Interface-compute

A Design Model for applications with Graphical User Interfaces

Rawld Gill

ALTOVISO, Inc.

First Published April 2009

Revised and Extended August 2010

Abstract

This article describes a new model for constructing non-trivial programs that are controlled with graphical user interfaces. The discussion is motivated by describing how requirements for such programs are increasing in complexity yet the abstractions used to control this complexity have stagnated. In particular, the omnipresent abstraction for building GUI-controlled programs, model-view-controller, is shown to be inadequate.

The core of the article describes a new model, termed “interface-compute”, for building GUI-controlled programs. Eleven prescriptions are detailed that codify the model and several less-formal suggestions are provided to help designers realize the full potential of the model. Finally, the article concludes with a discussion of how the interface-compute model may be applied to browser-hosted GUIs.

Motivation

Today, most computer programs that interact with a human user provide for that interaction with a GUI. Unfortunately, building non-trivial GUI-controlled programs remains a difficult endeavor. Indeed, over the last ten years, little has changed when considering the capabilities of native application frameworks (for example, Microsoft Foundation Classes, Qt, and the rest) targeted to desktop/laptop computing hardware.

On the other hand, demands on developers have increased. Ten years ago, targeting desktop/laptop computing hardware running Windows, X11, or OS X was sufficient to reach and satisfy most potential users. Today, this is not the case: users increasingly demand that their phones, tablets, and/or palmtops duplicate many applications previously reserved for their primary computing platform. These user demands are reasonable since these devices include substantial computing power. Consequently, in addition to the big three desktop environments, developers may be required to target the various operating systems and GUI environments that accompany phones, tablets, and the rest. And to make the computing system seamless to the user, applications must function continuously as a user moves from one device to another. Clearly this is a much harder problem than constructing a monolithic desktop application that is locally installed on a single computer.

At a minimum, we would like to find a better way to build the traditional GUI-controlled desktop application; ideally, we would like the solution to help build applications in the multi-target environment described above. In order to solve this problem, we must control its complexity. Complexity is the root cause of weak feature sets, poor performance, bugs, and construction cost. Controlling complexity is the key to building programs that do more, do it faster, do it with fewer bugs, and do it at less cost.

Complexity is not a unique problem to computer programs, but rather a problem common to all engineering sciences. Consequently, techniques for controlling complexity are well-established:[1]

- build abstractions that hide details
- build interfaces that allow componentization of systems
- build implementation languages that emphasize particular details while de-emphasizing others

Building GUI-controlled applications for just a single desktop platform is a difficult task that has not gotten much easier over the last decade. When we attempt to target multiple devices, operating systems, and GUI environments, often our strategies fail so badly that either a grossly substandard (or no) product is produced, or construction costs explode, or both. In the end, we must conclude that current strategies for controlling complexity, that is, our current abstractions, interfaces, and/or languages aren't good enough.

Since pinpointing the problem might at least point to where to look for a solution, let us ask, "What is the dominant abstraction we see, *ad nauseam*, in today's application frameworks?" I submit it is model-view-controller (MVC).

MVC is omnipresent. It is woven into many well-established and current class libraries (for example, the Microsoft Foundation Class library) as well as most web application frameworks (for example, Ruby on Rails). Indeed, it is woven into nearly every popular application framework intended for building enterprise-class applications.

If it is true that current frameworks aren't good enough because their abstractions or interfaces or languages aren't good enough, maybe a good place to start looking for errors is the single abstraction—MVC—that's a part of nearly all of these frameworks.

Model-View-Controller is Harmful

Since the many MVC implementations vary so greatly in design details, we'll concentrate on describing the big error in MVC and the solutions proposed by interface-compute, and ignore that one or another existing MVC implementation may not be completely faithful to the original model.

The view component is defined as a static component, never directly accepting input from the user. This is to say that reacting to user input is handled by a different component than presents user stimuli. But GUI programming environments do not draw a bright line between input and output components in this way: well-designed GUI programming environments are componentized into nested containers of user interface functionality.[2]

For example, a component that abstracts a bit of text may be aggregated with a component that abstracts a button that has a "pushed" and "unpushed" state into a component that represents a radio button. Next, a group of radio buttons may be aggregated to produce a radio group—a set of radio buttons where at most one may be "pushed". Many radio groups may be aggregated to produce a set of questions that might represent a quiz.

Notice what is being described: progressive layers of abstraction. Each layer provides an interface to the abstraction given by the layer and hides all other details. For example, a quiz component may provide an interface that only gives the values of the contained components and be utterly unaware of other attributes of the contained components. By dividing the problem of a complex user interface with many, often-nested components into progressive layers of abstraction, each

---

[1] The following three bullets are almost verbatim from [Abelson], which is an excellent treatment of this subject for software engineers.

