Working Paper No. 20
As of 7/31/2019

Universal Access to Application-building Tools: How It Can Happen

Overview

This research seeks to

> ***reframe*** *the way the brain-body relates to building software*
>
> ***from: a symbolic activity***
>
> ***to: a manual activity***.

Two kinds of lessons have come from this approach. The *technology* lesson occurred over years of experimentation, and the *practice* lesson has occurred recently.

To make this paper more digestible I'm presenting the practice lessons first. The technology lessons will follow for those who want to understand in detail what they have seen.

The *practice lesson* has two parts:

1. It introduces a model for application sessions that is static and nonprocedural.
2. It discusses the implications of this model on the development process.

The technology lesson also has two parts, which you will see heavily applied:

1. A set of User-Experience Design Principles called the Humane Dozen
2. A Pattern called DOer, SHOWer.

(TK)

Working Paper No. 20

As of 7/31/2019

Table of Contents

The Implications of This Research on Practice

For manipulation of domain-specific data, writing code is still necessary.

For construction of applications from lower-order parts, writing code is becoming <u>increasingly unnecessary</u> and, in some simple cases, is already so.

What is new is (1)a collaborative application architecture that gives non-programmers a key role in building and defining the character of an application, and (2)a development platform that supports an iterative workflow between these two groups.

Below is a model for the structure of an application session; this paper is about how this model can radically affect the development process.

Starting Point: The Functional-path Session Model

When IBM introduced the 3270 computer terminal in the 1970s, introduction of data fields instead of characters as the elements of communication between the computer and the terminal was viewed as an engineering optimization. What this conceptual shift does for us now is enable us to think of an application session in terms of static connections rather than procedural operations.

Here is a statement of the model:

Every item of data on an application's user interface is at the end of a path that originates in a business object and that includes zero or more functional transforms.

This enables us to draw lines from display items back to business objects, with some possible functional transformations along the way. A tool concept comes directly from this. We're going to make this concrete with the walkthroughs below, in which we start with a wireframe description of a use case and end up with a functioning wired application.[1]

[1] See http://melconway.com/talks/2019_consumer_apps/12.html and the following slide. I am considering creating a video that shows the entire process in one place.

The Research Mission

Viewpoint

The primary focus of my research has been to *maximize the number of people who have the ability to build useful information tools*.

Maximum access suggests finding a maximum set of skills common to all normal humans, and then designing to that skill set. ***This skill set is discoverable; just observe what Nature drives every human to practice relentlessly in the first few years of his or her life: hand-eye-brain coordination.***

My conclusion is that to maximize access ***we need to reframe the way the brain-body relates to building software***
from: a symbolic activity
to: a manual activity.

Consider this thought experiment. Imagine that you are a potter in the bowl-making business, but the individual potter's wheel technology does not exist. Instead what you have to do to make a bowl is write a bowl-making script in a text editor, email it to a bowl-making factory, and then wait for the bowl to be shipped back.

That is pretty much how we built software when I got started in the 1950s. Things are a little better now, but we're still a lot closer to the text-editor end of the hands-on spectrum than to the potter's wheel end. My goal, simply stated, is to move along that spectrum toward the potter's wheel.

Working Paper No. 20

As of 7/31/2019

An Extended Walk-through

From Wireframe to Functional Paths

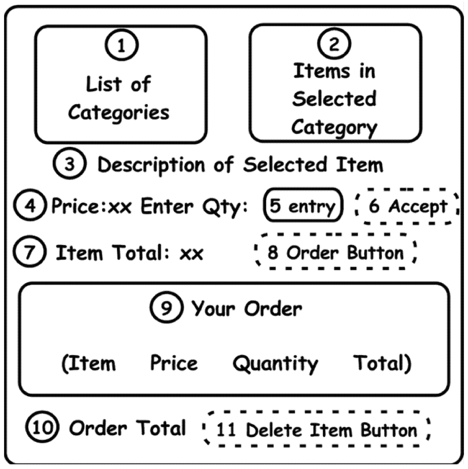
We're going to use the Functional-path model to go from a sketched wireframe to a running program in two steps:

1. Use common sense to draw the functional paths from what we know about the wireframe.
2. Make a direct conversion from the drawing to a wiring diagram that executes in the wiring tool.

The Restaurant Use Case

This walk-through illustrates a use case in which a waiter takes a customer order in a restaurant. The menu items are in a relational table called "Item". Every item is in one of six categories, and there is a "Category" table for navigating to items.

Here is a sketch of the user interface. The items are numbered for the purpose of the following usage scenario description.



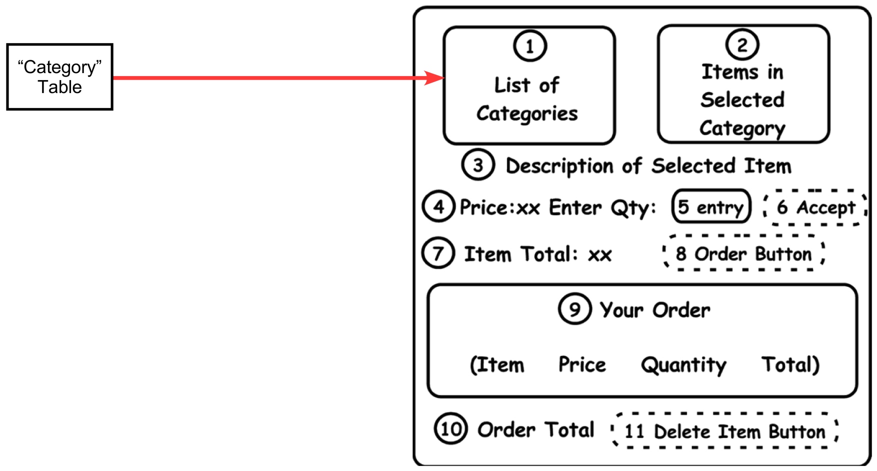
Select a category in (1). This displays all the items in the selected category in (2). Selecting an item shows a brief description in (3) and its unit price in (4). (5) is an entry field in which you enter the number of items being ordered. Clicking the Accept button (6) causes the "extended price" (total for the number of items) to be displayed in (7). If you want to add that number of items to the shopping cart, press the Order button (8) and it shows up in (9). At the same time the order total amount is recomputed and displayed in (10). Button 11 (not used here) will delete a selected item from the shopping cart.

Working Paper No. 20

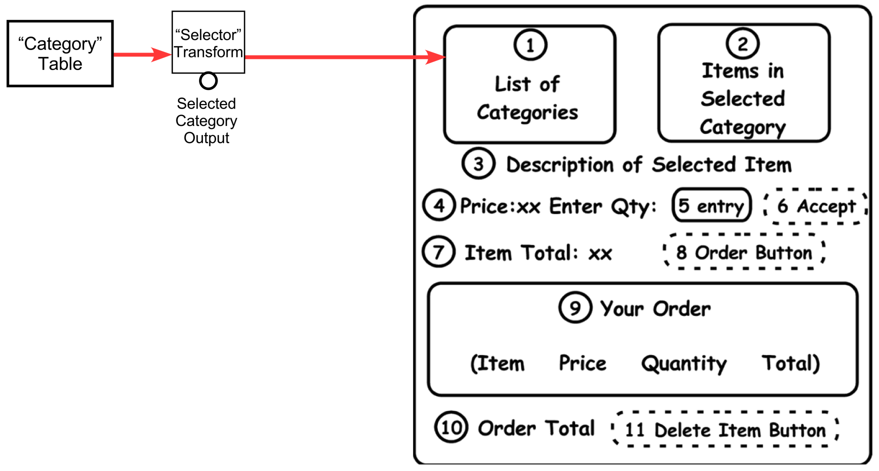
As of 7/31/2019

Connecting Paths to the Wireframe

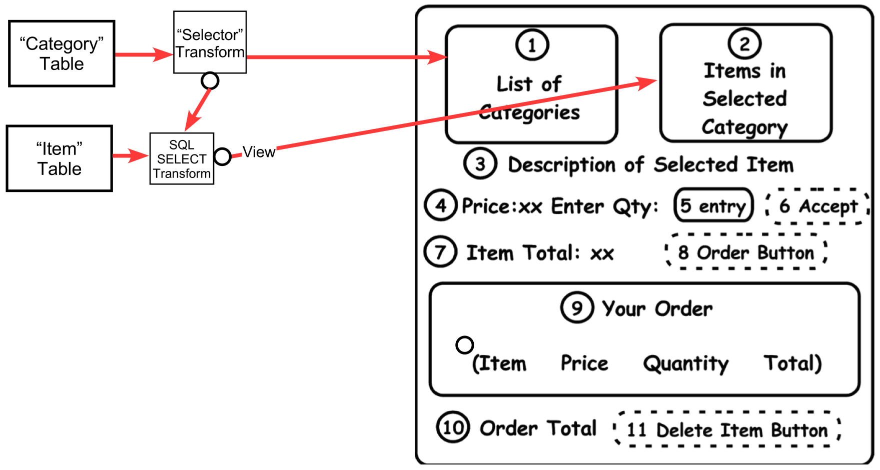
We will be using red lines to indicate the paths from data to display. For now, look only at connections; behavior will come later. Here is the path from the Category table to the Category list box:



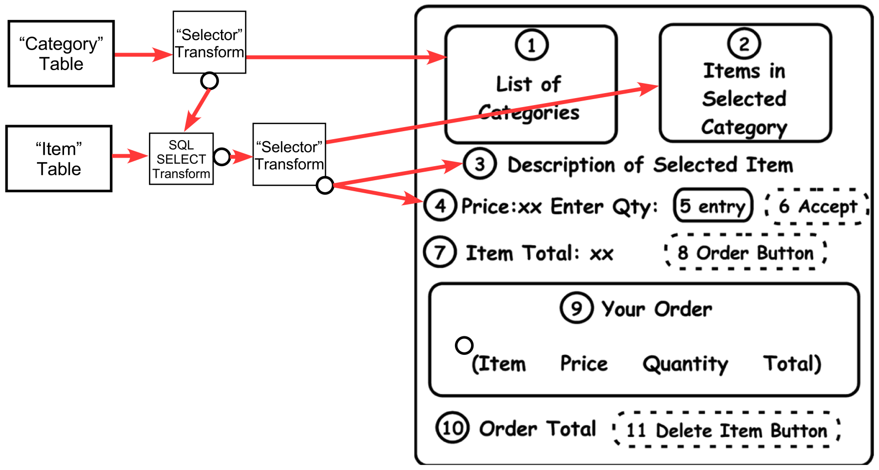
Now, if we were coding this with a relational database we would next perform an SQL SELECT on the Item table, selecting only the items in the chosen category. We're going to do that, in the functional path language. First we need a "Selector" transform box that is tightly coupled to list box (1) and that is always sourcing the chosen category. (If your mind is inclined to electrical circuitry as mine is, think of it as a rotary switch.) This output will come out a "connector" shown as a circle (we're beginning to sneak up on the wiring model). Here it is:



Now we'll make the path from the selected category to the SQL SELECT transform box, and from the resulting view to the Items list box (2):



Another Selector transform box inserted in this view path sources the selected Item, and two column values of the selected Item: Description and Unit Price. (The transforms that select those two columns are discussed below as Self-revealing Parameters.)

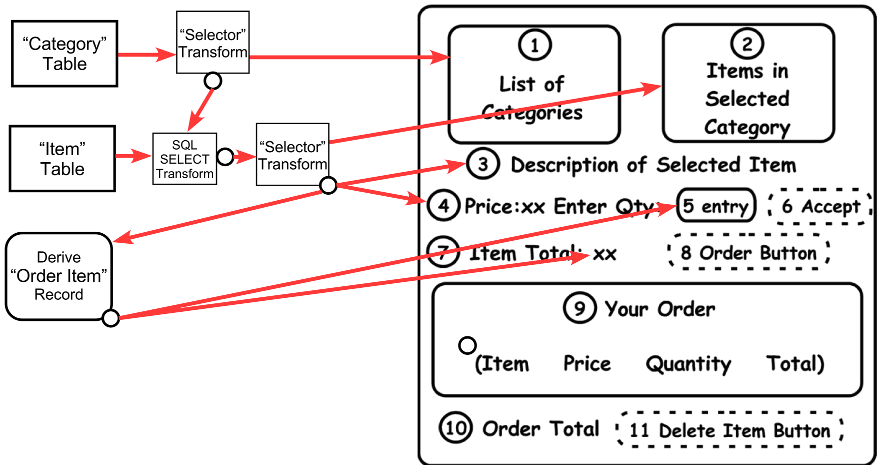


Now something new happens. It turns out there is a third table, called "Order Item", whose records are created from "Item" records by adding two columns: Quantity and Extended Price (Unit Price times Quantity). Furthermore, it has behavior: when an assignment is made to Quantity, Extended Price is computed. These Order Item records are the things that will be going into the Shopping Cart collection.

