Working Paper No. 10

As of 2/22/2018

The Self-revealing Message Pattern

(This is a design working paper. It is a work in progress that is a byproduct of a larger effort to simplify application development.)

The purpose of the self-revealing message is to enable an application builder to specify sending a message to an object, while knowing very little of that object's API. Its realization is vital to achieving the goals of humanizing application building.

The *self-revealing message* is a wrapper of the receiving object that standardizes, and provides needed information for, sending messages to the object.

A *self-revealing API* is a collection of self-revealing messages. The way to send a message to an object is to ask it for its self-revealing API, and then choose a self-revealing message from the collection returned from this request.

This paper describes a Smalltalk implementation of self-revealing APIs and messages being built for the author's wiring tool. Although the possibility is not discussed here, the pattern might be applicable in more conventional coding contexts.

Table of Contents

1 Example Application Description 2
- 1.1 Application Windows 2
- 1.2 Wiring Diagram 3
- 1.3 Theory of Operation 4

2 Informal Description of Self-revealing Messages and APIs 6
- 2.1 What is a Self-revealing Message? 6
- 2.2 What is a Self-revealing API? 6

3 Preliminary Specification 9
- 3.1 Justification for the Preliminary Specification 9
- 3.2 Specification Time: What Is a Self-revealing API? 11
- 3.3 Execution Time: What is a Self-revealing Message? 12
- 3.4 Instance Variables of Every Self-revealing Message 13
- 3.5 Class Diagrams 14
- 3.6 How Is a Self-revealing Message Associated With a Class? 16
- 3.7 Secondary Message Specification 16

Twitter: @conways_law

Date of pdf: 2/22/2018
conway.mel@gmail.com

Working Paper No. 10

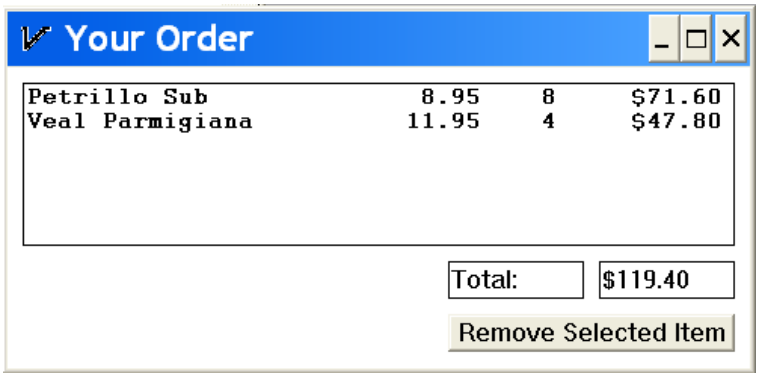
As of 2/22/2018

1. Example Application Description

1.1 Application Windows

The application I'll use in the discussion below is a simple restaurant menu order entry application fragment.[1] It has two windows.

- The "Choose an Item" window. It displays the menu by category and item within category. Once the user chooses an item, she sees the unit price and description. She can enter a quantity and then click the "Accept" button. This calculates and displays the "Extended Price" (unit price times quantity). When the user clicks the "Order" button the item is added to the shopping cart.
- The "Your Order" window. This is the shopping cart. It displays the list of ordered items and the sum of the extended prices of all items in the list. There is a button for removing a selected item from the shopping cart.



The data for the application is in a Microsoft Access relational database. It is transparently converted to internal objects for processing by the wired components. There are two tables: Category (6 rows) and Item (56 rows). A third internal table, OrderItem, is displayed in the Your Order window. Each of its rows is derived from an Item row by adding Quantity and Extended Price fields.

[1] Beginning at http://melconway.com/Home/prototyping/024.html you can see the construction of this application in the wiring tool. In the process you will see multiple illustrations of the self-revealing design principle.

Twitter: @conways_law

conway.mel@gmail.com

1.2 Wiring Diagram

This program was built, and runs, in the wiring tool, which conforms to eleven of the twelve humane design principles[2] (Undo is missing, due to the haphazard history of the research project). Below is an exploded view of the program's wiring diagram showing the top level and the insides of the three composite components. (This graphic was produced by combining screen shots from the wiring tool.) As you examine the wiring, keep in mind that data flows from left to right over the wires, from database to user interface.

