Free the Collaboration Barrier

Mel Conway

1. DDD Prototyping: Extend the tech-business collaboration

> Plan to throw one away; you will, anyway.
>
> Frederick P. Brooks, Jr.
> "The Mythical Man-Month"

What is the Collaboration Barrier?

The collaboration barrier is the moment on the timeline of a business application development project at which the major contribution to the end result shifts decisively toward the software developers and away from the business members of the design team.

The collaboration barrier typically occurs when enough is understood about the domain model that coding in earnest can begin. At that point there is confidence that the likelihood of disruptive change of ideas has become acceptably low, so the ideas that have been captured in the domain model can begin to be frozen in code.

Figure 1 shows a purely qualitative model of how the technical people and the business people share participation in determining the final outcome as the project progresses. There is a kink to the left of the collaboration barrier. Before this kink the business people are educating the tech people about the business. After the kink they have developed their ubiquitous language and are describing the domain model fully collaboratively. To the right of the collaboration barrier the contribution of the business people to the final product becomes minimal.

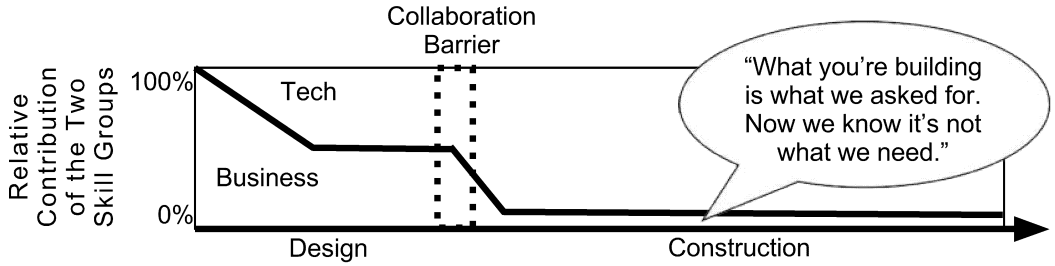

Figure 1

The contribution of business people drops when coding begins

conway.mel@gmail.com

Release candidate 1: 2018-10-19 12:48 PM

This can change if the team discovers that they must go back and revise the domain model. After the collaboration barrier the cost of actually going back and changing the domain model begins to rise, and it continues to rise as time progresses and more work must be undone. In other words, before the collaboration barrier the cost (in terms of ultimate project delay) of changing your mind is low and flat. After the collaboration barrier the cost typically rises monotonically. So after the collaboration barrier, if the realization occurs that there is something about the design that needs to change, the decision process may turn into a value tradeoff between product quality and project schedule.

Figure 2 shows, again qualitatively, the cost of change as time progresses. This cost is some combination of schedule delay and quality loss, depending on how the change is handled.

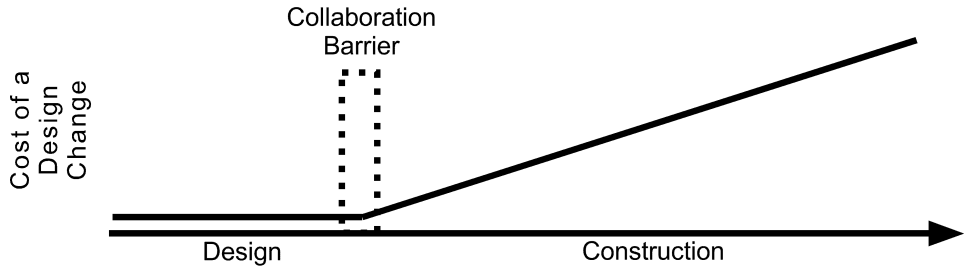

Figure 2

The more code that has been committed, the greater the cost of change

Obviously you want to get the design right before the collaboration barrier. The conundrum, in particular if this is a first implementation of the requirement, is that (to put it starkly) *you don't know what to build until you've built one*. IBM's experience building OS/360 led to Fred Brooks's lesson at the top.

conway.mel@gmail.com

The answer, then, is to build something cheaply and quickly that you can use in order to learn what you really need to build. I'm calling this thing a *prototype*. *The goal of a prototype should be to maximize learning at a controlled cost*.

