Working Draft

Wire Up Your Prototypes

This two-part paper proposes a role in application prototyping for a code-free wiring-model language and its hands-on development tool.

Part I: History. Part I summarizes a half century of research into programming simplicity and boils it all down to dozen attributes of a simplified developer-friendly workflow. Part I also links to a video demonstrating most of these attributes.

Part II: Opportunity. I believe that this technology can reduce the communication barrier between the development team and the client business by enabling business-side team members to build quick-turnaround, code-free prototypes that they share with their colleagues. Part II describes the opportunity technically, and it contains an illustrative video.

Twitter: @conways_law

September 10, 2017
conway.mel@gmail.com

Part I: History

Scope

This is about a code-free language for building important parts of interactive graphical user-interface (GUI) applications.

Part IA: Thinking Like a Computer Scientist

IBM RPG

In 1959 IBM introduced a tool, called Report Program Generator (RPG), for converting their customers' blue-collar labor force of punched-card machine operators into computer programmers. It was a brilliant solution, enabling most of a utility billing application, for example, to be written on a preprinted multi-column data-entry form with almost no algorithmic information.

Static is Good

Languages like RPG that are specialized for a particular class of applications and directed primarily to nonprogrammers are different from programming languages. I call them *application languages*.

> **The purpose of an application language is not to *express* algorithms but to *hide* them.**

The spreadsheet is the classic example.

Partitioning the Universe of Application Languages

RPG was a specific instance of this general principle:

> **The universe of all applications can be partitioned into classes according to their underlying algorithms.**

To build an application language for a particular class you put the class's underlying algorithm into the application's runtime and you devise a quasi-static parameterization of this algorithm as the application language.

So far I've found a few of these:

> **Linear files:** There is a loop that examines one record at a time until the end of the file. Each record might be compared to a current record in another file (sorting/merging) or to a stored data structure obtained from a previous record (billing/totaling). The treatment of one record, including what is written, is the basis of the application language. RPG and many early linear-file reporting languages fall into this category.

Twitter: @conways_law

conway.mel@gmail.com

Querying relational databases: SQL is almost an algorithm-free language. Query-by-example[1] is an even better illustration of the principle.

GUI apps: The dispatching event loop at the heart of a GUI application is the underlying algorithm. Early Visual Basic was an elegant example of an application language; the UI was drawn on the screen and a simple script handled each event.

Constraint languages: Most of the time a GUI application just sits there doing nothing; then a user event occurs and something has to happen. The flow machine I'll show you seeks a new equilibrium of a network of wired components after an equilibrium has been disturbed by a user event. The spreadsheet falls into the constraint-language category.

Flow Diagrams

Around 1992 I settled on the flow, or wiring, diagram[2] as the best simple application language for building GUI applications. Many groups were working with variants of this applicaiton model at the time[3,4,5].

I was looking for usable alternatives to conventional programming of real, not toy, GUI applications. That required scalability, which implied abstraction and multi-level reuse. In a wiring language abstraction looks like encapsulating a whole wiring diagram into a single component. The interface that would be created by the abstraction/encapsulation process (that is, the set of connectors on the outside of the new component) had to be as free of constraints as possible in order for the new component to be widely reusable.

The flow languages I was examining had problems this way, because the components that were created by the abstraction process had too many constraints on the resulting interface connectors.

1. Some schemes were object-oriented programming turned into flow diagrams. This required separate types of wires, and therefore

---

1 https://en.wikipedia.org/wiki/Query_by_Example
2 https://en.wikipedia.org/wiki/Flow_diagram
3 An early inspiration was http://www.ni.com/academic/students/learn-labview/graphical-programming/
4 https://msdn.microsoft.com/en-us/library/bb483088.aspx
5 http://sp.cs.msu.su/courses/smalltalk/Fabrik/Fabrik.html

Twitter: @conways_law

conway.mel@gmail.com

separate types of connectors, for messages and for objects (the message parameters).

2. In the more dataflow-oriented models, events and data flowed in opposite directions, so either there were bidirectional flow paths (an implementation nightmare) or there were distinct data paths and event paths.

After a while I settled on a hybrid unidirectional flow model.