[2] These components are variously called "controls" or "widgets".

component is less complex. Ideally, a large complex system of components can be divided and organized in a way that the implementation complexity of each individual component is almost trivial.[3]

The strategy described above is nothing new: it is the way we build modern GUIs. Indeed, without such abstraction techniques, it would be extremely difficult to build the feature-rich GUIs expected in today's applications. But, notice that each component is also self-contained at each level of abstraction. To divide the abstraction hierarchy into two trees—one for input and one for output as prescribed by MVC—would be ludicrous. Such a division could potentially double the number of components and greatly increase the communications path between these components.

For example, if detecting a mouse click is implemented in a separate component than the component that presents a pushed or unpushed button, then some non-trivial machinery must be constructed to communicate between the two components when a mouse click occurs on a button. This problem disappears when mouse input recognition and button presentation are implemented in the same component. While this example may be contrived in some environments that provide a "built-in" button control, it is not contrived for other, more-advanced components; I will give such an example in a moment.

The skeptical reader may object that "modern" MVC doesn't actually enforce or, even, demand such a division. While it is true that gifted software engineers can contort defective frameworks into well-designed systems, this does not negate the point that the frameworks themselves—because of the models they purport to prescribe—are still defective. But to say, for example, that some wonderful web application framework allows for sending a dynamically generated, self-contained form down to the browser and the server only processes the results of the form submit event misses the point entirely.

First and most important, was the form a member of some larger UI process that could be abstracted as a single unit? If so, why wasn't it? That is, why was the single abstractable unit divided (and complicated) in such a way that the implementation was spread across two processes, in two different environments, on two different machines? This is an example of improperly dividing input and output that is not contrived. We see frameworks, particularly, web application frameworks, proud of enabling these kinds of designs. If the machinery that detected the event that caused the form to be sent from the server could have been abstracted into a single component requiring no client-server transaction, it would have been less complex to do so.

Second, to say that "modern" MVC isn't strictly MVC—after all, some of the controller is really implemented in the view (and while we're at it, we might as well bleed all three components into each other)—is to regress into something that isn't MVC. It's just a monolithic mess. And, frankly, that's the state of many actual programs. This is not acceptable either. An application either follows a design model or not. If it follows MVC, it divides input and output and adds unnecessary complexity; if it purports to follow MVC but breaks the rules when convenient, then it’s probably just a monolith with all the implied cost associated with such designs. There is abundant evidence that real-world applications built on top of supposed modern frameworks are expensive, bug-laden, and often unsatisfying to their users and owners. Perhaps designs breaking to the point of monoliths can explain many of these failures.

Let's consider MVC defects from another perspective. Implementing most of the UI in one environment and the domain logic in another is a common application design model. Often this design is termed "rich client—server" (for example, the Eclipse Rich Client Platform). But how does MVC map on to this two-component division? The model in MVC may map to the domain logic, but where does the view and, particularly, the controller go? Is the controller split between the client and the server or exclusively at one side or the other? The client and server operate at the same level of abstraction and these are the only two components at this level of abstraction; yet MVC has three components at the same level of abstraction. Clearly these models are not isomorphic.

If we consider web application frameworks that purport to support constructing so-called "rich internet applications", the entire framework resides at the server, so, apparently both the view and controller are implemented at the server. This

[3] Of course this idea of divide-and-conquer is a cornerstone of the engineering sciences in general, and, due to the combinatorics common in our problems, computer science in particular.

leaves the browser completely absent from the design model. If this is our mental picture of the design, then the browser is left to function as little more than a terminal with nice output capabilities.

Indeed, many applications are built precisely to this model. But such applications are usually defective. Consider a simple web application that delivers an article. It would seem that the browser need do nothing more than display the article nicely. But what about navigation? Semantic navigation would be useful. And while we're at it, why not provide search features for words or phrases, perhaps with advanced languages like regular expressions. How about highlighting and annotations? Clearly an application that solves all navigation problems with the browser-provided scrollbar is defective when compared to the application just sketched. And the sketched application is a lot more than a terminal with nice output capabilities.

Alternatively, we could say the responsibilities of the view and controller are split between the server and the browser. Of course this vision is horribly flawed: intentionally splitting the implementation of two components into four components operating in two completely different environments in two different processes on two different machines obviously increases complexity to no benefit.

In the end, if we choose to build applications with rich clients—and all full-featured, browser-based applications are examples—then we're either contorting MVC in a way that increases complexity or we're adding a component outside MVC. Clearly, MVC fails to describe what we need to do.[4]

Interface-compute

We need a new design model that controls the complexity inherit in applications with GUIs. The new model must not build on MVC since it is clearly defective. The model must be independent of any particular GUI platform since, at the model level, all GUI platforms are identical (i.e., they signal user input and render graphics). In particular, the design must support using the browser as the GUI platform since it is such a pervasive platform today.

The essence of a GUI is to collect information from and present feedback to the user in an intuitive manner. It is wrong to say that a GUI exists to solely collect input (data and/or commands) and present output (data views and/or command results) that interact with the application domain logic because user interaction often has nothing to do with domain logic. For example, zooming the viewport is a popular and useful application feature that has nothing to do with the domain logic.