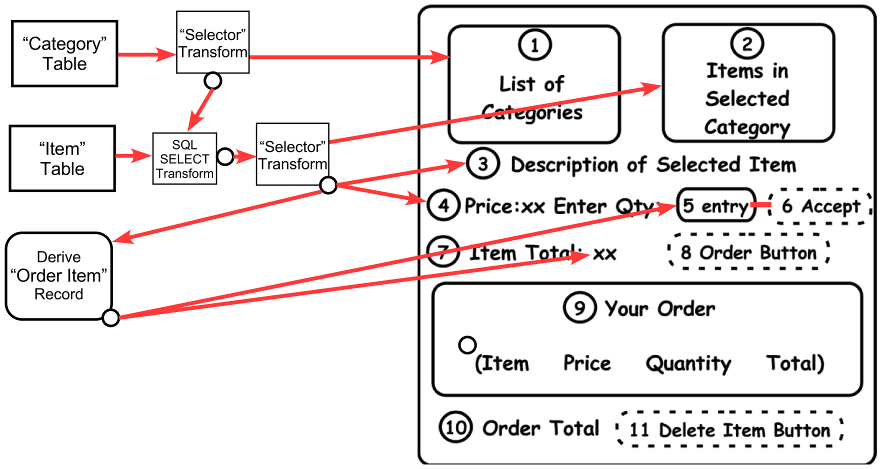
Working Paper No. 20

As of 7/31/2019

Every time a selection is made in the Item list box and the selected item is sourced from the Selector transform, a candidate Order Item record is made by a "Derive" transform box, and its Quantity and Extended Price fields go to user interface items 5 and 7:

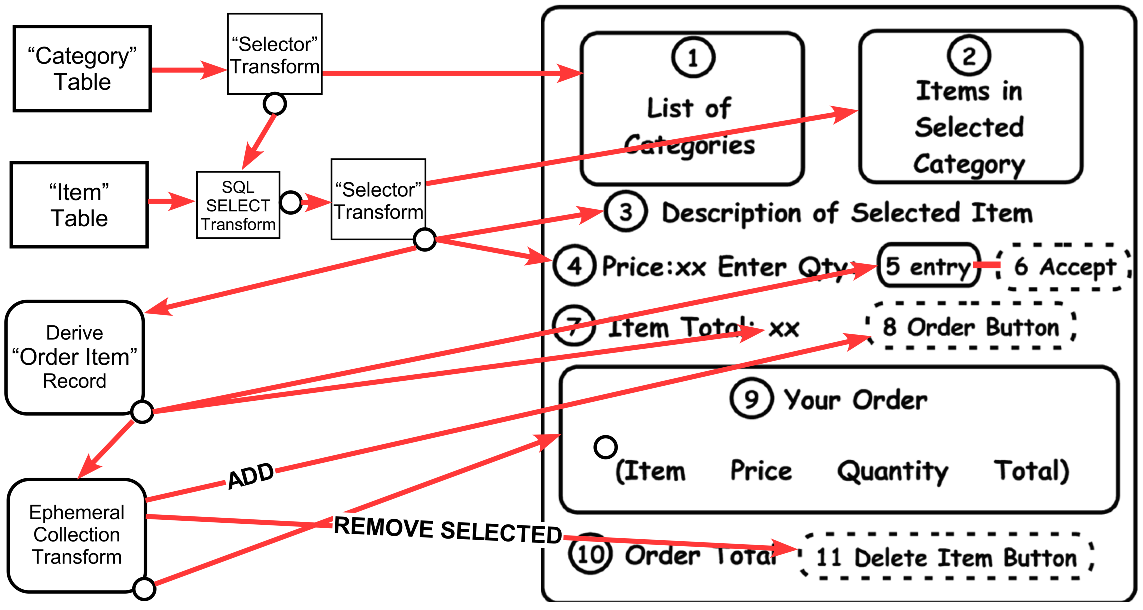


How does the assignment get made to Quantity? More precisely, how is the timing of the assignment from the entry field (5) back to the Order Item record determined? That's the function of the Accept button (6), which is connected to entry field (5) and tells it to make the assignment to its data source:

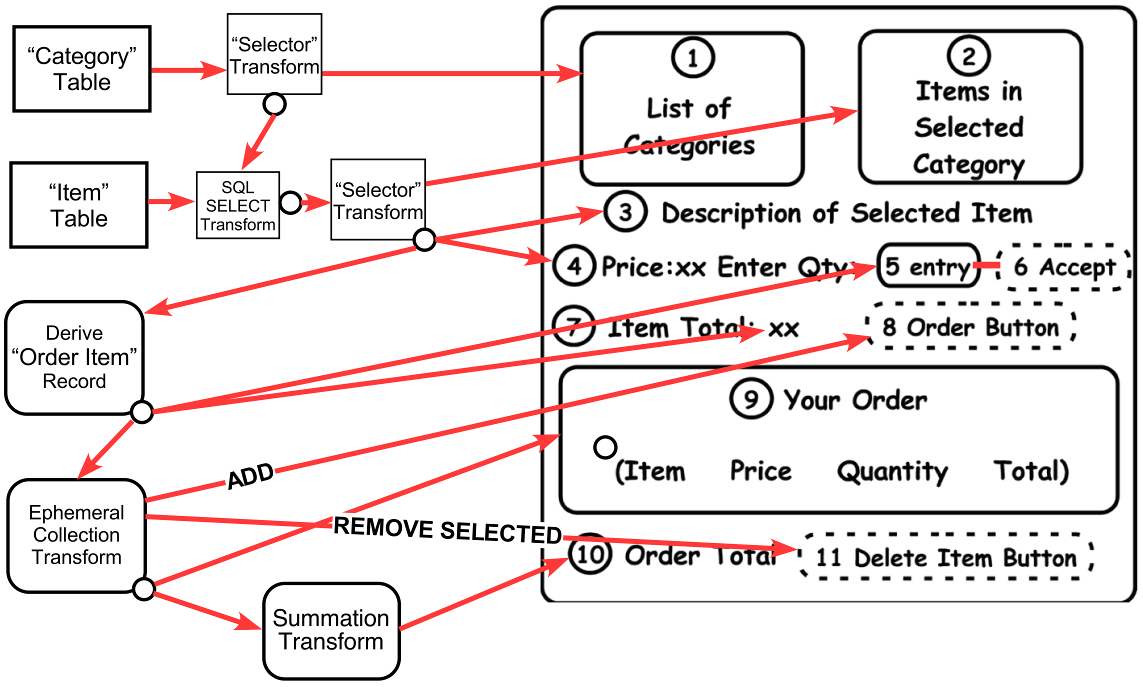


Now we need something (we'll call it a transform but whether it's "functional" purists might disagree) that holds the ephemeral collection that is the content of the shopping cart. The current Order Item record will be added to it when the Order button (8) is clicked.

(There are details being ignored here about paths to buttons; what are they? It turns out that they are data paths just like all the others, and no special treatment of user events is necessary. A button is a user-interface projector of an object I call a "doIt", and the normal update protocol used for keeping projections synchronized works for doIts also. To the extent that one wants to think about "flows" for data, there are no retrograde flows associated with user-interface events.)



Finally, we need to compute the total of the Extended Price columns in the shopping cart; we do this with another transform box. How we specify this and other transforms such as the SQL SELECT will be covered under Accessible APIs.



The Wiring Tool

From a Drawing to an Executing Wiring Diagram

Going from a sketch to a functioning wiring diagram in the wiring tool is straightforward because the wiring model looks so familiar. But first we'll exercise the wiring tool with a simple example.

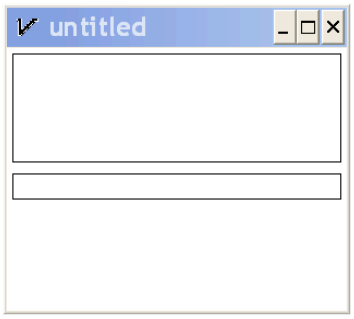
First Walk-through: A Simple Wired Example

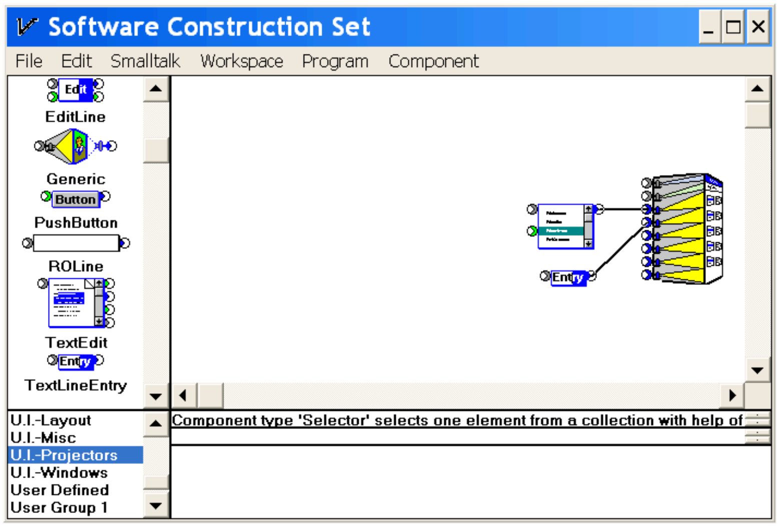
In this walk-through we build something simple that resembles the restaurant use case.

Assume left-to-right flow of data/objects on the wires.[2]

We're going to build a window with a list box showing a collection, and a text line showing the item selected in the list box. The window of the application being built will be on the left below.) The wiring tool will be on the right.

Some wired components, called *projectors*, have a special function: each renders its input object in its region of the application's user interface. The appearance on the user interface created by the projector is called the *projection* of its input object. On the right below are three wired projector components that render a multiline window frame, a list box, and a text line. The resulting application is on the left.





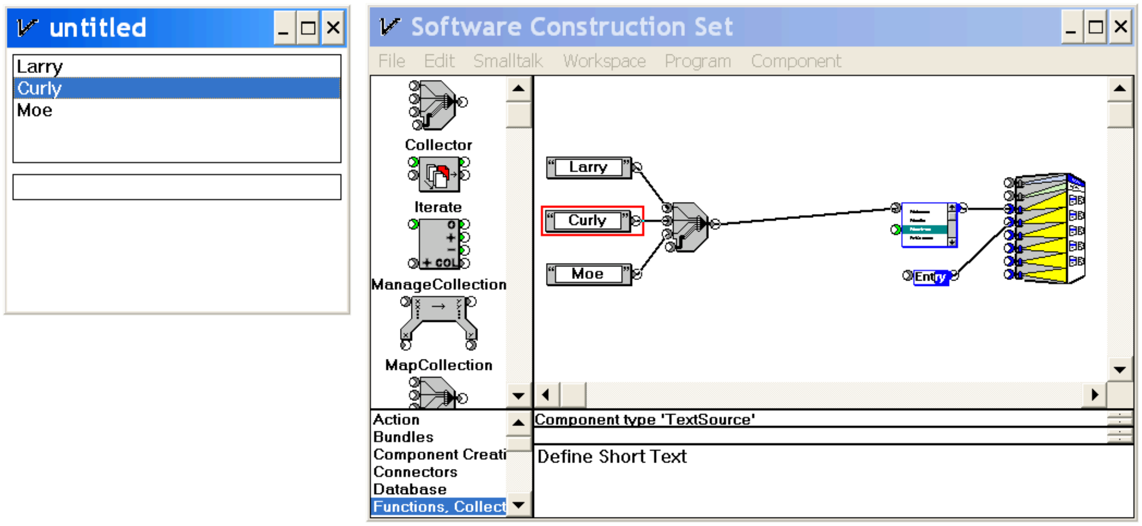
Component type 'Selector' selects one element from a collection with help of

As soon as these projector components were dragged onto the wiring workspace from the component

[2] The execution model started out as a flow model but has evolved to a set of message-based publish-subscribe trees. I have no reason to believe that it would run more slowly than conventional designs.

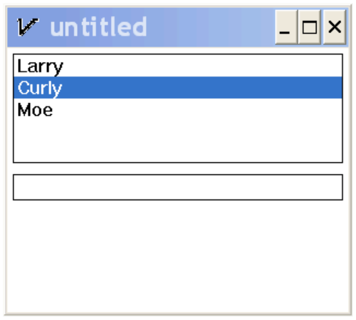
palettes on the left they were running; when they were wired together they showed up in the application window.

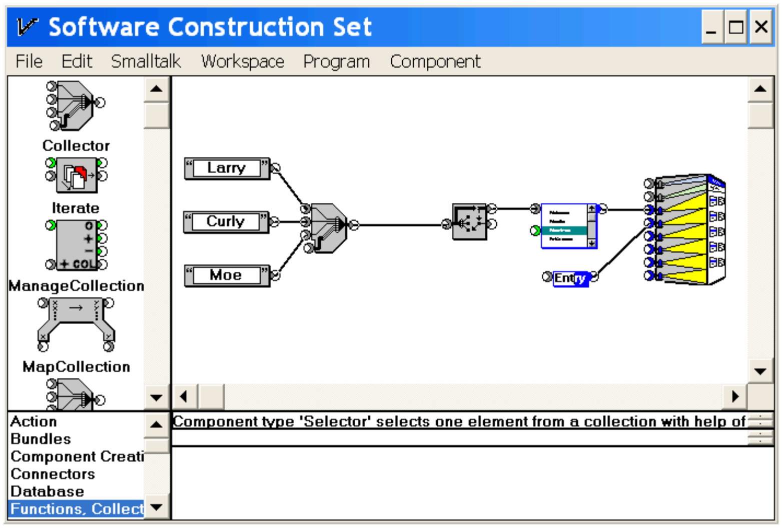
Now we'll wire up a test collection of three text constants and feed it into the list box projector; the result shows up at the left. (The selection of "Curly" is



the result of my having clicked in the list box.)

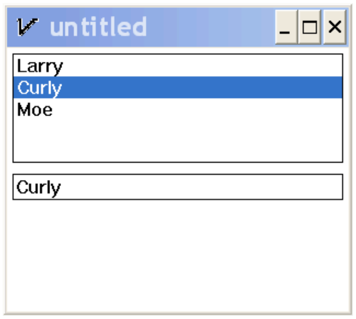
Notice that we have no way of feeding the result of a selection in the list box into the text line projector; that's the purpose of the *Selector* component, introduced below. Please note that the Selector is tightly coupled to the list box. There is nothing ad-hoc about this coupling; tight coupling between selectors and "choose one" components (of which the list box projector is but one example) is a natural consequence of the underlying messaging protocols.





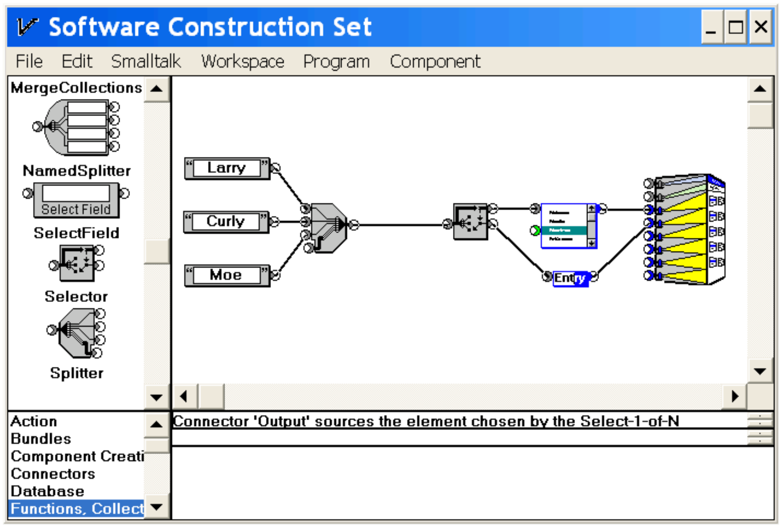
Component type 'Selector' selects one element from a collection with help of

After we wire the selector output to the text line component we see the result.



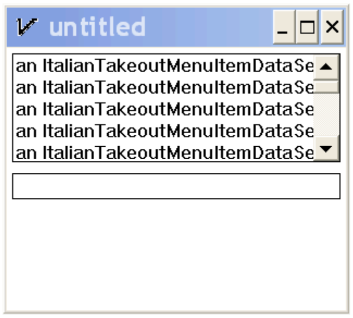
Larry
Curly
Moe

Curly

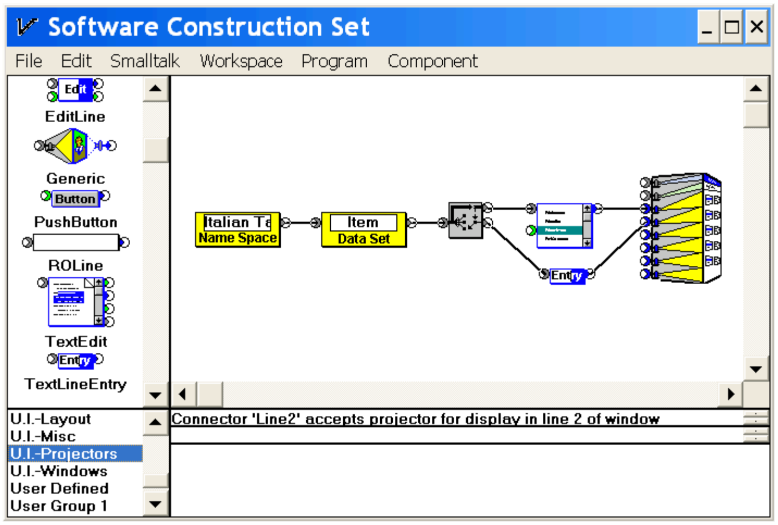


Connector 'Output' sources the element chosen by the Select-1-of-N

Now we'll replace the test collection source by the "Item" table from a relational database containing the take-out menu of an Italian restaurant. We do this by deleting the four components and their wires, and then replacing them. (There is some work selecting the database and table with parameter dialogs; this is not shown.)