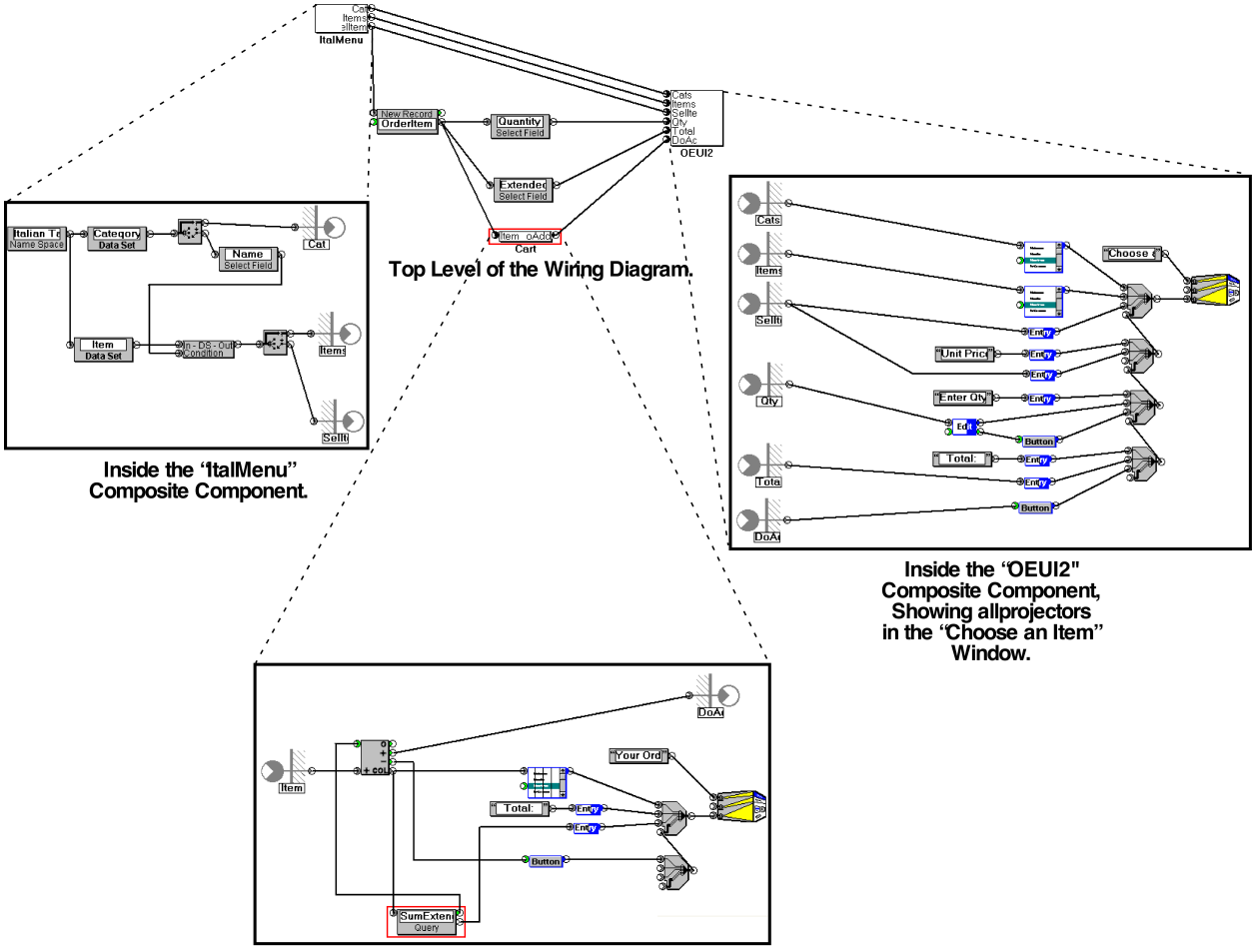


Figure 1

[2] http://melconway.com/Home/pdf/humanedozen.pdf

Twitter: @conways_law

conway.mel@gmail.com

1.3 Theory of Operation

As is explained elsewhere[3], all data flows are left-to-right, and event handling is inherited and need not be specified by the developer. The top-level wiring diagram has three composite components:

