Information Substrates: Interacting with Digital Matter

Michel Beaudouin-Lafon
Université Paris-Saclay, CNRS, Inria
F-91400 Orsay, France
mbl@lri.fr

ABSTRACT
This paper introduces the concepts of *information substrate* for organizing digital information and *interaction instruments* for manipulating these substrates. We present STRATIFY, a proof-of-concept implementation that combines a data-reactive approach to specify relationships among digital objects with a functional-reactive approach to handle agency, including user interaction. This unique combination makes it possible to create rich information substrates that can be freely inspected and modified, as well as interaction instruments that are decoupled from the objects they interact with, making it possible to use instruments with objects they were not designed for. We illustrate the flexibility of the approach with a number of examples and present directions for future work.

ACM Classification Keywords
H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

Author Keywords
Conceptual model, Information substrates, Instrumental interaction

INTRODUCTION

Today's computer interfaces are based on principles and conceptual models created in the late seventies. The Xerox Star [16], which pioneered today's graphical user interfaces (GUIs), was designed for executive secretaries to carry out office tasks. Thirty-five years later, computers come in many forms and are used for a wide range of tasks by a wide variety of users, yet their interfaces are still based on applications, documents, files and folders, with the same menus, buttons and dialog boxes and the same input devices as the early systems. Other forms of interaction, such as virtual reality, augmented reality and voice-based interfaces, have not replaced GUIs, except in some niche markets. Multitouch interfaces, designed for smartphones and tablets, are based on the same fundamental assumptions and basic interaction principles as GUIs. Similarly, web-based applications, while enabling a new category of applications, tend to resemble desktop applications.

Desktop, web and mobile apps are all trapped in the same application-centric model: each is designed for a single user, on a single device, with pre-defined tools to accomplish pre-defined tasks, such as editing a single document or posting a comment to a single social network. We identify three major issues with this model:

Walled gardens – Applications bundle together a set of tools for manipulating a particular type of content, stored in a proprietary format that can often be read only by the application that created it. Changing applications requires converting—and sometimes re-creating—documents. This lack of interoperability limits users to the application's tools, creating a *walled garden* around each application.

Information silos – Documents, and digital information in general, is more and more often trapped in the cloud, which arbitrarily insulates related pieces of content from each other and from users. Users no longer own their data and can no longer use the familiar file system to organize information.

Single-user, single device – Few applications support multiple, simultaneous users. Social networking apps and multiuser games offer limited forms of sharing, and dedicated web-based editors, such as Google Docs, are not interoperable with the single-user editors that users are familiar with. Although we commonly access on-line information via a variety of devices, these remain largely ignorant of each other. Distributed interfaces, such as when a smartphone controls a slide presentation on a laptop, are limited to predefined situations and often require a complex setup.

In contrast, the real world has no "applications"[1]: We use objects and tools carefully chosen for each specific task and we fluidly design and redesign our workspaces to adapt to the activity at hand. We engage in multiple projects and activities that require keeping together and assembling diverse, often inter-related digital artifacts.

The time has come to break out of the walled gardens and information silos created by the app-centric approach and to revisit the fundamental principles that underlie interactive software and digital information. The ability to share documents in real time, to distribute content and tools across devices, and to flexibly manage one's digital environment must be deeply embedded within the infrastructure, as an integral part of the fabric of the digital world. Addressing these limitations requires a paradigm shift, with a focus on two properties:

[1] Interestingly, the Xerox Star had no "applications" either.

Flexibility – Users must have choices: They must be able to use different tools to achieve goals in different ways, to work alone or collaboratively, to select one device over another, to configure their environments to support their favorite features, and to create several such configurations.

Extensibility – Systems must be open and interoperable: Users must be able to add features when they need them, rather than being subjected to the release cycles imposed by vendors. Third parties must be able to provide new features as well as alternative interactions for existing features. Users must be able to create their own extensions and reconfigurations and share them with others.

This paper introduces the concept of *information substrate* to organize digital information, and builds on *instrumental interaction* [1] to interact with these substrates. We then present STRATIFY, a proof-of-concept implementation of this conceptual model. STRATIFY combines a data-reactive approach to specify relationships among digital objects with a functional-reactive approach to handle agency, including user interaction. This combination makes it possible to create rich information substrates that can be freely inspected and modified, as well as interaction instruments that are decoupled from the objects they interact with, making it possible to use instruments with objects they were not designed for. We illustrate the flexibility of the approach with a number of examples, review related work and conclude with directions for future work.

MOTIVATION

Our goal is to create interactive systems based on a deep understanding of human skills, not on concepts and techniques that are convenient for computers. We want to create *digital matter* that, like physical matter, is comprehensible and appropriable by normal people, not just computer experts.

From affordances to information substrates

According to J. Gibson [10], affordances are properties of an object that enable specific actions by animals and humans. When perceived, they provide a natural way of understanding the capabilities of the environment. E. Gibson [9] studied the process of perceptual learning, in which children and adults learn to recognize affordances throughout life.

In order to build on these concepts, a digital object's capabilities should be perceivable—directly or through appropriate instruments—and those perceived qualities should correspond to the object's actual capabilities. For example, any piece of text should be selectable so it can be copy-pasted. In the physical world, we observe objects at various levels of detail to infer their capabilities. We recognize their material, shape, structure, and relationship to other objects, which is crucial when appropriating them for novel uses. For example, we know that a chair is heavy enough to prop open a door. If we are to take full advantage of affordances in the digital world, we need to support such multilevel representations.

The concept of *information substrate* is meant to operationalize affordances in the digital world. A substrate is a unified construct that holds digital information together with its constraints and relationships. A substrate can use information

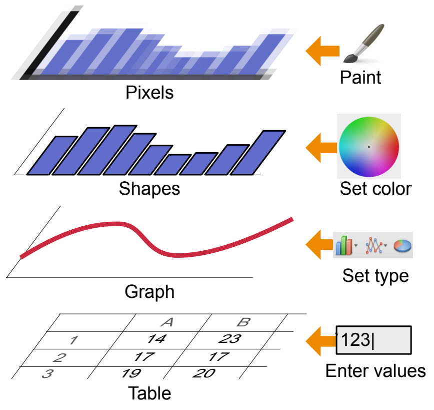

Figure 1. A stack of substrates and associated instruments

from, and provide information to other substrates. A collection of substrates creates a rich form of digital matter.

Figure 1 shows a data table substrate holding information that may come, e.g., from a sensor, with a new entry being added for each reading. The table can compute, e.g., the running average of the measures, and provide this information to a graphing substrate to plot it. The graph is a substrate that represents its input data as a set of graphical shapes. These shapes constitute yet another substrate, which a canvas substrate represents as a set of pixels displayed on the screen. The graph that the user sees on the screen is therefore a composition of four substrates: the data table, the graph, the graphical shapes and the pixels. Each level has its own properties, capabilities, and affordances. Different instruments can operate at each level (Figure 1, right) to add data to the table, change the type of graph, set the color of the bars, or edit the pixels to draw annotations. By letting the user interact at these different levels, the conceptual model provides an unprecedented level of flexibility. For example the color tool is also expected to work with a text substrate or a brush.