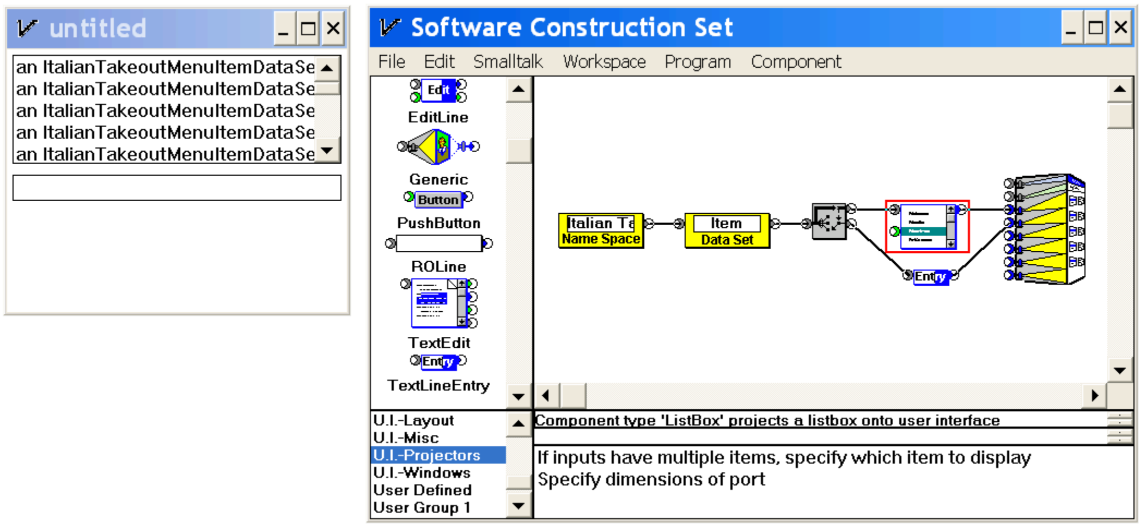


an ItalianTakeoutMenuItemDataSe
an ItalianTakeoutMenuItemDataSe
an ItalianTakeoutMenuItemDataSe
an ItalianTakeoutMenuItemDataSe
an ItalianTakeoutMenuItemDataSe



Connector 'Line2' accepts projector for display in line 2 of window

You see that the list box doesn't know which column of the table to display, so it shows the default: the Smalltalk class name of the table. This is our signal that we have to set a *parameter* of the list box component that specifies which column of each input item to display. So we select the component (selection is indicated by the red box); this reveals the list of commands the component makes available to the developer (the list box at the bottom of the window).

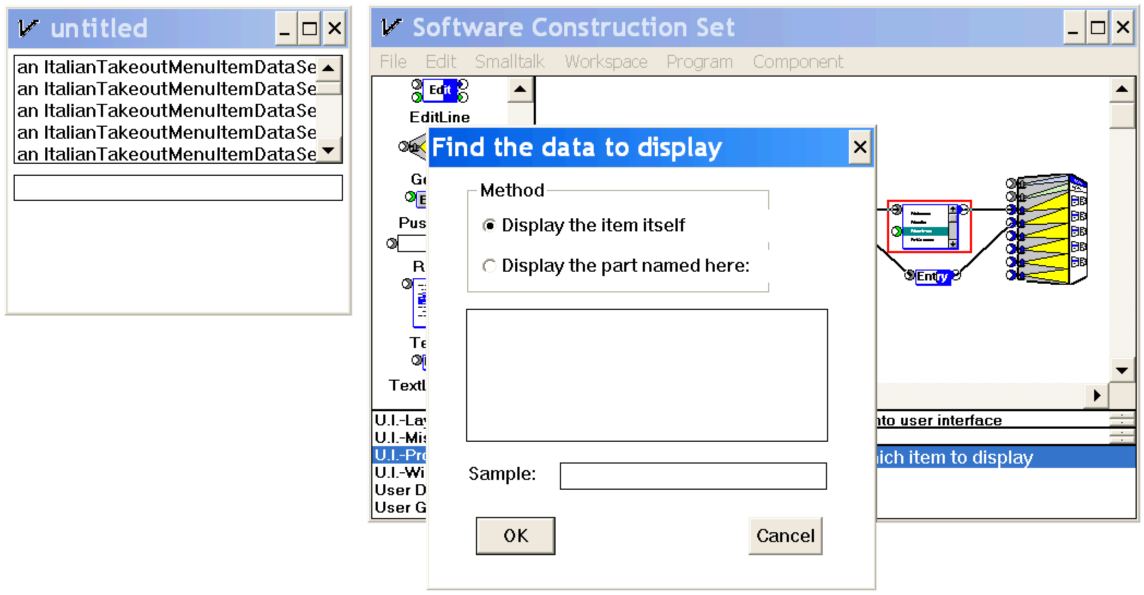


```
an ItalianTakeoutMenuItemDataSe
an ItalianTakeoutMenuItemDataSe
an ItalianTakeoutMenuItemDataSe
an ItalianTakeoutMenuItemDataSe
an ItalianTakeoutMenuItemDataSe
```

```
Component type 'ListBox' projects a listbox onto user interface
```

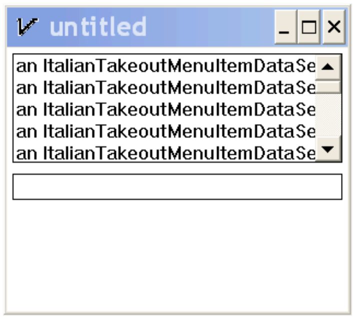
```
If inputs have multiple items, specify which item to display
Specify dimensions of port
```

Selecting the top command opens a dialog showing the default choice, which is not what we want:

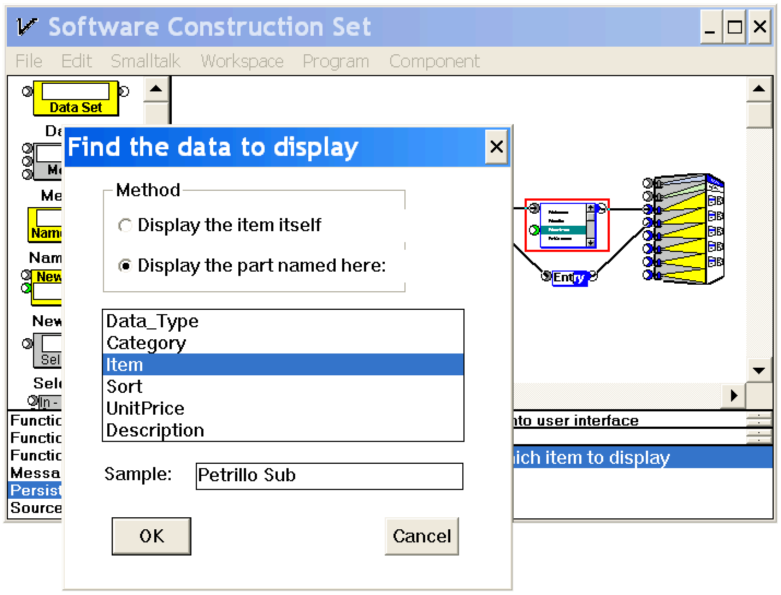

untitled
Software Construction Set
File Edit Smalltalk Workspace Program Component
Find the data to display
Method
Sample:
OK
Cancel

[✓ Display the item itself]
Display the part named here:

So we choose the other radio button:



an ItalianTakeoutMenuItemDataSe
an ItalianTakeoutMenuItemDataSe
an ItalianTakeoutMenuItemDataSe
an ItalianTakeoutMenuItemDataSe
an ItalianTakeoutMenuItemDataSe



Find the data to display

Method

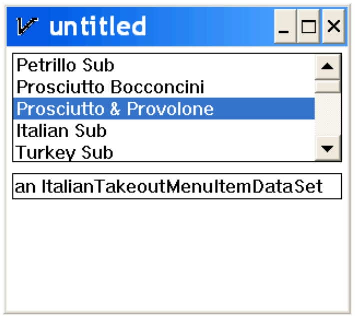
Display the item itself
[✓ Display the part named here:]

Data_Type
Category
[✓ Item]
Sort
UnitPrice
Description

Sample: Petrillo Sub

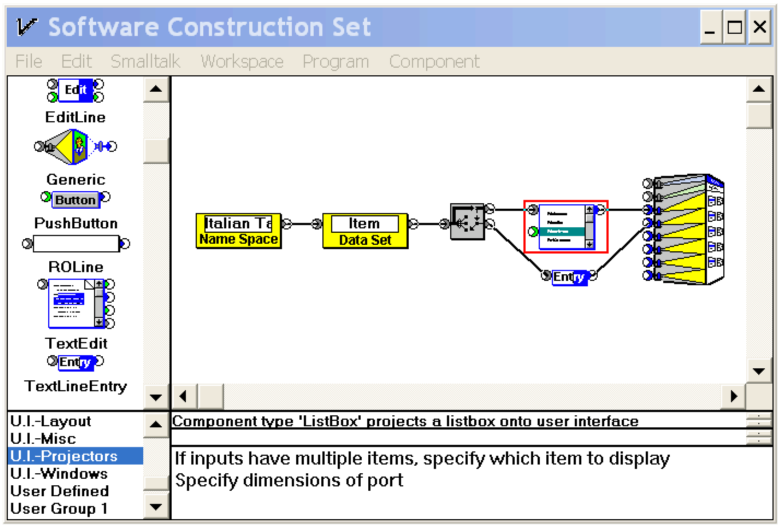
OK   Cancel

Here we see Humane Dozen #12, Alive With Your Data, in action. Not only do we see the list of column names but we are helped by being shown the value of the chosen column in the first record in the table. So we choose "Item", close the dialog, and see the correctly specified list box. (I clicked on the third item.)



Petrillo Sub
Prosciutto Bocconcini
Prosciutto & Provolone
Italian Sub
Turkey Sub

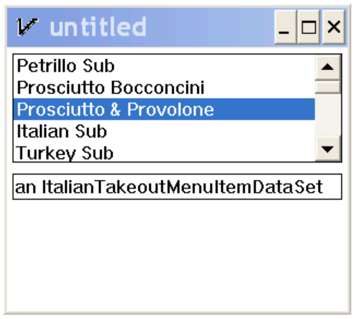
an ItalianTakeoutMenuItemDataSet



Component type 'ListBox' projects a listbox onto user interface

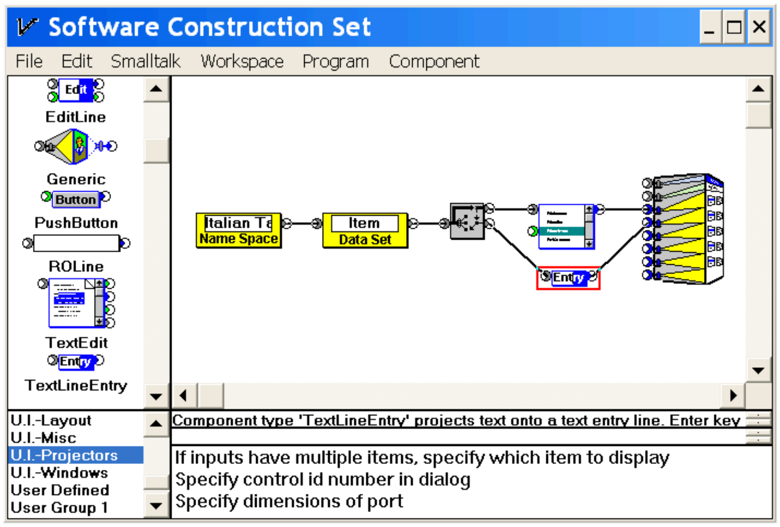
If inputs have multiple items, specify which item to display
Specify dimensions of port

Now we do the same thing for the text line. Select the component:



Petrillo Sub
Prosciutto Bocconcini
Prosciutto & Provolone
Italian Sub
Turkey Sub

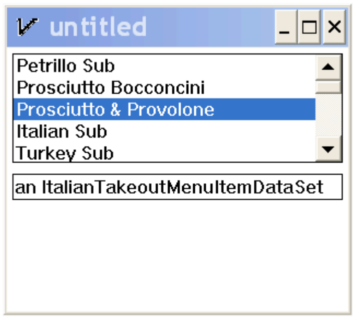
an ItalianTakeoutMenuItemDataSet



Component type 'TextLineEntry' projects text onto a text entry line. Enter key

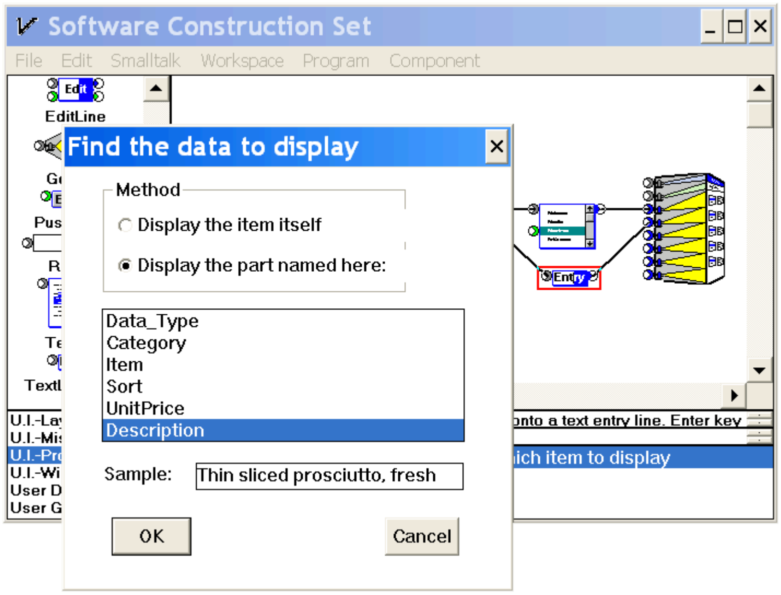
If inputs have multiple items, specify which item to display
Specify control id number in dialog
Specify dimensions of port

