design or organization, which software seeks to model, simulate, or visualize. The domain of "space" can vary; in the GIS context *spatial* is actually *geospatial*, where "space" is construed as features and locations on the Earth's surface (plus, sometimes, height above or depth beneath the ground). When mapping utilities, transit systems, traffic flows, and so forth, we are dealing with multi-part networks that are spread out over a geographic area, and geospatial relations are a fundamental dimension to the functional interrelation between disparate parts. But such systems are also comprised of non-geospatial (or not-entirely-geospatial) relationships: transfer points, in the case of transportation; highway exists and roadway capacity, in the case of traffic simulations; aquifers, water mains, and treatment centers, in the case of utilities (specifically water systems), etc. It is a spatial fact, for instance, that two roadways or train lines cross at a given point, but whether these points permit exits/transfers is a functional relationship — geospatial line-data alone does not report whether or

not drivers can turn off one road onto another, or riders switch from one line to another, at their contact-points.

Other domains have other concepts of space (not necessarily *geo*-spatial). Architectural and building systems assemble into the 3D form of a house or factory, but they also capture the machinery of modern-day construction, with electricity, heating/cooling, ventilation, fire safety, and so forth — a wall is not just a spatial border between rooms but a stopgap against a spreading fire, a barrier to sound, a perimeter enclosing rooms which have specific entry-ways and dimensions. Or, in robotics, Image Processing techniques parse visual input to create a spatial map of a robot's environment (here "space" is both the internal pixel expanse of an image and the projected ambient space where a robot moves), but robots also need to understand their surroundings in functional terms — understanding external objects as artifacts to be moved, obstacles to be avoided, affordance enabling the robot's own movements (walking steps, say). In all of these contexts spatial and functional systems are overlaid one on another, where spatial relations constrain and substantiate functional relations but additional parameters and axioms are needed to describe the functional systems in detail — simply mapping out their component's spatial networks is not sufficient to define functional-organizational properties (again, say, intersections between highways or train lines may or may not be a traffic link or transfer-point).

Formats such as Industry Foundation Classes have evolved alongside "green tech" and "smart grid" technology, or in general an increasing emphasis on employing computer models to promote sustainable, eco-friendly engineering; and these use-cases exemplify the co-engineering of spatially and functionally articulated data structures. For example, architects have learned that carefully selecting building plans and materials can lead to more efficient designs — ideally, "net zero" carbon footprint — by limiting a building's reliance on electricity (and environmentally harmful chemicals, such as freon) for, e.g., heating and cooling [33], [26]. Organizations such as the World Resources Institute have promoted icons and visual markers that may be introduced in graphic media to indicate buildings' energy-efficient measures: double or triple window panes, evaporative or radiative cooling, natural lighting and ventilation, and so on [4, page 22]. Such icons might be positioned on 2D images — photographs, artist's renderings, floor plans — or also on digital maps; consider street maps annotated with information showing how a particular community is planning to reach carbon-emissions goals. As a visual cue summarizing green-tech designs, such icons may also become interactive GUI features, where users could click on the icon to learn about the building materials chosen (specific kinds of windows or HVAC installations, say) and how they reduce carbon footprint.

As a case in point, warm-weather properties can minimize the use of air conditioners by orienting exteriors to create shaded areas and position windows away from sunlight, arranging ventilation systems to produce cooling air flows (which also has a drying effect, reducing interior moisture that could yield health hazards), selecting materials that naturally absorb heat — and potentially harness solar and/or wind to offset energy use. Computer models come into play because concepts such as "net zero" emissions are not abstract goals; they are concrete targets that may be measured and predicted. Software can estimate a property's energy efficiency while its architectural plans are still being finalized, and CyberPhysical sensors might track actual energy use post-occupancy (which in turn evaluates the accuracy of the initial simulations).

In this sense green technology has become an important component of Computer Aided Design in domains such as architecture and civil engineering. In addition to their roles as construction blueprints and visual tools to assess the aesthetics and livability of property designs, building plans thereby gain an additional dimension as data structures defining the functional organization of physical constructions, which in turn is subject to analysis by special-purpose software components which simulate air-flow, air quality, energy use, and related metrics that collectively quantify building efficiency. Although spatial relations among architectural elements are obviously significant factors in these simulations, such analyses require additional data based on mathematical models of (e.g.) air currents or solar-thermal heat gain [12]. Because CAD applications (for conventional AEC) are not "scientific computing" platforms in most contexts (apart from measuring buildings' structural integrity, perhaps) such "green tech" analyses require external software or plugins taking CAD data as a starting point but then incorporating additional, domain-specific parameters.