- ItalMenu at the left encapsulates the database access. The three source connectors are the category list, the list of items in the selected category, and the selected item in this list. These outputs connect directly to their projectors in OEUI2. It's worth noting that there is only one component (aside from transparent connector components), the square Selector component, between each component that sources a table/view and its list box projector. This Selector is what responds to selection events in its list box projector.
- OEUI2 at the right contains all the projectors of the "Choose an Item" window. The component at the extreme right projects the window frame; the others project subwindows. At the top are the two list box projectors for the category and item lists. The third sink connector component, named "SelItem", connects to two text projectors, for item description and unit price. What flows into these projectors are records. Before their encapsulation in the composite component, the projectors are parameterized to select the appropriate fields for display and to determine the location and size of the projection rectangle, and this paramerization persists in the composite component. The "Edit" projector projects the Quantity field of the OrderItem record. Note its short connection to the "Accept" button. When the Do-It sourced by the Edit projector is picked by this button, the Edit projector assigns its value back to the owner of the Quantity field, which is the New Record component in the top-level wiring diagram. *All behaviors in response to user events derive from message pathways that are inherited and/or determined by wiring connections, and*

[3] http://melconway.com/Home/pdf/pattern.pdf

Twitter: @conways_law

conway.mel@gmail.com

do not need to be specified by the application builder.

- The top-level wiring diagram connects together the three composite components and contains three primitive components for deriving a new OrderItem record from the selected Item record, and for isolating the Quantity and ExtendedPrice fields of this record.
- Cart in the middle contains the projectors for the "Your Order" window and two components of special interest that will be discussed below: the collection manager component that holds the collection of OrderItem records, and the Query message component (in the red box) that sends a message to this collection for computing the total of the ExtendedPrice fields.

There are two behaviors that make the wiring diagram algorithm-free. My thesis elsewhere[4] is that this ability to partition the application in a way that opens up a substantial code-free portion creates an opportunity for domain experts on the design team to contribute substantially to prototyping.

- Each OrderItem record is an object that, when an assignment is made by the Edit component to the Quantity field, computes and assigns the ExtendedPrice.
- The collection of OrderItem records is an object that responds to the SumExtendedPrice message sent by the Query message component. *The subject of this paper is generalization of this messaging behavior*.

[4] http://melconway.com/Home/prototyping/020.html

conway.mel@gmail.com

2. Informal Description of Self-revealing Messages and APIs

2.1 What is a Self-revealing Message?

The purpose of the self-revealing message is to enable an application builder to specify sending a message to an object, while knowing very little of the formal requirements of that object's API.

A *self-revealing message* is an object (not a message) associated with another object, called its *receiver*. Its function is to send to the receiver a message specific to the receiver and to this self-revealing message object. Let's call this message to the receiver the *designated message*; it is built into the self-revealing message object.

A self-revealing message may be seen as *a wrapper of the receiver object that standardizes, and provides needed information for, its simplified use*.

2.2 What is a Self-revealing API?

A *self-revealing API* is a collection of zero or more self-revealing messages. *Conventional objects or services can be wrapped by self-revealing APIs in order to obtain the benefits to the application builder of the self-revealing attribute*.[5]

Figure 2 below represents graphically a self-revealing API with one or more self-revealing messages.[6]

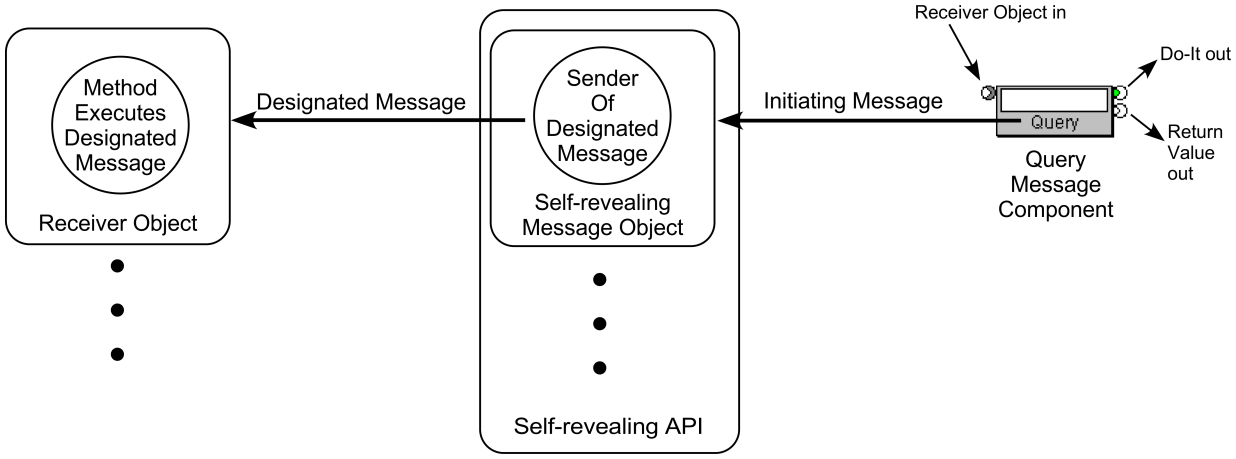


Figure 2

[5] http://melconway.com/Home/pdf/humanedozen.pdf page 3.