From human tool use to instruments

Humans are the only species that not only uses tools, but also creates new ones. When a person holds a tool, it becomes an extension of her body schema, as if it were a part of the body [19]. Holding a tool thus redefines the affordances of the environment, since it changes the person's capabilities. This provides a psychologically and cognitively sound basis for creating extensible systems, where new digital tools create new affordances. Osiurak [30, 31] suggests that tool use involves *technical reasoning*, a specific type of reasoning that complements the direct perception of what the tool can do. Technical reasoning is based on abstract technical laws, which, unlike skills acquired through procedural learning, do not require constant reinforcement. For example, people do not forget how to ride a bike, even after many years without practice, but commonly forget simple procedures, such as changing the time on a digital clock. Technical reasoning also supports appropriation based on the properties of physical tools, such as when we use a knife blade as a screwdriver.

Most interactive software systems are designed to require procedural knowledge instead of technical reasoning, which renders software tools difficult to learn, remember and master. Software tools are trapped within applications and inconsistent across applications, making them arbitrarily different and difficult to use in unexpected ways.

Our goal is to create digital tools, called *instruments*, that extend the body schema and rely on technical reasoning in the same way as physical tools. Instruments should be applicable to any substrate that presents appropriate data. Such an environment supports flexible and consistent tool use, where tools can be reused across domains and where users can reason about the effects of applying a tool in different situations.

THE SUBSTRATES CONCEPTUAL MODEL

In the physical world, objects are made of materials that have properties: mechanical, chemical, optical, etc. Moreover, human-made physical objects are also designed for a purpose: sheets of paper, for example are designed to be written on by pencils, pens and brushes. Physical objects are therefore polymorphic: they can be used for what they are made *of*, e.g., a sheet of paper can be used to start a fire because paper is highly flammable, or to level a table because it can be folded easily; Objects can also be used for what they were made *for*, e.g., a sheet of paper can be used to take notes or to make a drawing. Humans learn these affordances through a number of means, from perceptual learning [9] to cultural transmission.

Such polymorphism rarely exists with digital objects, which are usually reduced to their functional role. For example, an image editor manipulates pixels. Even if the pixels happen to represent characters, they cannot be manipulated as text. The windows in a traditional window manager are rectangles, but cannot be manipulated, e.g., rotated, distorted or aligned, as the shapes in a drawing editor. To make digital objects polymorphic, we introduce information substrates.

Information Substrates

The goal of an information substrate is to explicitly represent different levels of representation of a digital object and maintain consistency among them, so that the object can be used for what it is made *for* as well as for what it is made *of*.

The term substrate is used in many areas including geology, biology, chemistry, materials science and printing, each with different nuances. The general definition is *"a substance or layer that underlies something, or on which some process occurs"*. For example, biological substrates provide the surface on which an organism lives. In material science, a substrate is the material on which a process is conducted, and in printing, it serves as the base material that images are printed on. These definitions resonate with the need to provide context and constraints to interpret digital information and give it different capabilities.

Pixel substrates – This paper focuses on visual interfaces where digital information is depicted primarily on bitmapped displays. Since we receive visual stimuli through an array of pixels, the highest level of representation of digital information is a set of pixels[2]. We call these *pixel substrates*. Conceptually, they are 2D arrays of pixels, each representing a color. The same approach can be applied to other ways of perceiving digital objects, including tactile, kinesthetic and auditory.

Geometrical substrates – Of course, we rarely perceive pixels as such, but instead perceive shapes, images, text and other structured objects. Conversely, the computer typically only manipulates pixels at the last stage of its rendering pipeline, when "objects" such as text, shapes and images are turned into pixels. The next level of representation of a visual interface is therefore made of structured graphics, where geometrical objects are laid out according to layout constraints. We call these *geometrical substrates* to emphasize the geometrical nature of both the objects that populate it and of the layout constraints, such as alignment and containment. Scene graphs [34], HTML[3] and SVG[4] are examples of such substrates.

Structural substrates – While a geometrical substrate can be simply a representation of itself, for example a static web page or a hand drawing, the shapes it contains are often representations of "deeper" objects. In GUIs, these include the widgets such as buttons and menus, and the objects of the "model" of the application. In the Model-View-Controller pattern [22], model objects are often opaque and ad hoc. By contrast, we want to expose their structure and inner properties so that they can be manipulated in different ways. Since at this level of analysis, many objects are *containers* of other objects, we call them *structural substrates*.

A very common container is the sequence: a photo editor manages a stack of layers, a drawing editor a list of shapes, a window manager a set of windows, a menu a list of items. Yet each of these applications represents the sequence differently. By exposing their common structure, we open the door to alternative representations and manipulations. For example, a light table can be used to lay out the content of any sequence, letting users organize it as they see fit. The layers of the image editor can then be viewable separately instead of superposed, the windows of the window manager can be spread out as in Apple Mac OS X Exposé instead of overlapping each other. Alternatively, a sequence can be represented as a linear list of names or icons: we get a list of layers, a list of window titles, and the familiar linear menu.

Other common structures include trees, graphs and tables, which can be mapped to well-known representations such as indented lists for trees, node-link diagrams or matrices for graphs, grids for tables, or more specialized ones when needed. Structural substrates can be nested and can be heterogeneous: a table may contain a tree in one cell, and an image in another.

Structural substrates can also be mapped onto one another in order to provide alternative representations. This is a different from containment. For example, a sequence can represent the leaves of a tree, or the sorted version of another sequence. Combining such mappings makes it easier to reuse higher-level representations, such as using the light table described above to display the content of a tree, or displaying a list of window titles in alphabetic order.

[2]We place levels of representations "closer" to human perception, e.g., pixels, higher than those closer to machine representations

[3]https://www.w3.org/html/

[4]https://www.w3.org/Graphics/SVG/

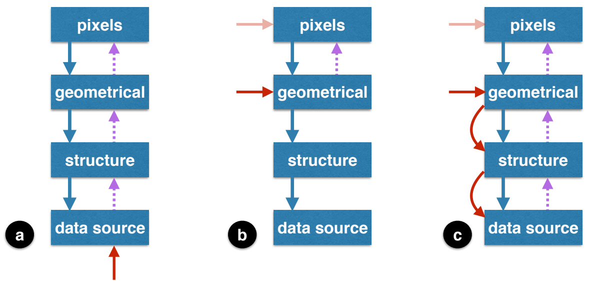

Figure 2. Update process when (a) a data substrate changes, (b) handing off an action at a lower level and (c) relaying an action. Substrates are in blue, with blue arrows depicting their source. Actions are in red, with the original action that caused the handoff in pink. Dotted purple arrows represent the update of dependencies.

Data – Structural substrates must eventually contain actual data. Data often comes from external sources, such as the content of a file or database, a sensor or a network service, and can therefore be dynamic. Since they often depend on a source, like other substrates, we call them *data substrates*. As new data is provided by the source of the data substrate, the higher-level substrates that depend on it update, all the way to the pixel substrate (Fig. 2a).