Similar principles apply to non-geospatial attributes within GIS data layers. Functional networks overlaid on GIS terrain can be analyzed much like structured systems within architectural space. As a concrete example, consider hydrology: indeed, I'll continue the earlier case-study of Ukrainian reconstruction. It has been well-documented that the 2022 war crippled the country's urban infrastructure in many places, so that utilities like water-distribution systems have to be rebuilt alongside

residential and industrial units. The multi-faceted scale of reconstruction priorities actually gives preliminary designs greater flexibility, because engineers are not limited to repairing utilities to meet the needs of a fixed residential population, or conversely accepting constraints on new construction due to limitations of existing urban infrastructure. Under the assumption that refugees might want to return to their prior cities or neighborhoods but not necessarily their exact previous address, planners have leeway to explore how to reconstruct both residential housing and utility systems to optimize factors such as clean water and inexpensive electricity: density of habitation can be structured to minimize the costs associated with providing energy, water, gas, waste disposal, etc.

In the case of water systems, rebuilding plans would reasonably start from existing aquifers and water sources (such as lakes and reservoirs) which are geological artifacts, not man-made ones. Once these sources are tapped, however, water-distribution networks have some level of flexibility, allowing for engineering choices: what kind of pipes to use, where to place occupied buildings and hydrological infrastructure (such as holding tanks), how to recycle wastewater (which points to the functional interrelationship between water and sewage systems), how to provide irrigation (which likewise connects water distribution to agricultural optimization), how to protect water-sources from contamination due to floods, existing hazard-sites, and so forth. Such assessments depend on structural models of water utilities as networked systems, where mathematical models can compute the quantity of water that may run through specific network-paths (to minimize the risk of building's losing access to water when one part of the system is damaged, for example), or to estimate water demand (based on the types of properties, considering the different needs of industrial, commercial, or residential locations) which are water systems' end-points, or to simulate how floods or environmental hazards might damage and/or contaminate the system. Computationally, such network models would normally represent water distribution via graphs, where (for instance) nodes represent physical sites (aquifers, buildings, treatment sites) and edges represent infrastructure like sewage drains or pipes, which might then be annotated with numeric data (for instance, a water main or secondary pipe's maximum capacity).

Graphs representing water systems in this sense could be geospatially visualized, because most of these structural elements have fixed geographic locations — i.e., the nodes in hydrological graphs may be assigned latitude/longitude coordinates and therefore the entire network can be laid out alongside street grids or geological feature-maps. At the same time, functional analysis of water networks would need to consider many engineering specifications apart from GIS position alone (e.g., the capacity and manufacturing details of pipes and storage tanks). This would be an example of non-GIS data that might be curated within the scope of a geospatial database but consumed by domain-specific client applications, rather than digital cartography renderers themselves.

Consider the following scenario: a company, or non-profit, that specializes in hydrology is tapped to produce proposals or assessments of reconstructed water systems considered for damaged Ukrainian cities. That work is encapsulated in structural models which simulate the design of new water networks, with system details mapped out via overlays on Ukrainian street maps. Simultaneously, architectural firms are at work formulating plans for new residential (and commercial/manufacturing) sites to replace infrastructure irreparably damaged during the war. When designing an apartment complex to house a specific number of families, for example, architects could then estimate the tenants' anticipated water-use and refer to the hydrological company's simulations to check that their proposed networks will provide adequate water for the new building, modeled as an end-node within the overall graph. From a computational perspective, we can see this situation as a kind of multi-part analytic framework where the software employed by the hydrologists and by the architects should interoperate with sufficient rigor to allow the water network models to answer questions arising in the architectural context. For example, the architect's software might need to obtain quantitative parameters specific to the water-distribution proposals and use such data as inputs to simulations of future residents' water consumption.

We may envision variations of this scenario, too: environmental engineers could similarity consult water-system proposals but run simulations of the network's performance, not in the context of residential demand, but rather for the effects of drought and floods, or the risk that water in the system could be tainted by environmental contaminants or flood runoff. Or, agricultural engineers may simulate the performance of water-supply when these systems are tapped to provide irrigation, which in turn would vary by season and in conjunction with choices related to crop-variety and cultivation methods (whose computational models would be derived from agricultural science).

