Customizing Software by Direct Manipulation of Tabular Data

Anonymous Author(s)

Abstract

In this paper we show how the behavior of a software application can be extended and adapted by direct manipulation, using table-driven customization, a new paradigm that allows end users to customize applications without writing traditional code.

Instead, users directly manipulate a tabular view of the structured data inside the application—rather than writing imperative scripts (as in most customization tools). This simple model also accommodates a spreadsheet formula language and custom data editing widgets, which provide sufficient expressivity to implement many useful customizations.

We illustrate the approach with Wildcard, a browser extension that implements table-driven customization in the context of web applications. Through concrete examples, we show that our paradigm can be used to create useful customizations for real applications. We share reflections from experience using the Wildcard system, on both its strengths and limitations relative to other customization approaches. Finally, we explore how this paradigm might lead to new software architectures that encourage this form of customization.

CCS Concepts • **Software and its engineering → Integrated and visual development environments**.

Keywords end-user programming, software customization, web browser extensions

ACM Reference Format:
Anonymous Author(s). 2020. Customizing Software by Direct Manipulation of Tabular Data. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Many applications don't meet the precise needs of their users. End user customization systems can help improve the situation, by empowering non-programmers to modify their software in the way they would like.

Most end user customization systems offer a simplified version of programming. Some scripting languages [7, 10] have a friendly syntax that resembles natural language. Some visual customization tools eliminate text syntax entirely. Macro recorders [4, 9, 10] remove some of the initial programming burden by letting a user start with concrete demonstrations.

Despite their many differences, these approaches all share something in common: an imperative programming model, with statement sequencing, mutable variables and loops. End users express their ideas in scripts made up of sequences of commands.

We have known for decades about an alternate approach: *direct manipulation* [20], where "visibility of the object of interest" replaces "complex command language syntax". Direct manipulation is the *de facto* standard in GUIs today, but when it comes to customizing software, it is rarely to be found. In this work, we ask: what would it look like to build a software customization interface that relies on direct manipulation? We take inspiration from spreadsheets and visual database query interfaces [1, 5], which have successfully enabled end users to run queries and computations through direct manipulation of data.

In this paper we present a technique called *table-driven customization*, which applies ideas from visual query interfaces in the context of software customization. An application's UI is augmented with a table view, where the user can see and manipulate the application's internal data. Changes in the table view result in immediate corresponding changes to the original user interface of the application, enabling the user to customize an application with live feedback.

We have developed a browser extension called Wildcard which uses web scraping techniques to implement table-driven customization for existing Web applications. In Section 2, we introduce the key ideas of table-driven customization through an example scenario.

In Section 3, we explain the architecture of table-driven customization. We focus on the *table adapter* abstraction, which allows many different types of underlying data to be bidirectionally mapped to a table. We describe several types of table adapters we've built in Wildcard, and also describe future adapters that are supported by the general paradigm.

We have used Wildcard to build real customizations for 11 different websites. In Section 4, we present reflections from this process, outlining the kinds of customizations we were

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

able to build, limitations we encountered, and reflections on the ease of integrating scraping logic with real websites.

In Section 5, we discuss some key themes from our work:

- *Customization by direct manipulation*: End users should be able to customize an application by directly examining and modifying its data, rather than by writing imperative scripts.
- *Wrapping applications for customization*: Typically, tools that don't rely on official extension APIs resort to offering low-level APIs for customization. Instead, we propose a community-maintained library of semantic wrappers around existing applications, enabling end users to work with domain objects rather than low-level representations.

Table-driven customization relates to existing work in many areas. In particular, our goals overlap with many software customization tools, and our methods overlap with direct manipulation interfaces for working with structured data, including visual database query systems and spreadsheets. We explore these connections and more in Section 6.

2 Example Scenario

To illustrate the end user experience of table-driven customization, we consider a real example of using the Wildcard browser extension to customize Hacker News, a tech news aggregator. Figure 1 shows accompanying screenshots.

When the user opens Hacker News in a browser equipped with the Wildcard extension, they see a table at the bottom of the page, listing information about each link on the site: its title, URL, number of upvote points, etc. (Figure 1, Note A) The end user didn't need to do any work to produce this table; Wildcard contains a library of adapters created by programmers for extracting data from websites.

The user's first goal might be to change the ranking of links on the homepage. Hacker News itself uses a ranking algorithm in which the position of an article depends not only on its point count (a measure of popularity), but also on how long it has been on the site. If the user hasn't been checking the site frequently, it's easy to miss a popular article that has fallen down the page. Sorting the page solely by number of points would achieve a more stable ranking, with the most upvoted articles remaining at the top of the list until they are no longer on the home page.

To achieve this sort order, the user simply clicks on the "points" column header in the table. This sorts the table view by points, and the website UI also becomes sorted in the same order (Figure 1, Note B). This sort order persists across page loads, so every time the user loads Hacker News, they will see articles with this stable ordering.