My premise here is that, in order that this opportunity to learn be maximized, the business people must be fully engaged in building the prototype. If this is the case, the effect of introducing a prototyping stage into development is to shift the collaboration barrier to the right, as shown in Figure 3.

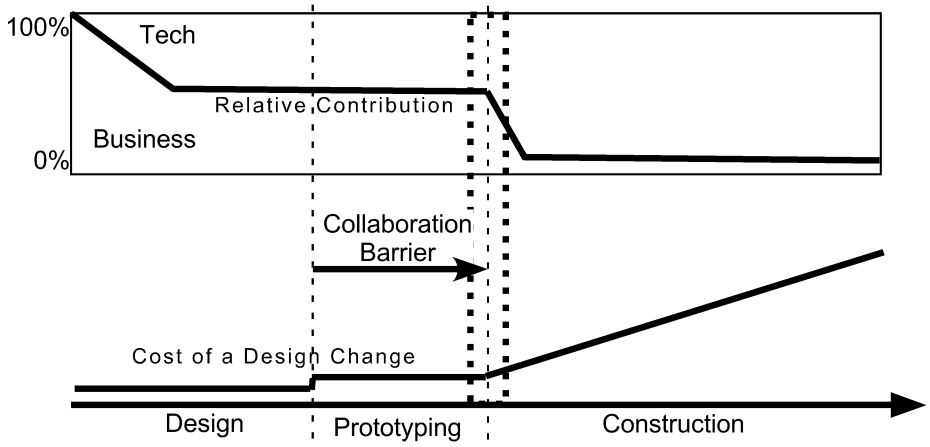

Figure 3

Engaging business people in prototyping moves the collaboration barrier to the right

If we accept that the prototype is a throwaway (and there is no reason at this point to doubt that) then the effort of building the prototype does not contribute to the final code. This effort is "lost", and it adds to the total cost of the project. What does this cost buy? To quote Brooks:

> "The management question, therefore, is not ***whether*** to build a pilot system and throw it away. You ***will*** do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers".[1]

I am taking the liberty of interpreting Brooks as follows:

> What building the prototype buys is avoiding having to scrap and rebuild the delivered system, or more optimistically, it buys building a more usable system the first time. What we have learned since Brooks is how to throw it away in little pieces instead of all at once.

Why can prototyping in which the business people are fully engaged lead to a more usable system? Because the business people, who are the only available prelease resources for measuring usability, are right there

conway.mel@gmail.com

participating in its construction. They are thinking about usability all the time.

So we have begged the question: how can we build the prototype in a way that the business people, who we assume are not programmers, are fully engaged in the process?

Partition the Work

The answer I propose here is to divide the prototype into a part built by the developers and a part built by the business people. Here are some requirements on this approach to dividing up the task.

- The business people must have control over use of the prototype so they can show it to their colleagues. The part they build must be lightweight in order to permit experimentation, so they can learn from experience and modify the design, and it must not require coding skills.
- The parts that the business and technical people build must correspond to the respective expertise and experience that these two groups bring to the table. Specifically, the business people should work at the level of use cases and the technical people should work at the level of domain objects.
- Change is inevitable as learning occurs. This learning should be reflected smoothly in the evolution of the prototype. (Ideally, as learning occurs, the prototype should evolve toward a small monolithic version of the ultimate system.)
- The sizes of the respective parts, the interface between them, and their relative rates of change must be such that the developers and business people can continue to work concurrently.

conway.mel@gmail.com

The Two-faced Prototype Model

The two-faced model (Figure 4) is an environment combining traditionally-built domain objects and pictorial use-case descriptions built by non-programmers. The Visual APIs enable these two parts to work together synergistically while maintaining the benefits of each.