Because this chapter is focused on computational meth-

ods rather than botany or hydrology *per se*, I will not examine the specifics of the different scientific domains in play for these various examples. Instead, I want to focus on the application-design concerns which these case-studies illustrate. We have here a hypothetical water-distribution network model serving as a core data source which multiple applications, each with their own specific domains (architecture, ecology, farming) will tap for parameters and calculations specific to their area of focus. All of this software has to interoperate according to data-sharing protocols that ensure each application's access to its requisite information despite their differences in scientific domain. Accommodating multi-faceted information networks along these lines is a more complicated task than engineering multi-part services spanning applications with largely overlapping functionality (e.g., passing Electronic Health Records between biomedical facilities, or supporting electronic payments/transforms among e-commerce and/or financial services applications).

Furthermore, in addition to the data-sharing issues just identified, the water-network example also points to the role of GIS data: although the most important network data is not intrinsically geospatial, it is nonetheless true that many details of a water system can be visualized in a GIS environment, and geospatial relations are among the parameters that would be considered during simulations (for example, the geographic distance between a water source and building location would presumably be a factor in predicting uninterrupted water supply based on, e.g., network redundancy). Visualizing water systems against a street-map background is a good framework for human users to picture the network in overview (as opposed to schematic diagrams, for example). We can thus imagine that front-ends for the various applications accessing the water-network data might use digital maps as an entry-point. Civil engineering software, for example, could give users the option of selecting a specific building by starting with a street map, zooming in to a desired location as necessary. In the context of hydrology-related simulations, this front-end functionality might be augmented by rendering water-distribution data within the maps which are presented to user; e.g., showing water routes drawn as GIS "ways" analogous to higways or transit lines, and providing icons marking individual building's connections to water systems as entry-points for requesting data about an individual property's water use/access.

In this environment, then, software simulating water networks would be depositing data in an overall GIS environment wherein specific data packages can be associated with geospatial locations, even if the encapsulated data is not intended for use by GIS software directly. Geospatial location then serves as a kind of index through which disparate applications could obtain non-GIS information so that they may then unpack and plug in to their own analytic capabilities. Here GIS databases act as way-stations for data packages routed between disparate applications. Pehaps an analogy to old-fashioned (non-electronic) mail is appopriate: the post office does not examine the contents of someone's letter, but it implements a deposition, storage, and delivery service allowing packages to be sent from one point in the country to another.

Notice GIS systems thereby potentially serve a purpose which is a variation on cross-application data sharing considered earlier in the chapter. Before, I mentioned the use of APIs allowing an application to receive third-party data from an external source, "exposing" specific capabilities which permit the host application to correctly interpret and process such external information. That prior discussion made the implicit assumption that applications in such an arrangement would be inter-communicating directly: an originating component explicitly connects to a target application via the latter's API. However, the case of non-geospatial data in a GIS database points to how cross-application interop can be more delayed and circuitous. Hydrological software could deposit data about water-networks through GIS systems which are subsequently consumed by third-party applications (for architecture, agriculture, and so forth) without either application having direct contact: all data would be routed and stored through GIS components (which, in the general, would not consider non-geospatial data directly, but at least would keep track of its association with geographical coordinates and GIS objects proper, such as buildings or geological features).

Having outlined the structure of this kind of mediated data-sharing — relying on intermediate components such as a GIS database to allow cross-application networking without direct inter-application connections — the next question would be to address the computational requirements of data intended to be shared through an indirect, multi-party environment and of the components wherein data packages are constructed, then stored, and then acquired and reused. The following chapter will present arguments that hypergraph data models are particularly well-suited for this kind of multi-part protocol, both with respect to formalizing data models and with respect to analyzing computer code through which the functionality to (respectively) construct, store, and acquire the respective data packages is implemented.