Next, the user might decide to attempt a more substantial customization: to show estimated read times next to each article, as additional context when deciding what to read. The table contains additional columns to the right of the original attributes for each link, where the user can enter spreadsheet-style formulas to compute derived values. The user enters a formula (Figure 1, Note C):

```
=ReadTimeInSeconds(link)
```

This calls a built-in function that takes a URL as an argument and makes a request to a public web service that returns an estimated read time for the contents of that link. The result of evaluating the formula is simply a numerical value in seconds, which is not a particularly legible format. The user enters another formula in the next column (Figure 1, Note D):

```
=Concat(Round(user1/60), "min read")
```

This converts seconds to minutes by dividing by 60 and rounding to the nearest integer, and then concatenates a label to the number. Finally, the user chooses to show the results of this new column in the original page (Figure 1, Note E). Each article on the page now shows an annotation with the estimated read time in minutes.

The user might also manually add notes to the table without using formulas. In this case, the user jots down a few notes about articles they might want to read, and the notes appear in the page as well.

The features presented in this scenario can support a broad range of other customizations. Section 4 explains many other use cases, but first we explain how the system works internally.

3 System architecture

Figure 2 summarizes the overall architecture of table-driven customization, using a simplified version of the Airbnb example above. (*todo: change this figure to hacker news*) In this example, the name of each listing is scraped from the web page DOM, the latitude and longitude of each listing is scraped from AJAX responses, and user annotations are loaded from the brower's local storage.

First, the three different data sources are each bidirectionally mapped to a table interface by a **table adapter**. The table adapter defines how to map a particular type of data to a table, and what effects edits should have on the original data source. In some cases, the mapping logic is straightforward: the local storage adapter stores a table of data, so the mapping to the table abstraction is trivial. In other cases, the mapping is more involved: the DOM scraping adapter implements web scraping logic to produce a table of data from the web page, and turns edits to the table into DOM manipulations like reordering rows of data on the page.

The three separate tables are then combined into a single table for the end user to view and edit. The **query engine** is responsible for creating this combined view, and routing the user's edits back to the individual table adapters. In this example, the query engine has joined the three tables together by a shared ID column, and sorted the result by the name column.

A) The user opens a table view which shows data about each article in the list: its title, link, the number of points and comments, etc.

B) When the user sorts the table by points, the web page becomes sorted in the same order

C) The user enters a formula to fetch estimated read times from an API

D) The user enters a formula to transform the result of the formula into a more readable label

E) The formula results, and manual annotations, are shown in the page next to each article