[6] Note that every self-revealing message has exactly one receiver object, but that receiver object can be the receiver of more than one self-revealing message (not necessarily in the same self-revealing API).

Twitter: @conways_law

conway.mel@gmail.com

In practice, when the application builder using the wiring tool wants to discover the API of an object and then send a message in that API to that object, this is done with a message component (here labeled Query), with the receiver object in its top sink connector. (Some message components have additional sinks for arguments.) The application builder asks the Query component to display the receiving object's self-revealing API, and then makes a selection from the list of presented self-revealing messages.

In the current realization of this concept the projection of the self-revealing API is in the list box in the "Choose A Message…" dialog in the following annotated screen shot, Figure 3.

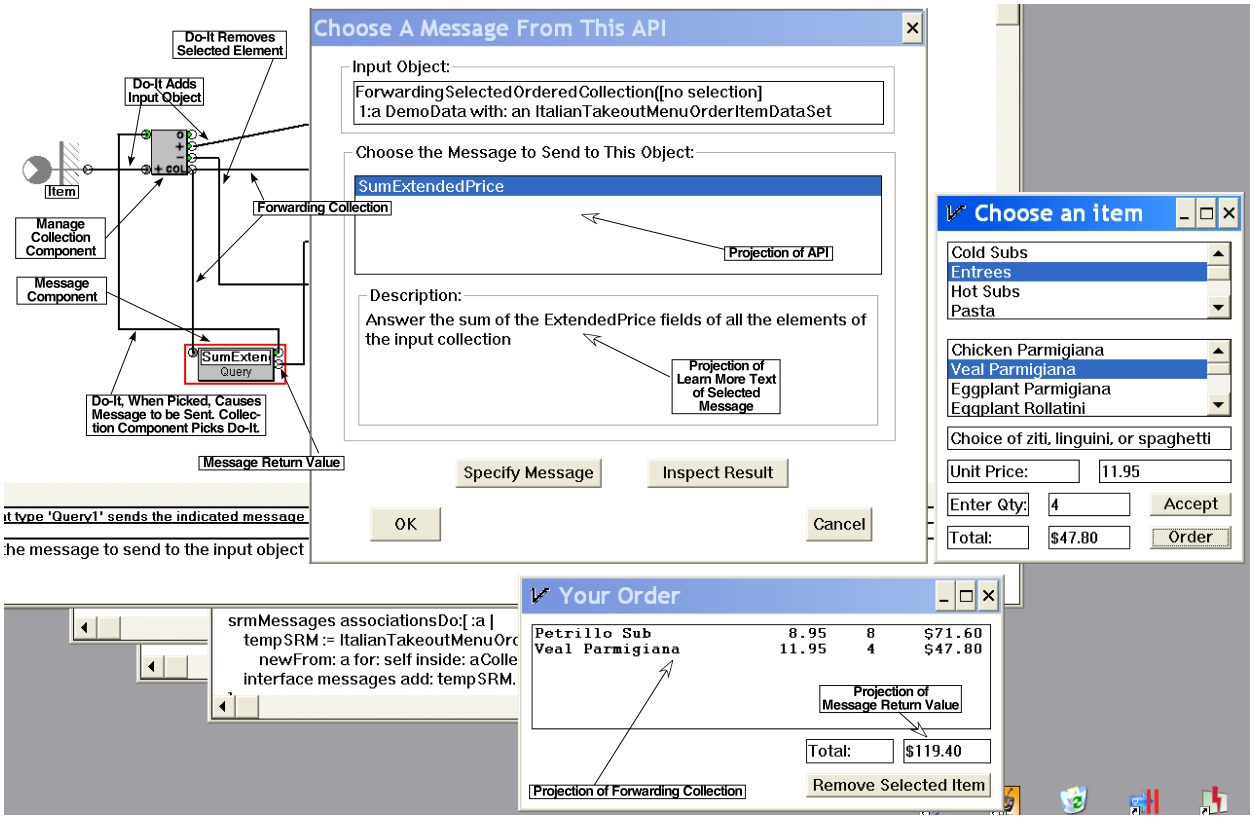


Figure 3

In Figure 3 there is, from left to right, (1)a fragment of a wiring diagram, including a Query messsage component (enclosed in red, indicating that it is selected) and a collection manager component above it, (2)covering the rest of the wiring diagram, the "Choose A Message" dialog that is opened by the selected Query component on receiving a signal from the wiring tool, (3)The "Your Order" application window (serving as a shopping cart), and (4) a "Choose an Item" application window, from which items are chosen and then sent to the shopping cart after a quantity is entered and accepted.