# 4 Conclusion

This chapter has briefly touched on several disparate themes in the realm of software engineering, themes which will be further examined, to varying degrees, over the next three chapters. In particular, domains such as Geographic Information Systems and Building Information Management present use-cases for multi-application data sharing through intermediaries such as a GIS database; for the juxtaposition of spatial (e.g. geospatial, image-space, or 3D/CAD designs, or some combination thereof) and functional structures, with spatial and functional facets analyzed and visualized side-by-side; and for application-front-end event-handling pipelines, discussed here in the context of users interacting with digital GIS maps. Each of these contexts present engineering complications that software components need to manage — these are the sort of implementational tangles that can, in the absence of well-constructed code libraries and design patterns, inhibit our ability to engineer applications in a timely and cost-effective manner.

Situations such as Ukraine reconstruction point to how software-development inertia has real-world consequences: ideally, a variety of special-purpose applications will come on board soon after an envisaged Russian cease-fire, helping the international community to rebuild Ukrainian cities and facilities in short order. The more user-friendly and feature-provisioned this custom software, the more effectively civil engineers can leverage the sheer scale of needed reconstruction to implement "smart city" technologies.

For each of these three themes (decentralized/mediated data-sharing, spatial/functional data juxtaposition, and event-processing) we may identify specific concerns that add layers of complexity and engineering requirements to the respective classes of applications. In terms of data sharing, programmers should identify how to package domain-specific data in formats that can be stored in a domain-neutral setting (like a GIS database) and then reconstituted by other applications with sufficiently overlapping capabilities, a problem which involves both how future applications will query for the specific information they need and how they will build in-memory structures (in their own coding environment) which are properly aligned with and behaviorally replicate the objects originally deposited. Programmers can certainly use object-oriented techniques (for example) to encapsulate the properties and behaviors of specific object-classes, but it requires extra layers of design to build object-systems which are sufficiently semantically explicit to orchestrate cross-application interoperability. In the context of event-handling, applications should engineer a centralized protocol for documenting event signals and response-routines. It is one thing to code event-handling procedures (e.g., via Qt signals and slots); situations such as GIS front-ends demonstrate however that applications require a further dimension of structural coherence, insofar as a code base needs not only to provide procedures responding to user-events, but to allow the total system of event signals and responders to be documented, queried, and augmented in an orderly manner. Frameworks like Qt provision support for the base-level implementations, but other paradigms are necessary to architect the overall system in this kind of logical manner.

And, finally, with respect to spatial/functional organization, applications need to construct data-packages encapsulating system-functional details which can be keyed to locations such that spatial coordinates serve as indices and handles to extra data layers (again, "space" here can have different or multiple meanings: geographic extent, image-space, the virtual space of CAD designs, etc.). At one level, associations between spatial points and data objects might be covered by a straightforward key-value mapping, with locations as keys and object-identifiers as values. But the larger problems of integrating spatial and functional data are more complex: how should GUI displays targeted at spatial arrangement (GIS maps in the geospatial context, say, or floorplan blueprints in architectural CAD) be extended to include icons or signal-generators such that users can access data (or access functionality leveraging data) handled by functional-analytic plugins/components coexisting with the spatial displays? How should underlying "spatial" data stores (such as GIS databases) support the depositing, querying, and retrieval of objects that are largely opaque to the stores themselves (as in non-GIS data integrated with GIS databases)?

To appreciate the full scope of these issues we should consider the problem-domain from the perspective of a programmer intending to support analytics concerning functional organization in an environment where data is predominantly organized in a spatial context (for some notion of "space"). In GIS, for instance, imagine the perspective of a developer building an application tracking data structures (consider water systems, as in the above examples) where functional units have geospatial coordinates but also many parameters which do not measure GIS magnitudes or attributes. Ultimately, such data must be encapsulated into persistable units and linked to GIS locations, so that peer applications can pull the relevant objects from a database via (say) latitude/longitude. How

would engineers know that the full range of parameters is properly reconstructed by external applications so that their computations or simulations are consistent with the originating applications' models?

It would be one thing if external software was simply cloning the original program's full data sets, with some guarantee that the later computing environment is sufficiently analogous to the former that both copies of the data will be behaviorally equivalent. However, working through a GIS intermediary occludes this synchronization: we assume that later applications are obtaining data piecemeal from specific GIS locations or areas, rather than getting a full-scale replica of an original information space (in the context of large-scale data sets, such as a hydrological survey of an entire country, cloning the data *tout court* could well be impossible; even for smaller-scale data sets, however, working with partial samples rather than full copies of original data might be more consistent with the later application's requirements). How can originating applications guarantee data integrity and analytic accuracy under the assumption that external software will run analyses over parts rather than full copies of their data sets? How can these applications identify the "operational closure" of any given sample within their data sets such that these integrity validations are guaranteed?