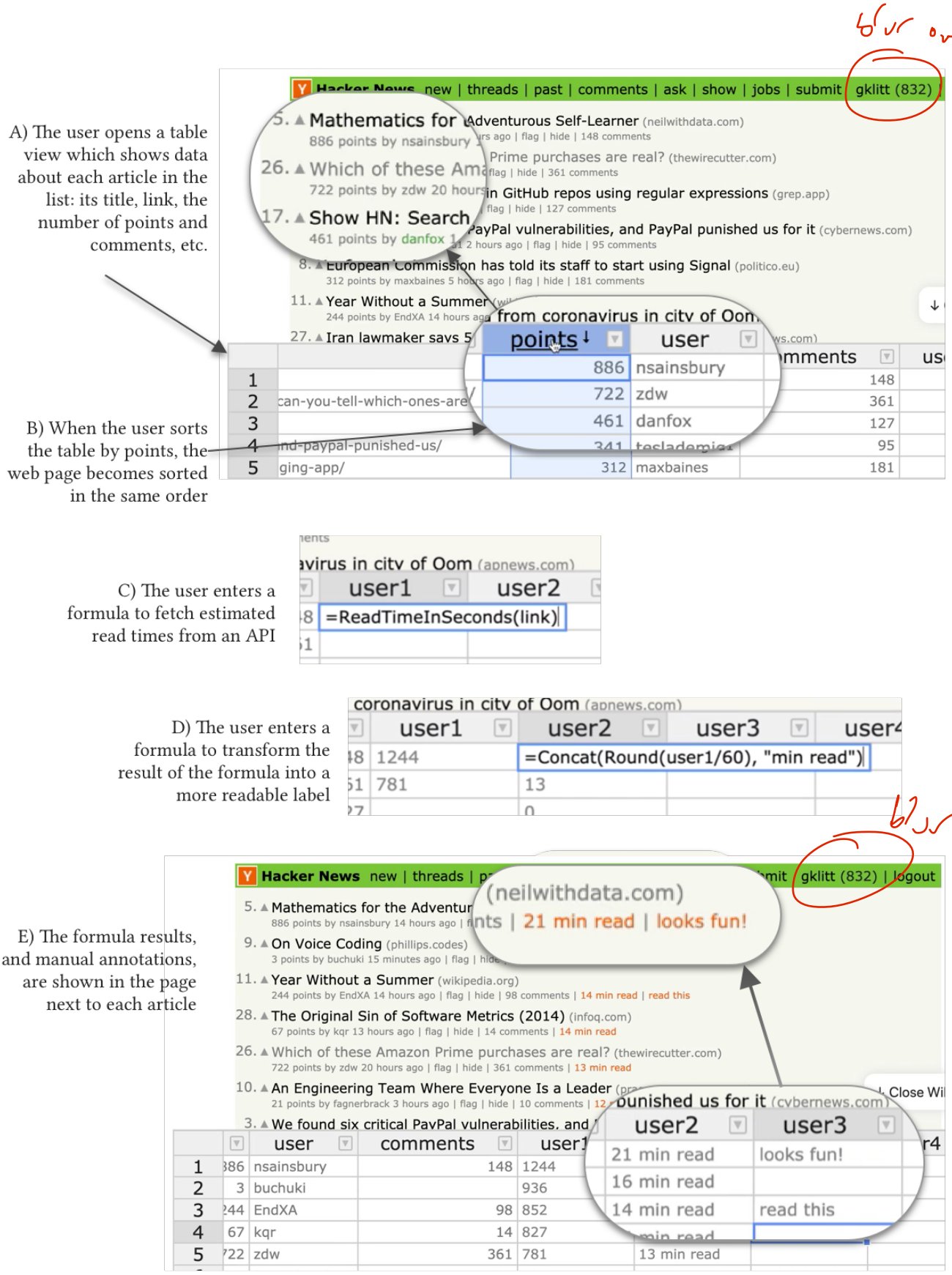

Figure 1. Customizing Hacker News by interacting with a table view

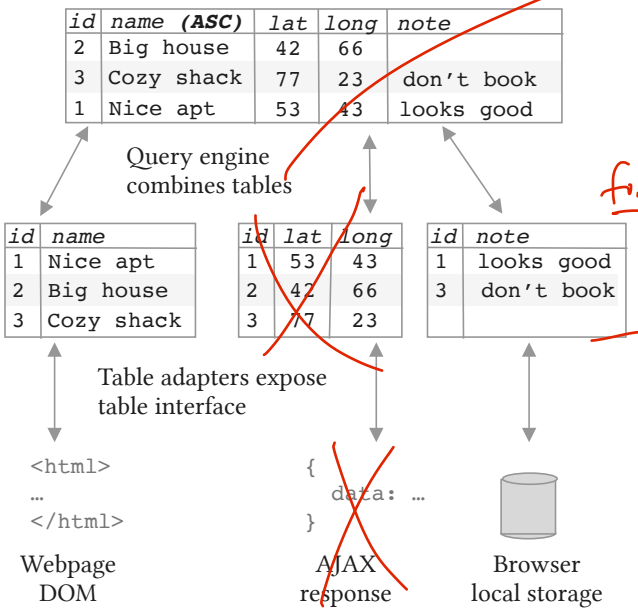

Figure 2. The table adapter architecture

We now examine each component of the system in more detail.

3.1 Table adapters

A key idea in table-driven customization is that a wide variety of data sources can be mapped to the generic abstraction of an editable table. In a relational database, the table matches the underlying storage format, but in table-driven customization, the table is merely an *interface layer*. The data shown in the table is a projection of some underlying state, and edits to the table can have complex effects on the underlying state.

Externally, a table adapter must satisfy an abstract interface. The first two parts of the interface resemble the interface exposed by a table in a relational database:

Returning a table: A table adapter exposes a table of data: an ordered list of records. Each record carries a unique identifier and associates named attributes with values. Tables have a typed schema, so the same attributes are shared across all records. A table adapter can update the contents of a table at any time, in response to changes in the underlying state (e.g., a DOM scraping adapter can update the table when the page body changes). When data changes, the query view is reactively updated in response.

Making edits: The query engine can issue a request to a table adapter to make an edit to a record. The meaning of making an edit can vary depending on the adapter: in the local storage adapter, an annotation may be persisted into local storage; in the DOM scraping adapter, an edit may represent filling in a form field. An adapter can also mark values as read-only if it wouldn't be meaningful to edit them; for example, the DOM scraping adapter typically marks page content as read-only, except for editable form fields.

The query engine also sends additional information about the combined query view to each table adapter. These functions are currently used to provide the DOM scraping adapter with sufficient information to manipulate the original application UI as the user manipulates the table view.

Sorting/filtering: When the user sorts or filters the query view, an ordered list of visible IDs is sent to each table adapter. The DOM scraping adapter uses this information to modify the list of visible rows in the web page.

Information from other tables: The query engine sends each table adapter the entire combined table of data being rendered to the user. The DOM scraping adapter uses this to add annotations to the page by looking for additional data columns joined to scraped rows, and rendering them in the page. (*todo: explain joined*)

Currently selected record: The query engine notifies each table adapter about which record is currently selected by the user in the table UI. The DOM scraping adapter uses this information to highlight the row in the page that corresponds to the selected record in the table. (*todo: clarify*)

3.1.1 Types of table adapters

Here we describe in more detail the table adapters that we have implemented in Wildcard to power the customizations shown in Section 2.

DOM scraping adapters are the essential component that enables Wildcard to interface with an existing website UI. In addition to the standard web scraping problem of extracting a table of data from the DOM, a scraping adapter must also manipulate the DOM to reorder rows, edit form entries, and inject annotations as the table is edited.

Because each website has unique content, we rely on programmers to create a DOM scraping adapter for each individual website to make it available for customization in Wildcard. To make this approach viable, we have built a library of functions that make it easy to implement a scraping adapter, requiring the programmer only to implement the minimal site-specific parts. The programmer writes a single Javascript function which can use standard scraping techniques like CSS selectors and DOM APIs to extract relevant elements from the page. The library functions then wrap this function to implement the rest of the needed functionality; for example, when the table is sorted, we find the DOM elements corresponding to the rows in the table, remove them from the page, and then reinsert them in the new order.