The conceptual model emerging from this analysis is that digital matter is made of a combination of substrates, linked together to maintain their consistency. Higher-level substrates are *representations* of lower-level ones, creating a fully inspectable stack of representations. The *sources* of a substrate are the substrates it depends on, i.e. those that it represents. New representations can be created dynamically, at any level, making it possible to address an object at different levels of representation. The ability to combine existing substrates in different ways supports *flexibility*: Users are not bound to the choices made by the designers and programmers. The fact that substrates can be inspected and interoperate supports *extensibility*: New representations, new data sources and new substrates can be injected into existing systems.

Agency

Information substrates let us represent digital information in a rich, polymorphic manner. They react to changes in their data sources to maintain consistency, but the network of dependencies is static. To modify substrates and their relationships, we need to provide a model for how user actions are interpreted and carried out by the system. We call (system) *agency* the way substrates *express* and *enact* actions.

Users only perceive the surface-level representations of the substrate stack, i.e. the pixel substrate in a visual interface. User actions expressed at this level, such as pointing and dragging, must be interpreted and handed off to lower levels so they can be addressed at the appropriate level of representation. For example, moving the icon of a file within a window does not affect the file system, and can be interpreted by the geometrical substrate. But moving a file icon from one window to another requires changing the location of the file in the file system, and must be handed off all the way down to the data substrate, which will interpret it as a file system operation. Of course, users should not be aware of these details.

Interaction protocols – In order to decouple the two aspects of agency, expressing and enacting actions, we introduce *interaction protocols*. The substrate emitting an action is its *origin*, the one enacting it is the *target*. An interaction protocol describes if and how a particular action, e.g., dragging, is turned into one or more operations on the target, e.g., moving it. Note that actions can be specified at any level of abstraction, not just at the user level as in this example. For example, a substrate can issue a sorting action to change the sorting order of a structural substrate.

When the target of an action has no matching protocol, the action is handed off to its source, down the stack of representations, unless the target blocks the action. This is how changing the name of a file is handed off all the way down to the data substrate representing the file, whereas moving a file icon within a window is handed off only to the geometrical substrate. When a target substrate enacts an action and changes its state, the higher-level substrates update to maintain consistency with the new state (Fig. 2b). This process is similar to the update that occurs when a data source changes, except that it does not start at the bottom of the stack of substrates.

Blocking and relaying actions – However, enacting an action at a certain level of representation does not necessarily –and cannot always– change its substrate directly. Indeed, many properties of a substrate depend on their source and should not be changed directly by the programmer. For example, consider a shape in a geometrical substrate whose color is linked to the "temperature" property of its source data substrate. A coloring action on the shape should be blocked not because it is not possible to change the color of the shape, but because it is bound by a dependency on the underlying data substrate.

Now consider the shape representing the minutes hand of an analog clock, whose angle depends on the "minutes" property of the time source displayed by the clock. To change the time by turning the clock's hand, the hand must provide an interaction protocol for the drag action. Instead of interpreting this drag operation as a change of the angle of the shape, it interprets it as a change of the number of minutes of its source, and issues the corresponding action to the source. We call this *relaying* an action (Fig. 2c). Assuming the time source accepts the action, it will update its state, which will update the analog clock –as well as any other representation it may have. If it blocks the action, dragging the hand will have no effect.

Forced actions – We can also let users experiment and take risks by *forcing* some dependencies, in the same way as we sometimes misuse a physical object on purpose. For safety, this should be enabled only in a user-controlled mode. In this mode, the user can bypass the blocking of actions and force the assignment of a property, even though it is dependent on a source substrate. For example, the user could force the change of color of the shape representing the temperature, breaking the representational link with the source temperature. In the clock example, the user could move the hand anywhere on the screen, breaking the layout of the clock. Since it is the angle of the hand that is linked to the time source, the moved hand would nevertheless continue to turn and reflect the time.

In summary, the ability to interact with different levels of interpretation of an object relies on two concepts:

1. Interaction protocols to find the level of interpretation at which the intended action makes sense; and
2. Interpretation of concrete actions, such as moving the clock hand, into lower-level ones, such as updating the time, to traverse the abstraction layers between successive substrates.

These two mechanisms ensure that all levels of representation of a digital object are potentially exposed and available for interaction, providing both flexibility and extensibility. Flexibility comes from the ability to reuse existing interactions and protocols, while extensibility comes from the ability to create new ones that can interoperate with existing ones. In addition, the reification of actions into first class objects makes it possible to reuse them and to support undo.

In order to take full advantage of these properties, designers should strive to create substrates that are as open as possible to incoming actions by defining appropriate protocols, e.g., making any text editable. Designers should also decompose complex substrates into simpler, more generic ones, to facilitate reuse and encourage recombinations. For example, creating a separate substrate to sort a sequence makes it possible to, e.g., insert a filter.

Instruments

After describing *system* agency, we turn to the *user* side of agency. In the physical world, we act on objects directly, e.g., folding a sheet of paper, or through tools, e.g., writing on a piece of paper. Using an object as a tool is the result of a dual process [4]: *Instrumentation* is the process by which the user adapts to the tool and the resulting extension of the body schema: the tool becomes an extension of the user's body and augments its capabilities; *Instrumentalization* is the process by which an artifact is turned into a tool, possibly through adaptation, once the user has determined that the artifact has the required properties for the task at hand. Mackay [23] describes the more general *co-adaptation* phenomenon, by which users both adapt and adapt to technology.

While co-adaption is common in the physical world, digital tools can often be used only for what they were designed for. Limited forms of instrumentalization exist, e.g., when creating a rectangle in a drawing tool to measure and duplicate the spacing between two objects. By exposing the various levels of representation of digital information and supporting a flexible model of system agency with the notion of *instrument*, we can support more general forms of co-adaptation.

An instrument is a substrate capable of agency, directly or indirectly controlled by user input. We start by modeling input devices as data substrates. For example, a mouse provides data about its movements and the state of its buttons, and controls the position of a cursor, whereas a touch screen provides information about touch points. Input devices emit actions that represent basic motor actions such as click, drag or pinch. The targets of these actions are one or more instruments, i.e. substrates that feature a protocol for these actions. Instruments typically transform these actions into higher-level actions, e.g., a sequence of button down, mouse moves and button up into a move action. They also determine which target(s) should receive these actions. For example, many mouse or touch actions apply to the object at the position of the cursor or touch point. The instrument must therefore query the various information substrates to identify that object, which will typically become the target of the output action of the instrument.

This model supports multiple simultaneous users, since multiple input devices can be represented, each triggering its own actions. Of course, substrates must be prepared to enact simultaneous user actions, or combine them together, or block them. This model also supports distributed interaction: the device and some substrates can reside on one machine, and the actions can be sent across the network to a remote target. For example, a mobile device may provide a color palette and a touch area to control a cursor on a desktop display. Moving the cursor and tapping a color sends the action to the desktop computer, which decides if the action can be applied. Another user can interact in parallel with another mobile device.

Designers should strive to create instruments usable with different types of substrates, e.g., to edit basic data types such as text, numbers, colors and fonts, or to move and transform shapes. The same instrument should let a user rotate a graphical shape in a drawing editor, a window in a window manager or a bar in a histogram. Generic instruments also include those for manipulating structured substrates, such as sorting and filtering sequences and tables, or creating new substrates that map these structures to other structures and the content of these structures to visual elements. For example, a histogram instrument creates a sequence substrate whose content is a set of geometric shapes bound to another sequence containing the values to be displayed. Other examples include instruments for annotating substrates, adding metadata and searching.