Current IFC standards are a good illustration of how these issues play out in practice. Seen as a design-model, CAD systems compose IFC elements as building-blocks which piece together to define an integrated data space through which architectural plans may be visualized, in various formats (blueprints, 3D models, "artist rendering" style 2D images). At the same time, the structural network evinced by interlinked IFC data can also be queried as a foundation for simulations involving buildings as functionally organized systems (with respect to energy consumption, fire safety, internal temperature dynamics, and so forth). Structured chains of IFC objects thereby serve as a multi-faceted foundation from which, on the one hand, spatially and visually oriented resources might be constructed (such as floor plans) and, on the other hand, data needed for functional simulations can be initialized. Such integration between spatial and functional structures is driven by the coexistence of parameters within IFC objects that address concerns related to buildings' functional-systematic profiles (energy efficiency, for example) as well as those related to CAD model-space. These facets of the IFC data model could potentially serve as an inspiration for analogous standardization of information classes within other domains where spatial and functional networks co-exist, such as GIS or — in the sense of visual object-recognition alongside structured environment-models — robotics and Computer Vision.

Setting aside the specific details of IFC, however, the issues of spatial/functional juxtaposition, event-handling, and mediated multi-application data sharing point to software-engineering challenges that — one could reasonably claim — point toward next-generation software-development environments. Existing programming tools and design patterns arguably do a relatively poor job of addressing such concerns effectively, measured by how readily developers have tools at hand to implement code which manages the relevant complexity in a rigorous manner (so that applications can be deployed more quickly, and maintained for a longer or more productive life-span). These are the kind of practical challenges which call for new ideas in Software Language Engineering, to circle back to the discussion at the start of this chapter. In the next chapter I will focus in specifically on one such paradigm, namely hypergraph models of computing processes, query evaluation, and data interoperability.

## References


[1] Renzo Angles, *et. al.*, "Foundations of Modern Query Languages for Graph Databases". http://renzoangles.net/files/csur2017.pdf

[2] Jeroen Bach, "Theory and Experimental Evaluation of Object- Relational Mapping Optimization Techniques: How to ORM and how not to ORM". Thesis, University of Amsterdam, 2010. https://homepages.cwi.nl/~jurgenv/theses/JeroenBach.pdf

[3] Jeffrey M. Barnes, "Object-Relational Mapping as a Persistence Mechanism for Object-Oriented Applications". https://digitalcommons.macalester.edu/cgi/viewcontent.cgi?article=1006&context=mathcs_honors

[4] Renilde Becqué, "Accelerating Building Decarbonization: Eight attainable policy pathways to net zero for all". https://wrirosscities.org/sites/default/files/19_WP_ZCB_final.pdf

[5] Jeang-Kuo Chen and Wei-Zhe Lee, "An Introduction of NoSQL Databases Based on Their Categories and Application Industries". https://mdpi-res.com/d_attachment/algorithms/algorithms-12-00106/article_deploy/algorithms-12-00106-v2.pdf?version=1559011753

[6] Pierre-Évariste Dagand, "A Cosmology of Datatypes: Reusability and Dependent Types". Dissertation, University of Strathclyde, 2013 https://citeseerx.ist.psu.edu/document?type=pdf&doi=1997710cea4b67b9e53df96d6dee7f2c3e5c5556

[7] Martin Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002 https://www.martinfowler.com/books/eaa.html

[8] Jan Hidders, "Typing Graph-Manipulation Operations" http://adrem.uantwerpen.be/bibrem/pubs/hidders.3.pdf

[9] Bart Jacobs, *Categorical Logic and Type Theory*. Elsevier, 2001.