On the other hand, the essence of application domain logic is to compute *something*. This computing is always realized by an abstraction of some real-world problem together with an interface that controls the abstraction. If the abstraction is intended to be controlled by a user, then the interface must be suitable for control by such a user.[5]

---

[4] MVC has other defects in addition to those just described. As real applications develop, bidirectional communications paths tend to arise between each of the three components. Such a design is, at least, a significant enhancement to the original MVC model. Often these bidirectional paths lead to logic (implementation) leaking between components and the bright line between components tends to blur. This may be as much a result of sloppy implementation as a flawed design, but, in practice, this seems to happen often, tending to suggest that the MVC design isn't helping implementers control complexity. The hardest problem is keeping the view and controller components separate. As mentioned above, the view and controller roles do not map to the way GUI components are abstracted and composed; either the design is compromised or the GUI system is unnecessarily complicated. In the end, we must ask, is the MVC mental model helping or hurting?

[5] For example, an application that expects 100 measurements per second for input is not suitable for control by a human user.

If we use these definitions for GUI and application domain logic, then these two components are orthogonal. It is possible to construct a fully functional program with just these two components: connect the GUI to the application domain logic and the user is given access to the full value of the application. Some applications may require additional machinery between the GUI and the application domain logic. For example, if a particular application is intended to service many users concurrently, then there may be components that manage concurrent requests, balance loads on multiple servers, ensure access control, and the rest. Notice that these kinds of components are also orthogonal to the GUI and application domain logic, and, as a direct consequence of this orthogonality, the GUI and application domain logic can be constructed and tested ignoring these additional components.[6]

At this point we can see a design model emerging. It has two components: the GUI, termed the "interface" component, and the application domain logic, termed the "compute" component. The overall design is termed "interface-compute" and is prescribed as follows:

P1. The compute component abstracts the domain problem, nothing more, nothing less.

P2. The compute component provides an application programming interface to the abstraction.

P3. The compute component is forbidden from containing any user interface features; this includes parts of the interface that may be very specialized to the particular domain problem.

P4. The compute component is forbidden from causing any interface component requirements.

P5. The interface component abstracts all user input and program output into a graphical user interface.

P6. The input and output abstracted by the interface component controls both the interface component and the compute component.

P7. The interface component is forbidden from containing any domain logic.

P8. The interface component implies requirements on the application programming interface provided by the compute component.

P9. The interface component initiates and controls all communications between the interface and compute components.

P10. All communications between the interface and compute components are asynchronous.

P11. The design of the interface and compute components should not cause process requirements on the components. They may execute in two separate processes on two separate machines, two separate processes on the same machine, or in the same process.

[P1] and [P5], which broadly describe the contents of the two components, define the two components as orthogonal to each other and form the core of the model. It may seem self-evident that any program that computes and then outputs a result based on some input can be divided into two orthogonal components. However, since examples of improper decomposition are so common it is instructive to formalize the idea; this is largely the purpose of the remaining prescriptions.

---

[6] However, performance requirements must not be ignored. For example, when building an application destined to be deployed to a high number of concurrent users, it is likely important that the application domain logic be designed with performance in mind and that the interface between the GUI and application domain logic be efficient. However, these requirements have no effect on the decomposition into GUI and application domain logic.

To begin, notice that, at the lowest level of interface abstraction, the interface and compute components are separate even when considering a monolithic program since the functions that collect input and render graphics (which, by definition, are part of the interface component) ultimately result in calls to operating system services and these services are clearly not part of the application domain logic. Of course, this is not the level of abstraction prescribed by [P1] and [P5].

To see how to divide any program into interface and compute components at the desired level of abstraction, imagine the program as a monolith. Like any program, it will have various units of computation (for example, function, classes, modules, and the rest, depending upon the paradigms employed). These units of computation are interconnected so that units variously demand/provide services from/to other units. The units and these connections can be visualized as a directed graph. By definition, the nodes in the graph that represent operating system calls to collect input and render graphics are part of the interface; mark these nodes. Next consider each node connected to a marked node: is it functioning to solve the domain problem or simply collect input or render output at a higher level of abstraction? If the latter, mark it. Continue this process until no additional nodes can be marked. At this point the graph is divided into two components: the unmarked nodes represent the compute component and the marked nodes represent the interface component, and the edges between marked and unmarked nodes represent the interface between the two components.[7]

Unfortunately, the procedure given is fairly loose and it is possible to adjudicate computational units as belonging to compute component when their function is better described as higher levels of input/output abstraction and therefore ought to belong to the interface component. Consider the computational unit of collecting vital signs for an electronic medical record. Since the information being collected is clearly domain-specific information, it may seem reasonable to assign this unit to the compute component. However, [P3] prescribes that this type of functionality be part of the interface component because rendering a set of user-interface components (static text, radio buttons, and the rest), navigating among those components, and interacting with those components are not exclusively connected to the problem being solved (persistent storage and availability of medical information). In fact, these tasks are completely generic across many applications. For example, the "date of collection" input is likely identical whether collecting vitals signs or a bank loan application.