Instruments provide the user side of flexibility and extensibility: well-designed instruments are polymorphic [2] and can be used with different types of substrates, even if they were not designed for them; Users can replace instruments, e.g., to use their preferred color picker, and add new instruments, e.g., an advanced tool to align and distribute objects [5].

PROOF OF CONCEPT: STRATIFY

STRATIFY is a proof-of-concept prototype that we have developed to demonstrate and experiment with the concepts of information substrates and interaction instruments. It is a web-based framework that runs in any modern web browser.

STRATIFY leverages the web environment and the Javascript language in several ways. It builds heavily on features of Javascript, including its prototype-based object-oriented model as well as meta-programming capabilities such as redefining a property by a pair of functions for getting and setting its value. It uses the DOM[5], the in-memory representation of the content of a page, as geometrical substrates for both HTML content, for document layout, and SVG content, for vector graphics. As a result, the pixel substrate is managed by the web browser and is not accessible. We leave it to future work to create a pixel substrate based on, e.g., the HTML5 Canvas or WebGL.

[5]Or rather the Inferno virtual DOM: https://infernojs.org/

```
class AtomicClock extends DataSource {
  constructor () {
    super();
    this.ms = Date.now();
    // update the millisecond counter every second
    setTimeout(() => this.ms += 1000, 1000);
  }
}

class Time extends Substrate {
  // 'get' declares a computed property
  get date() {return new Date(this.source.ms)}
  get hours() {return this.date.getHours()}
  get minutes() {return this.date.getMinutes()}
  get seconds() {return this.date.getSeconds()}
}

class AnalogClock extends SVGSubstrate {
  get hoursAngle() {return (this.source.hours + this.source.minutes/60) * 30)}
  get minutesAngle() {return this.source.minutes*6)}
  get secondsAngle() {return this.source.seconds*6)}
  get SVG() {
    // return 3 SVG lines rotated by the corresponding angles
  }
}

// create an AtomicClock, make it the source of a Time object,
// and make that object the source of an AnalogClock
new AnalogClock(new Time(new AtomicClock()));
```

Listing 1. Programming a clock with Stratify

```
1 // in class AnalogClock
2   set minutesAngle(a) {
3     this.source.minutes = Math.round(a/6);
4   }
5 // in class Time
6   set minutes(v) {
7     this.source.ms += (v − this.minutes)*60000;
8   }
```

Listing 2. Declaring setters for computed properties

Substrates: Data-Reactive Programming

An information substrate in Stratify is a collection of objects and properties linked by relationships. For example, an analog clock is made of graphical shapes, the clock face and the hands, linked to a time substrate, itself linked to a timer data source. In order to manage these relationships in a declarative way, we use a programming model that we call Data-Reactive Programming (DRP), implemented with the mobx library[6].

The central tenet of DRP is that objects expose their state as a set of properties. Some of these properties are computed from other properties of the same or other objects. These *computed properties* can be read but not written by default. Non-computed properties are *internal properties*: they belong to the object, and can be written directly. mobx ensures that whenever the value of an internal property changes, the computed properties that depend on it are recalculated[7].

In case of dependency cycles, successive iterations are computed until the values stabilize or a maximum number of iterations is reached. In practice, assuming that computed properties only depend on source substrates, cycles rarely obtain because sources normally form a tree or a DAG.

In the previous example, the hands of the analog clock are shapes whose rotation angle is a computed property, calculated from the time stored in the time substrate. The number of hours, minutes and seconds of that substrate are themselves properties computed from the timer data source. The counter held in the timer is an internal property whose value changes every second. As a result, the analog clock constantly reflects the current time, without any further programming (Listing 1). If a textual rendering of the time substrate is managed by another substrate, it too will be updated in real time.

The network of dependencies created by the computed properties resembles the formulas of a spreadsheet: changing the value of a cell triggers a recalculation of all the formulas that depend on it, updating their values. In Stratify, the cells are not organized in a table but are arbitrary object properties.

While this computational model is not novel, we add a twist that distinguishes it from other data-driven models. The programmer can define a *setter* for a computed value, i.e. what to do when setting the value of a computed property. In the clock example, to set the time by turning one of the hands, one must define a setter for the minutesAngle property that calculates the number of minutes based on the new angle and set the minutes property of the source time accordingly (Listing 2, lines 2–4). Since the time source depends on an AtomicClock data source, we also need a setter for its minutes property, otherwise nothing will happen (Listing 2, lines 6–8). That setter will trigger an update of computed properties that depend on it, i.e. the time and analog clock substrates.

Data-Reactive Programming provides a declarative model for dependencies among objects and keeps them up to date at all times, in a transparent and optimized way (thanks to the mobx library). It supports updates of computed values, by requiring the programmer to define *setters* that, in effect, compute an inverse of the computed property in order to update the source objects, which then propagate to all objects involved.

Note that this approach breaks traditional object-oriented programming: usually, objects do not expose state directly, but use methods that control access to their state. Stratify intentionally exposes state and expects it to be changed directly from outside. Objects can react to state changes either by *intercepting* changes to an internal property in order to modify or discard the change or trigger a side effect, or by *observing* a computed or internal property to generate side effects.

Listing 3 is a collection of structural substrates that represent sequences of objects. The base Sequence substrate is an array. The Sorter, Filter and Mapper recompute their sequence on the fly by sorting, filtering or mapping their source sequence. They are used in Listing 4 to create a to-do list whose items are sorted, filtered to eliminated the checked ones, and capitalized.

However, recomputing the entire sequence each time the source changes in the Mapper may cause a problem because

[6]https://mobx.js.org

[7]Only computed properties that are used by other properties are recalculated, greatly optimizing updates.

```
class Sequence {
  constructor(a) {
    this.seq = [];
    if (a) this.seq.replace(a);
  }
}

class Sorter extends Substrate {
  get seq() {
    return this.source.seq.sort(this.compare)
  }
  constructor(source, compare) {
    super(source);
    this.compare = compare || defaultCompare;
  }
}

class Filter // similar to Sorter using Array.filter
class Mapper // similar to Sorter using Array.map

class Mirror extends Substrate {
  seqChange(change) {
    // apply the change to this.seq, calling mirror for new items
  }
  constructor(source, mirror) {
    super(source);
    this.mirror = mirror; // mapping function
    // create the mapped sequence
    this.seq = source.seq.map(item => this.mirror(item));
    // reflect changes to the source locally
    intercept(source.seq, this.seqChange);
  }
}
```

Listing 3. Structural substrates for sequences

the resulting objects are re-created each time. In the to-do list example, a list of text strings is turned into a list of Todo items by creating a substrate that holds the internal property checked for each item. If the list of to-dos were recreated each time the list of text strings changed, the checked state of the items already in the list would be lost. By using a Mirror substrate instead of a Mapper, only the items that are added or changed are recomputed. Taking advantage of the observability of state changes, the Mirror can intercept the changes to its source sequence and mirror them in its internal sequence.