An **AJAX scraping adapter** intercepts AJAX requests made by a web page, and extracts information from those requests. When available, this tends to be a helpful technique because the data is already in a structured form, and often includes information not shown in the UI. As with DOM scraping adapters, we have made it easy for programmers to create site-specific AJAX scraping adapters. A programmer

writes a function that specifies how to extract data from an AJAX request, and the framework handles the details of intercepting requests and calling the programmer-defined function.[1]

The **local storage adapter** simply stores a table of data in the browser. As shown in the example use cases, this can be useful for persisting annotations in a private way, without uploading them to a web service.

3.1.2 Future Adapters

We have designed the table adapter API to be general enough to support other types of useful adapters in the future. Here are two such possibilities:

Integrated website adapters: A key benefit of the table adapter abstraction is that Wildcard is not coupled to web scraping as the only means for integrating with existing sites, but can also accommodate first party developers adding support directly into their own websites. An "integrated website adapter" installed by the developer could directly accesses the internal state of the application, providing the same functionality as a DOM scraping adapter but in a more robust way.

With the advent of rich frontend web frameworks, structured application state is now often available in the web client. We suspect it is possible to create plugins for frontend frameworks that expose this state to Wildcard with only minimal effort from the application developers. To test this hypothesis, we have created an early prototype of a Wildcard adapter for the Redux state management library, and used it with the TodoMVC todo list application. To install the adapter, the programmer only needs to make a few small changes to their app:

Exposing a table: In Redux, app developers already represent the state of a user interface as a single object. To expose a table of data, the developer simply writes a function that maps the state object to a table. *Editing data values*: The developer defines how edits to records in the table should map to edits in the state object. *Sorting/filtering the UI*: The developer inserts an additional data processing step before the state object is rendered in the UI, so that Wildcard can apply sorting and filtering. *Annotations*: The developer adds additional React components managed by Wildcard into their UI tree, which render additional data as annotations.

In certain types of software, customizability is often touted as a key selling point. An integrated website adapter would provide a way for developers to integrate with an existing ecosystem of formulas and customization tools, without needing to build all that functionality from scratch.

[1] So far we have only implemented AJAX scraping in the Firefox version of Wildcard, since Firefox has convenient APIs for intercepting requests. It appears possible to implement in Chrome as well, but we have not finished our implementation.

Shared storage adapter: It would be useful to share user annotations among users and across devices—for example, collaboratively taking notes with friends on options for places to stay on Airbnb. The existing Local Storage Adapter could be extended to share live synchronized data with other users. This could be achieved through a centralized web server, or through P2P connections that might provide stronger privacy guarantees.

3.2 Query engine

The query engine is responsible for coordinating across multiple table adapters. It joins data across multiple tables and creates a single result table which is shown to the user through the editor. It also handles all user interactions and routes appropriate messages to each table adapter.

Queries are processed in three steps. First, the query invokes a primary DOM scraping table adapter that associates records in the result with elements in the application's user interface. At minimum, the primary table adapter needs to return record IDs and have the ability to manipulate the application's UI. It can also optionally return data about each record. Next, additional tables (AJAX data, local storage data) are left-joined by ID. Finally, the result table can be sorted and filtered by any column.

This query model can be viewed as a tiny (and rather constrained) subset of the SQL query model. Despite its simplicity, this simple model has proven sufficient for meeting the needs of customization in practice, and minimizes the complexity of supporting more general and arbitrary queries. But because it fits into the general paradigm of relational queries, it could theoretically be extended to support a wider range of queries.

The query engine is also responsible for executing formulas. We have built a small formula language resembling the ones used in visual query tools like SIEUFERD. As in those tools, and unlike in spreadsheets, formulas automatically apply across an entire column of data, and reference other column names instead of values on specific rows. For example, the user would write `=price * 2` to create a column with a value derived from a price attribute. This is more convenient than needing to copy-paste a formula across an entire column as in spreadsheets, and none of our customization use cases have uncovered a need for writing different formulas on different rows of the table.

3.3 Table editor

In Wildcard we provide a basic table editor as the user interface on top of the query engine. It is built with the Handsontable Javascript library, which provides UI elements for viewing and editing a table, as well as basic query operations like sorting and filtering.

todo: other table instruments

Website	Description	LOC	Example customizations
Airbnb	Travel	73	Add Walkability Scores to listings. Sort listings by price.
Amazon	Online shopping	99	Sort used book sellers by total price, including delivery fees
Blogger	Blogging	36	Use alternate text editor to edit blog posts
Expedia	Travel	41	Use alternate datepicker to enter travel dates
Flux	Data portal	67	Use Wildcard as a faster table editor for editing lab results
Github	Code repository	62	Sort a user's code repositories by stars to find popular work
Hacker News	News	69	Add read times to links. Filter out links that have been read.
Instacart	Grocery delivery	48	Sort groceries by price and category. Take notes on items.
Uber Eats	Food delivery	117	Sort/filter restaurants by estimated delivery ETA and price.
Weather.com	Weather	51	Sort/filter hourly weather to find nice times of day.
Youtube	Videos	80	Sort/filter videos by length, to find short videos to watch.

Table 1. A list of table-driven customizations that we have implemented using Wildcard.

4 Evaluation

To evaluate table-driven customization in practice, we built the Wildcard browser extension, which implements table-driven customization in the context of existing websites. It is implemented in Typescript, and works across three major browsers: Chrome, Firefox, and Edge.

We have used Wildcard to customize 11 websites, including transactional sites like Amazon and Uber Eats, and media consumption sites like Hacker News and Youtube. Table 1 summarizes these customizations, showing the number of lines of code in the adapter configuration for each site.