Note that there are two different times in this story: *specification* time and *execution* time (corresponding roughly to "build" time and "run" time).

Specification time: Using the wiring tool the application builder selects the Query message component and asks it to show the self-revealing API of its input object (in Figure 2 a collection of two items)[7]. The "Choose A Message" dialog that opens contains a projection of the API of this input object in its list box. In this case this self-revealing API has only one self-revealing message, whose function is to add up the "ExtendedPrice" fields of each record in the collection. Selecting a message reveals additional information in the Description box. By clicking OK the application builder is specifying that *at execution time* the Query message component is to cause the designated message to be sent to the message component's input object (i.e., the receiver object). The message component remembers this *chosen* self-revealing message. The designated message is not sent to the receiver object at this time.

Execution time: In the application the designated message is sent to the receiver object when the message component receives a signal to do so, namely when the Do-It at the message component's top source connector is picked by the collection manager component. The collection manager does this whenever its collection changes in any way. (The vertical wire conveying the Do-It from the message

[7] The input object of a message component can be any object. (By default, the self-revealing API of an object without a specifically designed self-revealing API is an empty collection.)

Twitter: @conways_law

conway.mel@gmail.com

component to the collection manager component is shown in Figure 3.)

When it receives this signal the Query message component sends its standardized *initiating message* to the chosen self-revealing message it has remembered. The effect of the initiating message on the chosen self-revealing message object is to cause it to send the designated message to its receiver. The message component then pushes the return value of the designated message out the message component's second source connector. In this case the application wiring (shown in Figure 1) causes the result to appear in the "Your Order" application window as the $56.55 total.

3. Preliminary Specification

3.1 Justification for the Preliminary Specification

The preceding describes the first-generation message component I built to demonstrate a shopping cart total calculation[8] at MuCon London 2017[9]. The preliminary specification below describes a second-generation design that I am building for Craft Conference 2018[10]. It embodies a more general approach to a design for self-revealing APIs and messages.

This first-generation message component has three issues that require that I split the second-generation design into multiple cases. (In what follows I am going to use "SRM" for "self-revealing message" and "SRAPI" for "self-revealing API"). Here are the three issues.

Query Messages and Command Messages. In addition to an attempt to support wiring diagrams that conform to CQRS the following opportunity is raised by the Humane Dozen philosophy[11]. In the dialog that presents a list of SRMs (see the "Choose A Message" dialog in Figure 2) you can see an "Inspect Result" button that sends the chosen message, then opens a Smalltalk inspector window on its return value. Such a button enables testing a message without engaging the downstream portions of the wiring diagam. For this Choose A

8 http://melconway.com/Home/prototyping/031.html

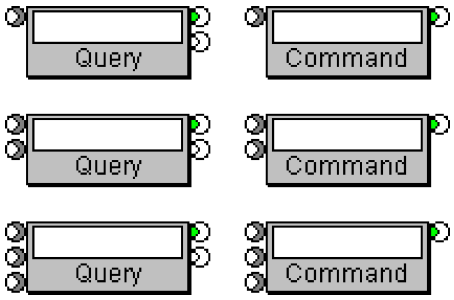
9 https://skillsmatter.com/conferences/8549-con-2017-the-microservices-conference

10 https://craft-conf.com

11 http://melconway.com/Home/pdf/humanedozen.pdf

Twitter: @conways_law

conway.mel@gmail.com



Message/Inspect Result combination to work deterministically it should have no side effects. I don't have a general approach to this but a partial approach is to split messages into query and command messages, with their corresponding Query and Command wired components. (Their images appear to the left.) The Command component would not have a result value and would not have the corresponding source connector. The Query component (ideally, although I do not know how to enforce this without a global Undo) would guarantee the absence of side effects.

Dependence on number of arguments. In Smalltalk the keyword held by the SRM that denotes the designated message must specify the exact number of arguments present. The person doing the wiring will be partially protected from this implementation detail by being required to choose a component with the number of sink connectors matching the number of input wires. The coding of the receiver must still check that the arguments are present.

