

Design Choices of Document-Oriented Programming Systems

Tomas Petricek^a , Jonathan Edwards^b , and Joel Jakubovic^a 

a Charles University, Prague, Czechia

b Independent, Boston, MA, USA

Abstract Many interesting programming systems treat programming as document manipulation. Examples include spreadsheets, data science notebooks, educational environments like Boxer and multiple recent research programming systems. In such systems, the programmer interacts with a document that contains a structured representation of both code and data, they modify the code and data, trigger computations and view the results from the unified document interface. Those *document-oriented programming systems* have a unique set of design choices. The concepts that we need to understand them and design them differ from the well-understood design choices known from programming languages and other programming systems.

The aim of this paper is to identify the key design choices that characterise different document-oriented programming systems. We review both historical and recent examples of such systems and identify twelve design choices that cover four aspects of the system design: (i) what structure and representation of document they use, (ii) how is programming done within the systems, (iii) how the user interface displays documents and allows for their editing, and (iv) how are computations within the system evaluated.

The catalogue is rooted in our own need to understand design choices and their consequences when designing multiple different document-oriented programming systems over the last multiple years. It provides a high-level map of the design space of document-oriented programming systems, makes it possible to identify differences and similarities across different systems and also suggests under-explored design choices and combinations of choices as areas for future research.

The key contribution of this work is perhaps not the catalogue of design choices itself, but the fact that we identify a new programming paradigm. Document-oriented programming systems have rich historical roots, widely-adopted contemporary examples, but they are also an active research area. We hope the review presented in this paper will aid future development of this new paradigm.

The Art, Science, and Engineering of Programming

Perspective The Engineering of Programming

Area of Submission Programming Systems

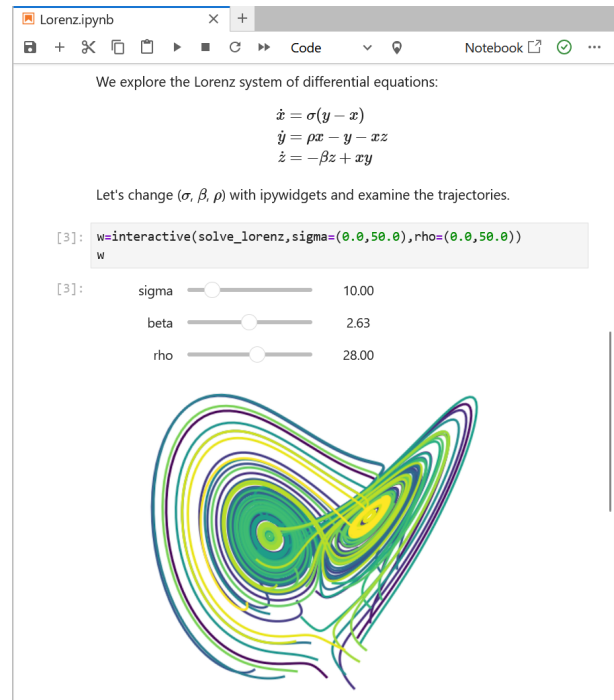
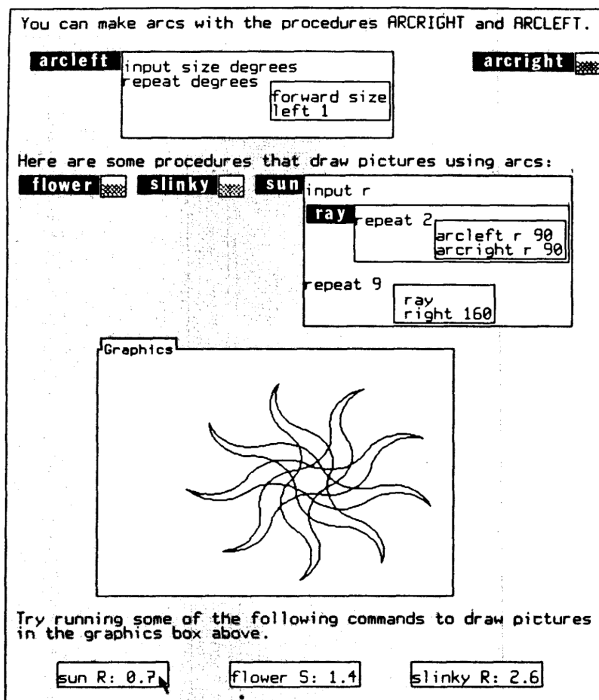


© Tomas Petricek, Jonathan Edwards, and Joel Jakubovic

This work is licensed under a “CC BY 4.0” license

Submitted to *The Art, Science, and Engineering of Programming*.

Design Choices of Document-Oriented Programming Systems



■ **Figure 1** In Boxer [18] (left), code and data are represented as nested boxes. All code and state exists in the document. Executing a box runs the code, which can make changes to the document. In Jupyter [33] (right), documents consist of sequence of cells. The results of evaluating a code cell are displayed in the document, but the full state is managed by the hidden underlying kernel.

1 Introduction

Document-oriented programming systems are programming environments that are built around a user interface modelled after a document editor. The document interface is typically used for editing the underlying program, for executing the program or its fragments, as well as for using interactive elements embedded in the document. The origins of the paradigm can be traced to the 1980s [18] and the paradigm has a rich history, but there has been a renewed interest in the design of document-oriented programming systems in recent years, both in practice and in research (Figure 1).

As historical, as well as recent, examples of document-oriented programming systems illustrate, there are many ways in which the document metaphor can be used as the basis for a programming system. In particular, there is a great variety in the kind of document structure used, in how rich user interfaces are embedded within the document, in how programming is integrated with the document and also in how computations are executed.

We argue that to advance research on document-oriented programming systems, we need to (i) precisely characterise what a document-oriented programming system is, (ii) understand the design choices available to the designers of document-oriented programming systems, and (iii) document the design choices made in existing document-oriented programming systems.

1.1 Definition

The term *programming system* has been used to talk about “an integrated and complete set of tools sufficient for creating, modifying, and executing programs” [30]. The term shifts the focus from programming languages to a more general notion that also encompasses the interactive graphical environments in which programming is done.

Document-oriented programming systems are a particular kind of programming systems, built around the central metaphor of a document. Such systems represent programs as documents that include data, code, documentation, but sometimes also evaluation state and other artefacts. The programming and using of programs in a document-oriented programming system is also done through the document interface.

To make the category of document-oriented programming systems more precise, we use Technical Dimensions of Programming Systems [30]. The framework identifies a number of axes along which programming systems can be placed and contrasted. Document-oriented programming systems occupy a particular sub-space of the entire design space mapped by the technical dimensions framework. The sub-space can be delineated in terms of three main categories of technical dimensions.