Data-Reactive Programming is well-suited to substrates because it forces the designer to identify and separate the multiple levels of representation of an object, leading to collections of highly reusable substrates. Each substrate typically depends on a single source, and can be easily combined with other substrates to create complex objects that are guaranteed to stay consistent and that can be freely inspected and modified.

Agency: Functional-Reactive Programming

In order to implement agency and instruments, Stratify uses another reactive paradigm called Functional Reactive Programming (FRP). FRP describes computations over *streams* of values called *signals*, which can be processed and combined using a rich set of operators. For example, a mouse is a signal that generates a new value every time it is moved or its buttons are pushed. This signal can be filtered to send moves only between a button down and a button up, and can be combined with a keyboard signal to also send changes in modifier keys.

```
class Todo extends Substrate {
  constructor(s) {
    super(s);
    this.checked = false;
  }
}

// Create a substrate that sorts, filters and maps a todo list
let list = new Sequence(['sleep', 'eat', 'work']);
let todos = new Mirror(list, a => new Todo(a));
let sorter = new Sorter(todos);
let filter = new Filter(sorter, a => !a.checked);
let upper = new Mapper(filter, a => capitalize(a.source));

// Changes to the list propagate immediately:
list.seq.push('relax');
todos.seq[2].checked = true;
```

Listing 4. A todo list based on sequences

Note that we cannot use DRP for programming agency as it has no notion of time (updates are –conceptually– instantaneous). Actions occur over time and therefore we need a model, such as FRP, where time is explicit. Stratify uses the RxJS[8] library.

Any substrate can be the origin and/or target of one or more signals. Each signal is typed: it carries values called *actions* with a type tag. In order for a target substrate to listen to a signal, a *protocol* must be found that matches the signal type to the target. Stratify implements several strategies to match protocols. The simplest is to look up the protocol by name in the target object and its prototypes. If the search fails, it continues to the source (if any) of the target substrate. If finding a protocol by name fails, a second search takes place using the names and/or types of the target object properties. For example, if a signal carries actions to set colors, the search starts with the protocol name setColor. If this fails, a new search starts for a property name that contains the word color and that holds a color object. If this fails and the user is willing to take risks, a last search is started for any property whose value resembles a color.

This process operationalizes a critical aspect of interaction with substrates: decoupling instruments from target substrates, so that one can use instruments on targets even though they were not explicitly designed to work with each other.

In Stratify, an instrument is a substrate that is the target of an input device such as the mouse (Fig. 3). The instrument processes the actions sent by the input device, and emits higher-level actions on an output signal. The target of that signal is the so-called *object of interest* of the interaction, i.e. the substrate for which a protocol matching the signal was found. The protocol instantiates an object that mediates the actions of the instrument on the target object, in effect reifying the interaction: When the interaction is over, the signal terminates and the interaction object disappears. This design provides tremendous flexibility by allowing new protocols matching instruments to target substrates without modifying either of them.

[8]http://reactivex.io

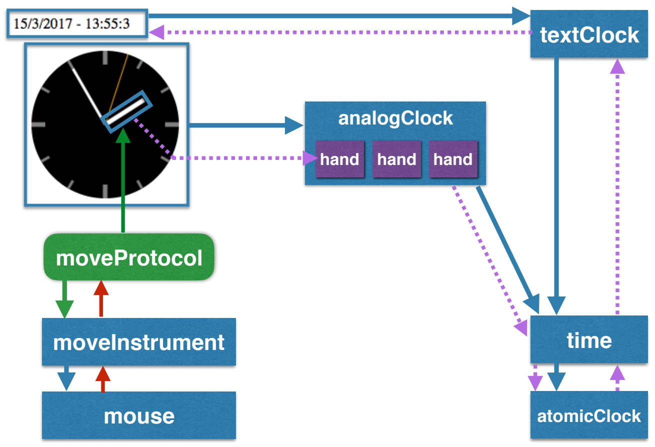

Figure 3. Substrates and instruments in Stratify. Substrates are in blue with their source indicated by a blue arrow. Protocols are in green, and signals in red. The dotted arrows show the update of dependencies when the move instrument rotates the clock hand.

Example: the Move instrument

Let us consider a generic, yet simple, instrument: the Move instrument. It is operated by an input device substrate such as the Mouse, and should work with any substrate that can be moved or whose content can be moved, such as the graphical shapes of a drawing canvas, the windows of a window manager, the items of a to-do list or the tools in a palette.

The Move instrument (see Appendix) emits start, move and stop actions in response to the signal coming from the mouse. Each action carries an incremental offset as well as an offset from the start of the move. Using the initial position of the mouse, the Move instrument identifies the substrate at this position and probes it for a protocol called move. The search for this protocol proceeds down the chain of sources of the target substrates. If not found it goes up the enclosing substrates in the geometrical substrate: if the original target was a text label enclosed in a rectangular frame, the search proceeds to the frame, and then its sources. If no protocol is found, the search starts over along the same path, looking for substrates with a position property whose value is a point (an object with an x and a y property that are numbers).

Moving list items – When the user wants to move a to-do item in a list, the first target is the HTML list item, and its source is the Todo item. Neither knows the move protocol. The parent of the list item is the HTML list, which *does* implement the move protocol. The protocol therefore instantiates the object that will represent the interaction. This interaction receives the move actions through the Move instrument output signal, and translates the vertical mouse movements into a change of the relative position of the HTML list item. As a result, the visual item moves vertically in response to the mouse movements.

At the end of the interaction, the HTML list needs to move the to-do item within the source sequence. Since it has access to the sequence, it could remove the item and insert it at the proper position. However, it first probes the source for the moveItem protocol, in case the source substrate implements the move properly. The Sequence substrate implements this protocol. The HTML list invokes it, resulting in an update to the sequence of to-do items, immediately reflected in the HTML representation, and in any other representation it may have. Note that the above interaction works with any sequence represented by an HTML list. We can use it to, for example, reorder items in a menu or tools in a palette.

Moving shapes – The situation is simpler when applying the move instrument to a shape in a drawing surface or to a window in a window manager, because the move does not reorder objects in a list. In this case, the protocol that is likely to match is the search for a position property: the protocol simply updates the position as it receives the move actions.

We implemented magnetic guidelines similar to Sticky-Lines [5]: graphical objects can be attached to a StickyLine; moving a StickyLine also moves the attached objects. A StickyLine is a substrate that holds the sequence of shapes attached to it. When the position of the StickyLine is changed, an observer propagates the horizontal or vertical component of the move (according to the type of StickyLine) to the positions of the objects in the sequence, using the same move protocol as the Move instrument. This illustrates a form of instrumentalization [4]: The StickyLine becomes an instrument.

Objects are attached to and detached from a StickyLine by moving them close to or away from it. Since the StickyLines are not part of the objects that are probed when the Move instrument is looking for its target, we create a move protocol that moves objects normally, but also checks if they are moving close to or away from a StickyLine. When such transitions occur, it sends attach/detach actions to the StickyLine. This example illustrates how new behaviors can be added to existing substrates without changing them. StickyLines are not limited to aligning objects in a drawing surface. They work with any substrate that matches the move protocol, or that has a position property. Hence it is possible to align windows in a window manager, icons in a file browser, etc.