1. There were only data paths. By convention they were left-to-right, and user-interface components were at the right end of the flow diagram. There were only two connector types: sinks (on the left of each component) and sources (on the right of each component).
2. User events originating at the UI produced not a flow but an implicit four-message-hop *update protocol*: two hops from UI to the changed data and two hops back to each UI view of the changed data.

There were several useful ideas that showed up along the way.

1. A *projector* component is a UI component that renders its input data onto a region of the UI. Often these are the components that have to handle user events. The list box as a projector of a linear collection is an example.
2. A special data type, which I now call *Do-It*, has projections that receive simple user events such as mouse clicks. Do-It projectors render buttons and menu items.
   The Do-It combined with the update protocol has eliminated the need for retrograde flow of user events.
3. All application data is wrapped by a wrapper called a *flow object*; thus there is one interface to data seen by all components. The role of the wrapper as an intermediary is the reason that the update protocol has two hops instead of one in each direction; the payoff for this is excellent decoupling between components.

Twitter: @conways_law

conway.mel@gmail.com

Part IB: Thinking Like an Anthropologist

The Goal of Universal Understanding

Some time around 2006 I decided that *this application flow model had to be so simple that every normal person would be able to understand it*, and possibly even build applications using it. That meant that, like arithmetic, writing, and calendar, the application model would evolve historically from being the monopoly property of a priest class (us) to being taught in elementary school.

I had to stop thinking:	And start thinking:
About technological particulars	About human universals
Like a computer scientist	Like an anthropologist

I taught myself to think about early man chipping away at a stone hand-axe 100,000 years ago as a predecessor in a continuous chain of human tool-building, leading up to a programmer writing a computer application today.

The epiphany came to me around 2010, as I was watching a grandson in his high chair struggling to grasp a Cheerio and put it in his mouth. I was amazed at his persistence, and I realized that I was witnessing the execution of a program built into every one of us, one that we execute relentlessly for years beginning with infancy:

> **Build the hand-eye-brain system**
> **by interacting with the environment.**

Every person is born with a built-in do-it-yourself project to build the brain through experience.[6] This was the human universal I was looking for.

> **In order to be universally comprehensible the application conceptual model must harness the massive investment Nature has made in the hand-eye-brain system of every human.**

The second part of the conceptual refaming came from abandoning what we as computer people take so for granted that it's invisible: the input-process-output construction model. I came upon *the potter at her*

[6] Dr. Michael Merzenich speaking on the brain as a machine that builds itself:
https://www.youtube.com/watch?v=UyPrL0cmJRs

wheel with her hands on the artifact, as a better example of a craft that exploits the hand-eye-brain system. The new construction model, ***transform-in-place***, replaces input-process-output.

> **The application builder metaphorically throws a lump of clay onto the wheel and, in stages, gradually reshapes it into the desired artifact.**

Clearly, the application conceptual model and the construction tool must be thought of together.

> **At every stage of transform-in-place the artisan has continuous feedback about how much has been done and how much more needs to be done.**

THE OLD WAY	**THE NEW WAY**
What the artisan does	
Keyboarding into a translator	Hands on the working material
The construction model	
Input-process-output	Transform-in-place

I found eleven attributes of a *hands-on software tool*, that I now regard as a reference model for thinking about humane application development. To this I added a twelfth: working with real data during development. These are *the twelve attributes of a modeless workflow*, seen on the next page. Most are present in the first demonstration video, described below. The first five attributes refer specifically to modelessness of the development process. The remaining seven attributes characterize hands-on construction.

Twitter: @conways_law

conway.mel@gmail.com

The Twelve Attributes of a Modeless Workflow

The "Modeless" Part (minimizes mental gear-shifting):

1. **Unified**. *The thing being manipulated and the end product are in the same conceptual domain*. In programming terms, the source and object languages are indistinguishable. A corollary necessary in order to simplify the overall process and eliminate debugger glitches: *The executing application being built is isomorphic to what is in the artisan's hands. There is no compiler that translates a wiring diagram into a model-view-controller application.*
2. **Symmetrical**. *The transition between "Build" and "Run" is modeless*. The tool and the application being built are peers. The artisan's next move can be on the user interface of either one or the other.
3. **Alive with actual data**. *The artisan is not asked to alternate attention between building and testing*. The working material exists with real data present; the effect on the appearance to the user of a change to either working material or application data is seen or can be examined immediately. (See 6 and 10.)
4. **Syntactically undemanding**. *The artisan is shown enough information to select among self-explanatory choices*. Nowhere is there a requirement for text input according to a formal grammar.
5. **Always on**. *During construction there is no concept of "starting the application"*. When a component instance is created in the workspace of the tool, it is already running, and it continues to behave according to its definition. To change an application, you don't stop it, fix it, then start it; you just fix it. (See 6.)