So far, most usage of Wildcard has come from members of the project team. Here we offer our reflections on using the system, focused on two key questions:

- How broad is the range of possible customizations in this paradigm?
- How feasible is it to build DOM scraping adapters for real websites?

4.1 Range of customizations

We have found that table-driven customization can serve a broad range of useful purposes. Here we expand on some archetypal examples that illuminate various aspects of using the system in practice.

4.1.1 Sorting/filtering

It might seem that most websites already have adequate sorting and filtering functionality, but we have found it surprisingly helpful to add new sorting/filtering functionality to websites using Wildcard.

Sometimes, websites have opaque algorithms which they use to rank results, which presumably maximize profit but don't offer much control to the user. For example, Airbnb doesn't allow users to sort listings by price, and Youtube doesn't allow users to sort the homepage by watch time. Wildcard enables users to take back some control over this process.

In other cases we have found that a lack of sorting options seems to be intentional, and simply represents a missing feature. For example, the Instacart grocery delivery service doesn't allow users to sort their grocery cart by price or category. Our guess is that they simply haven't gotten around to implementing the feature yet.

In the current implementation of Wildcard, users can only sort and filter entries that are shown on the current page, which means that users are not entirely liberated from the suggestions of the opaque algorithm. We could work around this restriction in the future by fetching content across multiple pages, but we've also found that sorting/filtering a single page of a paginated list is often an acceptable outcome (and sometimes even a preferable one). For example, it's more useful to sort (by length, say) 30 recommended Youtube videos than to try to sort all videos on Youtube, or to sort (by read time, say) the articles on the Hacker News front page than to sort all non-archived articles.

4.1.2 Annotating

Many other web annotation systems focus on annotating text or arbitrary webpage content, but Wildcard limits annotations to structured objects extracted by an adapter, resulting in quite a different set of use cases.

Annotating with Wildcard has proven most useful when taking notes on a list of possible options (e.g., evaluating possible Airbnb locations to rent). We have also used it with Instacart's online grocery cart, for jotting down notes like "should we get more milk?"

4.1.3 Formulas

Formulas are the most powerful part of the Wildcard system. So far, our language supports only a small number of predefined functions. Adding more will allow a very broad range of useful computations, as shown by the success of spreadsheets.

Formulas are especially useful for fetching data from Web APIs. We've used them to augment Airbnb listings with walkability scores, and to augment Hacker News articles with estimated read times. One challenge of the current language design is that supporting a new web API requires writing conventional code to add a new function to the language, because web APIs typically return complex JSON data structures that can't be easily displayed in a single table cell. Adding functions for handling JSON data to the formula language might make it possible to access new APIs without requiring an entire new function in the formula system.

We have also found instances where simple data manipulation is useful: adding together different subtotals into a total price, or manipulating the results of a formula expression with arithmetic or concatenating string labels, as shown in the example in Section 2. (*todo: this makes sense iff we switch the examples sectino to HN*)

4.1.4 Cell editors

(*note: written with the assumption that Expedia gets cut from the Examples section*)

Cell editors are UI widgets that expose a custom editing UI for a single cell of the table view. A programmer building a cell editor need only integrate it with the table viewer; propagating values into the website UI is handled by the site-specific DOM adapter. As a result, creating a new cell editor can be as easy as just writing some glue code between the table editor and an existing UI widget library.

An important benefit of cell editors is that they allow users to incorporate their personal information within the web UI but without uploading their personal data to the web server. To explore this idea, we created a cell editor based on the FullCalendar Javascript plugin, which can load data from a Google Calendar. This makes it convenient to enter dates into a website based on the user's personal calendar information.

Another benefit of cell editors is that a user can choose their preferred widget for editing some type of information. We built a cell editor based on the CKEditor rich text editor, and demonstrated its use by integrating it with Google's Blogger website and using it to edit blog posts, in place of Blogger's built-in editor.

4.1.5 Limitations

There are many useful customizations that are not currently possible in the table-driven customization paradigm. Wildcard can only make customizations that use the available data exposed in the table. UI modifications are limited in scope; deleting arbitrary buttons isn't possible, for example. At one point, we wanted to build an automation to repeatedly load a grocery delivery website to check for open delivery slots, but it wasn't clear how to achieve this. Some of these limitations are specific to the current implementation of the Wildcard extension, but some are more fundamental to the entire idea of table-driven customization.

We consider this an acceptable outcome; our goal is to make many useful customizations available to end users with a low threshold, and not to span all possible customizations. Section 5.1 discusses this point further.

One benefit of Wildcard is predictability: once we built a site adapter for a website, it was generally obvious to us what types of customizations were and weren't possible, and the UI guided us towards building customizations that matched the system's model. If data is not available in the table, for example, then it's abundantly clear that customizations relying on that data will not be expressible.

4.2 Viability of DOM scraping

Our second evaluation area relates less to the conceptual approach of table-driven customization, and more to the specific implementation of customizing existing web applications. In order for third-party customization through Wildcard to succeed, it is important that creating usable adapters for existing websites takes minimal effort.

Most of our DOM scraping adapters were created by members of our team. However, an external developer unaffiliated with the project contributed one adapter, designed to sort the Github page listing a user's repositories. They described the experience as "very straightforward."

The adapters for our test sites ranged from 36 to 117 lines of code, averaging 68 lines; Table 1 shows the full lines of code for each adapter.