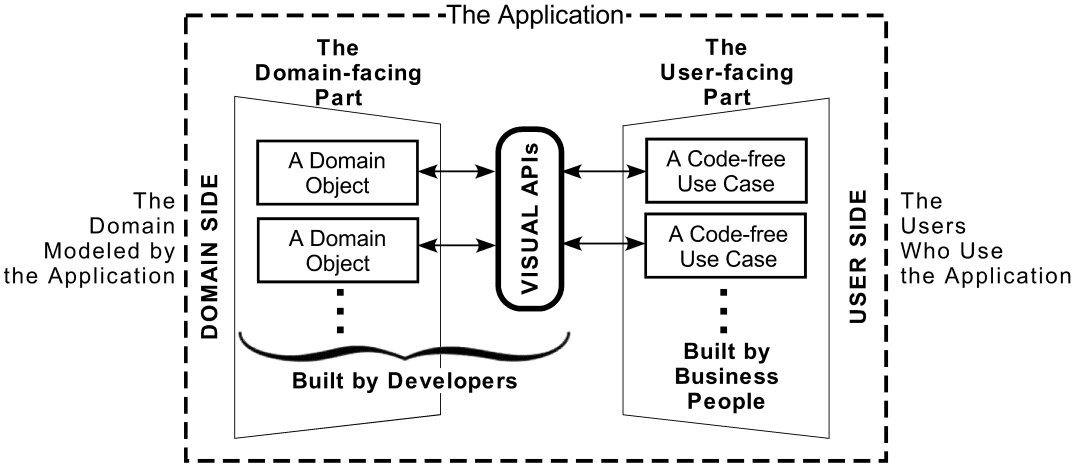

Figure 4
The two-faced prototype model

Code-free Use Cases

The code-free use cases execute value-delivering user interactions. They do more than simply describe a user interface because they communicate with domain objects, work with values derived directly from them, and respond appropriately to user events, possibly with state changes.

I have developed a unidirectional object-flow wiring language that seems to be sufficent for describing use cases according to the bulleted list of requirements above. I do not assert that it is the only solution, but it is an existence proof.

This wiring language is not powerful by itself, but it is seamlessly extended to encompass the business processes of the domain objects through the Visual APIs. Wiring is a pictorial, non-procedural flow language that doesn't do arithmetic or even the simplest algorithms involving looping or branching. It occupies a sweet spot between power and accessibility (that is, availability to non-programmers). It does not need to be made more complicated because, wherever more power is required such power is domain-specific, belongs in domain objects, and is accessible through Visual APIs, described in the following section.

conway.mel@gmail.com

Figure 5 illustrates how the wiring language fits into the power-accessibility tradeoff. It is capable of handling functional composition (by connecting wired components) and collections (business objects that flow on the wires), rendering user interfaces, responding to user events, and little else. It has been my observation that much of what happens in these wired use-case descriptions is assembling and disassembling collections. The objects that flow down the wires can be complex, for example, records whose elements have domain-specific behaviors.