The "Hands-on" Part (simulates the potter's experience):

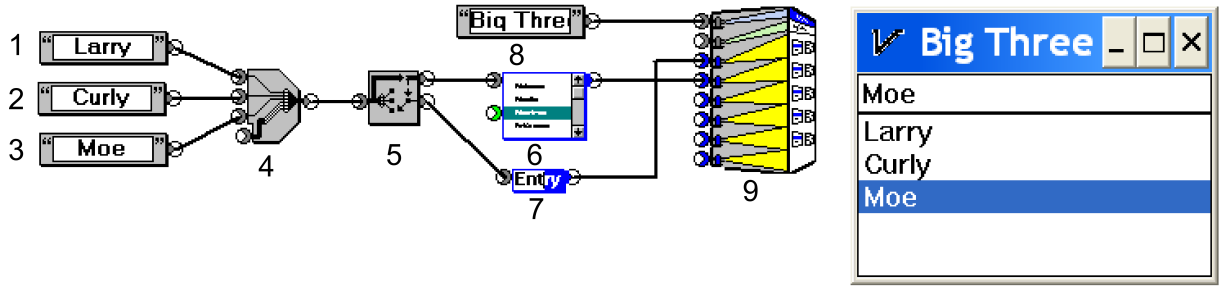
6. **Immediate**. *Every modification the artisan makes to the working material is immediately seen in its behavior*. There is no perceptible delay introduced by a translation phase.
7. **Continuous**. *From one step to the next there is obvious continuity in the working material's behavior*. Of course, software is severely nonlinear, but we can adapt the mathematical definition of continuity as follows: Small changes lead to predictable outcomes.
8. **Interactive**. *The result of each change helps to suggest the next change.* The artisan's brain is unconsciously engaged with the working material, like a child playing with a construction toy. (See 6.)
9. **Transparent**. *The tool supports the illusion that it is invisible and the artisan's hands are directly on the working material.* Metaphorically, the working material is embedded in the hand-eye-brain feedback loop. Given existing human-machine interface hardware, building a tool to create such a suspension of disbelief is a challenge, but we have examples pointing the way, such as some page-layout applications and spreadsheets.
10. **Inspectable**. *At any time all parts of the application can be inspected and the values so obtained can in turn be inspected*. (Keep in mind that an event-driven application is almost always doing nothing, except when it is briefly responding to a user event.)
11. **Intervenable**. *The artisan can modify any part of the application* (provided that doing so does not contradict the definition of an existing component used in the application).
12. **Reversible**. *A good UNDO means no regrets*.

Twitter: @conways_law

conway.mel@gmail.com

Part IC: Demonstration Video 1

Here is a screen shot of the wiring tool showing a version of the program that I'll be building in Video 1. (I've added component numbers for this discussion.) The wiring diagram (the "working material") is at the left side and the user interface of the artifact (the program being built) is at the right. In accordance with attribute 2 above ("Symmetrical") both the tool and the artifact are running at the same time.



This simple program illustrates the automatic coupling between components 5 and 6 that implements list box behavior. The user interface window is built by three projectors: component 9 projects the window frame with its title bar, component 6 projects the list box, and component 7 projects the text line.

The list box component 6 projects the collection it receives at its top sink connector from component 5. This collection ["Larry","Curly","Moe"] is created by component 4 from its three text inputs (component 4's graphic is meant to suggest wrapping individual wires into a bundle). The selector component 5 (whose graphic is meant to suggest a rotary selector switch) sends its input collection out its top source connector to list box component 6 and sends the selected element out its bottom source connector to text line component 7. Component 7's projection is above the list box in the user interface window. "Moe" has just been clicked in the list box and it appears in the text line.