Click on the top command, then the second radio button. We'll choose the Description column:



Petrillo Sub
Prosciutto Bocconcini
Prosciutto & Provolone
Italian Sub
Turkey Sub

an ItalianTakeoutMenuItemDataSet

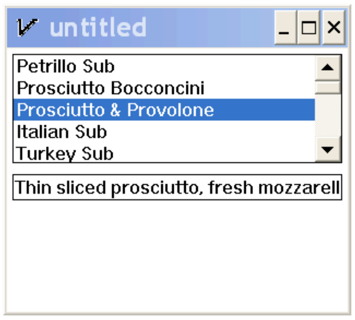


Method
( ) Display the item itself
(●) Display the part named here:

Data_Type
Category
Item
Sort
UnitPrice
Description

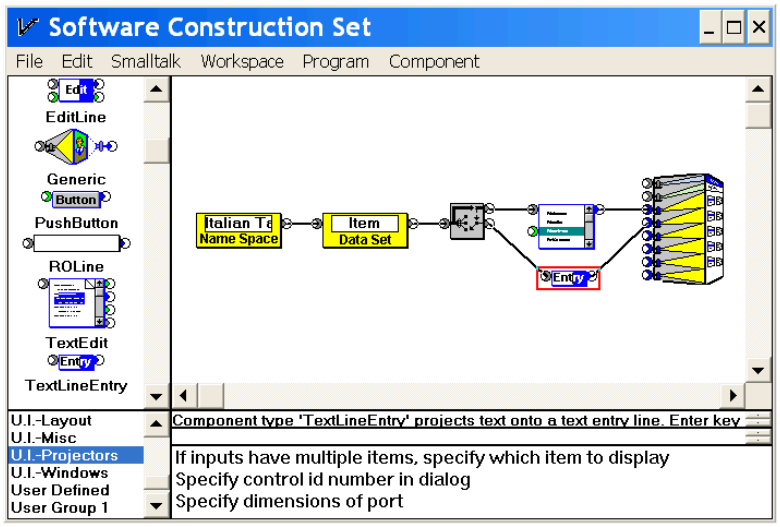
Sample: Thin sliced prosciutto, fresh

Close the dialog, and we get the desired result:



Petrillo Sub
Prosciutto Bocconcini
Prosciutto & Provolone
Italian Sub
Turkey Sub

Thin sliced prosciutto, fresh mozzarell



Component type 'TextLineEntry' projects text onto a text entry line. Enter key

If inputs have multiple items, specify which item to display
Specify control id number in dialog
Specify dimensions of port

Now We Wire the Actual Use Case

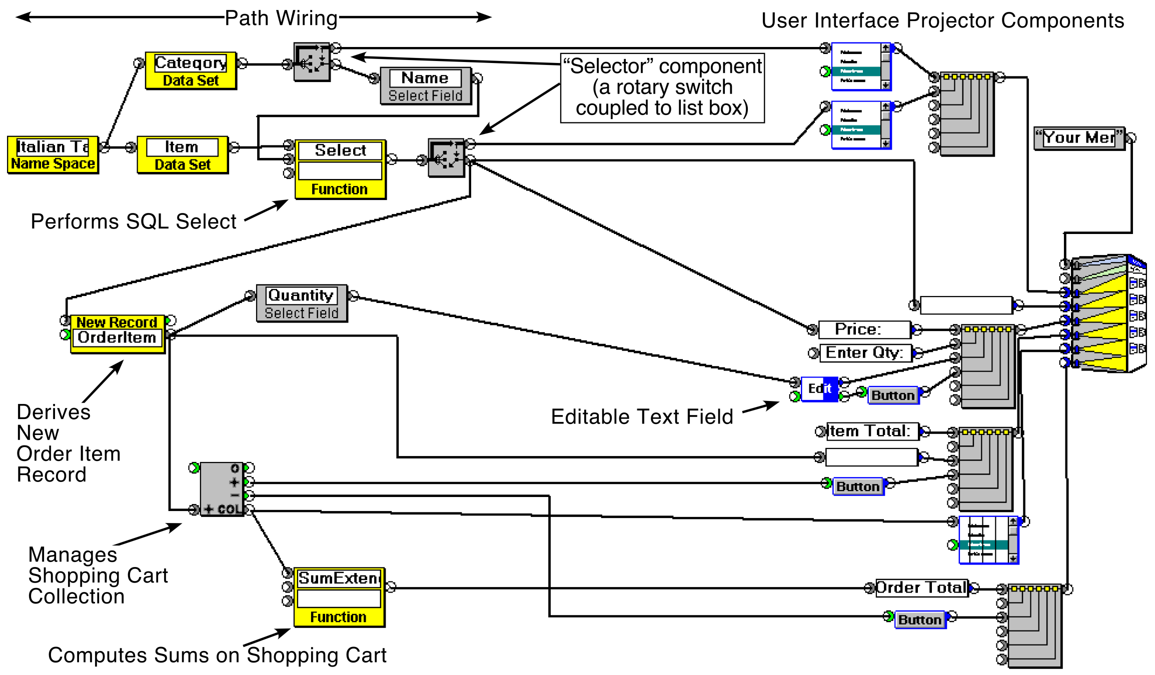
(placeholder: see http://melconway.com/talks/2018_gotober/13.html through http://melconway.com/talks/2018_gotober/21.html .)

Below you see the final result.

The Restaurant Use Case Wiring Diagram

The wiring tool as it exists has no WYSIWYG wireframe builder; instead each region of the screen is rendered with its own projector component. These feed into a projector for the multiline window frame.

Here is the wiring diagram for the restaurant use case, slightly annotated from a screen shot of the wiring tool executing the application. You can see it in operation at
http://melconway.com/talks/2019_consumer_apps/12.html
and the slide following that one. All the projectors are grouped on the right, and the wiring derived from the extended walk-through are grouped on the left.



Later on we will be paying special attention to the two yellow components labeled "Function"; they are "Gateway" components.[3]

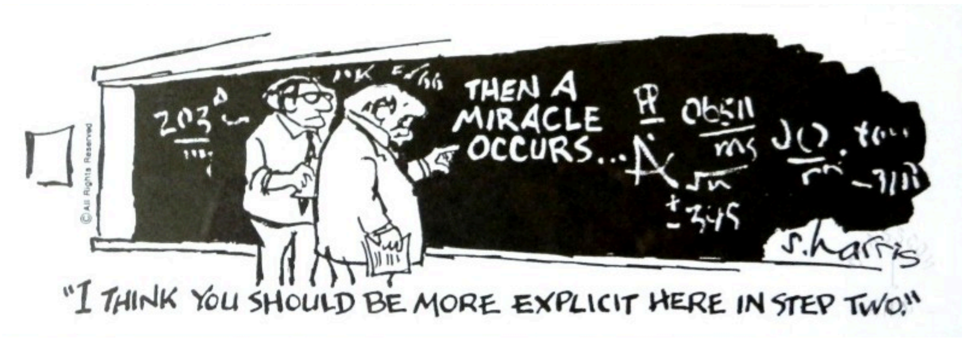
[3] There is no current video showing the whole process of building this use case, but an earlier version is very close: http://melconway.com/talks/2018_gotober/13.html through http://melconway.com/talks/2018_gotober/21.html .

Process: The Collaborative Application Model

The Functional-Path Model is a Basis for Programmer-Nonprogrammer Collaboration

The fact that you can go from a wireframe to a functioning wiring diagram might be an interesting exercise, but in terms of simplifying the development process, isn't it just squeezing the balloon at the small end, leaving most of the work for others to do?

Aren't those transform and other boxes to the left of the wireframe in the extended walk-through just "Then a Miracle Occurs" boxes?



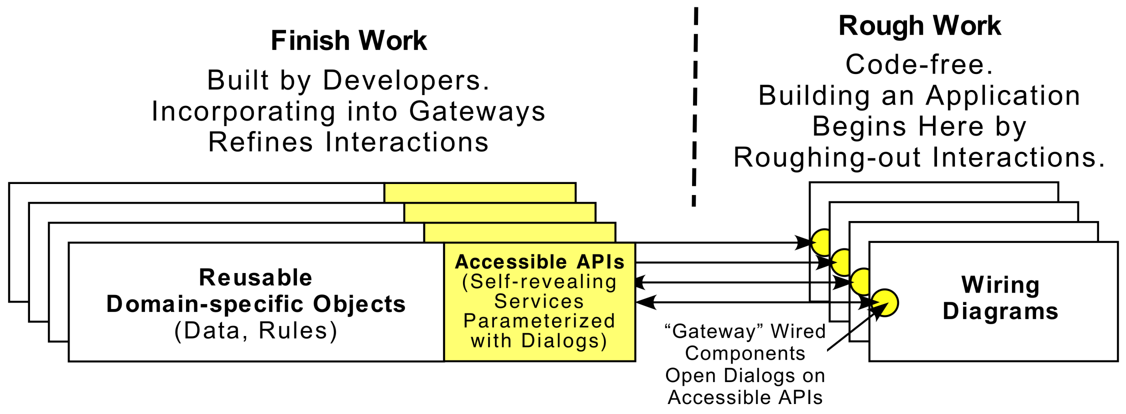
No, ***those boxes are the doors between the world of the programmer and the world of the non-programmer, and they lead us to an application model and development process that has an important role for both groups***.

The slots in those doors through which the two groups communicate without the non-programmers having to learn to code are *Accessible APIs* and *Self-revealing Services*, discussed below.

Rough to Finish: Iterative Development

When starting to build an application it can make sense to rough out the user interactions first, the way we did in the extended walk-through. Then we can add detail.

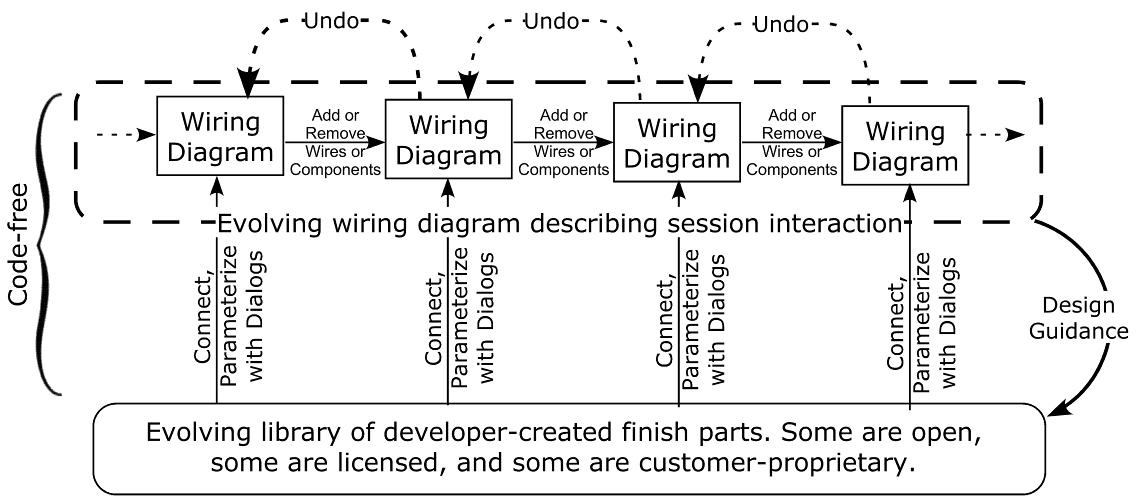
There is an interesting parallel to the manual craftsmanship analogy here too; the wiring diagram is the framework of the application and the details are added to refine the accuracy of the displays. Hence the names used in this *Collaborative Application Model*:



The Collaborative Application Model Supports a Quasi-decoupled, Iterative, Collaborative Workflow Between Programmers and Non-programmers

If the development environment conforms to *Humane Dozen #7: Robust*, then if a mistake is made with a choice of component or parameterization or wiring, the application might behave incorrectly but the tool will not break and the error will be reversible. This encourages a more experimental style of development, which is consistent with enlarging access to the general population.

The following figure suggests the workflow I envision. Wirers and developers proceed independently most of the time; their dependencies are at the points of contact in Gateway components, and their work might be subject to correction as the result of information gained when such connections are made. Together they iterate towards a solution.


Undo
Undo
Undo
Wiring Diagram
Add or Remove Wires or Components
Wiring Diagram
Add or Remove Wires or Components
Wiring Diagram
Add or Remove Wires or Components
Wiring Diagram
Evolving wiring diagram describing session interaction
Code-free
Connect, Parameterize with Dialogs
Connect, Parameterize with Dialogs
Connect, Parameterize with Dialogs
Connect, Parameterize with Dialogs
Design Guidance
Evolving library of developer-created finish parts. Some are open, some are licensed, and some are customer-proprietary.

The Theory Behind the Practice

The *Humane Dozen* Design Principles

Design Principles 1-7: Hands On the Working Material

The following seven attributes of a tool-language synthesis are what I consider necessary to move toward the hands-on end of the spectrum.

1. **Immediate** Every modification made to the working material is immediately seen in its behavior.
2. **Continuous** Small changes lead to predictable outcomes.
3. **Interactive** The result of each change suggests the next change, like a child playing with a construction toy.
4. **Transparent** The tool seems invisible and the artisan's hands seem to be directly on the working material.
5. **Inspectable** The behavior of all parts of the application can be inspected at any time. (Analogy: watching the behavior of a circuit by putting an oscilloscope probe on a test point.)
6. **Modifiable** The artisan can change his or her work in midstream.
7. **Robust** Every atomic gesture of the artisan that changes the working material is reversible and will not break the system. Of all possible states of the working material, many can be ugly or incorrect, but none are broken.[4]

Design Principles 8-12: Single-mode Workflow

A *Mode* is a set of cognitive constraints you have to submit to in order to do a particular task, for example, being in the mind-set of command-line syntax in order to compose a command, or programming-language syntax in order to write code.

Switching between modes is taxing and error-prone. The more modes, the greater the cognitive load. The following five attributes minimize cognitive load, minimize switching cost, and enhance fluidity in the workflow.

[4] The informal definition of "not broken" is: a determined novice can proceed without calling for help.

Notice *Self-revealing* in particular. This is the antithesis of coding. Instead of being required to *generate* a syntactically correct expression, you *choose* from presented alternatives, each of which can be explained if you need. When this attribute is put into practice consistently and without exception it transforms the construction process. *The DOer, SHOWer pattern directly addresses this attribute in multiple contexts*.