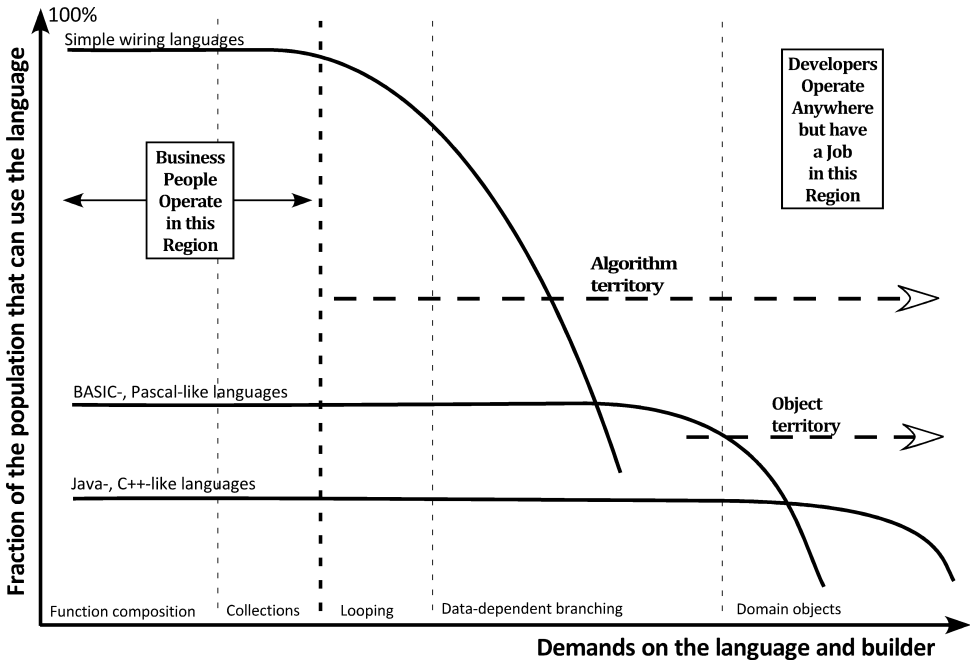

Figure 5

Simple wiring languages are less powerful (by themselves) but more accessible

Visual APIs

The Visual API replaces the demand on the builder to construct grammatical text by presentation of choices presented in dialog windows. It enables the business people to access domain objects and parameterize messages to them without leaving their comfort zones. These messages are sent by wired components, and their return values flow down the wires.

Figure 4 shows that Visual APIs are built by developers. The payoffs from this work are the ease of communication between the technical people and business people, and *extending the time that they are fully engaged further into the development process.* These are the business benefits from prototyping of the two-faced model.

conway.mel@gmail.com

2. Introducing the code-free API

This section combines multiple existing distinctions into one conceptual framework, in order to extend this framework. Figure 9 puts the concepts together in one place.

A. The Symmetry/Flexibility Distinction

Each API has two aspects.

- It is a ***formal specification*** of a request-response interface through which a ***client*** software component makes a request of a ***server*** software component.
- It is a ***contract*** to conform to the formal specification between two communities of developers. Developers in the ***producer role*** build conforming servers; developers in the ***consumer role*** build conforming clients. Somebody publishes the API; often, but not always, it is a producer.

Once the API is published and starts being used it might or might not be acceptable to change it, depending on the relationships between producers and consumers. This distinction divides APIs into two classes, *negotiable* and *fixed*; see Figure 6.

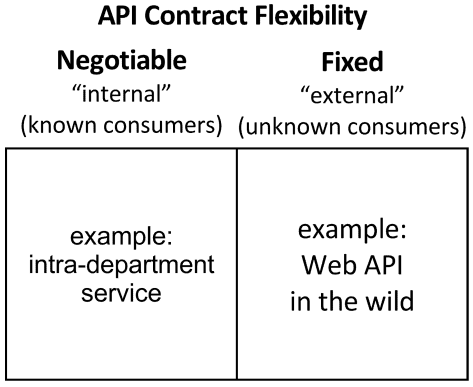

Figure 6

The API contract might be negotiable or fixed, once the API is in use

This corresponds to the distinctions in the general API literature. This literature almost always assumes that both producers and consumers are programmers, or at least can work with the technical details of the formal specification. However, the discusssion of DDD prototyping above contradicts this assumption because the consumers, the business people, are not usually programmers. So we need to turn this 1x2 into a 2x2; see Figure 7. The "symmetry" dimension of this array corresponds to the specification

conway.mel@gmail.com

aspect of the API; the "flexibility" dimension of the array corresponds to the contract aspect of the API.