- **Notation.** The *primary notation* in a document-oriented programming system is a document. Many systems are based on structured text documents, but some include other media such as images or video. Other document structures include spreadsheets, stacks of cards or text-based Markdown documents. The document often includes *secondary notations* representing formulas or code, possibly in multiple different programming languages. The *notational structure* is thus built around multiple complementing notations.
- **Interaction.** In the main *mode of interaction* in document-oriented programming systems, the user can modify the document, edit and execute code embedded in the document and also interact with any interactive elements in the document. The effects of interacting with the document are mostly immediate, although only a few systems are live. The *feedback loops* in document-oriented programming systems thus typically aim to minimize the gulf of evaluation. Within the main mode of interaction, there may be multiple more specific sub-modes, for example when editing code in a code editor embedded within the document.
- **Customization.** In document-oriented programming systems, it is typically possible to modify the document at any point during the use of the system. In terms of technical dimensions, the *staging of customization* is such that the modification of the document or code embedded in it does not require a special *mode of interaction*. The customization is also done using the original notations of the system. However, it is rarely possible to modify the system itself from within the document and this typically requires a different mode of interaction (such as editing system source code outside of the document and restarting the system).

The aforementioned aspects define *document-oriented programming systems*. While the design space can be further mapped using existing technical dimensions, many design choices that are specific to document-oriented programming systems, are not included in the framework and deserve further examination.

Design Choices of Document-Oriented Programming Systems

1.2 Contributions

Technical dimensions of programming systems [30] map the broad design space of programming systems. As the authors pointed out, programming systems built around text-based programming languages form one well-understood cluster in the space.

In this paper, we identify and zoom in on another interesting cluster of programming systems. Document-oriented programming systems deserve a close examination. The systems in this cluster share multiple technical characteristics and often aim to make programming more accessible. They have a rich history, include some of the most widely used programming systems (spreadsheets) and form an active research area.

- We characterize the notion of a document-oriented programming system in terms of the technical dimensions framework (Section 1.1), going beyond the informal metaphor of a programming system built around a structured document.
- We review a number of diverse past and recent examples of document-oriented programming systems (Section 2). These provide valuable reference points for discussion about system design.
- We present a catalogue of twelve central design choices that designers of document-oriented programming systems face (Sections 3, 4, 5, 6), alongside with interesting examples illustrating the different choices.

To develop the catalogue, we draw from our own experience developing multiple document-oriented programming systems [31, 44, 21, 26, 23], extensive literature review and also discussions at multiple workshops dedicated to past and future programming systems (Boxer Salon 2022, Substrates 2025).

1.3 Using this Paper

The aims of this paper are to highlight an interesting programming systems paradigm, develop a better understanding of the design choices within the paradigm and facilitate further research on document-oriented programming systems. The different aims are best served by different ways of using the paper.

- Readers who want to get an overview of document-oriented programming systems can focus on the introduction (§1) and examples (§2). They can then read the summary paragraphs for each of the design choices and learn about interesting designs by examining the examples (“Example” sections) throughout the paper.
- Researchers who are interested in comparing document-oriented programming systems can read the summary paragraphs for each of the design choices and then proceed to the details of the design choices that are relevant for their systems.
- Designers of novel document-oriented programming systems may find it useful to skim through the entire catalogue to explore interesting, but under-researched alternatives to some of the widely used and established design choices.

Although the paper can be read cover to cover, it is better approached with a specific document-oriented programming system or a design question in mind.

2 Examples

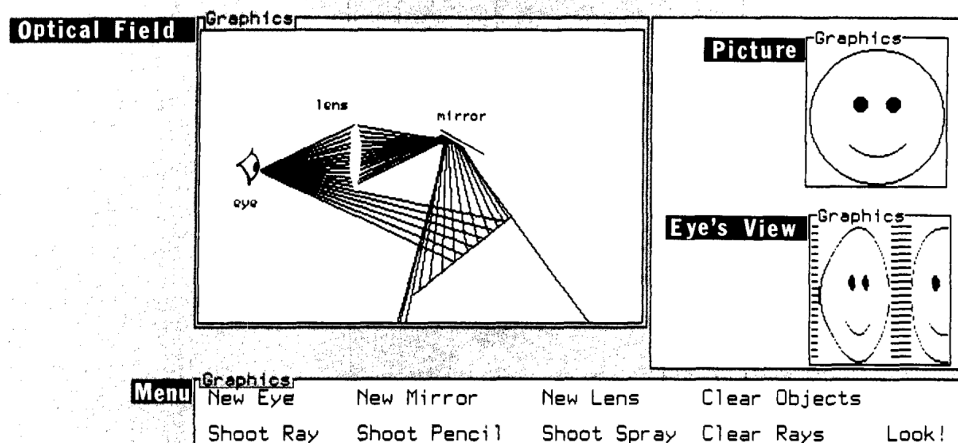
There are many programming systems that match our definition of a *document-oriented programming system*. In this section, we provide a brief overview of some of them, focusing on systems that appear frequently as examples in our catalogue.

2.1 Historical Origins: Boxer

The Boxer system [18, 19] was developed in the early 1980s as an educational interactive computational environment. One of the envisioned use cases of the system was production of interactive textbooks. The readers of such textbook would be able to explore interactive explanations in the textbook, modify parameters, but also see the logic implementing the explanations and potentially customize it (Figure 2).

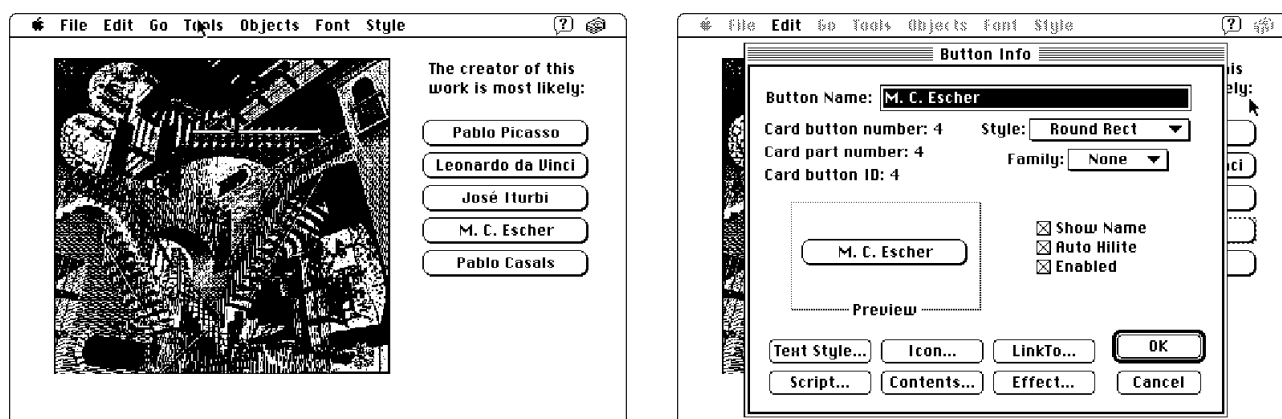
In Boxer, the document consists of nested boxes. There are two main kinds of boxes. DATA boxes contain text, graphics and other boxes. DOIT boxes contain program text and other DOIT and DATA boxes [35, p.2]. Boxes are used to represent structured data (such as a collection of records) as well as nesting structure in code (body of a loop). The system also includes GRAPHICS boxes that can display visual output and respond to user interactions.