The solution used here, also used by the Smalltalk **`perform:`** method, is to use a designated message keyword with exactly one argument that is an array containing the actual argument values. Then the coding of the receiver must also check the size and content of the array. The benefit of this approach is that it puts a bound on the size of the **`SelfRevealingMessage`** subclass tree, described below.

Whether or not the receiver of the SRM (the input to the message component) is a collection. In the wiring diagram at the left of Figure 3 the collection manager component assigns only a generic type to the collection that appears at the input of the Message component. How does the message component find the SRAPI of its input object without some type information? To require that the collection manager assign a type to its collection so that the message component will be able to find its SRAPI would introduce an undesirable coupling between these two components.

Twitter: @conways_law

conway.mel@gmail.com

In the case of a message to a collection whose function is to iterate over the collection, the solution is to use the first element of the collection as a proxy of the whole collection.[12] This requires the assumption that all elements of the collection are interchangeable in this respect, which is assured in Figures 1 and 3, since the collection is built of elements of the same type. The solution is to have the collection manager component create and own a collection of the generic class **`ForwardingCollection`**, which forwards the **`mySRAPI`** messages that request the collection's SRAPI to the first element of the collection. **`ForwardingCollection`** is a subclass of **`SelectedOrderedCollection`**[13].

3.2 Specification Time: What Is a Self-revealing API?

`SRAPI` is an instantiating subclass of **`SelectedOrderedCollection`**. The elements of the collection are self-revealing message objects, which will be described in the next section.

Every object must respond to these three messages: **`mySRAPI`**, **`mySRQAPI`**, and **`mySRCAPI`** with a self-revealing API object return value, which is an instance of the class **`SRAPI`**. **`mySRAPI`** returns a collection of all instances in the object's API. **`mySRQAPI`** returns the subset containing only those instances of **`SRQuery`**. **`mySRCAPI`** returns the subset containing only those instances of **`SRCommand`**.

There are three cases of interest:

1. The default return value of **`mySRAPI`**, **`mySRQAPI`**, and **`mySRCAPI`** is an empty collection.
2. When **`mySRAPI`** is sent to a nonempty **`ForwardingCollection`** the **`ForwardingCollection`** sends **`mySRAPIinside:self`** to its first element. (This

---

[12] This is a partial solution and has the smell of a hack, but will have to stand until deeper insight appears.

[13] In Smalltalk a **`SelectedOrderedCollection`** is a subclass of **`OrderedCollection`** with an additional member/instance variable that carries selection information. (List boxes are projections of **`SelectedOrderedCollection`**s; the selection information is typically projected as a distinct coloration of one or more lines of the list.) An **`OrderedCollection`** is a growable, shrinkable, integer-indexed linear collection.

Twitter: @conways_law

conway.mel@gmail.com

sends the **ForwardingCollection** to its first element so it can iterate over it.) Similarly, when **mySRQAPI** is sent to a nonempty **ForwardingCollection** the **ForwardingCollection** sends **mySRQAPIinside:self** to its first element. Also, when **mySRCAPI** is sent to a nonempty **ForwardingCollection** the **ForwardingCollection** sends **mySRCAPIinside:self** to its first element. In all cases the return value is returned back to the original sender. Thus, any object that can find itself inside a **ForwardingCollection** might be required to respond to **mySRAPIinside:**, **mySRQAPIinside:**, and **mySRCAPIinside:**. This is how the object acts as a proxy for the collection.

3. The **SRAPI** class has a constructor that returns new instances that can be populated; this constructor is called by the constructor of any new object that must return a nonempty response characteristic of itself to **mySRAPI**, **mySRQAPI**, or **mySRCAPI**.

3.3 Execution Time: What is a Self-revealing Message?

A self-revealing message is a descendant of the abstract class **SelfRevealingMessage**. The instance variables inherited by all its descendants will be described in the next section.

A Query or Command Message component that is ready to send a self-revealing message to its input object remembers the parts of this self-revealing message that will be used at execution time in a parameter. This parameter is set by the "Choose a Message" dialog, when a self-revealing message in its list box is chosen and OK is pressed.

Under **SelfRevealingMessage** are two instantiating classes: **SRQuery** and **SRCommand**. Every self-revealing message is an instance of one or the other.

- Instances of **SRQuery** respond to the initiating message **queryWith:<argumentArray>** and provide a return value. Their implementations should not create side effects. They might also be required to respond to the message **queryWith:<argumentArray>inside<coll>** ; in this case their implementation would perform some iteration over the collection **<coll>**. Note

that this latter message is not an initiating message; it is sent by the collection **`coll`**, not by a wired component.