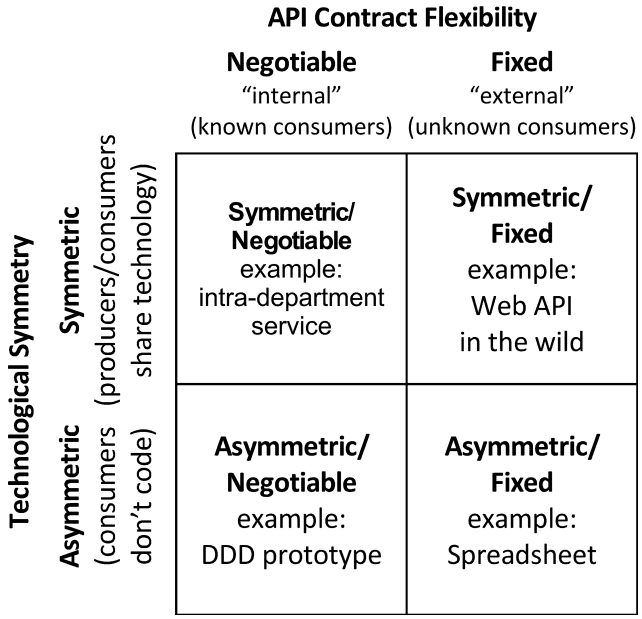

Figure 7

The Symmetry/Flexibility API diagram

Note that the spreadsheet, which has been recognized for a long time as some not-readily-classified species of programming language, falls into this classification scheme. A spreadsheet can be seen as an API, as follows:

- The consumer is the spreadsheet's user.
- The producer is the vendor, for example, Microsoft/Excel or Lotus/1-2-3.
- The formal specification is the *application model,* stated here approximately:
    - There is an expandable two-dimensional array of *cells*, addressable by row and column. Each cell contains an *entry*.
    - Each entry can be a literal or the value of a function whose definition is also part of the cell, but is usually invisible.
    - The arguments of the function are literals or cell references.
    - A request occurs every time the user changes an entry. The response is a network of constraint-resolution calculations that attempt to restore consistency among the entries, which have presumably been rendered inconsistent by the change. Sometimes this attempt will fail and an error message will appear.

conway.mel@gmail.com

Why mention the spreadsheet here? Because describing its application model will allow us later to segue to the two-faced model.

B. The Construction/Operation Distinction

There are two distinct stages in the development of any unit of software, the ***construction stage*** and the ***operation stage***. In order for us to understand fully what a Visual API is, these two stages need to be brought into the same conceptual framework and regarded as two aspects of the same thing: the ***life-cycle*** of that unit of software. For simplicity we limit this discussion to the development of a client-server interaction according to an API.

- During the construction stage the producers and the consumers each work with their respective tools building response code and request code, respectively, according to the contractual aspect of the API. Their tools build the respective codes to conform to the formal specification aspect of the API.
- During the operation stage the producers and consumers, and (usually) their tools, are not in the picture. Typically, the request code operates on some device and initiates an interprocess communication, which initiates the response code on some, possibly different, device.

The life-cycle/role diagram of Figure 8 captures this distinction for the Visual API. Notice that there is a new element: the Public VAPI Posting. This is visible to the consumer's tool. *It manages the interaction with the consumer; this interaction generates and stores the request code in the client*. All this interaction and code-generation capability is built in advance by the producer; this effort is an extra cost whose benefit is the simplicity experienced by the consumer.