The Boxer environment lets the user manipulate the document. They can create new boxes, modify text and code in existing boxes and delete boxes. Boxer uses *spatial metaphor*, leveraging the commonsense knowledge of space to make computing more comprehensible [18, p.2]. Another important principle of Boxer is *naive realism*, which means that what the users see on their screen is “their computational world in its entirety” [18, p.3]. Boxes can be shrunk to hide their contents and even put in closets to hide them [20], but they are always present in the document and can always be located and accessed.



■ **Figure 2** A page from a Boxer textbook on optics. Readers can construct new experiments, but also explore the implementation. (Image source [18])

Design Choices of Document-Oriented Programming Systems



■ **Figure 3** HyperCard. (Screenshot from Archive.org [15])

Computation in Boxer is represented as code in DOIT boxes. The code can be executed by clicking on the box. Conceptually, evaluation of code is based on *copying semantics*. When the running code references another box, the evaluation behaves as if the referenced box was copied in the place where it is referenced from. (The consequence of this model is that the system uses dynamic variable scoping.) Computation can modify state of the document. If it also produces a result, the result is placed in a new DATA box in the document.

Boxer also provides a range of functions for manipulating boxes in the document. For example, the BUILD command can be used to construct a new box from a box that serves as a template with placeholders, in a manner similar to meta-programming in Lisp. However, the underlying Boxer runtime that provides the basic user interface and evaluation logic remains hidden from the user and cannot be modified directly from within Boxer.

2.2 Successful: HyperCard, Spreadsheets and Jupyter

HyperCard

Spreadsheets

Jupyter

2.3 Current Research: Subtext and BootstrapLab

subtext [21] see the direct programming video - <https://vimeo.com/274771188>

list of edits tree of records and lists live evaluation of formulas like spreadsheets code (formula / ast) is also a record with fields for individual operations functional model native GUI with no custom projections evaluation is live final state is an annotation added to the doc

2.4 BootstrapLab

XX

3 Design Choices: Structure

The first set of design choices relates to the structure of the document. They determine what kind of document the user sees and works with, as well as how the document is represented internally by the system.

3.1 Document Shape

The document shape is the visible structure that the user manipulates and interacts with. This is the primary visible notation of the system. In many document-oriented programming systems, the shape is a tree consisting of different kinds of nodes.

- *Tree-Shaped Documents.* The tree can be displayed directly to user using some form of tree view widget that lets users directly manipulate the tree (Subtext [21], BootstrapLab [31]). The tree can be modelled after (or built directly on top of) structured mixed-media documents such as HTML (Webstrates [32], Webnicek [44]), in which case the user can often view and edit the document directly (WYSIWYG) or through a source view (tree view widget).
- *Sequence of Cells.* Computational notebooks for data science (Jupyter [33], Wrattler [45], Datnicek [44]) and programming systems built for journalist (Idyll [14], Planet Hazel [2], The Gamma [43]) use a flat structure where the document is a sequence of cells of different kinds, including textual cells, code cells, interactive cells or outputs of a computation.
- *Two-Dimensional Grid.* In spreadsheet systems [1], the document shape is a regular two-dimensional grid. Cells in the grid can contain data and formulas. In some systems, other structures are overlaid over the grid (such as named ranges) and there may be a secondary notation (macro language).
- *Other Choices.* Other choices are less common. HyperCard [3] documents are stacks of cards, which can contain freely positioned graphical, textual and interactive elements. An unstructured representation is used in Potluck [36], where the document is a plain-text Markdown document (although rendered as rich text).

Example: Ampleforth. In Ampleforth [7], the document shape is not fixed beforehand. The Ampleforth system is embedded in the live programming environment of the object-oriented programming language Newspeak [10], which runs in the web browser.

Internally, Ampleforth documents are Newspeak objects, but they are displayed as rich text documents containing media (using standard capabilities of the web browser) and also interactive Ampleforth elements (which send messages to the underlying Newspeak objects). An Ampleforth document can thus have many different specific shapes. As illustrated by the DocuApps project [8], this includes structured documents, presentation slides and spreadsheets.

The BootstrapLab [31] system has a similar capability. Its underlying representation is a graph (primarily a tree, but with arbitrary links), but what is displayed to the user can be reprogrammed within the system itself. The example implemented in the prototype is a tree view, but other renderings are possible.

3.2 Document Representation

Whereas *Document Shape* is concerned with the user-facing side of a document-oriented programming system, *Document Representation* focuses on the internal data structures that the system maintains. In the technical dimensions framework [30], the two aspects are referred to as *surface notation* and *internal notation*.

- *Document Itself*. In many systems, their internal notation closely resembles to the surface notation and what they display to the user is a direct rendering of their document representation (Boxer [18], HyperCard [3], Jupyter [33]).
- *Underlying Model*. Document-oriented programming systems can be embedded in another programming system with a more basic underlying representation. This can be a soup of objects (Ampleforth [7]) or a graph structure (BootstrapLab [31]).
- *Edit History*. In a number of systems, the ground truth is not the document itself, but a sequence of edit operations through which the document has been created (Subtext [23, 25], Denicek [44]). The document is computed by reapplying the operations, but the history can be used to implement programming by demonstration.

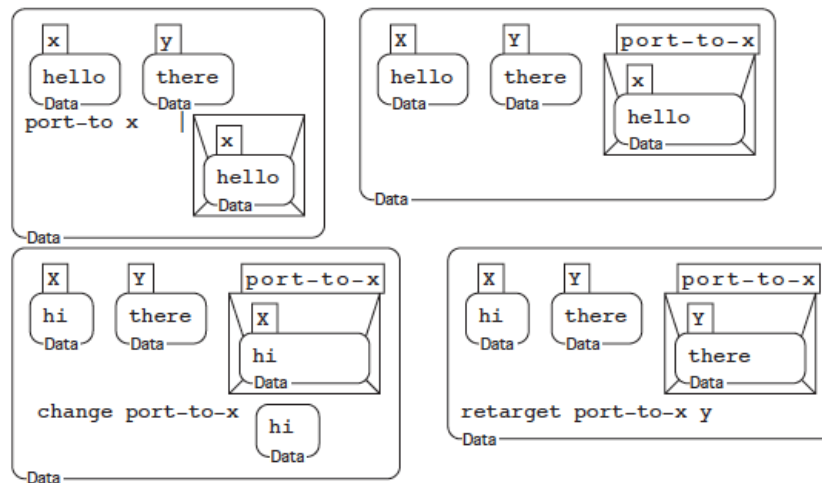
Example: Webstrates. In Webstrates [32], there is no gap between the *internal notation* and the *surface notation*. The Webstrates system is built on top of the web platform and it uses the web browser’s Document Object Model (DOM) as the canonical document representation. Webstrates make the document collaboratively editable by synchronizing it between devices. However, editing is done directly to the document through the browser developer tools or through Codestrates [48], which is a development platform built on top of Webstrates.