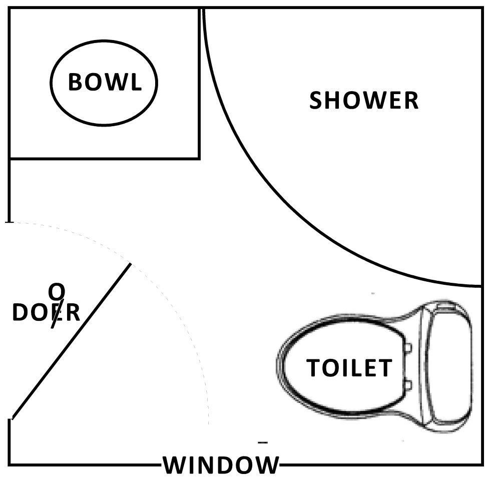
8. **Isomorphic** The executable form of the application is isomorphic to the working material. (This seems to be necessary for WYSIWYG development.)
9. **Self-revealing** Interfaces *present* (and, if necessary, explain) *choices*; they don't require formal constructions.
10. **Symmetrical** The tool and the application being built are *side-by-side peers*, running at the same time with consistent state. Your next user-interface action can be on either.
11. **Always on** There is no concept of starting or stopping applications or components during construction. When a component is introduced onto the workspace it is running.
12. **Alive with your data** You can see the flow of data through the application moment by moment.

The DOer, SHOWer Pattern

Why the Funny Spelling?

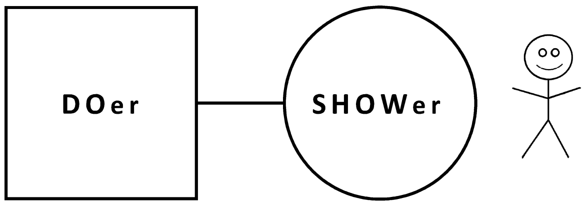
So you won't confuse the names with a misspelled architectural drawing like the one below, and they will therefore help you remember the intended pronunciation:

DOer, SHOWer rhymes with MOO-er, BLOW-er.



The Pattern is Simple

There are two things in the pattern: one thing does, one thing shows. The pattern is a generalization of the Model-View-Controller pattern. Here is a figure we will be using.[5]



It has two parts. As you will see later they are not necessarily software objects.

- The **DOer** is a thing that has a specific job that it Does.
- The **SHOWer** is a user interface on the DOer. It permits the user (shown as a stick figure on the right) to see aspects of the DOer that the SHOWer's designer wants the user to see, and for the user to change or edit (some of) those

[5] Stick figure courtesy of Wikipedia, created by Jleedev: https://en.wikipedia.org/wiki/Stick_figure

aspects.
The SHOWer might be part-time. For example, it might exist as a factory that creates a dialog box object when the user requests.

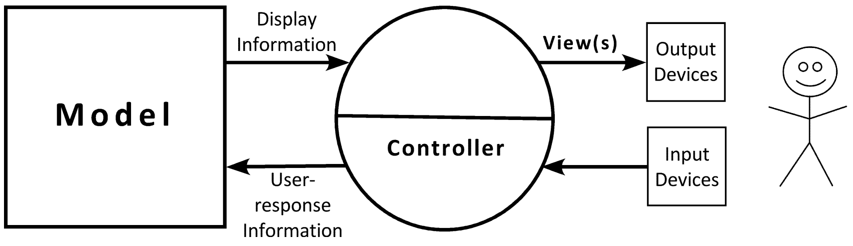
The user is not part of the pattern.

Pattern Examples 1-3

The variety of the six examples suggests the wide applicability of the pattern.

Example 1: Model-View-Controller (MVC)

Model-View-Controller is the underlying model for the GUIs (graphical user interfaces) of almost all well-architected contemporary software. It was formalized in the 1970s in the Smalltalk project at Xerox PARC.[6]

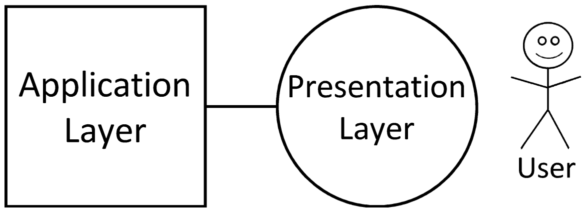


(In many descriptions, the terms View and Controller are not parallel; "View" means a display in a particular format (there might be several), and "Controller" is the part of the software that handles events from the user. Hence the asymmetry in the figure.)

6 https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Example 2: Application-Presentation Layered Model

The common GUI operating systems such as Smalltalk[7], Microsoft® Windows®[8] and Macintosh® OS X®[9] employ variants of MVC. As a class of models we can refer to them as Application-Presentation Layered Models:



Example 3: The Application-Development Life Cycle (Conventional)

In this example we'll see that the pattern is hierarchical.

The thinking behind Conway's Law[10] can be simply stated in the slogan "Think life cycle, not artifact". The slogan is a device to help one back away from the center of attention in order to find a larger, containing system.

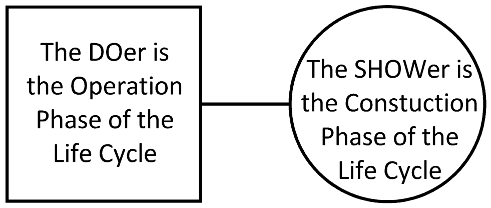
In its simples form, the life cycle of an interactive business application has two phases.

- **The Operation Phase**. The object code of the application is executing.
- **The Construction Phase**. This is the design/development phase of building the software. In this conventional example, the artifact (the thing being built) comprises developer-readable *source code*, from which a software toolset creates computer-executable *object code*.

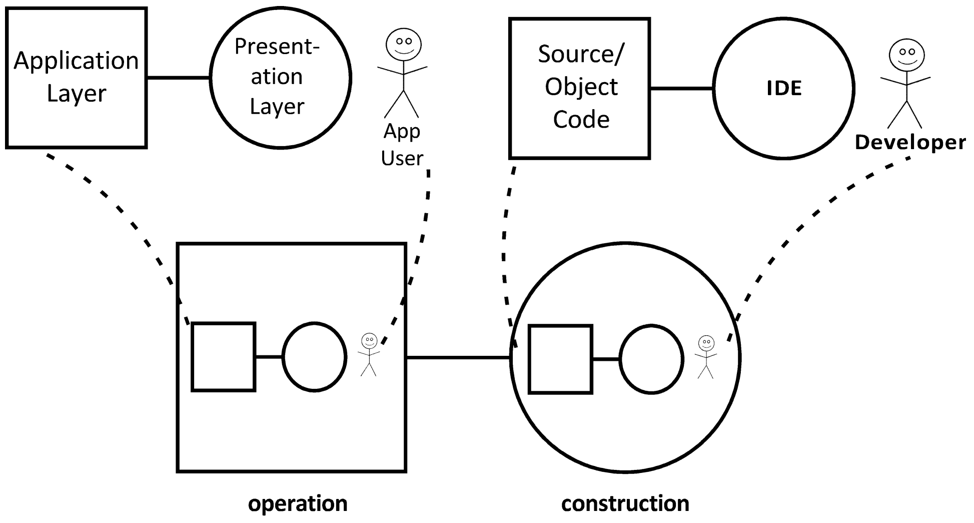
7 https://en.wikipedia.org/wiki/Smalltalk
8 https://en.wikipedia.org/wiki/Microsoft_Windows
9 https://en.wikipedia.org/wiki/MacOS
10 https://en.wikipedia.org/wiki/Conway%27s_law

Mapping this onto our pattern we get:

- The **DOer is the Operation Phase of the application's life cycle**. The code is executing in an operational environment.
- The **SHOWer is the Construction Phase of the application's life cycle**. The code is being examined and modified in a development environment.


The DOer is the Operation Phase of the Life Cycle
The SHOWer is the Constuction Phase of the Life Cycle

Looking inside the *operation* phase of the application's life cycle, we can account for the application's GUI with one or more instances of the Application-Presentation Layered pattern (see the left below). In the *construction* phase (see the right below) we will see the developer viewing and editing the source code using an Integrated Development Environment as the SHOWer:



Pattern Example 4: Self-revealing Parameters

Definition

- A *Self-Revealing Parameter* is a parameter equipped with sufficient viewer(s)/editor(s) for examining and/or specifying it. The normal way the parameter is specified or examined is through these viewer(s)/editor(s), *which conform to Humane Dozen #9, Self-revealing*.

Implementation

Here is the design behind the “Find the data to display” dialog in the above wiring-tool walk-through.[11] It is an application of the DOer, SHOWer pattern.

Any component that needs to project a string value that is a named element of an input collection, for example the two projectors in the walk-through, will need to have as one of its parameters something whose value is an instance of the Smalltalk class **`DisplayMethodParameter`**.

Every **`DisplayMethodParameter`** *value has the ability to raise the “Find the data to display” dialog on the host component’s input value.*

> Here is an assumed underlying convention: Every object will respond to the message **`hasNamedField`** with a Boolean. Each object answering **`true`** must be able to supply a list of its named fields and will respond to **`namedFieldAt:<string>`** with the instance variable that has that name.

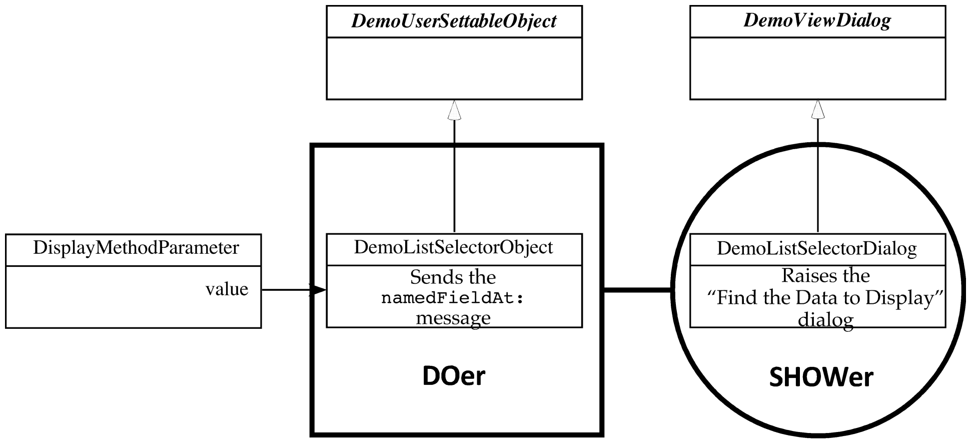
Every parameter of every wired component has a “value” instance variable. In the case of **`DisplayMethodParameter`** the parameter value is an instance of the class **`DemoListSelectorObject`**, which is a subclass of the abstract class **`DemoUserSettableObject`**.

`DemoListSelectorObject` is the DOer part of the pattern. Its twin, **`DemoListSelectorDialog`**, is the SHOWer part of the pattern; it raises the “Find the Data to Display” dialog.

[11] The wiring tool in its present form is a 16-bit Windows application. Other implementations of this example might work differently.

This is made explicit in the figure below. The abstract class **`DemoUserSettableObject`** has a twin abstract class **`DemoViewDialog`**, and each subclass of **`DemoUserSettableObject`** has a twin subclass of **`DemoViewDialog`**.

Each pair of twin descendants of **`DemoUserSettableObject`** and **`DemoViewDialog`** *is a realization of the DOer, SHOWer pattern.*



All parameters whose values are specified through a dialog use this pattern.

When its respective parameter needs to be edited (that is, when the developer clicks the component's command corresponding to that parameter), each descendant of **`DemoUserSettableObject`** creates an instance of its twin, which will raise a Windows dialog through which the user can edit its instance variables.

Following is a table of some self-revealing parameters in the wiring tool that conform to this pattern.

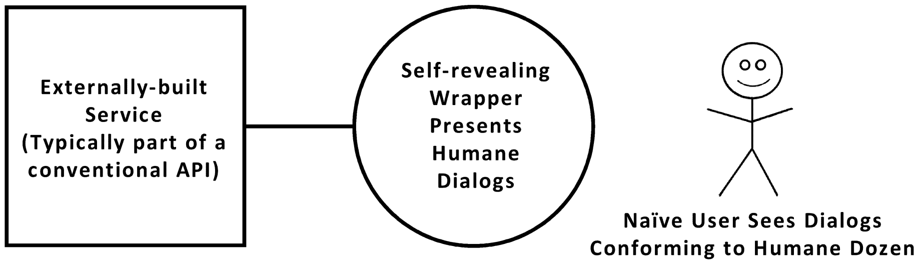
Parameter class	**DemoUserSettableObject subclass**	**DemoViewDialog subclass**	**Windows resource name**
ColumnListParameter	DemoColumnListObject	DemoColumnListDialog	columnlist
ConnectorNameParameter	String	Smalltalk Prompter	
DatabaseNameParameter	DemoDatabaseObject	DemoDatabaseDialog	databasename
DisplayMethodParameter	DemoListSelectorObject	DemoListSelectorDialog	listselector
FieldSpecParameter	DemoFieldSpecObject	DemoFieldSpecDialog	fieldspecs
FrameParameter	DemoFrameObject	DemoFrameDialog	demoframe
NameQuadParameter	DemoNameQuadObject	DemoNameQuadDialog	splitternames
NameTripletParameter	DemoNameTripletObject	DemoNameTripletDialog	Collectornames
NameSpaceNameParameter	PersistentNameSpace		
QueryNameParameter	DemoQueryObject	DemoTableDialog	queryname
SimpleBooleanParameter	Boolean		
SimpleNumericParameter	Integer		
SortItemParameter	DemoSortItemObject	DemoSortItemDialog	sortitem
TableNameParameter	DemoTableObject	DemoTableDialog	tablename
TextStringParameter	String	Smalltalk Prompter	

Pattern Example 5: Self-revealing Services

Definition

- An *Accessible API* is a collection of one or more *Self-revealing Services*.
- A *Self-revealing Service* looks like the service analog of a self-revealing parameter. It is a service equipped with sufficient viewer(s)/editor(s) for examining and specifying a request to it. The normal way a request is specified is through the viewer(s)/editor(s), *which conform to Humane Dozen #9, Self-revealing*.

But Self-revealing Services are very different in this respect: systems are built by combining externally-built services. Therefore it should be possible to take a conventional service built elsewhere and have a way to turn it into a Self-revealing Service. We do this by treating the externally-built service as the DOer of our pattern, and adding a SHOWer to it that appears to the external user as a Self-revealing Service. The SHOWer acts as a wrapper that presents a standardized interface to the naïve user that conforms to the Humane Dozen:


Externally-built Service (Typically part of a conventional API)
Self-revealing Wrapper Presents Humane Dialogs
Naïve User Sees Dialogs Conforming to Humane Dozen

For each service the wrapper should present a single, standard service. It answers the simple question: “Show me what you can do”. The form of the answer is a list of zero or more Self-revealing Services. This list is presented in the standard dialog raised by the Gateway component.

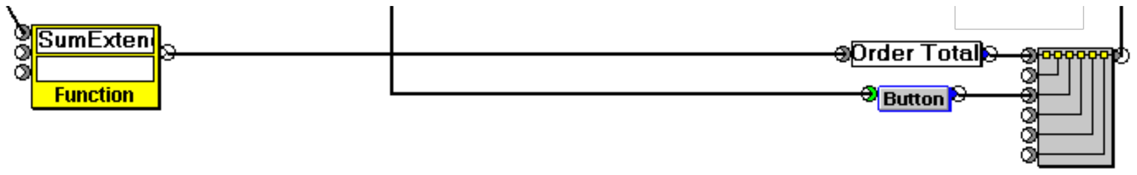
I’ll show two examples from the restaurant use case.