- Instances of **`SRCommand`** respond to the initiating message **`commandWith:<argumentArray>`** . They return no meaningful value. They might also respond to the message **`commandWith:<argumentArray>inside<coll>`** ; in this case their implementation would perform some iteration over the collection **`<coll>`**. (Since no value is returned this seems that it would not normally occur.) Note that this latter message is not an initiating message; it is sent by the collection **`coll`**, not by a wired component.

3.4 Instance Variables of Every Self-revealing Message

Here are the instance variables common to all self-revealing messages in the Smalltalk implementation.

`receiver:` The Receiver Object of Figure 2.

`publicSelector:` A text string that appears in the list box. It should be descriptive, one short line long, and can contain spaces.

`publicLabel:` A very short text string that appears in the Message component icon.

`privateMessage:` The Smalltalk keyword symbol of the Designated Message (see Figure 2). The self-revealing message has the job of checking and mapping the elements of **`<argumentArray>`** of the initiating message into the arguments of the designated message. It will be built into a **`perform:`** message.

`learnMoreText:` An expanded text string of unspecified length. It shows up in the Description area of the "Choose A Message" dialog when its public selector is selected in the list box. It can contain description, operational details, instructions on usage, etc. Its purpose is maximally to simplify the work of the application builder.

`learnMoreObject:` Not yet defined. It might correspond to the "learn more" link on web pages; it might even open up a web page.

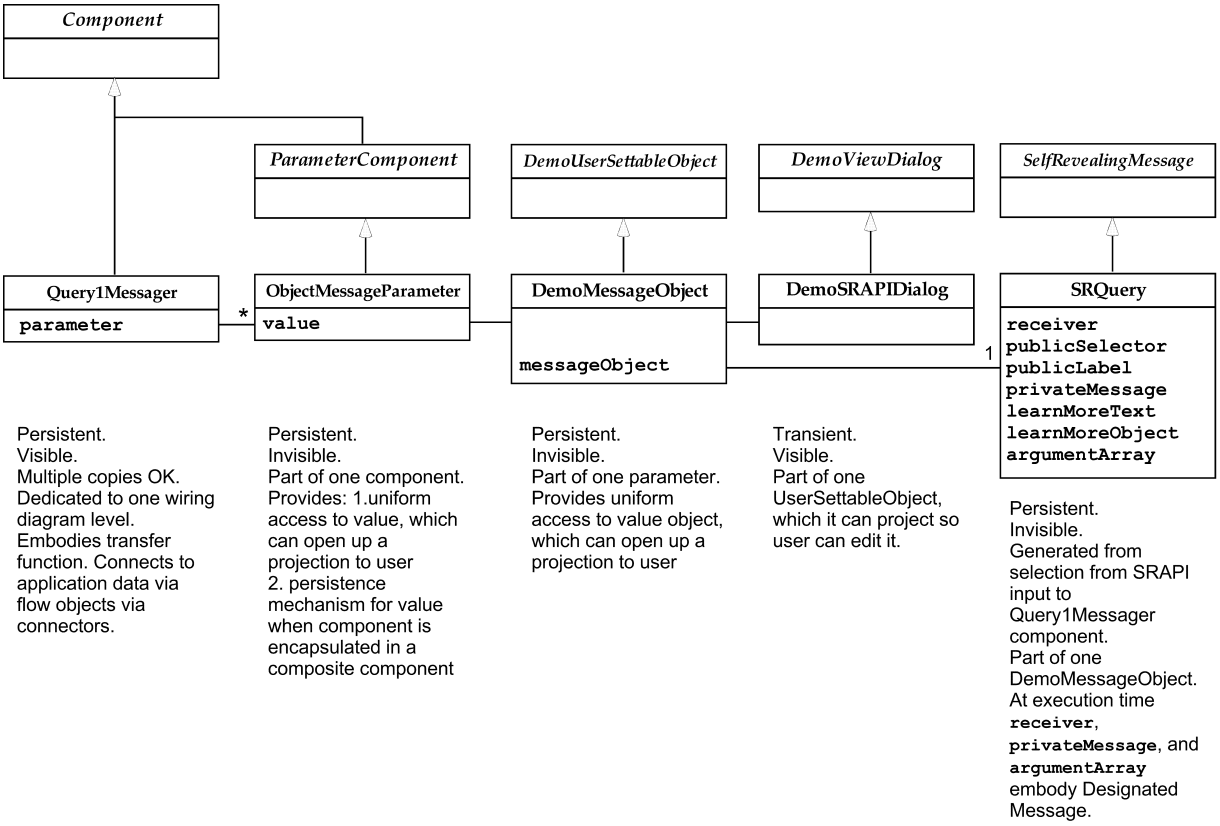
Twitter: @conways_law

conway.mel@gmail.com

`argumentArray:` This is passed on with the **`perform:`** message.