3.3 Modularity

The choice of modularity mechanism determines how can aspects of programs be reused. The design choice corresponds directly to the technical dimension *factoring of complexity*. In programming languages, this is often the key concern and possible solutions range from classes and inheritance to domain-specific languages and type classes. In document-oriented programming systems, most programs are more direct and only implement more basic modularity mechanisms (possibly posing an interesting open question for the future of the paradigm).

- *Transclusions*. Transclusions [38] is a hypertext concept, where a resource, or a part of a document, is included in multiple different places at once. In document-oriented programming systems, they allow modularity without introducing additional concepts (such as a function call) to the end-users. They can be used to reuse a component existing inside the document (Boxer [18]), as well as content that may exist elsewhere in the underlying model (Ampleforth [7]).
- *Internal References*. A more basic mechanism for modularity and code reuse is a reference to an entity defined elsewhere in the document. (See §6.3 for different referencing mechanisms.) Unlike with transclusions, the entity is not displayed

Design Choices of Document-Oriented Programming Systems



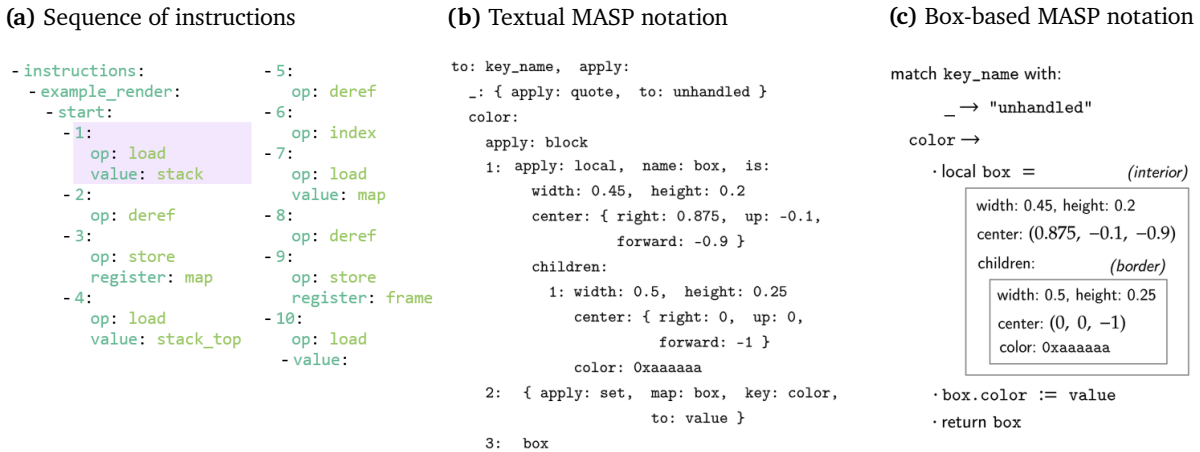
■ **Figure 4** Portals (a transclusion mechanism) in Boxer. A port is created using the port-to command (left top) and is itself a box that can be named (left right). Changing the value of the portal box changes the original source (bottom left) and portals can be redirected (bottom right). Image source [20].

in the document location where it is referenced (the user typically sees the entity name). The reference can refer to functions (BootstrapLab [31]), interactive components (Livelits [41]), or more generally ancestral nodes (Subtext [21]).

- *External References.* In many document-oriented programming systems, the ability to define new abstractions within the document is limited and so modular components or code are defined outside of the document. Those can be external libraries (Jupyter [33]) or interactive components ([14]).
- *Edit History Reference.* In systems where the underlying representation of a document is a history of edits (see §3.2), it is also possible to specify behavior by referring to a part of the history (replaying the edits then performs the behavior again). The mechanism can be used to implement programming by demonstration [17] (Subtext [21], Denicek [44]).

Example: Boxer. In Boxer, parts of a document can be reused through ports (Figure 4). They are views of another box and, despite looking differently, behave identically to their target. The Boxer Structures report [20] lists a number of typical uses of ports.

Ports can be used as a user interface mechanism to provide easy access to another box hidden somewhere in the document, or to create an easy to follow hyperlink. They can be used to share data between parts of a document. Ports are also useful as object references, for example to create a list of objects (existing elsewhere in a document) to be processed. Although Boxer supports named references to variables and procedures, ports can also be used for this purpose to avoid name mix-ups.



■ **Figure 5** Two different programming models in BootstrapLab [31]. **(a)** Imperative instructions to mutate document that read the top of a stack and store the result in a register (a special document node). **(b)** A functional document transformation that replaces color field with a filled rectangle that has the given color. **(c)** Visual rendering of the same MASP code. (Image source [29])

4 Design Choices: Programming

An essential characteristic of document-oriented programming systems is that they allow some kind of programming. However, our definition admits both end-user document systems with limited programming capabilities and systems that can be fully reprogrammed from within themselves. Systems also differ in their programming model, as well as where and how they store program code.

4.1 Programming Model

The basic programming model of a document-oriented programming system can be built around the standard programming language paradigms.

- *Imperative.* In the imperative programming model, code embedded in the document consists of operations that mutate some state (see §6.2). In some systems, commands can modify the document itself (HyperCard [3], Ampleforth [7], Boxer [20], Webstrates [32]), while in others, commands can only modify state that exists behind the scene but not the document structure (Jupyter [33]).
- *Declarative.* In the declarative model, code in the document describes a computation that should be performed in order to obtain some result (Subtext [21], Spreadsheets [1]). The result may be displayed alongside the document, or placed in the document (see §6.2), but the code does not explicitly modify the document.
- *Reactive.* When the declarative programming model makes it possible to write code that explicitly responds to user interactions with the system, it can be referred to as reactive (Renkon-pad [40], Lopecode [34]). Note that spreadsheets support live updates, but only in response to change of the source (data).

Design Choices of Document-Oriented Programming Systems

- *Transformation.* Another variation on the declarative programming model exists in systems where computations operate on document fragments and their result is also a document fragment. The evaluation may transform the document (Web-nicek [44]), but the model has origins in systems that generate new documents as the result (XSLT [13], Document calculus [16])

Example: BootstrapLab. BootstrapLab [31] shows that a document-oriented programming system may support multiple different programming models. The primitive programming model in BootstrapLab is imperative. Code is represented as a sequence of low-level instructions that modify a set of registers (Figure 5 (a)). The instruction pointer register determines the next instruction to be executed. The programming model includes operations such as key lookup, copying, and conditional jump.

A high-level programming model envisioned in BootstrapLab is the MASP language, modelled after LISP (Figure 5 (b)). The document structure (nested maps) represents the abstract syntax tree of functional-first programs. The code can be rendered as textual tree, but also visually (Figure 5 (c)). The MASP interpreter stores all immediate information, as well as the final result, directly in the document (see §6.2). Although this was not done in the BootstrapLab prototype, a MASP interpreter or compiler could be built in the low-level programming model.

4.2 Metaprogramming Capabilities

Document-oriented programming systems differ by the degree to which they can be modified from within themselves. This is known as *self-sustainability* in the technical dimensions framework [30]. As most document-oriented programming systems are not self-sustainable (i.e., not implemented and modifiable from within themselves), we refer to the design choice as *Metaprogramming Capabilities*.