We implemented two other tools: the Spacer and the Pusher. The former is based on our observations of designers who use the width of a graphical object as a “spacer” to control object distribution. The latter is inspired by the physical world, where one can push an object with another. Applying these instruments to existing shapes turns them into spacers or pushers. Spacers bump into other objects and do not overlap them, and other objects bump into spacers. Pushers push objects, and other objects bump into pushers. Moving objects with the Shift key depressed ignores these constraints. Spacers and pushers are implemented in a similar way as StickyLines and work with any object with an extent property, e.g., windows.

Other examples

We developed a number of prototypes with Stratify to experiment and illustrate the power of substrates. We already mentioned the clock and the to-do list, implemented as a collection of substrates supporting synchronized views and direct manipulation using the Move instrument. We also implemented a collection of polymorphic instruments, including:

- a TextEdit instrument to edit any text property of a substrate or one of its sources, including the text labels of the instruments themselves;

- a ColorPicker instrument to change the value of any color property of a substrate or one of its sources;
- a Move instrument that can move shapes and reorder the content of sequences, including tool palettes and to-do lists;
- a Delete tool to remove items in a sequence, including shapes in a canvas, tools in a palette and windows; and
- a Select instrument to select a single item in a sequence, including a tool in a palette, a shape in a canvas or an item in a menu.

***Canvas* –** The Canvas substrate holds graphical shapes. A series of tools let the user create basic shapes and StickyLines, move them and delete them. The pusher and spacer instruments can be applied to any shape in the canvas. An AddTool instrument lets users select a tool in the palette and add it to the canvas, where it can be selected, illustrating the fact that the canvas can hold objects other than simple graphical shapes.

***Table* –** The Table substrate holds tabular data. Cells can hold arbitrary substrates as long as they have a value property. The table can also be connected to an input signal to receive data from a live source and add it to the table.

We turn the table substrate into a spreadsheet by creating a Formula substrate whose value is computed according to a formula, editable with the TextEdit tool. Formulas can reference the values of other cells through the function (row, col). The Data-Reactive programming model ensures that the values of formulas are automatically recomputed.

***DataViz* –** The Histogram substrate takes a Sequence as source and maps it to a sequence of rectangular shapes that can be used as a source for a Canvas. The sequence being mapped can be a row or column of the Table, providing immediate visualization of its content. Updating a table cell is immediately reflected in the histogram.

The Resize tool can be used to change the size of any object with an extent property. Histogram bars use a special resize protocol to ensure that the height of the bar reflects the source value. If the source value cannot be changed, e.g., it is a formula of the spreadsheet rather than a simple value, the height cannot be changed. This does not require any modification to the Resize tool.

***Window Manager* –** Each window of the Window Manager (Fig. 4) can contain an arbitrary substrate, including a web page, a clock, a to-do list, a canvas or a table. Windows can be moved and deleted with the Move and Delete instruments. StickyLines, pusher and spacer instruments can be applied to windows, as in the canvas.

***Sharing* –** To support sharing, we experimented with storing a JSON representation of substrates in ShareDB[9]. STRATIFY uses observers to send all substrate changes to ShareDB. Messages from ShareDB are dispatched to the proper substrate to update its state, maintaining consistency across replicas. Users can thus share a collection of substrates in real time and interact with them simultaneously. For example, one user can reorder items in a to-do list while another is renaming them.

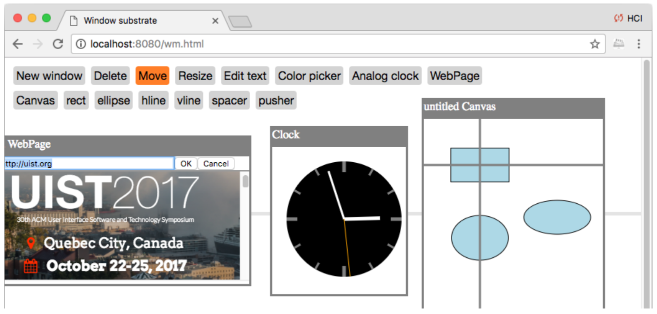

Figure 4. Window Manager substrate with a web page, a clock, a canvas, and a StickyLine to align the windows horizontally.

Although instruments can be shared in this way as well, this is not a good idea: Since a shared instrument is connected to a different input devices at each site, the state of the instrument will be inconsistent across sites, leading to erratic behavior. Sharing the substrates manipulated by the instruments is therefore sufficient. To support awareness of other users, each user should share a substrate that represents the user's activity, such as the real-time cursor position to create a telepointer.

For multi-device support, we experimented with the distribution of a set of substrates and instruments between a desktop computer and a smartphone. For example, the desktop displays a canvas and the smartphone a tool palette. The user can select the tool on the smartphone and use it on the desktop. The smartphone communicates with the desktop by relaying the actions issued on the smartphone to a proxy of the instrument on the desktop.

In summary, STRATIFY uses a combination of Data-Reactive and Functional Reactive programming to implement the conceptual model of information substrates and interaction instruments. The examples illustrate the power of the model and how a small number of substrates and instruments can give rise to a variety of use scenarios, demonstrating the flexibility and extensibility of the resulting interactive systems. We now compare this work to the state of the art.

RELATED WORK

Conceptual models

According to Norman [28], a conceptual model describes both the designer's mental model of the system and the mental model created by users as they interact with the system over time. Unfortunately we have no agreed-upon formalism to describe these models. Johnson & Henderson's [15] conceptual model is based on objects and operations, but remains very high-level. Reality-Based Interfaces [14] promote leveraging qualities of physical objects from the real world into the digital world, but do not propose a clear set of concepts, nor a proof-of-concept implementation.

The Tokens+Constraints [35] conceptual model for tangible interaction is closer to our work, as it is more precise, and can serve as a basis for implementation. Our model builds on and expands Instrumental Interaction [1] and the design principles derived from it [2], with an emphasis on generative power: the model helps imagine new solutions to design problems by analyzing them in terms of substrates and instruments.

[9] https://github.com/share/sharedb

Substrates in HCI

The notion of substrate has been used before in the context of Human-Computer Interaction. Paper Substrates [8] let music composers create complex pieces by linking and layering pieces of interactive paper that represent and interpret digital data. In particular, transparent paper substrates can be used to interpret data visible through them. They are, in a sense, a literal version of our layers of representation.

Graphical substrates [25] are mental models created by graphic designers to represent their layout ideas. The authors observe that digital tools do not support these constructs and present several prototypes that address this gap. Like Paper Substrates, these prototypes are good examples of novel substrates and instruments that could be created with STRATIFY.

Webstrates [21] uses web technologies to implement shareable dynamic media, sharing the DOM across multiple clients in real time with ShareDB. Like our model, it uses interaction instruments and blurs the distinction between applications and documents. Unlike our model, it does not support multi-level representations of digital information.

Finally, Repening [32] introduces AgentSheets as a "programming substrate" for creating interactive learning environments. This is quite different from our use of the term, although we could certainly analyze AgentSheets with our model.

Interactive Environments

Over the years, many environments have been created to support flexibility and extensibility. Most are programming environments targeted at developers rather than end-users. Alan Kay talked of software as "clay" [18] and created environments such as Smalltalk [12]. More recently, Lively [13] offers a similar approach, running in a web browser. While we can envision a programming environment for Substrates, this is not the primary goal of this work. Rather, we seek to provide a conceptual model for designers and developers that leads to more flexibility and extensibility for users.