The main purpose of Video 1 is not to show the program but to show *an interactive style of building the program* (see attribute 8 above). As you watch it you will see a casual, almost experimental, approach to putting the pieces together, while having full knowledge at every stage of construction what data is

September 10, 2017
conway.mel@gmail.com

at key points of the wiring diagram, even before the whole program is connected together.

> The left-to-right flow convention puts the user interface projector components at the right of the wiring diagram and the data source components at the left. Given that the user is on the outside looking in, you can think of the right side as the "outside" and the left side as the "inside" of the program.
>
> In this dataflow application model we don't think so much of building "top-down" or "bottom-up" (that seems to be an artifact of procedural programming) but of building "outside-in" (right-to-left) and "inside-out" (left-to-right). In my experience, what seems to be most natural is building from both ends toward the middle. (You will see that here and again in Video 2.)

[Video 1 is in development. Here is an earlier version as a temporary placeholder. You might have to start each video manually.]

http://melconway.com/HumanizeTheCraft/Video_1/

You can learn how this program works in

http://melconway.com/Home/pdf/pattern.pdf .

Twitter: @conways_law

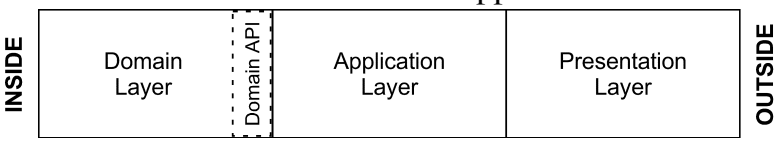
conway.mel@gmail.com

Part II: The Prototyping Opportunity

Part IIA: Identifying the Part of an Application Appropriate for Wiring

Assumptions

We assume the application structure shown here.

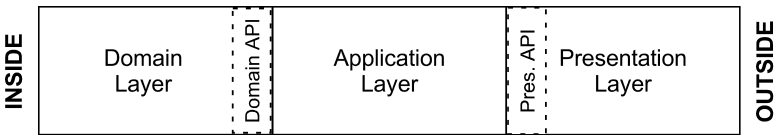


The Domain Layer contains all the domain objects, which embody knowledge and behavior of the system being modeled. In this model it includes persistent storage. Other parts of the infrastructure, such as networking, are ignored here. The presence of the Domain API in the figure indicates the assumption that there is a clean cleavage between the Application Layer and the Domain Layer. The Domain API presents a set of services that respond to requests initiated by the Application Layer.

The Presentation Layer controls appearance to the user, including accounting for the differences in multiple user agents such as browsers and mobile devices.

The Application Layer determines the way the software presents the application's use cases to the user. It makes service requests to the Domain Layer, and it calls on the Presentation Layer to manage multiple windows/screens and sequences of presentation, in response to user events.

This figure shows that the Presentation Layer has its own API, which offers presentation services to the Application Layer.



Twitter: @conways_law

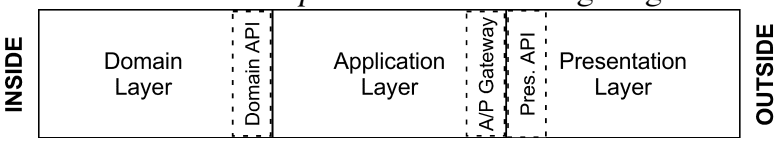
conway.mel@gmail.com

The Opportunities

- Language opportunity: ***The wiring diagram is a coding-free language for building the Application Layer.***
- Process opportunity: ***Domain-expert members of a developer/domain-expert design team can build their own application prototypes*****.** This can enhance communication between the design team and the client organization, which can accelerates the iterative process that will converge on an acceptable prototype.

Application/Presentation Gateway Components

If the Application Layer is a wiring diagram, communication with the Domain and Presentation APIs is embodied in *dedicated reusable "gateway" components* in the wiring diagram.



There is a set of Application/Presentation gateway components shown in Video 1 and described below. Video 2 shows a set of Application/Domain gateway database components.