The key point is that the types of input and output (for example, static text, radio buttons, aggregation of less-specialized types to make more-specialized types) are generic across many different application domains. It is the values of the input and output that vary. It follows that since the types are identical across many different application domains, they must not be the exclusive property of a particular application domain. Ergo, to consider them as part of the application domain problem would be to include something more than the application domain problem in the compute component which is a violation of [P1]. Further, since the question of whether or not application-specific input or output is part of the domain logic is encountered in nearly every program—and, often decided incorrectly—[P3] prescribes that such a functionality shall always be apportioned to the interface component.

Another way to see where a particular computational unit resides is to notice the relative stability of the requirements of that component. GUI builders spend a significant amount of time working out the organization and appearance of data presentation and collection functions. It is not uncommon for a simple collection function like vital signs to be redesigned multiple times before being accepted by the program owner. Conversely, domain logic is usually quite stable. A program that models an electronic circuit must follow the laws of physics and the parameters of the devices and wires contained in the circuit. Once a simulation strategy is chosen, logic requirements are quite stable. They may be augmented to add support for new devices, but, the intention of the program never changes: it must output the real-world behavior of circuits presented as input. Similarly, in the vital signs example, the associated set of data (height, weight, temperature, heart rate, blood pressure) is usually decided once and left to the database architect to codify into a relational model. The relative stability of a particular function can help decompose a program. Generally, stable requirements indicate a unit that belongs to the compute component while unstable requirements indicate the interface component.

---

[7] Of course this procedure is intended to take place in the "mind's eye" as a mental exercise during the design phase of application construction.

Notice that the compute component will often rely on units of computation that are generic across different domains. For example, a business econometrics application may rely on many standard and well-understood statistical functions. Such functions are part of the library that is used by the compute component. Analogously, the unit of computation that detects a keystroke is generic across different interfaces. So, considering the compute and interface components, we have four categories of units of computation:

1. Units that do not interact with the user and provide generic functionality (that is, functionality not specifically connected to the problem domain) that is consumed by the application domain logic; these should be provided by an external library.
2. Units that interact with the user and provide generic functionality that is consumed by the application GUI; these should also be provided by an external library.
3. Units that do not interact with the user and provide some domain logic function; these belong to the compute component.
4. Everything else; these belong to the interface component.

The catch-all Item 4 is intentionally broad. In fact, the interface component will contain a wide variety of functions:

- Some of which are specific to a particular application domain (for example, collecting vital signs); [P3] prescribes, indirectly, that such functions shall always be apportioned to the interface component.
- Some of which are specific to the GUI itself (for example, zoom the display); [P6] prescribes that such functions shall always be apportioned to the interface component.
- Some of which seem to almost define a separate component (for example, maintain and manage a session cache of data values). These are apportioned to the interface component since they fail to meet the requirements of [P1] and since they are usually not—at least completely—readily available in an external library.

Notice that preferring to implement functionality in the interface component stands in direct opposition to how most current frameworks are intended to be employed. In particular, most web application frameworks favor implementing most functionality within the framework at the server; contrast this to interface-compute which favors implementing everything except the domain logic within the browser (see the Browser—Compute section below for details).

Turning back to the compute component, different application domains abstract their domain problems in different ways. Simulators are abstracted differently than database systems, and there is substantial literature describing how to solve these and other problems. Therefore, not much more can be said about the domain logic unless we constrain the discussion to a particular application domain. Details describing how to abstract a particular problem domain are beyond the scope of this article.

The intent of the interface-compute model is for the interface component to control the compute component. Indeed it is impossible for the compute component to control the interface component since the compute component is forbidden from implementing any interface requirements [P1, P3, P4]. It follows that the compute component must provide an application programming interface (API) to make its feature set available [P2]. The interface component uses this API to provide access to the compute component's feature set to the user. In this sense, the interface component is a kind of middleware that provides a communications channel between the user and the application domain logic. It provides the user with a mental model of the application through a GUI and translates user interaction with this GUI to and from the domain logic by means of the compute component API.

Like all middleware, the interface component causes requirements on both sides. Obviously, the interface component implies requirements on the GUI since providing a GUI is the central purpose of the interface component. Less obvious, the interface component implies requirements on the compute component API since it must connect to a defined API with which to service user requests as provided through the GUI [P8]. For many problem domains, the compute component API is obvious and that the interface component will come to rely and therefore require this particular API is inconsequential. However, for some kinds of applications—typically those with highly dynamic output—early and careful consideration of the conceptual integrity of the mental model of the domain problem as presented by the GUI can ease the design of both the interface component and the compute component API.

A common design error is to allow the compute component to initiate notifications to the interface component, thus causing requirements on the interface component. At first, this seems harmless. After all, there is only one interface component and requiring just a couple of signal receiving functions in it shouldn't be harmful. Unfortunately, these kinds of designs tend to grow to the point where each component is providing a fairly significant API to the other. This has two undesirable consequences. First, the components tend to become more and more tightly coupled resulting in logic starting to leak from one component into the other. This is a severe violation of the model, causes increased complexity, and will diminish many of the benefits of the model. Second, as the two components become tightly coupled, it becomes impossible to replace once component without keeping legacy requirements implied on that component by the other. [P9] prohibits such design errors.