- *No Metaprogramming.* In many document-oriented programming systems, computation can only produce results that are displayed in the document (Spreadsheets [1], Idyll [14], Potluck [36]).
- *Limited Metaprogramming.* In most notebook systems, code written by the user cannot arbitrarily modify the structure of the notebook, but there may be a special operation for, for example, adding a new cell (Jupyter [33], Observable [39]).
- *Document Metaprogramming.* In a number of document-oriented programming systems, code running in the system has the capabilities to freely modify the structure of the document, but it cannot modify the system itself (HyperCard [3], Webstrates [45], Wrattler [45], Boxer [18]).
- *Full Metaprogramming.* Only document-oriented programming systems where much of the system itself is implemented within itself can be modified from within code in the document. (Ampleforth [7], BootstrapLab [31], Lopecode [34]). The degree depends on what the minimal unmodifiable runtime of the system is.

Example: Lopecode. Metaprogramming capabilities of most notebook systems are very limited, but Lopecode [34] shows that this is not inevitable. The system is built on top of the reactive kernel from Observable [39], which maintains a dataflow graph of the computation. Everything else, including the document model based on cells and the editor is implemented (and can be modified from) Lopecode notebooks.

4.3 Code Representation

How is code represented in a document-oriented programming system and where can it be found? We consider these two aspects together as a single design choice.

- *Naive Realism vs. Hidden Representation.* In the naive realism model, everything that exists in the system is available in the document (Boxer [18], Jupyter [33], Subtext [21]). In contrast, systems where code exists outside of the document use the hidden representation model (HyperCard [3], Ampleforth [7]).
- *Structured Representation vs. Textual Representation.* Most document-oriented programming systems use a structured representation of a document, such as a tree (see §3.1). In such systems, code can be represented as an AST using the same structure and edited using a strucute editor (Boxer [18], Subtext [21], Denicek [44], Hazel [41]). The alternative is to use a textual representation of code alongside with an (embedded) text editor (Jupyter [33], Potluck [36], Ampleforth [7]).

4.4 Document Structure Checking

In programming languages, static typing is the most common mechanism for early *error detection* (see technical dimensions [30]). In document-oriented programming systems with *Document* or *Full Metaprogramming* (see §4.2), it is possible to use a mechanism akin to type checking to ensure that operations transforming the document result in a well-formed document structure.

Documents manipulated by systems that use the *Tree-Shaped Document* model (see §3.1) may be irregular containing, for example, different structure of data in different document sections. As such, it is also an interesting problem to define what is considered a well-formed document structure.

- *Minimal Structure.* Systems where document structure is very regular (Spreadsheets [1]) do not typically need sophisticated structure checking to ensure the document is well-formed (but even they have possible constraints, such as cycles). In computational notebooks (Jupyter [33]), checking may be useful for the embedded code, but not to ensure the well-formedness of the notebook structure.
- *Implicit Structure.* Document systems that let users programmatically manipulate documents with more complex shapes may allow arbitrary modifications of the structure (Webnicek [44], BootstrapLab [31]). Changing the structure in a way that is incompatible with other code in the document then results in a runtime error, reported when the other code is executed.

Design Choices of Document-Oriented Programming Systems

- *Explicit Structure.* In some systems, the document structure can be explicit using a mechanism such as a type system or using prototypes. This is an under-explored area of research in programming systems (Subtext'21 [25]), but examples exist in document manipulation (XDuce [28], Ur/Web [12]).

Example: Subtext 10 and Baseline. An example of a system with explicitly checked document structure is Subtext 10 [24] (which served as the basis for a more recent prototype implementation Baseline [26]). Subtext 10 uses a form of static type checking that is based on concrete values. The type of a collection is determined by a special element (with index ***) that serves as a default value (or a prototype [6]).

The default value is used when adding a new item to the collection to ensure that all items have the same structure. Similarly, a structural document edit (e.g., adding, renaming or removing a record field) can only be applied to the default value and results in the change being applied to all items of the collection. In Subtext 10, default values were also proposed for function arguments.

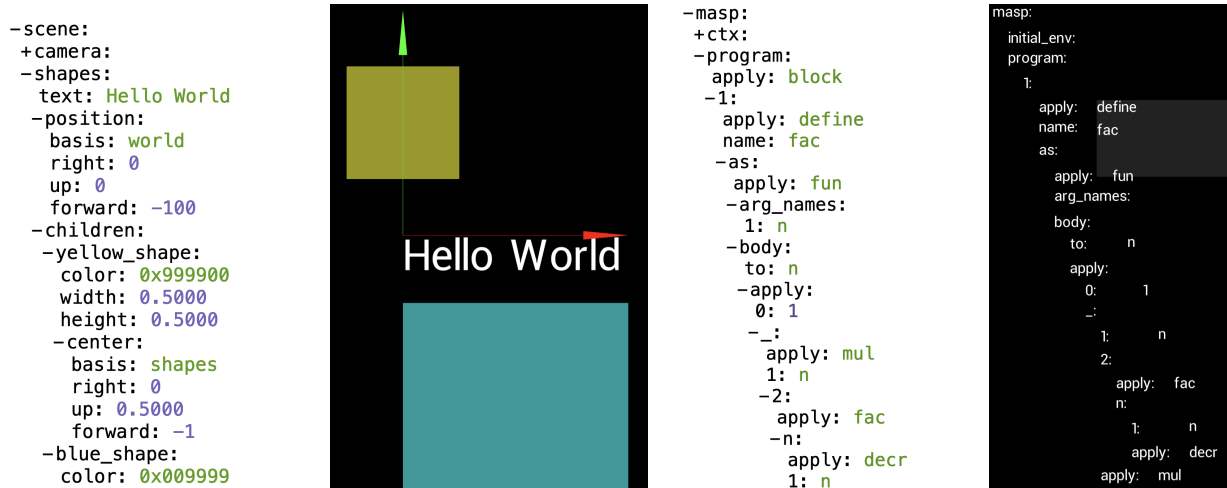
5 Design Choices: Graphical Interface

Whereas some of the design choices related to structure (§3) and programming (§4) coincided with the more general technical dimensions [30], the design choices concerning user interface are new in this paper. We consider two complementary mechanisms, for display and for editing.

5.1 Display Mechanism

A defining characteristic of document-oriented programming systems (§1.1) is that the user sees and interacts with a document. There are multiple display mechanisms through which the document can appear on screen.

- *Naive Realism.* Systems where the document representation is the document itself, or edit history from which the document is obtained (see §3.2), the system may have a standard mechanism for displaying the document (Boxer [18], Webstrates [32], Webnicek [44]). Custom graphical elements can be constructed using components of the standard mechanism such as graphics boxes [20] or HTML.
- *Renderer with Widgets.* In systems where the document representation is less expressive, custom graphical elements can be implemented as widgets that are defined outside of the document itself. They can then be created programmatically from within the document or used automatically to display (and edit) specific types of values. (Livelits [41], Jupyter Widgets [46]).
- *API Calls.* Document-oriented programming systems where code exists outside of the document can modify the displayed document through some API provided by the system, for example to construct document elements and draw (HyperCard [3]) or generate document fragments (Ampleforth [7]).