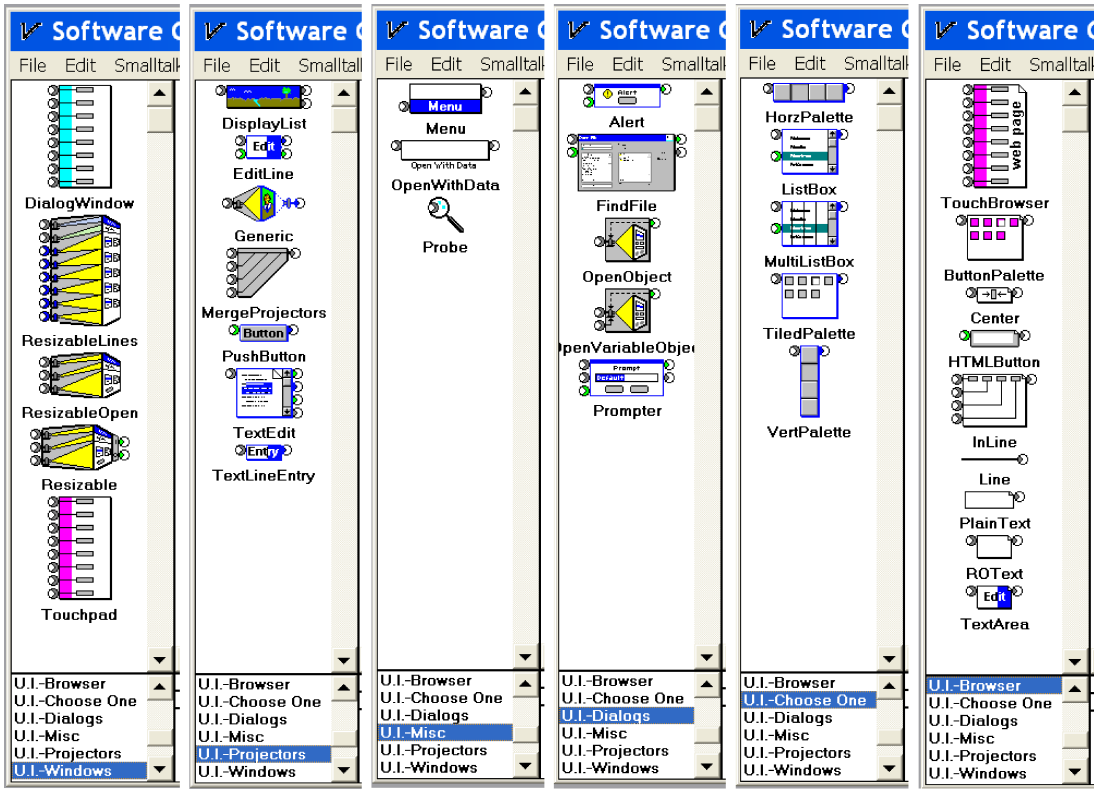
There are actually two Presentation Layers in the existing prototype, the Microsoft Windows user interface management system (UIMS), and a web-browser UIMS. (Ideally this distinction would be abstracted out and only one Presentation Layer would be present.)

> **The gateway presentation components that are the interfaces to the Windows UIMS are exactly the projector components shown in Video 1.**

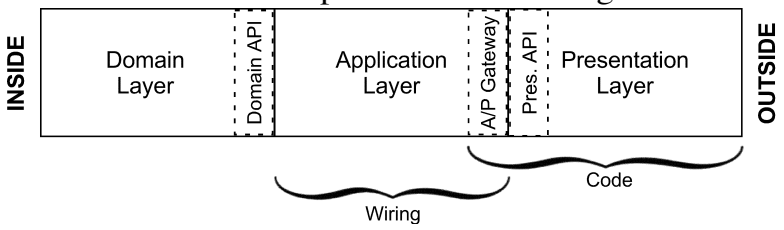
The Windows UIMS is accessed by five categories of primitive components (i.e., components whose bodies are built with code). The web projectors are less mature in the prototype and comprise only the rightmost component category. These Presentation-Layer gateway component categories are shown below. Note that all the Choose One projectors (including ButtonPalette) work interchangeably with the Selector component.

Twitter: @conways_law

conway.mel@gmail.com

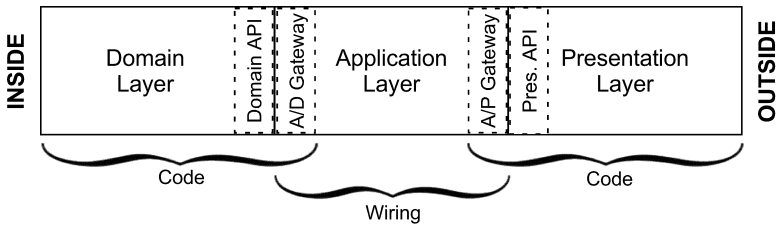


The overlapping curly braces in the following figure say that the Application/Presentation gateway components are built with code and that they appear as components in the wiring tool.