conway.mel@gmail.com

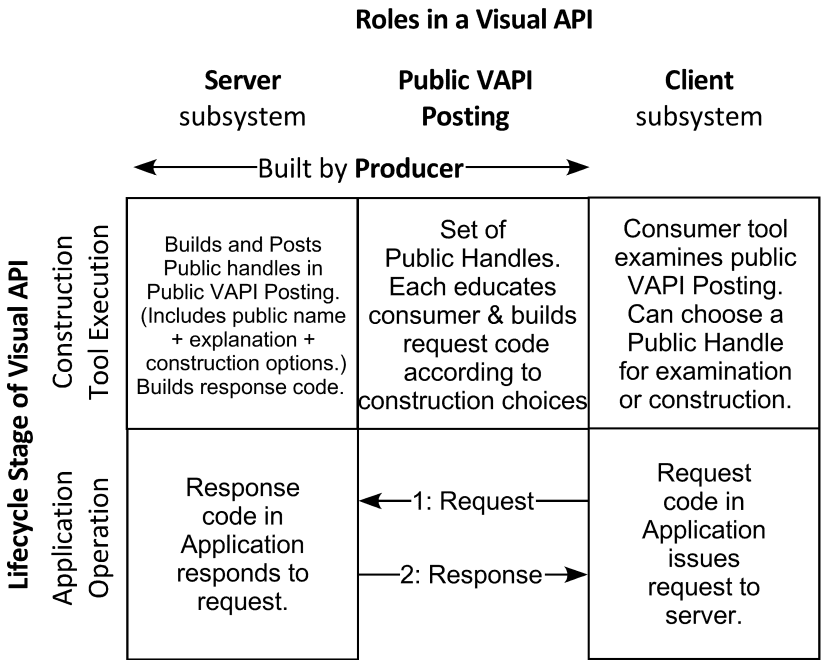

Figure 8

The life-cycle/role diagram of a Visual API

Figure 9 summarizes the concepts presented so far.

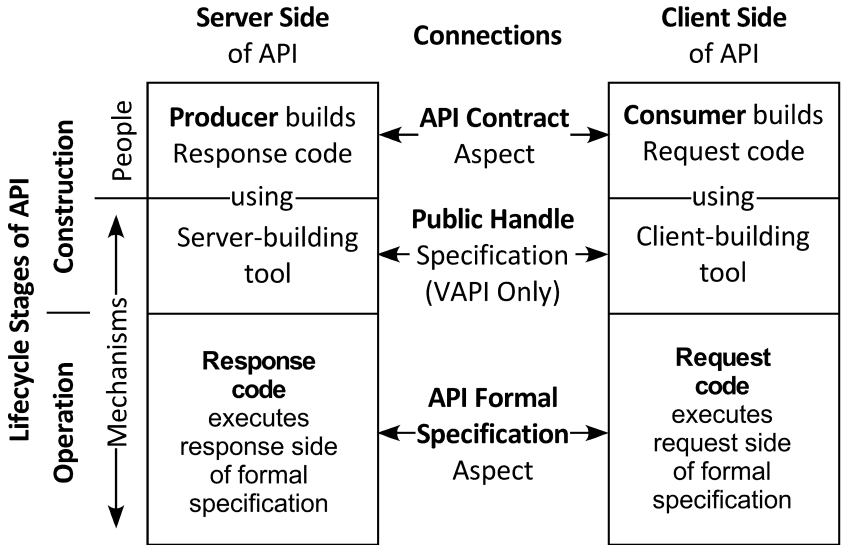

Figure 9

A Summary of the Concepts

conway.mel@gmail.com

3. Proposal: Release the Visual API into the wild

Product Concept

Now let us revisit Figure 7 replacing the spreadsheet example by an example that doesn't exist yet. I'll call it a do-it-yourself app builder, "DIY App Builder" for short. It belongs in the Asymmetric/Fixed quadrant of Figure 7. That is, its API contracts are not negotiable, and its clients can be created by non-programmers. Figure 10 simply replaces "Spreadsheet" in Figure 7 by "DIY App Builder".