■ **Figure 6** Memory-mapped display mechanism in BootstrapLab. A sample scene consisting of text, rectangles and arrows (a) with rendering (b). A tree representing a MASP function, rendered outside of BootstrapLab (c) and using the memory-mapped display mechanism (d). Image source [29].

Example: BootstrapLab. An under-explored approach for document rendering has been used in BootstrapLab [31]. The document in the system is a tree structure consisting of nodes that map keys to other nodes or primitive values. The document is not displayed directly to the user. Instead, it contains a special sub-tree (scene) that defines a visual structure to display (as a scene consisting of vector shapes).

The system can display arbitrary part of the document by filling the scene sub-tree with shapes rendering the specified sub-tree (Figure 6). The mechanism, inspired by memory-mapped screen on microcomputers, makes it possible to keep all code and data in a tree-shaped document, but fully control what is displayed on the screen.

5.2 Editing Mechanism

The editing mechanism used in a document-oriented programming system is usually tightly linked with the display mechanism. However, it is worth separating the two design aspects as a system can, for example, display a document visually, but use a plain text editor.

- *Structure Editing.* Common approach is to let the user directly edit the structure of the document through a structure editor that works directly with the underlying structure such as tree nodes (Denicek [44], Webstrates [32], Boxer [18]).
- *Projectional Editing.* If the display mechanism lets users construct widgets for some parts of the document, the widgets may behave as projectional editors [51] and propagate interactively made edits back to the document (Livelits [41]). Some widgets are display-only and do not modify the document (Jupyter Widgets [46]).

Design Choices of Document-Oriented Programming Systems

- *Plain Text.* In some systems, the primary means of editing is text-based. Text editing may be used only for code and markup (Spreadsheets [1], Jupyter [33]), but there are also systems where the whole document is edited as plain text (Potluck [36]).
- *In-System Editor.* Systems that provide sufficient expressive power and rendering capabilities to fully control the display and handling of user events can, in principle, support any kind of editing interface, even if they typically use one of the above design choices (Ampleforth [7], BootstrapLab [31], Lopecode [34]).

6 Design Choices: Evaluation

The fourth and last category of design choices discussed in this paper relate to how are computations in a document-oriented programming system evaluated. The design aspect is related to the choice of a *Programming Model* (§4.1), but here we focus more specifically on how the evaluation of code happens.

6.1 Evaluation Mode

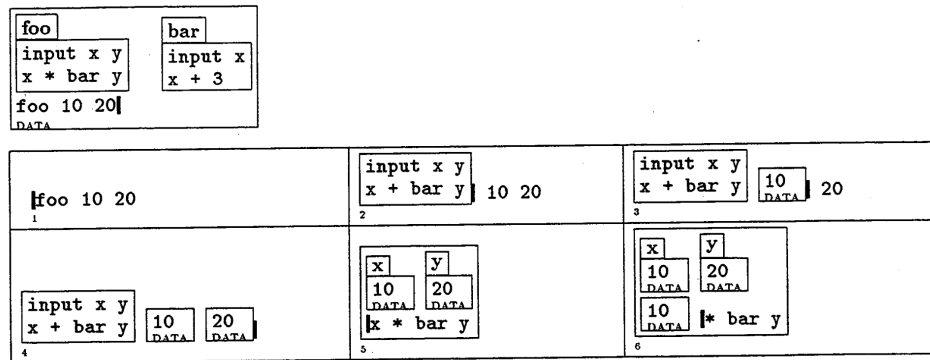
The first choice is concerned with what triggers evaluation. The different choices have been discussed extensively in the context of live programming and our list below partly follows the established levels of liveness [49, 50]. In technical dimensions [30], the evaluation mode is determined by the structure of *feedback loops*.

- *Explicit Trigger.* In this mode, computation is triggered explicitly by a user action. The choice is common in systems where computation modifies state (imperative *Programming Model*, see §4.1) or where computations are long-running (Jupyter [33]).
- *Implicit Trigger.* More immediate feedback may be provided by automatically recomputing the affected parts of the document when code or data change. This is a widely used approach for declarative systems and systems focusing on small code snippets (Spreadsheets [1], Potluck [36], Ampleforth [9]).
- *Live Update.* The last mode refers to systems where ongoing evaluation reflects changes to data and code made during evaluation (liveness level 4 [49]). This mode is under-explored, but can likely be supported by reactive document-based programming systems (Lopecode [34], RenkonPad [40]).

6.2 Evaluation State

A design concern specific to document-oriented programming systems is the interaction between the evaluation mechanism and the document representation. A number of interesting design options arises when the evaluation mechanism stores information directly in the underlying document during (or at the end of) the evaluation.

- *Ephemeral Results.* Many systems do not modify the document in any way when evaluating computations (Spreadsheets [1], Jupyter [33]). The results are stored in some representation that exists outside of the document and are displayed to the



■ **Figure 7** First six step of the execution of `foo 10 20` in the Boxer movie stepper. The box above shows the original source code. The box below shows the successive steps. (1) The evaluation starts, (2) function definition replaces the reference, (3, 4) arguments are wrapped in data boxes, (5) input is replaced with boxes passed as arguments and (6) `x` is replaced with the variable value. (Image source [35])

user, but cannot be directly programmatically manipulated by other computations (besides from constructing other computations that refer to them).

- *Materialized Results.* In systems based on the imperative programming model (see §4.1), the computation directly modifies the document and result of a computation is also stored in the document (HyperCard [3], Webstrates [32]). This could be the case in declarative systems too, although such systems typically use one of the subsequent two choices.
- *Internalized Execution.* A document-oriented programming system can choose to use no other memory or information storage aside from the document that it is working with. In this case, the individual steps of the execution repeatedly modify the state of the document until the computation terminates (BootstrapLab [31], Boxer [18], see the example below).
- *Materialized Execution.* Finally, a system can record every single step of the execution of a computation in the document. This way, the full execution trace is materialized in the document and can be accessed from other code inside the document, for example in order to analyse provenance or execution of tests. The approach has only been implemented experimentally and would require optimization to work well in practice (Denicek [44], Subtext'17 [22], BootstrapLab/MASP [31]).

Example: Boxer. The evaluation model of Boxer [18] is based on the idea of *copying semantics*. In this model, the evaluation of a reference to another box proceeds by replacing the reference with the box itself. The user does not see this during normal use of the system, but the Boxer movie stepper [35] makes it possible to see how the execution proceeds step-by-step. As illustrated in Figure 7, the execution can be seen as a sequence of (mutable) transformations of the document.

6.3 Identification of Nodes

When manipulating documents, the document-oriented programming system (and code that it runs) needs some mechanism for referring to other elements in the document. There are multiple approaches and a programming system may support a combination of these.