Application/Domain Gateway Components

Similarly, there is a layer of Application/Domain gateway components on the domain side of the Application Layer, as shown in the following figure.



Application/Domain gateway components send service requests to the Domain API, using the returned objects to compute the component outputs. (These service requests are sent as part of a component's computation in response to receipt of a `compute` message from a sink connector.[7])

[7] See Update Protocol in http://melconway.com/Home/pdf/pattern.pdf.

Part IIB: Demonstration Video 2

Video 2 is the restaurant order-entry applet I prepared for the DDD-Europe conference in February 2017. It is at

http://melconway.com/HumanizeTheCraft/DDD_Europe/025-were-going-to-build.html

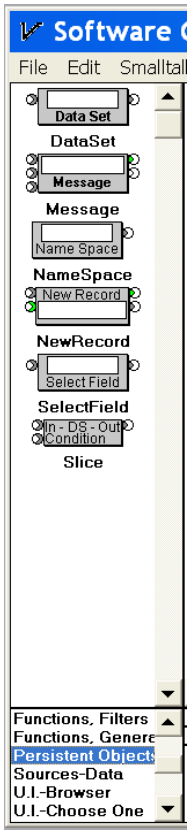
through

http://melconway.com/HumanizeTheCraft/DDD_Europe/038-complete-order-entry.html .

(You might have to start each of the four videos manually.)[8]

(This menu application was initially conceived and implemented on an iPad as part of a proof-of-concept mobile order-taker device.)

Object Database Gateway Components



In 1995 I attached the Microsoft Access relational database to the Smalltalk prototype. In 2016 I added an object-relational management (ORM) layer to Smalltalk as a wrapper of the database in order to have all the domain-dependent logic in the Domain Layer and none in the wired Application Layer. (There is one piece of domain-dependent logic in this video: computing total price from unit price and quantity; there is no arithmetic in the wiring diagram.)

I implemented the object database with the "Persistent Objects" category of gateway components, shown at the left. The object database schema is in two parts: the Microsoft Access relational schema and some Smalltalk classes, one for each record type in the relational schema. Each of these classes is a subclass of the class DemoDataSet, each of whose instances encapsulate one record. The wiring sees only these DemoDataSet subclass objects, and no relational database records. The DemoDataSet hierarchy specifies which record types can be derived from each type. For example, only OrderItem and Item can be derived from Item.

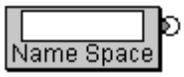
There is also a Smalltalk object called a "slice", which is the object-oriented equivalent of a table or view;

[8] This program contains a rule violation that has since been corrected. In private correspondence Jonathan Edwards pointed out that the use of a FieldOwner component violates one of my rules. I have since removed this component, and the current version of this application complies with the Flow Object Pattern paper.

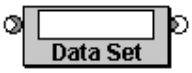
Twitter: @conways_law

conway.mel@gmail.com

slices act as collections of DemoDataSet subclass instances. Slices are instances of the Smalltalk class PersistentDataSet.

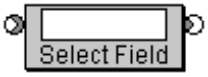


NameSpace is the name given to the relational database equivalent. This component sources the named NameSpace object.



DataSet is the name given to the relational table equivalent. This component sources the named PersistentDataSet object in the input NameSpace.

The NewRecord component creates and sources a new instance of a record that is derived from the input record, including its inherited instance variable values. The dialog from which this new record type is chosen permits selection only of the derived types permitted by the DemoDataSet hierarchy, i.e., by the object database schema.



The SelectField component sources the named instance variable of the input record. (Keep in mind that, in keeping with the way pure object systems are built, no values, only references, are present. The object that is sourced is a DemoFieldProxy object that encapsulates the name of the instance variable.)

The Slice component sinks and sources a PersistentDataSet object (i.e. a table or view), and sinks an optional condition expression. It implements an SQL SELECT statement specified in a dialog the artisan can open when the component is selected.