Some of the challenges of writing a DOM scraping adapter are the same ones as with writing normal web scraping code, but the more interactive nature of Wildcard introduces additional challenges. rve changes, but it may prove challenging to observe changes on some sites only through the DOM. One challenge is triggering updates to the spreadsheet data in response to UI changes that happen after initial page load. Site adapters are responsible for recognizing these changes by observing the DOM. So far, we have been able to use event listeners and the MutationObserver API to successfully obse

Another challenge is persisting updates to the DOM—some websites use virtual DOM frameworks that can occasionally overwrite changes made by Wildcard.

So far, we've managed to work around these issues for all the websites we've tried, but we don't claim that any website can be customized through DOM scraping. As web frontend code gets increasingly complex (and starts to move beyond the DOM to other technologies like Shadow DOM or even WebGL for rendering), it may become increasingly difficult to customize websites from the outside without first-party support.

5 Key themes

5.1 Customization by direct manipulation

Hutchins, Hollan and Norman [13] define a direct manipulation interface as one that uses a model-world metaphor rather than a conversation metaphor. Instead of presenting an "assumed" but not directly visible world that the user converses with, "the world is explicitly represented" and the user can "[act] upon the objects of the task domain themselves."

Although most GUIs today employ direct manipulation, software customization tools typically use an imperative programming model, which implements the conversational metaphor rather than direct manipulation. Here, for example, is how a user retrieves a list of of calendar names from the Calendar application in Applescript [10], the scripting language for customizing Mac OS applications:

```
tell application "Calendar"
  name of calendars
end tell
```

Some customization environments like Mac Automator and Zapier forego textual syntax and let the user connect programs and construct automations by dragging and dropping icons representing commands. These environments still do not constitute direct manipulation, though: the objects being manipulated are in the domain of programming, not in the domain of the task at hand. Imperative programming is a reasonable choice as the model for building customizations. Turing-complete programming provides a high ceiling for possible customizations, and a sequence of commands is a natural fit for automations that simulate a series of steps taken by the user.

Imperative programming is a reasonable choice as the model for building customizations. Turing-complete programming provides a high ceiling for possible customizations, and a sequence of commands is a natural fit for automations which simulate a series of steps taken by the user.

There is, however, a serious drawback to this approach. MacLean et al [18] describe an ideal for user-tailorable systems: a "gentle slope" from using to customizing, where small incremental increases in skill lead to corresponding increments of customization power. Requiring users wanting to customize their applications to learn programming creates an abrupt "cliff," exacting a significant investment in learning even to implement the simplest customizations. Another goal of MacLean et al is to make it "as easy to change the environment as it is to use it"—at least for some subset of changes. But in scripting languages, even user-friendly ones like Applescript or Chickenfoot, the experience of customization does not remotely resemble the experience of use, so these systems can't meet this goal.

With table-driven customization we aim to provide a gentler slope, by using direct manipulation for software customization. The data shown in the table view is the domain data from the original application. The user makes changes to the data by selecting areas of interest in the table, e.g. sorting/filtering by clicking the relevant column header, or adding annotation by clicking on the relevant row. These interactions are common in GUI applications, and Wildcard therefore meets MacLean et al's goal: some one-click customizations are as easy as using the original application.

One aspect of directness that we have chosen not to maintain in Wildcard is enabling customization in the context of the original user interface, as explored by tools like Scotty [11]. We have found that augmenting the original UI with a distinct, additional representation provides a consistent experience across all applications, and clearly shows the user what structured data is available to work with.

Ainsworth et al provide a helpful taxonomy of the value of multiple representations [2]. In their terms, Wildcard plays a complementary role by supporting a different set of tasks from the original application, while displaying shared information. Wildcard may also help construct deeper understanding by subtraction. By stripping away details and only showing the essential data in an interface, Wildcard helps users think of new ways of using that data, outside the specific constraints of the original application.

It is important to note that there are many specific customizations that can be achieved in systems like Chickenfoot [7] that cannot be reproduced in Wildcard. We consider this an acceptable tradeoff in exchange for a gentler slope in customization, and we show in Section 2 and Section 4 that our model can still implement many useful customizations in practice. Also, there is sometimes a way to reframe an imperative script in terms of our direct manipulation model; for example, a script that iterates through rows in a page adding some additional information can be reproduced using a formula in Wildcard.

5.2 Semantic wrappers

Software customization tools typically fall into one of two categories: ad hoc, or semantic.

Ad hoc customization tools enable customization without using official extension APIs, enabling a broader range of customizations on top of more applications. For example, web browser extensions have demonstrated the utility of customizing websites through manipulating the DOM, without needing explicit extension APIs to be built in. However, ad hoc customization comes with a corresponding cost: these tools can typically only operate at a low level of abstraction, e.g. manipulating user interface elements. This makes it harder for end users to write scripts, and makes the resulting scripts more brittle.

Anticipated customization tools, in contrast, use explicit extension APIs provided by the application developer. Examples of this include accessing a backend web API, or writing a customization in Applescript that uses an application-specific API. The main benefit is that this allows the extension author to work with meaningful concepts in the

application domain—"create a new calendar event" rather than "click the button that says new event."—which makes customizations easier to build and more robust. However, this style limits the range of extensions that can be built to those that the official plugin API supports.

todo: copy over more content from gdoc

With Wildcard, we use a hybrid approach that aims to provide the best of both worlds. Programmers implement an API wrapper that is internally implemented as an ad hoc customization, but externally provides a high-level interface to the application, abstracting away the details of the user interface. These wrappers are added to a shared repository, available to all users of the system. When an end user is using a site that already has an adapter, they get a semantic customization experience that avoids low-level details.