- *Human-Readable Identifiers.* The most common approach is to let users name elements they want to refer to using a variable name or human-readable identifier for a document element (Jupyter [33], HyperCard [3], Boxer [18], Webstrates [32]). Names are typically global, although a system could also support namespaces.
- *Absolute Paths.* In systems with a richer document structure, elements can be referred to by a pre-defined addressing scheme (Spreadsheets [1]) or by using a path (a sequence of node names) from the root (Denicek [44], HyperCard [3]). In HyperCard, elements on different cards can have the same name and one refers to them using a path consisting of the stack name, card name and an element name.
- *Internal Unique Identifiers.* To sidestep fragile named references, some programming systems use internal unique identifiers, not visible to the user. The user may see a name, a visual representation of the reference or the referenced value, directly in place of the reference (Subtext [21]). Note that maintaining internal unique identifiers typically requires using a structure editor.
- *Relative References.* In addition to other mechanisms, document nodes can refer to document other nodes through paths that are relative to their location in the document (Spreadsheets [1], Denicek [44]). In trees, this typically requires the ability to refer to a parent element. In spreadsheets dollar references (e.g., \$B\$10) are absolute and ordinary references (e.g., B10) are relative.
- *Selectors.* A richer referencing mechanism allows the user to specify nodes using a combination of paths and selectors, for example to refer to all children of a node, children of a specific tag or class (as in CSS selectors), or an element of a given shape anywhere in the document (Webstrates [32], Denicek [44]).

Example: Denicek and Subtext. Identification of nodes poses an interesting problem in systems where document nodes can be moved and renamed, or in systems based on edit histories (see §3.2) where edits need to be merged or reconciled. Two systems that tackle this problem are Subtext [21] and Denicek [44].

For example, say we have a document with a root node named *pioneers* containing two children named *lovelace* and *hamilton*. In Denicek, one can refer to a specific node using an *Absolute Path* such as `/pioneers/lovelace` or to multiple nodes using a *Selector* `/pioneers/*`. Subtext, by contrast, uses *Internal Unique Identifiers* and the selection made by the user results in a collection of IDs, referring to the selected nodes.

The first consequence of the design choice is how the systems handle renaming of nodes. In Subtext, renaming *pioneers* to *compsci* has no effect on the unique identifiers. In Denicek, the operation requires updating all references, for example, from `/pioneers/*` to `/compsci/*`.

The design choice has interesting consequences when merging edits. If a user makes edit using a selector (such as */pioneers/**) in a system identifies nodes through *Selectors*, the edit affects all nodes including those that may have been created independently and have been merged with the current document (a scenario illustrated in [27]). When the system identifies nodes through *Internal Unique Identifiers*, only nodes known when the edit was done will be affected. On the other hand, merging of edits is challenging when the system uses expressive language of selectors (and Denicek restricts the expressivity of selectors to make merging tractable [44]).

Design Choices of Document-Oriented Programming Systems

[45] [33] forms/3 [11]
Lopecode [34]
spreadsheets [1] subtext [21] denicek [44] hypercard [3] jupyter [33] c.f. wrattler
more live boxer [18]
ampleforth [7, 8] bootstraplab [31] hazel [41] [42] webstrates [32]
potluck [36]
ampleforth can be anything
—
[40] – maybe not exactly document but around DOM
Videostrates <https://pure.au.dk/ws/portalfiles/portal/164699070/VideostratesUIST2019.pdf>
are docs with videos
Active [47] EmbeddedButtons [5, 4]
spreadsheets[1] subtext [21] hypercard [3] jupyter [33] c.f. wrattler more live
boxer [18]
ampleforth [7, 8] hazel [41] [42] bootstraplab [31] denicek [44] webstrates [32]
potluck [36]

7 Related work

not about programming sytems [16] json editor, but similar to livelits [37] Self - not quite documents, but objects as UI
antranig's thing - maybe Varv - maybe
transclusion - used in ampleforth
c.f. Rein's literature review - but it is more difficult to google this

References

- [1] Robin Abraham, Margaret Burnett, and Martin Erwig. *Spreadsheet Programming*, pages 2804–2810. John Wiley & Sons, Ltd, 2009. doi:10.1002/9780470050118.ecse415.
- [2] Cyrus Omar Alexander Bandukwala, Andrew Blinn. Toward a live, rich, composable, and collaborative planetary compute engine. Extended abstract. Programming for the Planet (PROPL) workshop, London, UK, 2024. URL: <https://hazel.org/papers/propl24.pdf>.
- [3] Bill Atkinson. HyperCard. Apple Computer, Inc., 1987.
- [4] Eric A. Bier. Embeddedbuttons: documents as user interfaces. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, UIST ’91, page 45–53, New York, NY, USA, 1991. Association for Computing Machinery. doi:10.1145/120782.120787.
- [5] Eric A. Bier and Ken Pier. Documents as user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’91, page 443–444, New York, NY, USA, 1991. Association for Computing Machinery. doi:10.1145/108844.108994.
- [6] A. H. Borning. Classes versus prototypes in object-oriented languages. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, ACM ’86, page 36–40, Washington, DC, USA, 1986. IEEE Computer Society Press.
- [7] Gilad Bracha. Ampleforth: A live literate editor. Presented at Workshop on Live Programming (LIVE 2022), 2022. Workshop presentation. URL: <https://blog.bracha.org/Ampleforth-Live22/out/primordialsoup.html?snapshot=Live22Submission.vfuel>.
- [8] Gilad Bracha. Docuapps: Ampleforth documents as applications. Presented at Workshop on Live Programming (LIVE 2024), 2024. Workshop presentation. URL: <https://www.youtube.com/watch?v=4GOeYylCMJI&t=28690s>.
- [9] Gilad Bracha. Newspeak by example. Technical report, The Ministry of Truth, 2025. URL: <https://newspeaklanguage.org/samples/Literate/literate.html>.
- [10] Gilad Bracha. The newspeak programming system. Technical report, The Ministry of Truth, 2025. URL: <https://newspeaklanguage.org/pubs/newspeak-2025-tutorial.pdf>.
- [11] Margaret Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11(2):155–206, 2001. doi:10.1017/S0956796800003828.
- [12] Adam Chlipala. Ur/web: A simple model for programming the web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’15, page 153–165, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2676726.2677004.