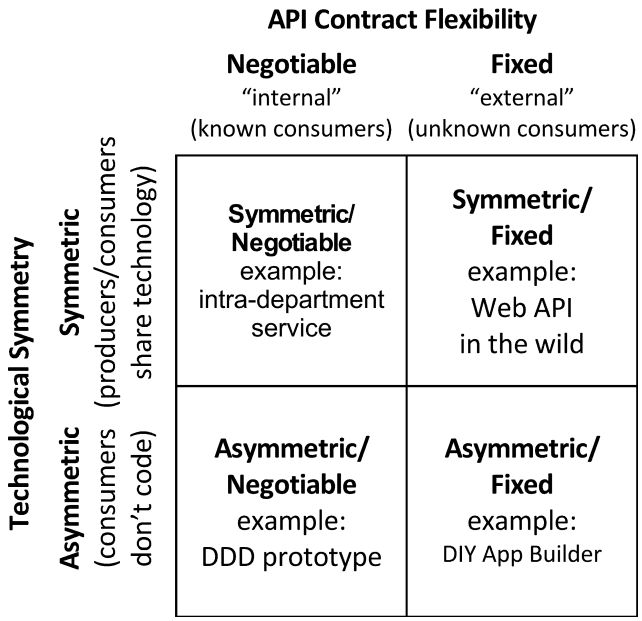

Figure 10

The DIY App Builder Replaces the Spreadsheet in Asymmetric/Fixed

What is a DIY app builder? Think of it as something that builds business applications that conform to the two-faced model (Figure 4) and that don't require that the consumers of its APIs be programmers. The vision for DIY app builder APIs is that they can be released into the wild the way Web APIs and spreadsheets are; they will allow nonprogrammers to build certain classes of business applications. What this looks like is the subject of the next section.

conway.mel@gmail.com

Figure 11 portrays my conception of a DIY App Builder product.

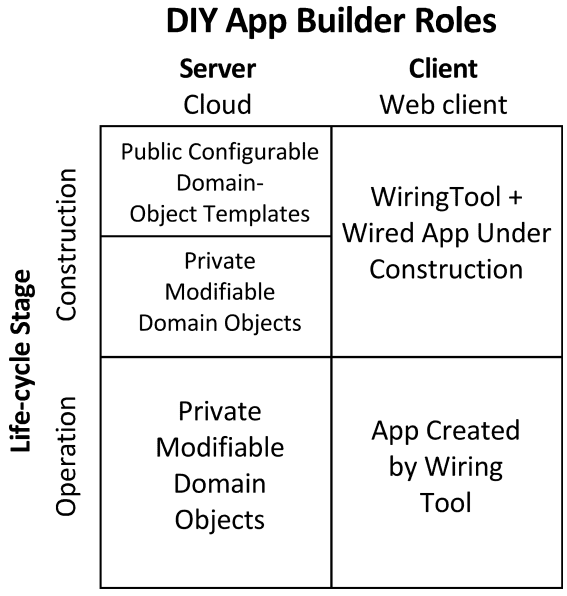

Figure 11
The life-cycle/role diagram of the DIY App Builder

- **Client role.** Applications are built in a web browser and, initially at least, they are also executed in a browser. The user's construction tool is a wiring tool that runs in a browser concurrently with the application it is building.
- **Server role.**
    - **Construction stage.** In Figure 11 "Public" means available to everybody; "Private" means available to a particular consumer account.
      The consumer builds applications based on domain objects that have been configured from a publicly available library of domain-object templates. These templates cover a range of small- and medium-size business objects that have already been abstracted by the creators of specified-function small-business software packages now available. (The abstraction process has been done; now it needs to be adapted to the two-faced model.)
      The consumer can operate in two modes.
        - Adapting a public template for specific use as a private modifiable domain object.
        - Wiring an application that accesses the consumer's repertoire of private modifiable domain objects through its visual APIs.
          What does "modifiable" mean? This is not yet clear, and

is probably context-dependent. The intent is to maximize the consumer's opportunity to change his/her design even after operation has begun.

- **Operation stage.** These are the same private modifiable domain objects.

Implications

This concept might well be limited, at least initially, to the small/medium business software market. One way products are delivered in this market is through local consultants who customize existing packages for their clients. A few examples include: retail cash register/inventory/ordering, non-profit donor management, and client project management/billing. This product should enable these value-add consultants to be more productive and to build proprietary domain objects that can increase their added value. Some small fraction of business users might venture into building their own applications.

If this product is managed appropriately it will build a community of users who will interact and support each other. In the long run I can imagine the community of Free and Open-Source Software, now mostly limited to developers, splitting into two branches (corresponding to the two parts of the two-faced model): the existing developer branch and a new end-user branch, whose members trade encapsulated wiring diagrams. Some number of the developers in the developer branch will build domain object templates for this product, enlarging its market.

---

[1] Brooks, Jr., Frederick P. (1975, 20th Anniversary Edition, 1995). "The Mythical Man-Month". Addison-Wesley

conway.mel@gmail.com