1. Computing the shopping-cart total. This is a “shallow” interaction: the wirer chooses from the two available kinds of total with one click in the standard dialog raised by the Gateway component.

2. Specifying the SQL “SELECT” verb. This is a “deep” interaction: the “Specify Message” button in the standard dialog tells the Gateway component to turn control over to the wrapper, which raises its own dialog(s) to further specify the service.

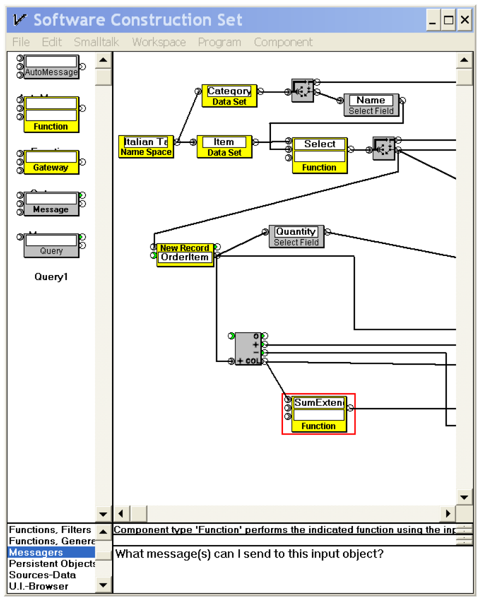
A Shallow Interaction

We’ll be looking at the Gateway component that computes the total of the shopping cart:



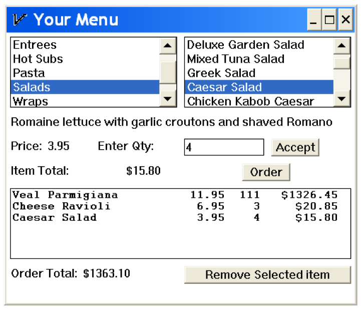
Here we start with the shopping cart in the final state of the video at
http://melconway.com/talks/2019_consumer_apps/12.html .

In the following we have selected this Gateway component. Its input is the collection of three Order item records in the shopping cart. The service that has been selected is “Sum Extended Price Fields” and the result shows up at the bottom of the application window.



Component type 'Function' performs the indicated function using the inp

What message(s) can I send to this input object?



Your Menu

Entrees
Hot Subs
Pasta
Salads
Wraps

Deluxe Garden Salad
Mixed Tuna Salad
Greek Salad
Caesar Salad
Chicken Kabob Caesar

Romaine lettuce with garlic croutons and shaved Romano

Price: 3.95 Enter Qty: 4 Accept

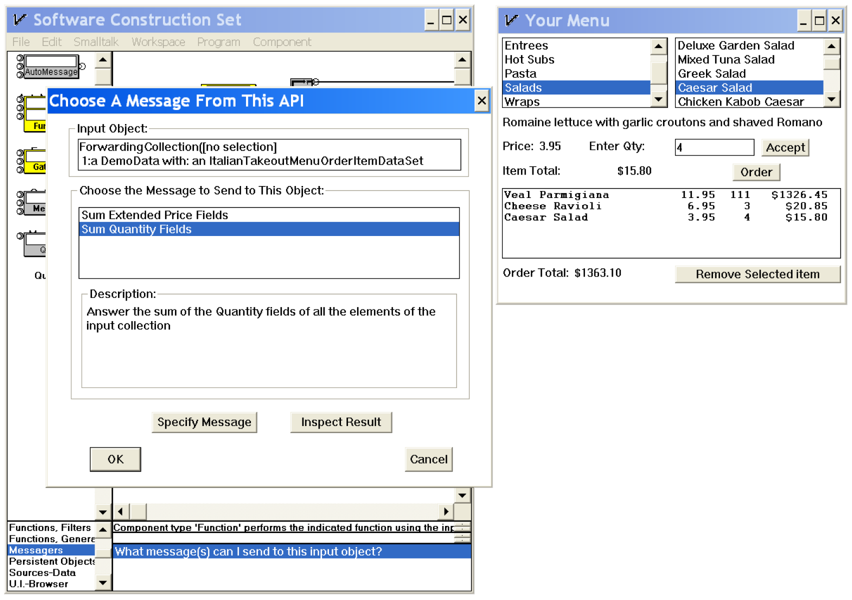
Item Total: $15.80 Order

```
Veal Parmigiana        11.95   111   $1326.45
Cheese Ravioli          6.95     3     $20.85
Caesar Salad            3.95     4     $15.80
```

Order Total: $1363.10 Remove Selected item

We’re going to see what other service is available so we click on the “What message(s)…” command at the bottom, and the Gateway component's standard dialog opens up, showing two services: "Sum Extended Price Fields" and "Sum Quantity Fields".

Notice that a brief description is shown below the selected service.


Software Construction Set
File Edit Smalltalk Workspace Program Component
AutoMessage
Your Menu

Choose A Message From This API

Input Object:
ForwardingCollection([no selection]
1:a DemoData with: an ItalianTakeoutMenuOrderItemDataSet

Choose the Message to Send to This Object:
Sum Extended Price Fields
Sum Quantity Fields

Description:
Answer the sum of the Quantity fields of all the elements of the input collection

Specify Message | Inspect Result
OK | Cancel

Functions, Filters
Functions, Generc
Messagers
Persistent Objects
Sources-Data
U.I.-Browser

Component type 'Function' performs the indicated function using the inp
What message(s) can I send to this input object?

Entrees
Hot Subs
Pasta
Salads
Wraps

Deluxe Garden Salad
Mixed Tuna Salad
Greek Salad
Caesar Salad
Chicken Kabob Caesar

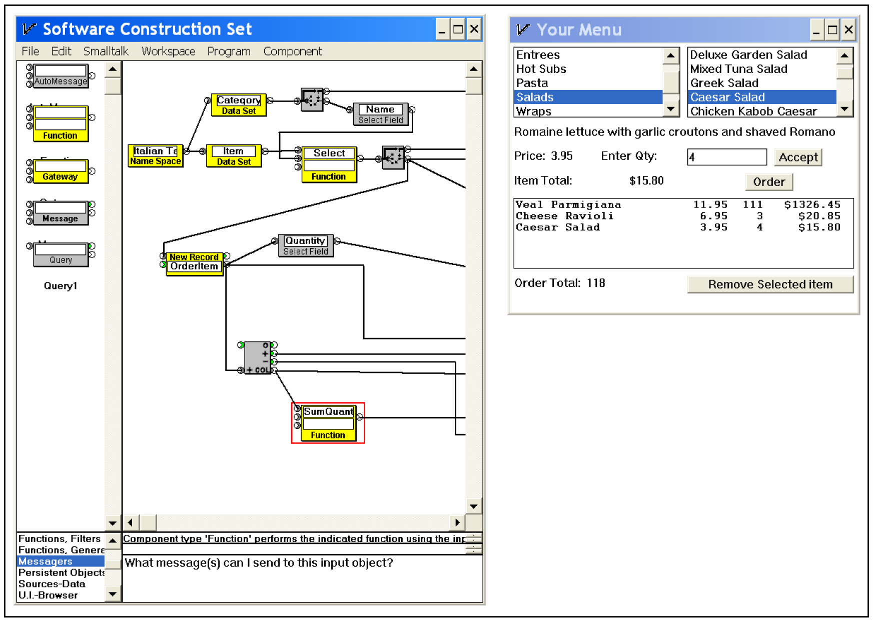
Romaine lettuce with garlic croutons and shaved Romano

Price: 3.95 Enter Qty: 4 Accept
Item Total: $15.80 Order

```
Veal Parmigiana      11.95  111   $1326.45
Cheese Ravioli        6.95    3     $20.85
Caesar Salad          3.95    4     $15.80
```

Order Total: $1363.10 Remove Selected item

We select "Sum Quantity Fields". After we click OK the dialog closes and we immediately see the new value.



Functions, Filters
Functions, Generc
Messagers
Persistent Objects
Sources-Data
U.I.-Browser

Component type 'Function' performs the indicated function using the inp
What message(s) can I send to this input object?

Entrees
Hot Subs
Pasta
Salads
Wraps

Deluxe Garden Salad
Mixed Tuna Salad
Greek Salad
Caesar Salad
Chicken Kabob Caesar

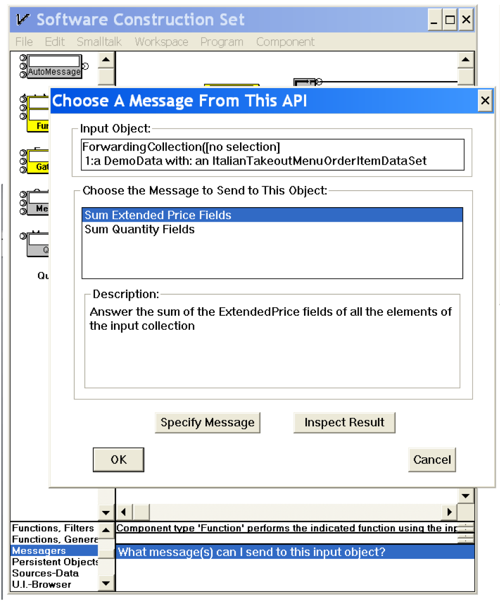
Romaine lettuce with garlic croutons and shaved Romano

Price: 3.95 Enter Qty: 4 Accept
Item Total: $15.80 Order

```
Veal Parmigiana      11.95  111   $1326.45
Cheese Ravioli        6.95    3     $20.85
Caesar Salad          3.95    4     $15.80
```

Order Total: 118 Remove Selected item

Now we reverse the process:


Software Construction Set
File Edit Smalltalk Workspace Program Component
AutoMessage
Choose A Message From This API
Specify Message
Inspect Result
OK
Cancel

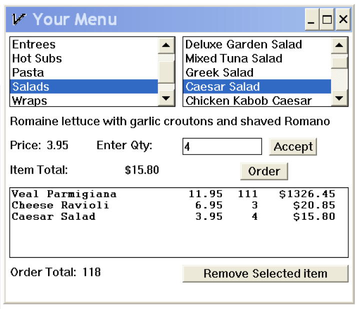
Input Object:
ForwardingCollection([no selection]
1:a DemoData with: an ItalianTakeoutMenuOrderItemDataSet

Choose the Message to Send to This Object:
Sum Extended Price Fields
Sum Quantity Fields

Description:
Answer the sum of the ExtendedPrice fields of all the elements of the input collection

Functions, Filters
Functions, Genera
Messages
Persistent Objects
Sources-Data
U.I.-Browser

Component type 'Function' performs the indicated function using the inp
What message(s) can I send to this input object?



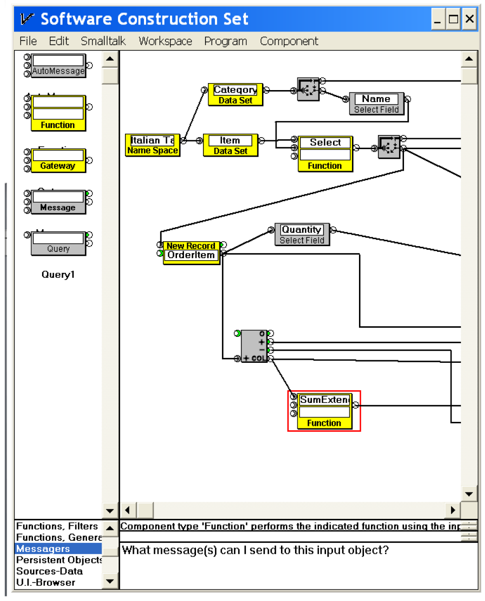
Romaine lettuce with garlic croutons and shaved Romano

Price: 3.95 Enter Qty: 4

Item Total: $15.80

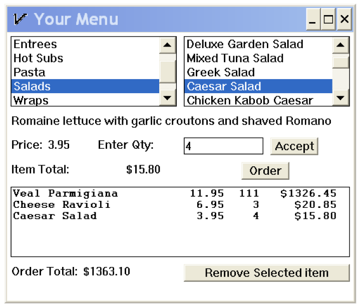
```
Veal Parmigiana       11.95  111   $1326.45
Cheese Ravioli         6.95    3     $20.85
Caesar Salad           3.95    4     $15.80
```

Order Total: 118



Functions, Filters
Functions, Genera
Messages
Persistent Objects
Sources-Data
U.I.-Browser

Component type 'Function' performs the indicated function using the inp
What message(s) can I send to this input object?



Romaine lettuce with garlic croutons and shaved Romano

Price: 3.95 Enter Qty: 4

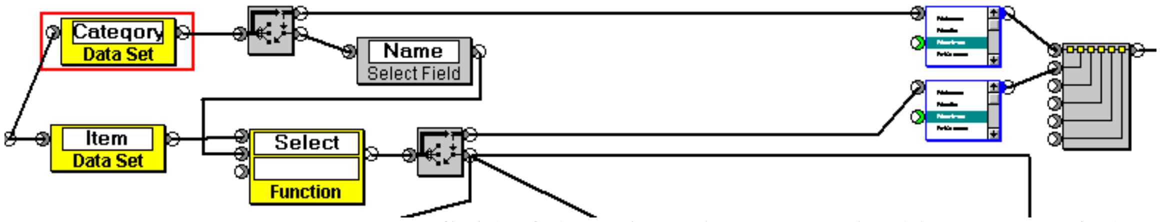
Item Total: $15.80

```
Veal Parmigiana       11.95  111   $1326.45
Cheese Ravioli         6.95    3     $20.85
Caesar Salad           3.95    4     $15.80
```

Order Total: $1363.10

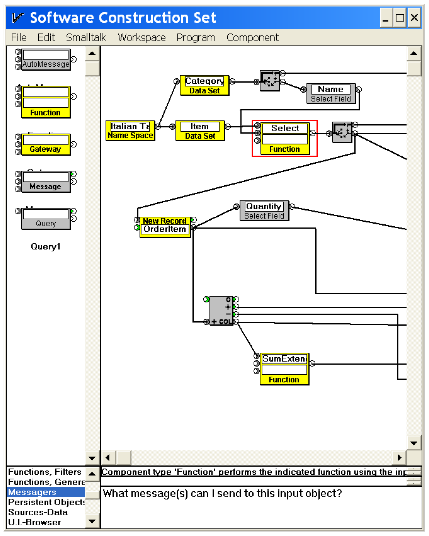
A Deep Interaction

In this case we'll be looking at the Gateway component that specifies an SQL SELECT applied to the Item table input, using as a parameter the Name



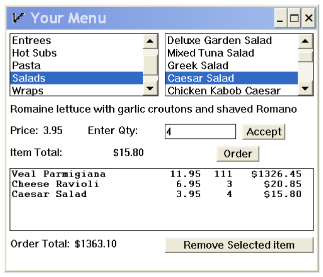
field of the selected category, in this case, "Salads":

Here we have selected the Gateway component and we see the "What Message(s)…" command at the bottom:



Component type 'Function' performs the indicated function using the inp

What message(s) can I send to this input object?