One way to view this approach is as introducing a new abstraction barrier into third-party extension. Typically, a third party customization script combines two responsibilities: 1) mapping the low-level details of a user interface to semantic constructs (e.g., using CSS selectors to find certain page elements), and 2) handling the actual logic of the specific customization. Even though the mapping logic is often more generic than the specific customization, the intertwining of these two responsibilities in a single script makes it very difficult to share the mapping logic across scripts.

With Wildcard we propose a decoupling of these two layers: a repository of shared wrappers maintained by programmers, and a separate repository of specific customizations built on top of these wrappers. This general architecture has been successfully demonstrated by projects like Gmail.js, an open source project that creates a convenient API for browser extensions to interface with the Gmail web email client.

(*todo: mention automated wrapper induction here or somewhere else?*)

6 Related Work

This paper extends work reported in a workshop paper by Litt and Jackson [16] which presented an early prototype version of Wildcard. We have substantially extended their work in this paper by creating the table adapter abstraction and reimplementing the system around that abstraction, evaluating the system more fully on many more websites, and by characterizing the design of the system in more detail.

Table-driven customization relates to two broad areas of related work. Our problem statement is related to software customization tools, and our solution approach is related to spreadsheets and other direct manipulation interfaces.

6.1 Customization tools

Table-driven customization is most closely related to other tools that aim to empower end users to customize software without traditional coding.

This lineage goes back at least to the Buttons system by MacLean et al [18], where Xerox Lisp users could share buttons that performed various "tailoring" actions on the system. The authors proposed the "gentle slope" idea which has greatly influenced our approach to table-driven customization (as discussed in Section 5.1). The authors also point out the importance of a "tailoring culture" where people with different skillsets collaborate to produce useful customizations; in their system, Lisp programmers create buttons that others can use, modify, and rearrange. This division of labor corresponds to our idea of semantic wrappers, where end user customization is supported by programmer-created building blocks.

Some recent web customization tools aim to enable end users to modify web interfaces without programming.

Sifter [14] enables end users to sort and filter lists of data obtained by web scraping, much like Wildcard's sorting features. The main difference between the systems is that table-driven customization has many other use cases besides sorting and filtering. Also, Sifter involves end users in a semi-automated data extraction process, rather than having programmers create wrappers. This provides coverage of more websites, but at the expense of complicating the end user experience. We might explore such techniques in Wildcard in the future, but we think that it's valuable for end users to have a customization experience decoupled from the challenge of web scraping the underlying data. Sifter implements scraping across multiple pages, a valuable feature for sorting and filtering that isn't present in Wildcard.

Thresher [12] helps end users create wrappers that map website content to Semantic Web schemas, making customizations available using the schema information. Wildcard shares the general idea of wrappers with Thresher, but maps to a single generic data type, rather than more specific schemas, increasing the range of websites and data that Wildcard is able to support.

There are many software customization tools that offer simplified forms of programming for end users. Chickenfoot [7] and Coscripter [15] offer user friendly syntax for writing web automation scripts; Applescript [10] has a similar goal for desktop customization. There are visual programming environments for customization that don't involve writing any text: Automator for Mac and Shortcuts for iOS are modern options for customizing Apple products, and Zapier enables users to connect different web applications together visually. As mentioned previously, these tools all require writing imperative programs in some form, in contrast to the more declarative and direct approach of table-driven customization.

6.2 Direct manipulation programming interfaces.

Another relevant area involves direct manipulation interfaces for interacting with structured data. We take inspiration from these tools in our work, but apply them in a different

domain: customizing existing software applications, rather than interacting with databases or constructing software from scratch.

The most closely related work is in systems that offer spreadsheet-like querying of relational data, as proposed by Liu and Jagadish [17]. SIEUFERD by Bakke and Karger [5] is an example; their paper presents a survey of other similar tools. Our work is particularly influenced by the authors' observation that direct manipulation requires that the user manipulate the results of a database query rather than the query itself, and that the user must see intermediate results at every step of constructing the query. SIEUFERD's interface supports a far wider range of queries than Wildcard, but the basic ideas of the user interface are similar.

Airtable is another example of a modern commercial product that offers spreadsheet-like interaction with a relational database.

Our work is also inspired by the many projects that have explored using spreadsheets as a foundation for building software applications, including Object Spreadsheets [19], Quilt [6], Gneiss [8], Marmite [21], Glide. These projects share the idea that a spreadsheet is a convenient interface for editing the data underlying a GUI application. We share that idea, but apply it to software customization, rather than building software from scratch.

Another system in this space is ScrAPIr, by Alrashed et al. [3], which enables end users to access backend web APIs without programming. ScrAPIr shares our high level goal of end user empowerment, as well as the idea of wrappers, by creating a shared library of wrappers around existing web APIs. Unlike Wildcard, however, ScrAPIr targets explicit APIs exposed by developers. It also focused on backend services and doesn't aim to extend the frontend interfaces of web applications.