Design Choices of Document-Oriented Programming Systems

- [13] James Clark. XSL Transformations (XSLT) Version 1.0. Technical report, World Wide Web Consortium (W3C), November 1999. W3C Recommendation. URL: <https://www.w3.org/TR/1999/REC-xslt-19991116/>.
- [14] Matthew Conlen and Jeffrey Heer. Idyll: A markup language for authoring and publishing interactive articles on the web. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, page 977–989, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3242587.3242600.
- [15] Dixie Coutant. Art history (HyperCard card stack). Sierra Vista High School, Archived by archive.org, 1994. URL: https://archive.org/details/hypercard_art-history.
- [16] Will Crichton and Shriram Krishnamurthi. A core calculus for documents: Or, lambda: The ultimate document. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi:10.1145/3632865.
- [17] Allen Cypher and Daniel Conrad Halbert. *Watch what I do: Programming by demonstration*. MIT press, 1993.
- [18] A. A diSessa and H. Abelson. Boxer: a reconstructible computational medium. *Commun. ACM*, 29(9):859–868, September 1986. doi:10.1145/6592.6595.
- [19] Andrea A. diSessa. A principled design for an integrated computational environment. *Hum.-Comput. Interact.*, 1(1):1–47, March 1985. doi:10.1207/s15327051hcio101_1.
- [20] Andrea A. diSessa. Boxer structures. Technical report, The diSessa Family Foundation, 2021. URL: <https://boxer-project.github.io/docs/Boxer%20Structures%202021.pdf>.
- [21] Jonathan Edwards. Subtext: uncovering the simplicity of programming. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 505–518, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1094811.1094851.
- [22] Jonathan Edwards. Reifying programming. Presented at Workshop on Live Programming (LIVE 2017), 2017. URL: <https://vimeo.com/228372549>.
- [23] Jonathan Edwards. Direct programming. Video recording. Online, 2018. URL: <https://vimeo.com/274771188>.
- [24] Jonathan Edwards. The subtext programming language, 2021. Unpublished. URL: <https://github.com/JonathanMEdwards/subtext10/blob/master/doc/language.md>.
- [25] Jonathan Edwards and Tomas Petricek. Typed image-based programming with structure editing, 2021. Presented at Human Aspects of Types and Reasoning Assistants (HATRA'21), Oct 19, 2021, Chicago, US. URL: <https://arxiv.org/abs/2110.08993>, arXiv:2110.08993.
- [26] Jonathan Edwards and Tomas Petricek. A playable demo of live database/-document programming, 2025. Unpublished. URL: <https://thebaseline.dev/LIVE25submission/>.

- [27] Jonathan Edwards, Tomas Petricek, Tijs van der Storm, and Geoffrey Litt. Schema evolution in interactive programming systems. *The Art, Science, and Engineering of Programming*, 9(2):1–34, 2024.
- [28] Haruo Hosoya and Benjamin C. Pierce. Xduce: A statically typed xml processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, May 2003. doi:10.1145/767193.767195.
- [29] Joel Jakubovic. *Achieving Self-Sustainability in Interactive Graphical Programming Systems*. PhD thesis, University of Kent, 2024.
- [30] Joel Jakubovic, Jonathan Edwards, and Tomas Petříček. Technical dimensions of programming systems. *The Art, Science, and Engineering of Programming*, 7(3):13:1–13:59, 2023. URL: <https://programming-journal.org/2023/7/13/>, doi: 10.22152/programming-journal.org/2023/7/13.
- [31] Joel Jakubovic and Tomas Petricek. Ascending the ladder to self-sustainability: Achieving open evolution in an interactive graphical system. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2022*, page 240–258, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3563835.3568736.
- [32] Clemens N. Klokmoose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. Webstrates: Shareable dynamic media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST ’15*, 2015. doi:10.1145/2807442.2807446.
- [33] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks—a publishing format for reproducible computational workflows. In *Positioning and power in academic publishing: Players, agents and agendas*, pages 87–90. IOS press, 2016.
- [34] Tom Larkworthy. The lopecode tour, 2025. URL: <https://observablehq.com/@tomlarkworthy/lopecode-tour>.
- [35] Jr. Leigh L. Klotz. Boxer: The programming language. Bsc thesis, MIT, Cambridge, MA, January 1989.
- [36] Geoffrey Litt, Max Schoening, Paul Shen, and Paul Sonnentag. Potluck: Dynamic documents as personal software. Online, 2022. URL: <https://www.inkandswitch.com/potluck/>.
- [37] Andrew M. McNutt and Ravi Chugh. Projectional editors for json-based dsls. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2023, Washington, DC, USA, October 3-6, 2023*, pages 60–70. IEEE, 2023. doi: 10.1109/VL-HCC57772.2023.00015.
- [38] Ted Nelson. *Literary Machines: The Report on, and of, Project Xanadu Concerning Word Processing, Electronic Publishing, Hypertext, Thinkertoys, Tomorrow’s Intellectual Revolution, and Certain Other Topics Including Knowledge, Education and Freedom*. Mindful Press, Sausalito, CA, 1981.

Design Choices of Document-Oriented Programming Systems


- [39] Observable. Observable: Explore and visualize data together, 2025. URL: <https://observablehq.com/>.
- [40] Yoshiki Ohshima, Adam Bouhenguel, and Matthew Good. Renkon-pad: a live and self-sustaining programming environment in functional reactive programming. In *Proceedings of the 11th Programming Experience Workshop (PX'25)*, Prague, Czech Republic, 2025. To appear.
- [41] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. Filling typed holes with live guis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021*, page 511–525, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454059.
- [42] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL '17*, page 86–99, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009900.
- [43] Tomas Petricek. The gamma: Programming tools for data journalism. Extended abstract. Computation + Journalism Symposium, New York, USA, 2015. URL: <https://tomasp.net/academic/papers/data-journalism/extended-abstract.pdf>.
- [44] Tomas Petricek and Jonathan Edwards. Denicek: Computational substrate for document-oriented end-user programming. In *Proceedings of the 38th ACM User Interface Software and Technology Symposium (UIST 2025)*, Busan, Korea, 2025. To appear.
- [45] Tomas Petricek, James Geddes, and Charles Sutton. Wrattler: reproducible, live and polyglot notebooks. In *Proceedings of the 10th USENIX Conference on Theory and Practice of Provenance, TaPP'18*, page 6, USA, 2018. USENIX Association.
- [46] Project Jupyter. Jupyter widgets 8.1.7 documentation, 2025. Available online. URL: <https://ipywidgets.readthedocs.io>.
- [47] Vincent Quint and Irène Vatton. Making structured documents active. *Electron. Publ.*, 7(2):55–74, 1994.
- [48] Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmoose. Codestrates: Literate computing with webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology, UIST '17*, page 715–725, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3126594.3126642.
- [49] Steven L. Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, 1990. doi:10.1016/S1045-926X(05)80012-6.
- [50] Steven L. Tanimoto. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 31–34, 2013. doi:10.1109/LIVE.2013.6617346.
- [51] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. In Benoît Combemale, David J. Pearce, Olivier

Barais, and Jurgen J. Vinju, editors, *Software Language Engineering*, pages 41–61, Cham, 2014. Springer International Publishing.


Design Choices of Document-Oriented Programming Systems

About the authors


Tomas Petricek is an assistant professor at Charles University. He is interested in finding easier and more accessible ways of thinking about programming. To do so, he combines technical work on programming systems and tools with research into history and philosophy of science. His work can be found at tomasp.net and he can be reached at tomas@tomasp.net.

 <https://orcid.org/0000-0002-7242-2208>

Jonathan Edwards is an independent researcher working on drastically simplifying programming. He is known for his Sub-text series of programming language experiments and his blog at alarmingdevelopment.org. He has been a researcher at MIT CSAIL and CDG/HARC. He tweets [@jonathoda](https://twitter.com/jonathoda) and can be reached at jonathanmedwards@gmail.com.

 <https://orcid.org/0000-0003-1958-7967>

Joel Jakubovic todo

 <https://orcid.org/0000-0003-0252-9066>