Entrees	Deluxe Garden Salad
Hot Subs	Mixed Tuna Salad
Pasta	Greek Salad
Salads	Caesar Salad
Wraps	Chicken Kabob Caesar

Romaine lettuce with garlic croutons and shaved Romano

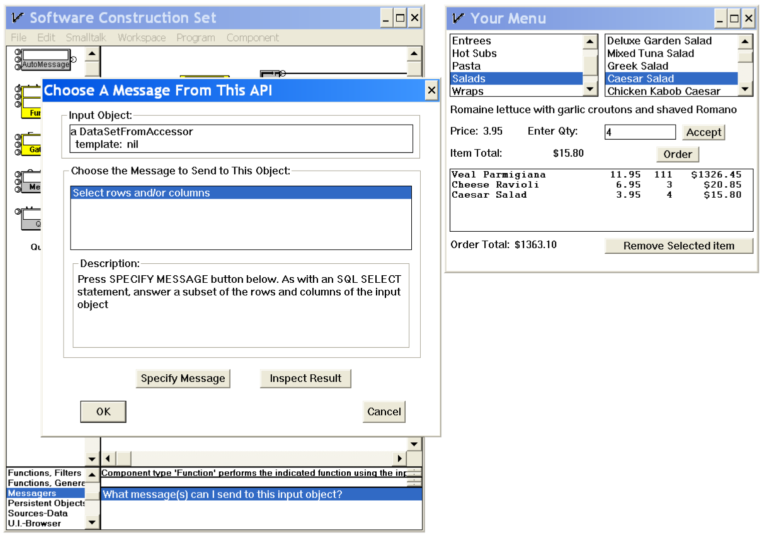
Price: 3.95 Enter Qty: 4

Item Total: $15.80

```
Veal Parmigiana        11.95  111   $1326.45
Cheese Ravioli          6.95    3     $20.85
Caesar Salad            3.95    4     $15.80
```

Order Total: $1363.10

We select it, and the Gateway component raises its standard dialog. There we see only one choice, “Select Rows and/or Columns”.


Software Construction Set
File Edit Smalltalk Workspace Program Component
AutoMessage
Choose A Message From This API
Specify Message
Inspect Result
OK
Cancel
Your Menu
Accept
Order
Remove Selected item

Input Object:
a DataSetFromAccessor
template: nil

Choose the Message to Send to This Object:
Select rows and/or columns

Description:
Press SPECIFY MESSAGE button below. As with an SQL SELECT statement, answer a subset of the rows and columns of the input object

Functions, Filters
Functions, Genera
Messagers
Persistent Objects
Sources-Data
U.I.-Browser

Component type 'Function' performs the indicated function using the inp
What message(s) can I send to this input object?

Entrees
Hot Subs
Pasta
Salads
Wraps

Deluxe Garden Salad
Mixed Tuna Salad
Greek Salad
Caesar Salad
Chicken Kabob Caesar

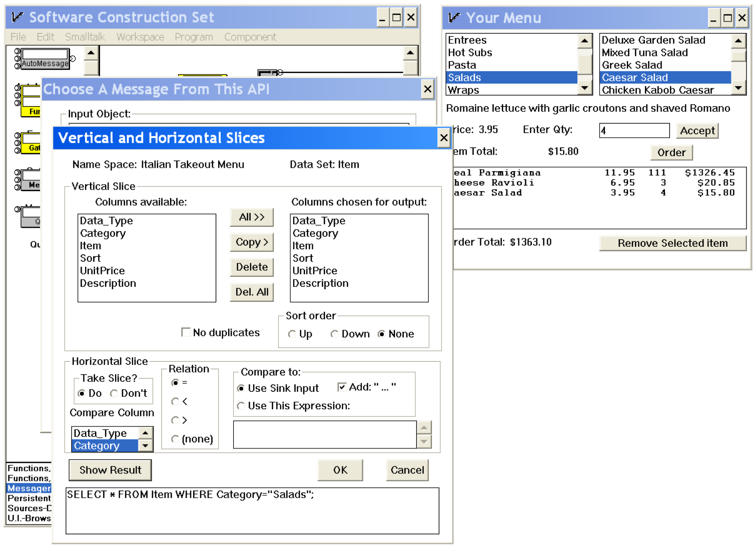
Romaine lettuce with garlic croutons and shaved Romano

Price: 3.95 Enter Qty: 4
Item Total: $15.80

```
Veal Parmigiana         11.95  111   $1326.45
Cheese Ravioli           6.95    3     $20.85
Caesar Salad             3.95    4     $15.80
```

Order Total: $1363.10

We choose it and follow the instruction to click the “Specify Message” button. This gives control to the wrapper, which continues the process. It opens a dialog that projects a “Slice” object (part of the Table input object), which manages an SQL SELECT verb:


Software Construction Set
File Edit Smalltalk Workspace Program Component
AutoMessage
Choose A Message From This API
Input Object:
Vertical and Horizontal Slices
All >>
Copy >
Delete
Del. All
Show Result
OK
Cancel
Your Menu
Accept
Order
Remove Selected item

Name Space: Italian Takeout Menu   Data Set: Item

Vertical Slice

Columns available:	Columns chosen for output:
Data_Type	Data_Type
Category	Category
Item	Item
Sort	Sort
UnitPrice	UnitPrice
Description	Description

☐ No duplicates

Sort order: Up Down [✓ None]

Horizontal Slice

Take Slice? [✓ Do] Don't

Relation: [✓ =] < > (none)

Compare to: [✓ Use Sink Input] ☑ Add: " ... " Use This Expression:

Compare Column: Data_Type, Category (selected)

SELECT * FROM Item WHERE Category="Salads";

Functions,
Functions,
Messager
Persistent
Sources-D
U.I.-Brows

Entrees
Hot Subs
Pasta
Salads
Wraps

Deluxe Garden Salad
Mixed Tuna Salad
Greek Salad
Caesar Salad
Chicken Kabob Caesar

Romaine lettuce with garlic croutons and shaved Romano

rice: 3.95 Enter Qty: 4
em Total: $15.80

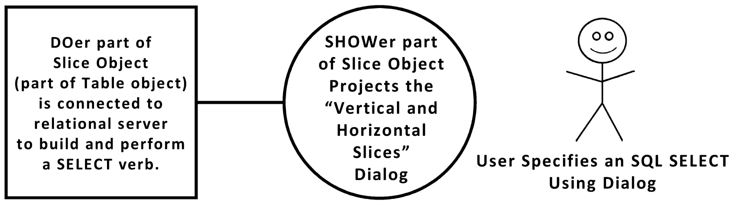
```
eal Parmigiana          11.95  111   $1326.45
heese Ravioli            6.95    3     $20.85
aesar Salad              3.95    4     $15.80
```

rder Total: $1363.10

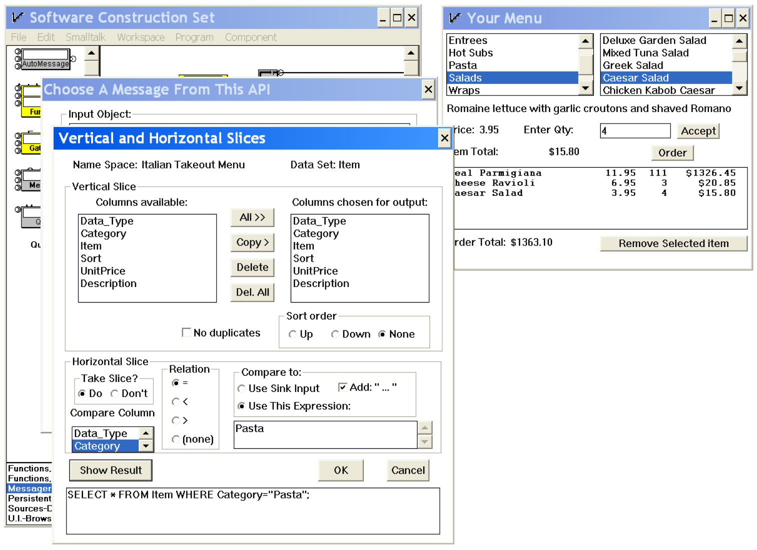
Working Paper No. 20

As of 7/31/2019

This is yet another application of the DOer, SHOWer pattern:


DOer part of Slice Object (part of Table object) is connected to relational server to build and perform a SELECT verb.
SHOWer part of Slice Object Projects the "Vertical and Horizontal Slices" Dialog
User Specifies an SQL SELECT Using Dialog

Now, in order to reassure ourselves that we really are in touch with what is going on, we decide to break the connection to the selected category input and force "Pasta" as the category in the SELECT. We do this by clicking the "Use this expression" radio button and typing "Pasta" into the text box. We can examine the resulting SQL by clicking the "Show Result" button:


Software Construction Set
File Edit Smalltalk Workspace Program Component
AutoMessage
Choose A Message From This API
Input Object:
Vertical and Horizontal Slices
All >>
Copy >
Delete
Del. All
Show Result
OK
Cancel
Your Menu
Accept
Order
Remove Selected item

Name Space: Italian Takeout Menu   Data Set: Item

Vertical Slice

Columns available:	Columns chosen for output:
Data_Type	Data_Type
Category	Category
Item	Item
Sort	Sort
UnitPrice	UnitPrice
Description	Description

No duplicates: ☐

Sort order: Up Down [✓ None]

Horizontal Slice

Take Slice? [✓ Do] Don't

Compare Column: Data_Type, [✓ Category]

Relation: [✓ =] < > (none)

Compare to: Use Sink Input [✓ Use This Expression:] Add: " ... " ✓

Pasta

```
SELECT * FROM Item WHERE Category="Pasta";
```

Entrees
Hot Subs
Pasta
Salads
Wraps

Deluxe Garden Salad
Mixed Tuna Salad
Greek Salad
Caesar Salad
Chicken Kabob Caesar

Romaine lettuce with garlic croutons and shaved Romano

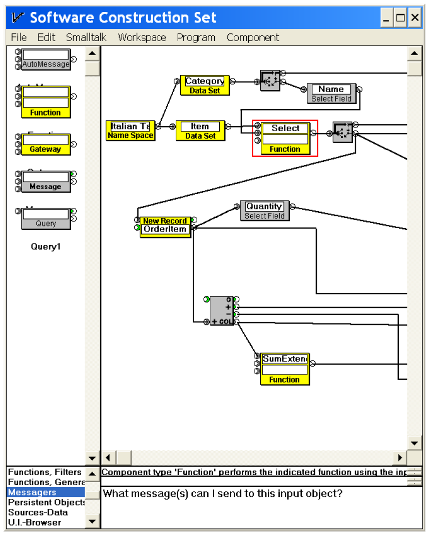
rice: 3.95   Enter Qty: 4

em Total: $15.80

```
eal Parmigiana          11.95   111   $1326.45
heese Ravioli            6.95     3     $20.85
aesar Salad              3.95     4     $15.80
```

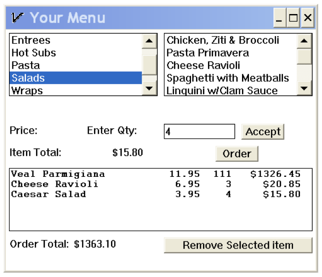
rder Total: $1363.10

We click two OK buttons and see that, indeed, the "Items" list box shows Pasta items in spite of "Salads" being selected in the "Categories" list box.[12]



Component type 'Function' performs the indicated function using the in

What message(s) can I send to this input object?



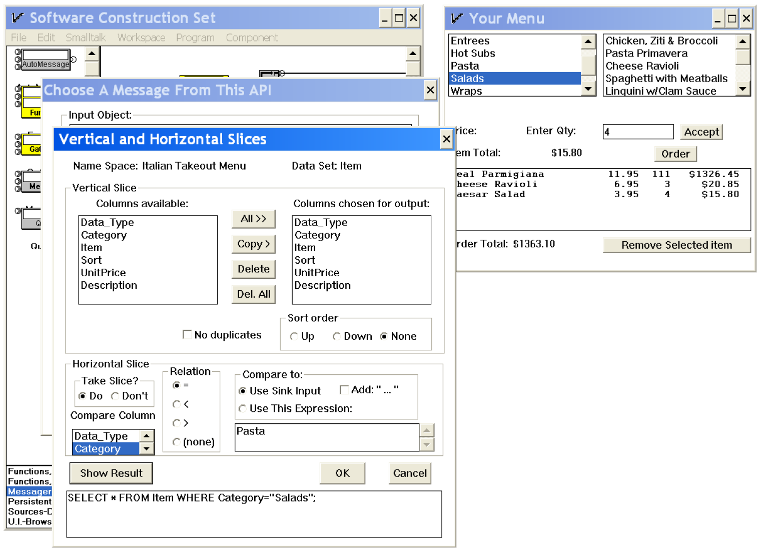
Price: Enter Qty: 4

Item Total: $15.80

```
Veal Parmigiana    11.95   111   $1326.45
Cheese Ravioli      6.95     3     $20.85
Caesar Salad        3.95     4     $15.80
```

Order Total: $1363.10

Now we re-select the "Use Sink Input" radio button, and click "Show Result" again:



Input Object:

Name Space: Italian Takeout Menu   Data Set: Item

Vertical Slice

Columns available:	Columns chosen for output:
Data_Type	Data_Type
Category	Category
Item	Item
Sort	Sort
UnitPrice	UnitPrice
Description	Description

☐ No duplicates

Sort order: ( ) Up ( ) Down [✓ None]

Horizontal Slice

Take Slice? [✓ Do] ( ) Don't

Relation: [✓ =] ( ) < ( ) > ( ) (none)

Compare Column: Data_Type, [✓ Category]

Compare to: [✓ Use Sink Input] ☐ Add: " ... " ( ) Use This Expression:

Pasta

SELECT * FROM Item WHERE Category="Salads";

rice: Enter Qty: 4

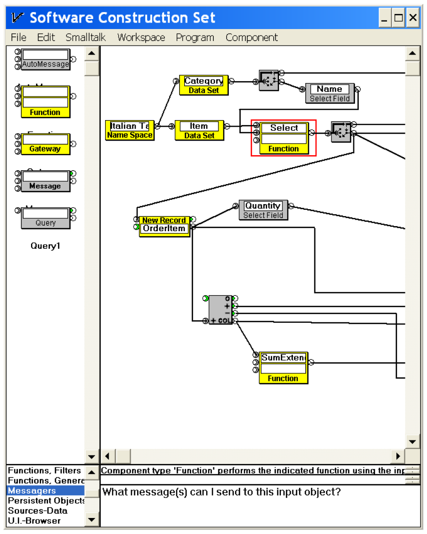
em Total: $15.80

```
eal Parmigiana    11.95   111   $1326.45
heese Ravioli      6.95     3     $20.85
aesar Salad        3.95     4     $15.80
```

rder Total: $1363.10

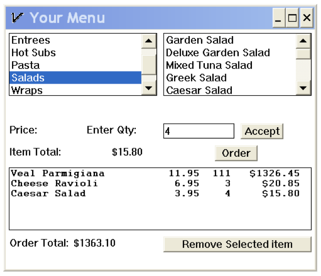
[12] The "4" in the Quantity box is a bug.