The Message component is the Joker in the set of Application/Domain gateway components. When the Do-It from the top source connector is picked the component sends the named message to the input domain object at the top sink connector, with one or two optional parameters. (My intent is to use it to add an OrderItem to, and remove an OrderItem from, a shopping cart list.) The dialog from which the message is chosen only offers as options the messages that can be received by the input object.

Twitter: @conways_law

conway.mel@gmail.com

How It Works

Of particular interest is the video at

http://melconway.com/HumanizeTheCraft/DDD_Europe/038-complete-order-entry.html

which is described in part below. It bears watching at least once, because it illustrates several Application/Domain gateway components in action. Note also that it illustrates many of the Modeless Workflow attributes.

The Category and Item classes are straightforward; their records are the immutable value objects in the restaurant menu. The members of the OrderItem class are the objects that will populate the shopping cart. They are mutable domain entities created by the application; each is derived from an instance of Item by the NewRecord Application/Domain gateway component shown here, whose specific purpose is creating new records of a type that may be derived from its input.

The top sink connector is receiving an existing Item record. Opening the dialog that presents the choices for the derived record type reveals only two classes: Item and OrderItem, because the object databse schema knows that these are the only two possibilities. (Watch this dialog in the video.) Choosing OrderItem creates a new instance of the OrderItem class with its inherited instance variable values and pushes it out the lower source connector.

The OrderItem instances are the line items that are to be added to the shopping cart (which has not yet been built). Each OrderItem instance has two instance variables of particular interest that are not in Item: Quantity and ExtendedPrice.

OrderItem instances are domain entities of interest that demonstrate the domain API because they perform extended (i.e., total) price computation as follows: storing into the Quantity instance variable causes the total price to be computed using the unit price value and puts the result in ExtendedPrice. This storing operation is performed by an editable text line projector that has no knowledge of the thing it's changing.[9] There is no arithmetic in the wiring language. Thus, domain-specific behavior (price computation) is kept in the Domain Layer and out of the Application Layer.

[9] This operation is very similar to the *Edit a String* use case on page 11 of http://melconway.com/Home/pdf/pattern.pdf

September 10, 2017
conway.mel@gmail.com

Part IIC: Extreme Prototyping

In this section I speculate about some possible consequences of adoption of this technology.

There are two independent variables that would affect these outcomes.

- The degree to which the technology is adopted, whether just for prototyping, or whether wiring technology is carried into building production applications. At this time, neither is true.
- The development practices of the organization using the technology, in particular how closely the development team communicates with the organization at large, and how closely the team members follow the Evans model[10] in which technical and domain experts are peers.

Development Process

By allowing the business representatives on the development team to build and experiment with meaningful prototypes the opportunity exists for a more dynamic communication between these representatives and their colleagues in the larger client organization. Given some process discipline, this can inject reality into the development at an earlier stage, shortening the development process.

Given a reasonable set of presentation services and some initial set of developmental domain objects, some of which might even start as mocks, prototype development can be fast and will tend to get ahead of domain object development. I see this as an opportunity for the design process as a whole, and domain object definition in particular, to become more experimental.

A hybrid technology such as this can divide the software into slow parts (coding) and fast parts (wiring). If this rhythm is allowed to play out, the fast parts will drive the definition of the slow parts. Because the presentation (projector) components are externally determined and will change little, the slow parts will be the services of the Domain Layer. Given full engagement by the technical and non-technical

---

[10] **Domain-Driven Design: Tackling Complexity in the Heart of Software**, Eric Evans, Addison Wesley 2003 ISBN: 0-321-12521-5

Twitter: @conways_law

conway.mel@gmail.com

sides of the development team this can lead to more of an outside-in (i.e., potentially user-driven) design process.

System Structure

The necessity that the Application Layer communicate with the Domain and Presentation Layers exclusively via wired gateway components imposes a discipline on the partitioning of the artifact whose violation would be difficult and obvious. If the wiring technology carries into production systems, this can have long-term benefits for system integrity.

My observation is that the only significant coupling between components seems to be through the domain objects they share. To the extent that this is a problem, it seems to be of the same nature as in object-oriented programming in general, and is addressed the same way, by careful domain-object design.

Twitter: @conways_law

conway.mel@gmail.com