[10] Christopher Ireland and David Bowers, "Exposing the Myth: Object-Relational Impedance Mismatch is a Wicked Problem"

[11] Samiya Khan, et. al., "Storage Solutions for Big Data Systems: A Qualitative Study and Comparison". https://arxiv.org/pdf/1904.11498.pdf

[12] Julian Keil, et. al., “Preparing the HoloLens for user Studies: an Augmented Reality Interface for the Spatial Adjustment of Holographic Objects in 3D Indoor Environments”. https://d-nb.info/1208303023/34

[13] M. B. Luther, et. al., “A simple-to-use calculator for determining the total solar heat gain of a glazing system”. https://anzasca.net/wp-content/uploads/2014/08/p511.pdf

[14] Divya Mahajan, et. al., “Improving the Energy Efficiency of Relational and NoSQL Databases via Query Optimizations”. https://www.sciencedirect.com/science/article/pii/S2210537918301112

[15] Mike Marti, “Translating Between Graph Database Query Languages”. Thesis, ETH Zürich, 2022. https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/575738/1/Marti_Mike.pdf

[16] Wim Martens, et. al., “Representing Paths in Graph Database Paern Matching”. https://arxiv.org/pdf/2207.13541.pdf

[17] Bálint Molnár and András Benczúr, “The Application of Directed Hypergraphs for Analysis of Models of Information Systems”. https://www.mdpi.com/2227-7390/10/5/759

[18] Martin Ledvinka and Petr Křemen, “Formalizing Object-ontological Mapping Using F-logic”. https://kbss.felk.cvut.cz/reports/2019/19ruleml-report.pdf

[19] Chris Lattner and Vikram Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. https://llvm.org/pubs/2004-01-30-CGO-LLVM.pdf

[20] Tommi Laukkanen, et. al., “Virtual technologies in supporting sustainable consumption: From a single-sensory stimulus to a multi-sensory experience”. https://e-tarjome.com/storage/panel/fileuploads/2022-06-27/1656317001_e16754.pdf

[21] Martin Ledvinka and Petr Křemen, “A Comparison of Object-Triple Mapping Frameworks”. https://www.semantic-web-journal.net/system/files/swj1910.pdf

[22] Daniel Martin, et. al., “Multimodality in VR: A survey”. https://arxiv.org/pdf/2101.07906.pdf

[23] Anders Olofsson, “Behavior-driven Tile Caching in Web GIS Applications”. Chalmers, Thesis. https://mdpi-res.com/d_attachment/ijgi/ijgi-08-00373/article_deploy/ijgi-08-00373.pdf

[24] Radoslav Radev, “Extending Object-Relational Mapping with Change Data Capture”. http://www.cs.ubbcluj.ro/~studia-i/contents/2015-1/04-Radev.pdf

[25] Ismo Rakkolainen, et. al., “State of the Art in Extended Reality – Multimodal Interaction”. https://humanoptimizedxr.org/wp/wp-content/uploads/2021/06/HUMOR-State-of-the-Art-in-Extended-Reality-Multimodal-Interaction-1.pdf

[26] Ismo Rakkolainen, et. al., “Technologies for Multimodal Interaction in Extended Reality – A Scoping Review”. https://www.mdpi.com/2414-4088/5/12/81

[27] Danielle Richardson, “Sustainable Urban Planning: Turning the Concrete Jungle into Green Buildings”.

[28] Filippo Sanfilippo, et. al., “A Perspective Review on Integrating VR/AR with Haptics into STEM Education for Multi-Sensory Learning”. https://www.mdpi.com/2218-6581/11/2/41

[29] Patrick Schultz, et. al., “Algebraic Databases”. https://arxiv.org/pdf/1602.03501.pdf

[30] Joshua Shinavier, et. al., “Algebraic Property Graphs”. https://arxiv.org/pdf/1909.04881.pdf

[31] Srinath Srinivasa, et. al., “LogicFence: A Framework for Enforcing Global Integrity Constraints at Runtime”. https://ieeexplore.ieee.org/document/4041608

[32] Valter Uotila, et. al., “MultiCategory: Multi-model Query Processing Meets Category Theory and Functional Programming”. https://arxiv.org/pdf/2109.00929.pdf

[33] D.W.F. van Krevelen and R. Poelman, “A Survey of Augmented Reality Technologies, Applications and Limitations”. https://ijvr.eu/article/view/2767/8825

[34] Jose D. Velazco-Garcia, et. al., “A Modular and Scalable Computational Framework for Interactive Immersion into Imaging Data with a Holographic Augmented Reality Interface”. https://pubmed.ncbi.nlm.nih.gov/33045556/

[35] Natalie C. Wheating, et. al., “Embodied carbon: A framework for prioritizing and reducing emissions in the building industry”.

[36] Xiaodong Zhou, et. al., “A PDF Tile Model for Geographic Map Data”. https://www.mdpi.com/2220-9964/8/9/373