Notice that [P9] does not prohibit the compute component from sending event notifications to the interface component; indeed, this is an important technique for some interface designs. However, such notifications can only be sent once requested and as specified by the interface component, typically by some publish-subscribe or connection point interface defined by the interface component. The key point here is that the interface component defines how it should receive notifications, not the other way around.

Finally, the interface component should communicate with the compute component asynchronously. Although designers often convince themselves early in the implementation that the compute component will always return results in less than user reaction time (about 0.2s), as the program grows it either slows or features are added that cannot meet this requirement. At this point, the implementation has two choices: either accept a temporarily frozen interface or implement an asynchronous interface between the interface and compute components. The former is clearly unacceptable for any non-toy program. If the latter is chosen, then there are two ways to communicate with the compute component (synchronously and asynchronously), and this adds complexity. It is far better to plan on this channel being slow and/or unreliable from the beginning and design the communications channel between the interface and compute components to be asynchronous [P10]. By using modern techniques such as promises, the cost of this decision is negligible while the savings down the line are significant.

[P11] naturally falls out from all of the discussion above. If the interface and compute components are designed to be orthogonal, the compute component contains absolutely no UI logic, and the interface component communicates with the compute component API asynchronously, then any process architecture is possible. [P11] is included as a sort of check point to ensure that the intent of the model and all of the other prescriptions are followed as well as a reminder to designers of possible deployment scenarios.

Alternatives

Before discussing the benefits of interface-compute, let's briefly consider alternative designs. Essentially, at the level of the interface-compute model, there are two alternatives: either divide a program into more than two components or don't divide it at all.

By not dividing a program into clearly delineated components, the design degenerates into a monolith where everything is interconnected within one implementation unit. This has the benefit of ensuring fast communications channels. However, considering modern inter-process communications techniques, communications channels between two separate components are never so slow that a user could perceive degraded performance, particularly if the two components are on the same machine (which is to compare apples to apples since a single process is, by definition, on a single machine). Thus, this potential benefit is without value.

On the other hand, to bundle the functions contained in the compute and interface components in a single monolithic implementation unit will guarantee that unit is larger than either the compute or interface units individually. It is well established that program size is a key factor—usually *the* key factor—in program complexity, and further that complexity grows geometrically as program size grows linearly. Therefore, the increased size of the single implementation unit implied by a monolith will have a severe negative impact on complexity.

What about dividing the program into more than two units? First, the model does not prohibit further dividing the interface and compute components; in fact, a good design will do so. Thus, to choose another model with more components is to divide something out of both components into a third component. While it's certainly possible to create such a design (MVC being an example), I can see no such design that has significant advantages over interface-compute. Of course this is an impossible argument to prove since there is no way of listing or knowing all of the design possibilities.

Benefits

Now that the model is fully disclosed, let's examine its benefits. First, the problem of building the application is divided into two separate problems. Thus the total lines of code for the completed application is divided between two independent implementation units, and a linear decrease in code implies a geometric decrease in complexity. For applications that have non-trivial interface and compute components, assuming about 50% of the code of an application goes to implementing the GUI and the other 50% goes to implementing the domain logic,[8] each component is about one-quarter as complex to implement. These numbers will vary somewhat given any particular program, but they are in the correct order of magnitude. This is a huge advantage.

Second, since each component is independent of the other, they can be built in parallel. This advantage can decrease the calendar time of implementation significantly compared to implementing a single monolithic component. Further, adding resources to the implementation teams up to the point of having two rationally composed and completely separate teams will actually result in decreasing construction time. This often gives management an option of decreasing implementation time by adding resources; this option stands in stark contrast to the futility of adding resources to large teams implementing monolithic projects. Finally, two smaller teams are always much more efficient than a single large team, assuming the total number and quality of the members are the same, each team is working on a single isolated project, and other such factors are equal.

Third, the interface-compute prescriptions result in several testing advantages. Notice that the division of components implies a division of testing code which results in decreasing the size of the code contained in each separate unit. This effect alone can cause a geometric decrease in testing code complexity. Further, since the compute component has no user interface but rather exposes its features through a programming interface, writing test code requires simply invoking interface functions and checking the results returned by those functions. This is the easiest kind of testing to accomplish since results are already encoded in binary objects that are easily verifiable. While testing the interface component is always a difficult problem, one significant variable—the correctness of the compute component's logic—can be eliminated from the testing by employing a simple simulator in place of the compute component. Such a simulator can be implemented as a trivial static dictionary system where a precomputed result is returned for a particular compute function invocation. The division of testing and relative ease of writing tests (particularly for the compute component) makes it easy to localize program faults when they occur during program acceptance testing and after deployment. All of these factors combine to make better testing more likely, earlier, and more often, which can dramatically increase the quality of the final product.

Finally, since each component is independent of the other, it is possible to replace one component independent of the other. For example, the interface component could be replaced as interface paradigms and styles evolve while leaving the compute component untouched. This allows the cost of the compute component to be amortized over multiple versions of a program. The lifetime cost of the program is thereby decreased and/or more resources can be expended on the interface component.