Our work is closer in spirit to the Alternate Reality Kit [33], where objects can be manipulated with other objects, based on a physical metaphor. While we do not seek the same type of literalism, we pursue a similar goal of facilitating the free combination of objects through polymorphic tools.

Beyond interactive environments, a number of systems have demonstrated the power of breaking out of the app model. For examples, Buttons [24] let users encapsulate a piece of interactive behavior in a configurable button that can be shared. Interface attachments [29] augment existing applications, based on their surface representations. Scotty [7] injects code into existing applications to augment or replace its features. Our work seeks to create more open environments where such features can easily be added, rather than relying on ad hoc solutions.

Finally, previous work has recognized the need to better support distributed interfaces and multi-user interaction. For example, Recombinant Computing [27] emphasizes the need for systems and services to be fluidly recombined even with very limited knowledge of each other. We have a similar goal, even though our approach is based on different concepts.

Software architectures

The dominant architectural pattern for interactive systems is the Model-View-Controller (MVC) [22] and its variants. MVC is based on the dichotomy between the world of the system (the model) and the world of the user (the view). Our approach is more powerful: a substrate can be both a view (represent another substrate) and a model (be represented by another substrate). MVC also does not reify interaction, unlike our instruments and interaction protocols.

Interactive systems are usually programmed with events and listeners, but event-driven programming is notoriously error-prone and hard to maintain [26]. Other approaches such as functional reactive programming [36] and dataflow [6] are powerful, but rarely used in current systems due to limited toolkit support. Our combination of Functional and Data-Reactive programming smoothly integrates with the host language, making it easy to use. The use of computed values rather than bidirectional constraints as in, e.g., ThingLab [3] makes the behavior more predictable at the expense of having to write some property setters. Compared with the many data-binding frameworks for the web, such as React or Angular[10], STRATIFY provides a more general reactive model well-integrated with the language.

Finally, VIGO [20] operationalizes instruments into an architectural model and Shared Substance [11] introduced data-oriented programming. STRATIFY builds on this previous work and provides a novel unified model that combines substrates and instruments as well as a sample implementation.

CONCLUSION

This paper introduces a conceptual model informed by how humans interact in the world, with a much higher level of flexibility and extensibility than current environments. *Information substrates* represent rich digital "matter" by supporting multiple, interdependent levels of representation of digital information. *Interaction instruments* enable powerful user and system agency by decoupling the sources of actions from the way they are enacted by the target substrates. We developed STRATIFY, a proof-of-concept implementation to validate and experiment with this conceptual model. STRATIFY uses Data-Reactive Programming which, combined with Functional Reactive Programming, provides a powerful computational model for static dependencies and dynamic changes. The collection of prototypes we created demonstrates the power and generality of this model.

Future work will continue to explore the conceptual model by refining it and applying it to a wider range of interaction styles. At the implementation level, we plan to experiment with pixel substrates and better support persistence and multi-user, multi-device interaction. In the longer term, we want to create a full-fledged environment and test it in real settings in order to explore its "adjacent possible" [17], to see how users will adopt and appropriate it.

[10] https://facebook.github.io/react/, https://angularjs.org

ACKNOWLEDGMENTS

This work was partially supported by European Research Council (ERC) grant n° 695464 ONE: Unified Principles of Interaction.

REFERENCES

1. Michel Beaudouin-Lafon. 2000. Instrumental Interaction: An Interaction Model for Designing post-WIMP User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '00)*. ACM, New York, NY, USA, 446–453. DOI: http://dx.doi.org/10.1145/332040.332473
2. Michel Beaudouin-Lafon and Wendy E. Mackay. 2000. Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '00)*. ACM, New York, NY, USA, 102–109. DOI: http://dx.doi.org/10.1145/345513.345267
3. Alan Borning. 1981. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Trans. Program. Lang. Syst.* 3, 4 (Oct. 1981), 353–387. DOI: http://dx.doi.org/10.1145/357146.357147
4. Pascal Béguin and Pierre Rabardel. 2000. Designing for instrument-mediated activity. 12 (2000), 173–190.
5. Marianela Ciolfi Felice, Nolwenn Maudet, Wendy E. Mackay, and Michel Beaudouin-Lafon. 2016. Beyond Snapping: Persistent, Tweakable Alignment and Distribution with StickyLines. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 133–144. DOI: http://dx.doi.org/10.1145/2984511.2984577
6. Pierre Dragicevic and Jean-Daniel Fekete. 2001. Input Device Selection and Interaction Configuration with ICON. In *People and Computers XV—Interaction without Frontiers (HCI & IHM 2001)*. Springer, London, 543–558. DOI:http://dx.doi.org/10.1007/978-1-4471-0353-0_34
7. James R. Eagan, Michel Beaudouin-Lafon, and Wendy E. Mackay. 2011. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 225–234. DOI: http://dx.doi.org/10.1145/2047196.2047226
8. Jérémie Garcia, Theophanis Tsandilas, Carlos Agon, and Wendy Mackay. 2012. Interactive Paper Substrates to Support Musical Creation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. ACM, New York, NY, USA, 1825–1828. DOI: http://dx.doi.org/10.1145/2207676.2208316
9. Eleanor Jack Gibson. 1969. Principles of Perceptual Learning and Development. (1969).
10. James J Gibson. 1986. *The Ecological Approach to Visual Perception*. Lawrence Erlbaum.
11. Tony Gjerlufsen, Clemens Nylandsted Klokmose, James Eagan, Clément Pillias, and Michel Beaudouin-Lafon. 2011. Shared Substance: Developing Flexible Multi-surface Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, New York, NY, USA, 3383–3392. DOI: http://dx.doi.org/10.1145/1978942.1979446
12. Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
13. Daniel Ingalls, Tim Felgentreff, Robert Hirschfeld, Robert Krahn, Jens Lincke, Marko Röder, Antero Taivalsaari, and Tommi Mikkonen. 2016. A World of Active Objects for Work and Play: The First Ten Years of Lively. *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (2016), 238–249. DOI: http://dx.doi.org/10.1145/2986012.2986029
14. Robert J.K. Jacob, Audrey Girouard, Leanne M. Hirshfield, Michael S. Horn, Orit Shaer, Erin Treacy Solovey, and Jamie Zigelbaum. 2008. Reality-based Interaction: A Framework for post-WIMP Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, New York, NY, USA, 201–210. DOI: http://dx.doi.org/10.1145/1357054.1357089
15. Jeff Johnson and Austin Henderson. 2011. *Conceptual Models: Core to Good Design*. Morgan Claypool.
16. Jeff Johnson, Teresa L. Roberts, William Verplank, David Canfield Smith, Charles H. Irby, Marian Beard, and Kevin Mackey. 1989. The Xerox Star: A Retrospective. *Computer* 22, 9 (1989), 11–26.
17. Steven Johnson. 2010. *Where Good Ideas Come From: The Natural History of Innovation*. Riverhead Books.
18. Alan C. Kay. 1984. Computer Software. 251 (1984), 53–59.
19. Roberta L Klatzky, Brian MacWhinney, and Marlene Behrmann. 2012. *Embodiment, Ego-Space, and Action*. Psychology Press.
20. Clemens Nylandsted Klokmose and Michel Beaudouin-Lafon. 2009. VIGO: Instrumental Interaction in Multi-surface Environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 869–878. DOI:http://dx.doi.org/10.1145/1518701.1518833
21. Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable Dynamic Media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software &#38; Technology (UIST '15)*. ACM, New York, NY, USA, 280–290. DOI: http://dx.doi.org/10.1145/2807442.2807446

22. Glenn E. Krasner and Stephen T. Pope. 1988. A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80. *J. Object Oriented Program.* 1, 3 (Aug. 1988), 26–49. http://dl.acm.org/citation.cfm?id=50757.50759

23. Wendy E. Mackay. 2000. Responding to cognitive overload: Co-adaptation between users and technology. *Intellectica* 30 (Jan. 2000), 177–193.