Finally we click two OK buttons and we see that the Salad Items show up in the Items list box:



Component type 'Function' performs the indicated function using the inp

What message(s) can I send to this input object?



Entrees	Garden Salad
Hot Subs	Deluxe Garden Salad
Pasta	Mixed Tuna Salad
Salads	Greek Salad
Wraps	Caesar Salad

Price: Enter Qty: 4

Item Total: $15.80

```
Veal Parmigiana        11.95  111   $1326.45
Cheese Ravioli          6.95    3     $20.85
Caesar Salad            3.95    4     $15.80
```

Order Total: $1363.10

Pattern Example 6: Immediate-Turnaround WYSIWYG Development Tools (wTools)

Definition

In *Immediate-Turnaround Development* every change to the source program is immediately reflected in the behavior of the object program. At the very least we need a very fast compiler or an alternative to the edit-compile-link-run-debug cycle.

In *WYSIWYG development* we further reduce the cognitive load on the developer with this requirement:

> Eliminate the distinction in the artisan's mind between an *executable* object language and a *readable* source language. Therefore no debugger will be necessary to translate between the two, and no distractions will arise from the need to manage the correspondences between expressions in two languages.

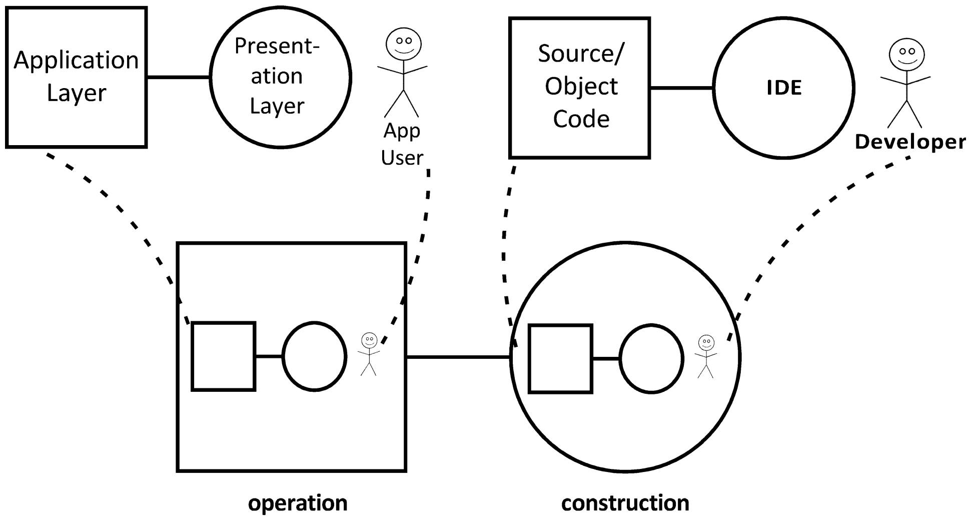
Here we apply just about everything we have seen above in order to address the question:

> How do you build an Immediate-Turnaround WYSIWYG Development tool?

For the sake of brevity, we'll call it a wTool. We need a name for the Application Under Construction, which we'll call the App.

Multiple Applications of the Pattern

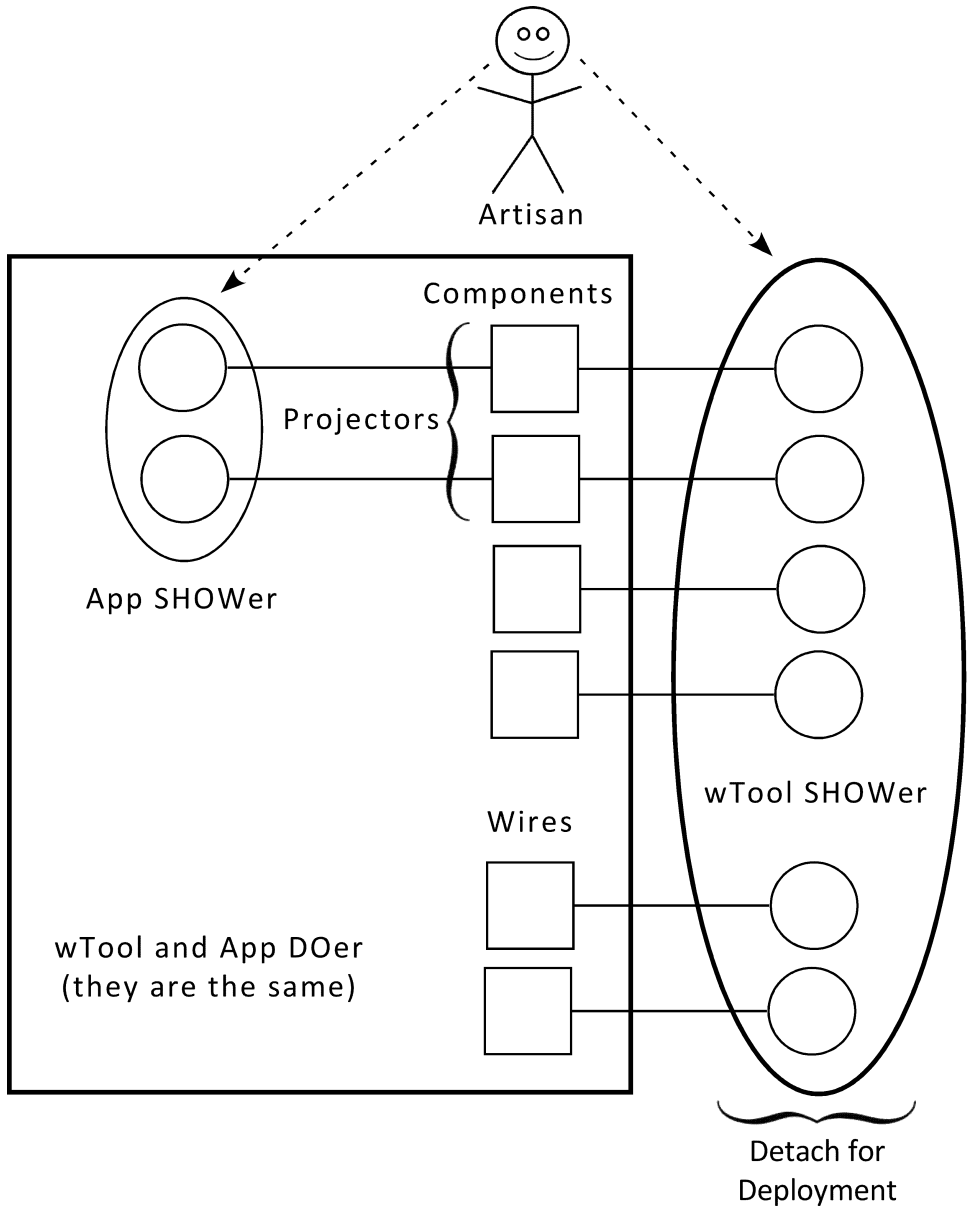
Let's return to Pattern Example 3, which describes the conventional application-development life cycle:



How does a wTool differ?

1. The Operation and Construction phase of the life cycle (namely, the DOer and SHOWer parts of the wTool) *are running concurrently*.
2. As you have seen many times above, their SHOWers are projecting their results side-by-side.
3. The User and Developer are the same person; we'll call her the Artisan.

How to represent this? This is an interesting demonstration of the expressive power of the pattern. It also relies heavily on *Humane Dozen #8, Isomorphic*.



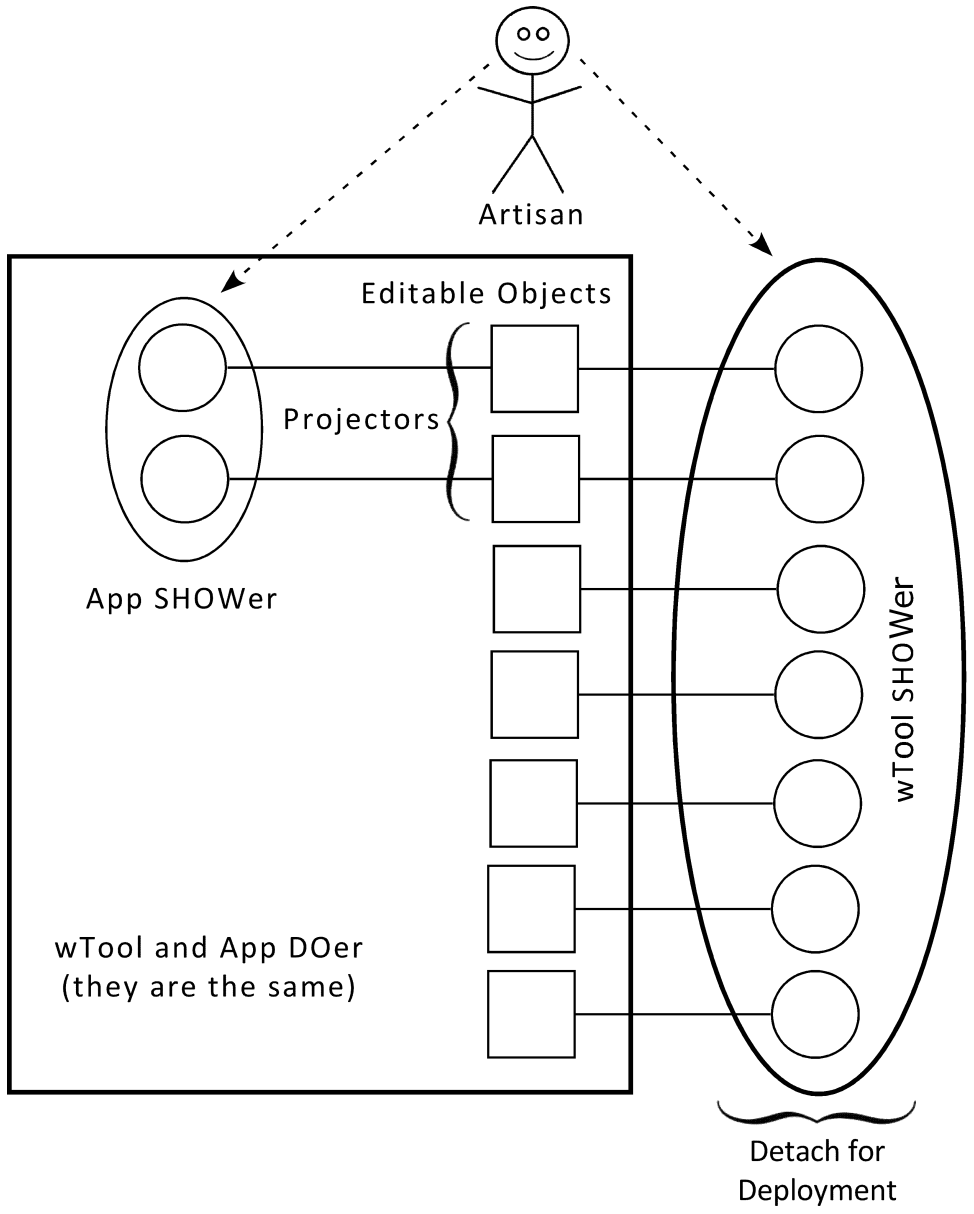
You can see these elements in the walk-throughs.

- The objects of the executing application (that is, the App DOer) that are reflected in the wTool's UI (that is, the wTool SHOWer) that have behaviors in response to Artisan events in the wTool's UI are Components and Wires. These are the elements of the projection of the App in the wTool's UI for which atomic editing gestures exist.
    - A new component instance can be dragged out from a component palette.
    - A showing components can be deleted, after all its connecting wires are removed.
    - A showing wire can be deleted.
    - A new wire instance can be created (subject to compatibility constraints embodied in the components) by dragging between a source and a sink connector of two different showing components.
- Completion of any of these gestures triggers a change in the App DOer, which can generate a shower (the other kind) of internal events. Those four events and their responses define the semantics of the wTool/Artisan interaction.
- Note that, at the risk of oversimplification, deployment means detaching the Tool SHOWer, and maintenance means re-attaching it.[13]
- The semantics of Artisan events in the App SHOWer are defined by the respective projector components.

[13] It's not so simple with Smalltalk, where hidden dependencies across the detachment surface need to be attended to.

General Conjecture On wTool Design

I conjecture that wTools in general can be built according to a general form of this design; we remove mention of components and wires and refer generally to objects in the executing App that have editable projections in the wTool's SHOWer:



Appendix: Evaluation of the Wiring Tool's Fidelity to the *Humane Dozen*

Here is the Humane Dozen with comments about the applicability of the wiring language and tool.

> Keep in mind that the Humane Dozen applies to interactive event-driven applications, which, except for a very brief time after a user event, are doing nothing, waiting for the next user event.

The workspace is a graphical canvas that allows drag-and-drop drawing and editing of wiring diagrams.

1. **Immediate** "Every modification made to the working material is immediately seen in its behavior." True.
2. **Continuous** "Small changes lead to predictable outcomes." This is an imprecise notion but is true in spirit.
3. **Interactive** "The result of each change suggests the next change, like a child playing with a construction toy." This has been my experience. I believe that this property has implications for development style; see the discussion of workflow on page 6 of [[14]].
4. **Transparent** "The tool seems invisible and the artisan's hands seem to be directly on the working material." False, but it must be maintained as a goal for WYSIWYG tools such as page-layout applications.
5. **Inspectable** "The behavior of all parts of the application can be inspected at any time." True, in part. There are big opportunities here.
6. **Modifiable** "The artisan can change his or her work in midstream." Not strictly true when working with components that have previously been packaged from wiring diagrams. Only the top-level workspace is currently modifiable.
7. **Robust** "Every atomic gesture of the artisan that changes the working material is reversible and will not break the system." True and

[14] http://melconway.com/Working/WP_19.pdf

significant. The four atomic gestures are: insert and remove a wire, and insert and remove a component.

8. **Isomorphic** “The executable form of the application is isomorphic to the working material.” True. I had to invent a wiring-diagram-shaped execution model.
9. **Self-revealing** “Interfaces *present* (and, if necessary, explain) *choices*; they don’t require formal constructions.” How true this is will depend on the faithfulness of implementations.
10. **Symmetrical** “The tool and the application being built are *side-by-side peers*, running at the same time with consistent state. Your next user-interface action can be on either.” True and significant.
11. **Always on** “There is no concept of starting or stopping applications or components during construction. When a component is introduced onto the workspace it is running.” True and significant. This was not true initially; it took years to learn this lesson.
12. **Alive with your data** “You can see the flow of data through the application moment by moment.” True and significant. It implies a different style of development, in which you have data present from the start, even if it is fake.