Refining the Interface Component

[8] Of course these percentages will vary depending upon the application domain, but this variance does not negate the points being made.

Unlike the compute component, the requirements of the interface component are largely generic across all problem domains:

1. It must be capable of recognizing input from available input devices (keyboard, mouse, touchpad, and the rest).
2. It must be capable of producing output; this can be further divided into three levels of capability:
    a. Rendering stylized boxes that contain stylized text; rendering images.
    b. Playing animations, audio, and/or video.
    c. Rendering generalized graphics (points, lines, curves, and the rest).
3. The interface component must communicate with the compute component asynchronously.
4. Often the interface component requires machinery that provides for temporary local storage.

Considering just the input-output functions (Item 1 and Item 2), the requirements as listed above are far too primitive to build a GUI efficiently. Thus, a set of abstractions with appropriate interfaces must be built. Since the purpose of these abstractions is to solve the computational problem of interacting with the user, they must abstract the kinds of interaction that are desired (for example, bits of text and graphics that give the illusion of a "radio button"). This is to say, we need to abstract things that exist in the real world, even though that real world may itself be an illusion. The examples given so far (text, buttons, radio buttons, radio groups) are not intended to suggest that abstraction activities should terminate at the simple component (often termed a “widget”) level. Quite the opposite is true:

1. The entire interface component for a particular program must be factored into a hierarchy of abstractions.
2. Siblings of any parent must be maximally orthogonal.
3. Abstractions must be progressively divided to the point that any single abstraction is easy to understand and implement.
4. Some of the abstractions will be containers that contain and operate on other abstractions. The pervasive example is a "form" that contains different kinds of components. Usually, the operations must be polymorphic: the container must be able to demand an operation without knowing the particular types of the components it contains.[9]
5. Some of the abstractions will aggregate more-primitive abstractions (for example, radio groups are multiple radio buttons).
6. Some of the abstractions will refine other, more-general abstractions (for example, an integer input field is a specialization of a text input field).

Items 1 through 3 simply restate well-established design and implementation principles. Since we are going to use some kind of abstraction mechanism a lot, it must be easy to use, its output easy to read and understand, and its runtime space and time complexity sufficiently cheap to not noticeably affect program performance. Further, the abstraction mechanism must support polymorphism (Item 4), aggregation (Item 5), and inheritance (Item 6). Obviously, this set of requirements suggests a modern object-oriented system. However, I resist characterizing (and reducing) these requirements to the simple phrase, "the system must be object-oriented" because this is both a loaded and meaningless statement. It is loaded because it means different things to different people. For example, Ruby and C++ both provide for "object-oriented" software construction, but proponents of either often find the other an abomination. It is somewhat meaningless because there is no universally accepted model that defines "object-oriented".[10]

While these observations seem self-evident—indeed, state-of-the-art user interface frameworks are built with systems that employ this design (for example, Qt, Cocoa)—many application frameworks utterly ignore this advice and suffer costly consequences. For example, all web application frameworks that dynamically generate HTML with supposedly wonderful templating systems are implementing a completely different model for the user interface. In these systems, the level of abstraction is extremely low (HTML), the coupling between user stimuli (HTML) and user input is weak or non-

[9] If the system required enhancements to every container every time a new containable component was constructed, the complexity of adding new components and/or containers is increased.

[10] Date and Darwen make a noble attempt to formalize several “object-oriented” concepts [x]. Sadly, this work is not universally known, let alone understood and agreed upon.

existent, the interface component is spread among two different processes in two grossly different environments, and aggregation, inheritance, and re-use are implemented by the programmer's editor cut-and-paste functions. Each of these realities increases complexity. It is also true that, no matter how many Ajax tricks are incorporated into these systems, they more-or-less look and feel like a web application instead of a native application. This design must be rejected if the true potential of browser-based applications is to be achieved.

Notice that the computational demand on the interface component is limited to filling a viewport with stimuli, catching all user input, and updating the viewport consequent to that user input. This is not a large computational burden. The idea that it is necessary to spread it across processes and, even, machines is plainly nonsense—so long as the interface component restricts itself to implementing the GUI and doesn't cross into the problem domain. Thus, since there is no good reason to divide the interface component into multiple processes and since doing so will increase complexity, we have stumbled into another strong suggestion for the interface component: it should be implemented in a single process. So far as web applications are concerned, this is quite a dramatic statement. It says the interface component should be implemented completely in the browser, and server-side dynamic generation of the user interface is forbidden.

Given a hierarchy of well-designed abstractions, the basic model for the interface component is a tree of instantiated variables of those abstractions that facilitate the user's control of the application. At any instant in time, the user's focus is on a leaf node of the tree. For example, if the user is reading or entering the systolic blood pressure field on a vital signs form in a electronic medical record application, the path of focus (i.e., mental focus, but also likely keyboard focus) might be application → workspace → vital signs form → blood pressure field → systolic field. Similarly, it is easy to imagine how the tree might be filled out at various levels. For example, the application level may have the children top-menu, toolbar, workspace, status bar.