24. Allan MacLean, Kathleen Carter, Lennart Lövstrand, and Thomas Moran. 1990. User-tailorable Systems: Pressing the Issues with Buttons. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90)*. ACM, New York, NY, USA, 175–182. DOI: http://dx.doi.org/10.1145/97243.97271

25. Nolwenn Maudet, Ghita Jalal, Philip Tchernavskij, Wendy Mackay, and Michel Beaudouin-Lafon. 2017. Beyond Grids: Interactive Graphical Substrates to Structure Digital Layout. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 11 pages. To appear.

26. Brad A. Myers. 1991. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-backs. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology (UIST '91)*. ACM, New York, NY, USA, 211–220. DOI: http://dx.doi.org/10.1145/120782.120805

27. Mark W. Newman, Jana Z. Sedivy, Christine M. Neuwirth, W. Keith Edwards, Jason I. Hong, Shahram Izadi, Karen Marcelo, and Trevor F. Smith. 2002. Designing for Serendipity: Supporting End-user Configuration of Ubiquitous Computing Environments. In *Proceedings of the 4th Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques (DIS '02)*. ACM, New York, NY, USA, 147–156. DOI:http://dx.doi.org/10.1145/778712.778736

28. Don Norman. 1998. *The Design of Everyday Things*. Doubleday.

29. Dan R. Olsen, Jr., Scott E. Hudson, Thom Verratti, Jeremy M. Heiner, and Matt Phelps. 1999. Implementing Interface Attachments Based on Surface Representations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*. ACM, New York, NY, USA, 191–198. DOI: http://dx.doi.org/10.1145/302979.303038

30. François Osiurak. 2010. What Neuropsychology Tells Us about Human Tool Use? The Four Constraints Theory (4CT): Mechanics, Space, Time and Effort. (2010). DOI: http://dx.doi.org/10.1007/s11065-014-9260-y

31. François Osiurak and Arnaud Badets. 2016. Tool Use and Affordance: Manipulation-Based Versus Reasoning-Based Approaches. (2016). DOI: http://dx.doi.org/10.1037/rev0000027

32. Alex Repenning. 1994. Programming Substrates to Create Interactive Learning Environments. *Interactive Learning Environments* 4, 1 (1994), 45–74.

33. Randall B. Smith. 1987. Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic. *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface* (1987), 61–67. DOI: http://dx.doi.org/10.1145/29933.30861

34. Paul S. Strauss. 1993. IRIS Inventor, a 3D Graphics Toolkit. *SIGPLAN Not.* 28, 10 (Oct. 1993), 192–200. DOI:http://dx.doi.org/10.1145/167962.165889

35. Brygg Ullmer, Hiroshi Ishii, and Robert J. K. Jacob. 2005. Token+Constraint Systems for Tangible Interaction with Digital Information. *ACM Trans. Comput.-Hum. Interact.* 12, 1 (March 2005), 81–118. DOI: http://dx.doi.org/10.1145/1057237.1057242

36. Zhanyong Wan and Paul Hudak. 2000. Functional Reactive Programming from First Principles. *SIGPLAN Not.* 35, 5 (May 2000), 242–252. DOI: http://dx.doi.org/10.1145/358438.349331

Appendix

The code in this appendix shows the generic Drag instrument, the Move instrument that subclasses it, and a move protocol to move an object by setting its position property.

```
class DragInstrument {
  // Methods to be redefined in subclasses
  draggable() {return null;}       // return a protocol
  startAction() {return null;}     // return an action
  dragAction() {return null;}        // return an action
  stopAction() {return null;}      // return an action

  // Called in response to incoming signal from input device
  startDrag() {
    // Find the target
    let res = this.draggable();
    if (! res) return;
    // Initialize state
    this.dragging = true;
    this.delta = {x: 0, y: 0};
    this.incr = {x: 0, y: 0};
    // Instantiate protocol and connect our signal to it
    this.signal = new Signal();
    let interaction = new res.protocol(this, res.target);
    interaction.connect(this.signal);
    // Send start action
    let action = this.startAction();
    if (action) this.signal.next(action);
  }

  drag() {
    // Update state
    this.delta.x += this.source.delta.x;
    this.delta.y += this.source.delta.y;
    // Send drag action
    let action = this.dragAction();
    if (action) this.signal.next(action);
  }

  stopDrag() {
    // Send stop action
    let action = this.stopAction();
    if (action) this.signal.next(action);
    this.dragging = false;
    // Shutdown signal
    this.signal.complete();
    this.signal = null;
  }

  // Called to activate/deactivate this tool
  activate() {
    // subscribe to source signal, call startDrag, drag, stopDrag
    // when mouse down/move/up values are received
  }

  deactivate() {
    // unsubscribe from source signal
  }

  constructor(input) {
    this.source = input;
    this.signal = null;
  }
}
```

Listing 5. The Drag instrument is designed to be subclassed by instruments that move, resize or otherwise respond to a drag interaction.

```
class MoveInstrument extends DragInstrument {
  draggable()  {
    // look for a target with a 'move' protocol
    let protocol = source.findNamedProtocol('move');
    if (protocol) return protocol;

    // else look for a target with a 'position' property
    return this.source.findNamedPropertyProtocol('position',
     Point);
  }

  startAction() { // return 'start' action
    return { type: 'start' }
  }

  dragAction() {
    // return 'move' action with incremental and total move
    return {
      type: 'move',
      incr: this.source.delta,
      delta: this.delta,
    };
  }

  stopAction() { // return 'stop' action with total move
    return { type: 'stop', delta: this.delta }
  }
}
```

Listing 6. The Move instrument simply redefines the methods of DragInstrument to find a target and send actions.

```
class MovePositionProtocol extends Protocol {
  connect(signal) {
    // Call move when receiving 'move' actions, but only
    // once the cursor has moved more than 3 pixels
    signal.filter(a => a.type === 'move')
         .skipWhile(a => norm(a.delta) < 3)
         .subscribe(a => this.move(a));
  }

  move(a) {
    // Set the position of the target object
    this.target.position.x += a.incr.x;
    this.target.position.y += a.incr.y;
  }
}
```

Listing 7. The protocol to move the position of a substrate filters the input signal and updates the 'property' position of the target