3.5 Class Diagrams

Here are the classes active at specification time.[14]



This diagram is typical for almost all wired components. The additional part is the SelfRevealingMessage object that occurs only with Query and Command components. To review:

- Visible wired components are the objects whose projections appear in the tool as component icons.[15]
- Each component typically has one or more hidden parameters. For example, a button

---

[14] Given the "Always on" property, specification time and execution time coexist. The distinction is only one of user focus.

[15] http://melconway.com/Home/pdf/pattern.pdf describes other object associated with components that connect to data and other components: connectors and flow objects.

projector might have two: a rectangle specifying its size and position, and a text string specifying the text appearing on the button's projection. In this implementation parameters are implemented as hidden wired compoennts.

- These parameters are wrappers for value objects, which can be of any type. These value objects are instances of subclasses of DemoUserSettableObject, which gives them all the ability to open a dialog on the object to permit the user to edit its values.
- Each DemoUserSettableObject descendant has a paired DemoViewDialog descendant, which has its own Windows dialog resource that defines the form of the dialog and the correspondence between the dialog controls and the instance variables of the value object.

The **ObjectMessageParameter** is a hidden component associated (one-to-one) with each Query and Command component. (Unlike other non-parameter wired components) it has a **`value`** which, in the case of Message components, is a **DemoMessageObject**.) **DemoMessageObject**, along with every other descendant of **DemoUserSettableObject**, has a dedicated sibling dialog class, which is a subclass of **DemoViewDialog**. Each **ParameterComponent** descendant can open the respective dialog as a projector of its respective **`value`**. This is how parameters are set. Once the "Choose a Message" dialog has chosen a **SelfRevealingMessage**, the parts of the **SelfRevealingMessage** necessary for execution are stored in the **ObjectMessageParameter**. These parts are embodied in a simpler object called a **SelfRevealingExecution**. The instance variables of a **SelfRevealingExecution** are **`receiver`**, **`publicLabel`**, **`privateMessage`**, and **`argumentArray`**, which are given the same values as in the originating **SelfRevealingMessage**.

3.6 How Is a Self-revealing Message Associated With a Class?

Every object must respond to **`mySRAPI`**, **`mySRQAPI`**, and **`mySRCAPI`**. Most don't have anything interesting to offer. The default value is an empty **`SRAPI`**, which is returned by **`Object`**.

Each classes whose instances must respond specifically to the my **`mySRAPI`**, **`mySRQAPI`**, and **`mySRCAPI`** message is given methods that look like this.

```
mySRQAPI

    "Answer the self-revealing query API of the receiver"
    | collection query |
    collection := SRAPI new.
    query := (SRQuery
        newFor: self
        publicSelector: 'Select'
        publicLabel: 'Select'
        privateMessage: #select:
        learnMoreText: 'As with an SQL SELECT statement,
answer a subset of the rows and columns of the input object'
        arguments: Array new).
    query parameter1: SliceObject new.
    collection add: query. "Repeat as necessary"
    ^collection
```

The SliceObject in parameter1 will project into the "Vertical and Horizontal Slices" dialog shown below, which allows the user to specify the parameters of the SELECT statement (or its functional equivalent). This dialog will open when the "Specify Message" button in the "Choose a Message" dialog is clicked.

3.7 Secondary Message Specification

In many cases the Choose A Message dialog is sufficient for completely specifying a message. This is the case in the example application at the beginning of this paper. Other messages require a secondary level of specification. For example, if a DataSet (see the "ItalMenu" composite component wiring diagram in Figure 1) is the input to a Query component one of the messages that can be sent to this object wll perform the equivalent of an SQLSelect. The following figure shows a Query component (selected, in red) to whose input an "Item" DataSet object has just been wired. In this case the "Specify Message" button has then been

Date of pdf: 2/22/2018
conway.mel@gmail.com

clicked, which opens up a "Slice" dialog that permits the user to specify the variables of a Select. So you see the initial appearance of the Slice dialog on top of the Choose A Message dialog.