Thus, modeling the interface component as a hierarchy of variables has two benefits:

- The model accurately reflects the user's mental model of the interface.
- The model controls the scope of any implementation unit.

Note carefully that there are two hierarchies: a hierarchy of abstractions that define the component *types* used to build the interface component, and a runtime hierarchy of *variables* that contain specific values of those component types.

The interface component begins life as some predefined runtime tree value. The tree is modified consequent to three events:

- The user provides input that is targeted to the interface component; the interface component computes and presents the response.
- The user provides input targeted to the compute component; the interface component sends the service request to the compute component and then presents the associated response.
- The compute component signals an event to the interface component; the interface component presents the consequences of the event.

Let's turn our attention to requirements for the communications channel between the interface and compute components. As previously discussed, there is very little that we can say for certain about the compute component. In particular, it may be in the same process or a different process. If in a different process, it may be on-host or off-host. Depending upon the kind of problem the compute component solves, it may provide a very fast and reliable interface or a very slow and/or unreliable interface. In addition to planning for an unreliable and/or slow compute component, the design ought to gracefully handle programming errors unwittingly caused by the interface component since the combinatorics consequent to a GUI interface operated by a human are staggering.

As per the model, the channel between the interface and compute components must be an asynchronous channel where the interface demands some service from the compute component and then, at some time later the compute responds with an answer, signals an error, or times out. The compute API need not be asynchronous, but rather the manner in

which the interface component communicates with the compute API must be so. This can be implemented by generic machinery such as promises.

The last area of interface component design to examine is data storage and management. Since the compute component is forbidden from including any user interface functionality, it cannot contain any machinery to store data as it is being added, edited, and/or deleted by the interface component. In particular, the interface component must manage data while it is in a transitional state where the user has edited the data in some way but has not signaled that the edits should be committed. In some types of applications, such transitional states can be quite lengthy and contain significant data. If the GUI is designed in such a way that a single data item is never referenced by more than one variable, then the management of any particular data item can be delegated to that single variable that references that data item. Unfortunately, this almost never works since it is common for multiple user interface components to reference a single data item. For example, a contact information form holds a name that is also on a contact list grid. If the name is edited in either location, the edit should be reflected in both locations.

In applications with these kinds of design problems, the interface component should employ a session data store (that is, a data store that has a lifetime of the user session) that manages session data. The data store should employ an interface that sets and gets data item values and signals to all interested interface components when any particular data item changes status.

During the design of the session data store, it is important to keep in mind the limited requirements of the machinery:

- The store lifetime will be short—the length of a single user session.
- The services required by the store are exactly and only those required by the interface component.
- The data contained in the store will be very limited—precisely the data with which the user is currently interacting.

A common mistake is to attempt to mine functionality out of the session data store that simply is not reliably possible. The data store is in no way a database that can be queried to answer interesting questions because it doesn't have a predictably complete set of data. It is simply a cache for use by the interface component.

Browser—Compute

The interface component prescriptions and refinements are platform- and tool-independent. It could be implemented in many languages (for example, Java, C++, Ruby, and others), on top of many GUI frameworks (for example, MFC, Qt, Cocoa, and others), on many different computing platforms (for example, desktop, palmtops, phones, and others). If a particular application is required to exist on a single platform, then choosing a set of implementations technologies and tools specialized for that platform may be ideal. However, if an application must operate on many platforms, then the interface component should be implemented on top of technologies that are stable and readily available for all of those platforms. Today, that means the browser.

Targeting the browser is not without challenges. I've pointed out several gross defects in current web application frameworks. However, if the principles set forth so far are followed faithfully, the browser is a powerful platform upon which to implement the interface component.

Notice that the modern browser contains the foundational capabilities for a user interface. It can recognize input from input devices, render stylized boxes of text and images, play animations, audio, and video, and even render generalized graphics. The problem is that most designers don't visualize a browser as a GUI platform, but rather as an HTML platform that renders markup and takes input from HTML form controls. From this starting point, the browser becomes a very clumsy GUI platform.

The alternative is to visualize the browser as a JavaScript programming environment that includes sufficient input capabilities and powerful output capabilities through the DOM. JavaScript can be used to build up the hierarchy of abstractions and then instantiate a hierarchy of variables as described above. In some cases these abstractions may be

built on top of existing HTML controls. But often the HTML controls are too primitive and it is easier build controls by direct manipulation of the DOM instead of trying to contort one of the limited HTML controls to do something that is was not designed to do. Also notice that since the entire GUI will be built at the browser as per the prescriptions, there is no need to group controls within forms for form submit processing.

Surprising to some, JavaScript is extraordinarily powerful. In many ways, it is closer to Lisp than Java. Of course, JavaScript as provided in the browser is also fairly miserable out-of-the-box. The power of the language must be used to augment the library since the built-in library is completely inadequate. This task has largely been completed (for example, Dojo, YUI, and others). Some of these libraries include full-featured abstraction mechanisms (for example, user-defined types, inheritance, polymorphism, aggregation), support multiple programming techniques (for example, object-oriented programming, functional programming, aspect-oriented programming), and tools for modularizing large programs. Some of these libraries also go a long way toward normalizing the various vendor-specific browser idiosyncrasies so that the platform behaves identically across various browsers and browser versions.