7 Conclusion and Future Work

In this paper, we have presented table-driven customization, a new paradigm for customizing software by direct manipulation of the underlying structured data. We have demonstrated the paradigm using a browser extension, and have used it to create useful customizations for a variety of websites.

Our primary goal for future work is to evaluate the system with a broader group of users. What barriers do end users face when using the system? What types of customizations do they choose to create? What formulas prove most helpful?

Another area for future work is exploring whether it is possible to express a broader range of customizations by extending our table-editing paradigm. Are there ways to offer an increase in power and functional complexity, while retaining a programming model that is simpler for end users than conventional coding?

As computing plays an ever greater role in our lives, it is increasingly important that end users have agency over the behavior of their software, rather than having every detail be dictated by companies whose incentives are not always aligned with the user's interests. We hope that table-driven customization can serve as one point, and maybe a guidepost, on our path from normal use to deep modification, in support of a more adaptable experience for all computer users.

References

[1] 2020. Airtable. https://airtable.com.

[2] Shaaron Ainsworth. 1999. The Functions of Multiple Representations. *Computers & Education* 33, 2-3 (Sept. 1999), 131–152. https://doi.org/10.1016/S0360-1315(99)00029-9

[3] Tarfah Alrashed, Jumana Almahmoud, Amy X. Zhang, and David R. Karger. 2020. ScrAPIr: Making Web Data APIs Accessible to End Users. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, Honolulu, HI, USA, 1–12. https://doi.org/10.1145/3313831.3376691

[4] Vinod Anupam, Juliana Freire, Bharat Kumar, and Daniel Lieuwen. 2000. Automating Web Navigation with the WebVCR. *Computer Networks* 33, 1 (June 2000), 503–517. https://doi.org/10.1016/S1389-1286(00)00073-6

[5] Eirik Bakke and David R. Karger. 2016. Expressive Query Construction through Direct Manipulation of Nested Relational Results. In *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*. ACM Press, San Francisco, California, USA, 1377–1392. https://doi.org/10.1145/2882903.2915210

[6] Edward Benson, Amy X. Zhang, and David R. Karger. 2014. Spreadsheet Driven Web Applications. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology - UIST '14*. ACM Press, Honolulu, Hawaii, USA, 97–106. https://doi.org/10.1145/2642918.2647387

[7] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. 2005. Automation and Customization of Rendered Web Pages. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology - UIST '05*. ACM Press, Seattle, WA, USA, 163. https://doi.org/10.1145/1095034.1095062

[8] Kerry Shih-Ping Chang and Brad A. Myers. 2014. Creating Interactive Web Data Applications with Spreadsheets. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology - UIST '14*. ACM Press, Honolulu, Hawaii, USA, 87–96. https://doi.org/10.1145/2642918.2647371

[9] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *The 31st Annual ACM Symposium on User Interface Software and Technology - UIST '18*. ACM Press, Berlin, Germany, 963–975. https://doi.org/10.1145/3242587.3242661

[10] William R. Cook. 2007. AppleScript. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages - HOPL III*. ACM Press, San Diego, California, 1–1–1–21. https://doi.org/10.1145/1238844.1238845

[11] 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 Press, Santa Barbara, California, USA, 225. https://doi.org/10.1145/2047196.2047226

[12] Andrew Hogue and David Karger. 2005. Thresher: Automating the Unwrapping of Semantic Content from the World Wide Web. In *Proceedings of the 14th International Conference on World Wide Web - WWW '05*. ACM Press, Chiba, Japan, 86. https://doi.org/10.1145/1060745.1060762

[13] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1985. Direct Manipulation Interfaces. (1985), 28.

[14] David F. Huynh, Robert C. Miller, and David R. Karger. 2006. Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology - UIST '06*. ACM Press, Montreux, Switzerland, 125. https://doi.org/10.1145/1166253.1166274

[15] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, Florence, Italy, 1719–1728. https://doi.org/10.1145/1357054.1357323

[16] Geoffrey Litt and Daniel Jackson. 2020. Wildcard: Spreadsheet-Driven Customization of Web Applications. In *Companion Proceedings of the 4th In- Ternational Conference on the Art, Science, and Engineering of Programming*. Association for Computing Machinery, Porto, Portugal., 10. https://doi.org/10.1145/3397537.3397541

[17] Bin Liu and H. V. Jagadish. 2009. A Spreadsheet Algebra for a Direct Data Manipulation Query Interface. In *2009 IEEE 25th International Conference on Data Engineering*. 417–428. https://doi.org/10.1109/ICDE.2009.34

[18] 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 Empowering People - CHI '90*. ACM Press, Seattle, Washington, United States, 175–182. https://doi.org/10.1145/97243.97271

[19] Matt McCutchen, Shachar Itzhaky, and Daniel Jackson. 2016. Object Spreadsheets: A New Computational Model for End-User Development of Data-Centric Web Applications. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2016*. ACM Press, Amsterdam, Netherlands, 112–127. https://doi.org/10.1145/2986012.2986018

[20] B. Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16, 8 (Aug. 1983), 57–69. https://doi.org/10.1109/MC.1983.1654471

[21] Jeffrey Wong and Jason I. Hong. 2007. Making Mashups with Marmite: Towards End-User Programming for the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '07*. ACM Press, San Jose, California, USA, 1435–1444. https://doi.org/10.1145/1240624.1240842