The abstraction mechanisms provided by some popular JavaScript libraries are particularly important. For example, Dojo's so-called "class definition" function (dojo.declare) allows defining complex data types that includes a form of multiple inheritance. Dojo also includes machinery to modularize a large program into manageable implementation units and later pre-assemble the units into a release version, thereby eliminating any potential performance penalty. These two mechanisms—user-defined data types and modularization—together with the core input/output capabilities allow the construction of complex interface components completely within the browser.[11]

And building the interface component completely within the browser is the key point of browser—compute. This is to say the server does nothing more than deliver the static files (JavaScript code and resources) that implement the interface. In particular, there is no dynamically generated HTML since this would imply the interface component is divided between two processes which is a violation of the interface-compute design.

Furthermore, the program files themselves ought not to contain much, if any, HTML. The program is written in JavaScript and any need to interact with the DOM is handled by lower levels of abstraction within the interface component. A GUI is a complex program; an appropriate, powerful language is needed to express the solution to this program. HTML is neither appropriate nor powerful, particularly when compared to JavaScript. The interface component must be primarily written in JavaScript when the browser is selected as the target platform.

Recall that units that interact with the user and provide generic functionality that is consumed by the application GUI should be provided by an external library. Certainly some current JavaScript libraries contain some of this functionality. But caution is warranted when selecting among the "general purpose" and popular JavaScript libraries currently available. Most of these libraries aim to aid in the development of web sites rather than GUIs. And web sites have quite different requirements than the interface component: they present mostly static information one page at a time and have very limited user interaction—click a link/button, perhaps after filling out a few simple HTML controls—before the page is replaced. This is clearly not how an application GUI operates. Consequently, many of JavaScript libraries are not general purpose at all, but rather are highly biased to help solve the problems encountered by web site designers. In the end, the popular and easy choice may have significant deficiencies down the road when the library proves to be missing key features required to build fully functioning GUI components. If the library appears to concentrate on manipulating HTML rather than building higher level abstractions, then it is probably inappropriate.

While some popular JavaScript libraries (Dojo, for example) have many of the features required to use the browser as the platform for the interface component, none purport to provide a framework to build this kind of software. This is not surprising since browser—compute (at least in the detail specified in this article) is a fairly novel idea. The backdraft framework (which is built on top of Dojo) appears to be the first library that provides a framework specifically for the purpose of constructing interface components faithful to the interface-compute model.

[11] See [x], Chapters 9 and 11 for an extensive discussion of these features.

All this talk about JavaScript begs the question, "what about other, more powerful systems like Adobe ActionScript, Microsoft Silverlight, or Java?" These are all powerful systems that are fully capable of implementing the interface component and worthy of consideration. However, when choosing any of these technologies it is important to understand that the choice implies a platform choice, *not* an implementation technology choice within the browser platform. To choose ActionScript is to choose the Adobe platform, not the browser platform, since such a choice requires developers to write to the ActionScript specification and users to install and maintain the Adobe plugin and suffer the pain that comes with every such locally installed and proprietary technology. The same is true of Silverlight and Java. No matter how popular these technologies may be, they have the considerable drawbacks of being much less pervasive and potentially more painful than a standards-based browser running a JavaScript program. Further, these kinds of technologies tend to grow heavy as they advance, taxing system resources. This may be a significant problem on small-device platforms (phones and the rest). They are also vender-specific solutions which may have business-related drawbacks. To realize the full power of the browser as a GUI platform, the system must be constructed absent any plugins whatsoever.

The advantages of choosing the browser as the platform for the interface component are considerable. First and foremost, interface components built for the browser are, by definition, runnable on any platform with a browser. Today, this includes every general-purpose computing system that is intended to interact with human users. Clearly, this is a huge advantage when the compute component is implemented on a widely available remote server, since any user with a browser has automatically fulfilled all requirements to access the program.

However, it's also a significant advantage when the program is intended for local execution. Although the compute component must be constructed to target particular platforms, it need not concern itself with platform-dependent user input and graphical output idiosyncrasies; these are still handled by the browser platform. Taking away these kinds of platform dependencies can greatly simplify targeting a compute component to multiple platforms since the platform differences are mostly reduced to process, thread, and file system control and inter-process communications. Libraries are often available to abstract these differences into a platform neutral environment. For example, a browser—compute program that is intended for local installation could target Windows, Unix/Linux, and OS X by constructing a single interface component targeted to the browser platform and a single compute component that is conditionally compiled to the three different targets. This would be extremely difficult to achieve if the interface component was written to the native Windows, X11, and Cocoa graphical environments.

Assuming browser technology continues to evolve, it is hard to find any significant advantage in constructing platform specific GUIs. This is the central point to browser—compute.