Tags
Find a Tag
analysis application availability_section best_practices cad change_management changes compare compare_revisions customizations customize database db2 exchange find_available_times gantt_chart gantt_scheduler group memory_footprint modifications modify object_label object_revision operating_system oracle performance platform problem_determination reports reserve reserve_performance revision revisioning single_sign-on snapshot space sql_server sso support system system_performance tags: track_customizations tririga troubleshoot tuning upgrade ux version versioning
Cloud | List

Introducing UX

Like | Updated March 4, 2019 by Jay.Manaloto | Tags: *None* Add tags

Edit | Page Actions

UX Framework | UX App Building | UX in Classic Tools | UX App Designer Tools | UX Best Practices

See the UX Article 1 "Introducing UX" PDF for previous versions of this content. What is UX? The standard definition of "UX" is user experience. But for simplicity, I'll refer to the TRIRIGA UX framework as "UX".

Introducing UX: Shifting to a more modern IBM TRIRIGA UX framework

CONGRATULATIONS! So you've heard about the new UX application framework. You've thought about it and you're ready to dive in. But what exactly is TRIRIGA UX? How do you get started? First, let's introduce a few key concepts. After all, this isn't the same old TRIRIGA anymore.

- What are the key concepts?
- What about the existing framework?
- What are the key challenges?
- What exactly is TRIRIGA UX?
- Will you be required to use UX?
- What are our future plans?
- Still confused or curious?

What are the key concepts?

First of all, in the words of Mike Herbert, our (former) principal architect on the TRIRIGA applications team, the UX framework *"implements an MVC architecture for TRIRIGA applications."* This is key. To break this down, UX implements *MVC*. In turn, MVC is applied to our *applications*.

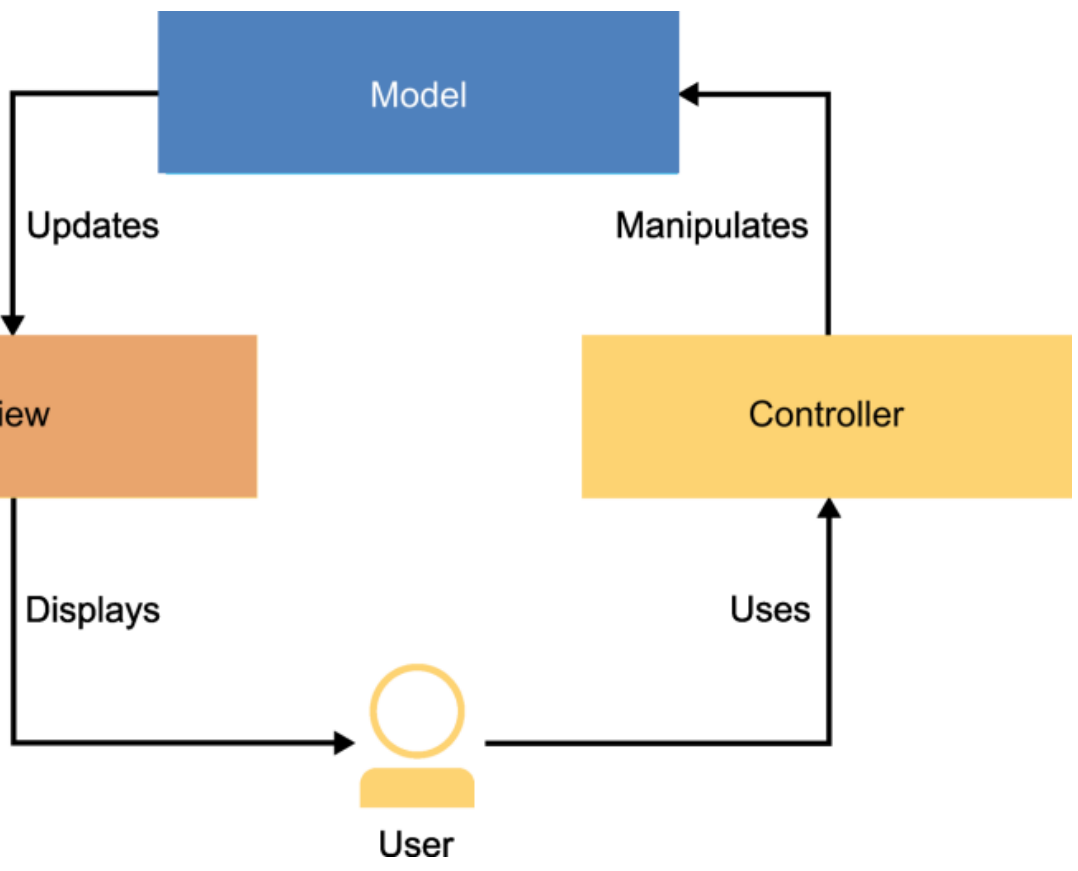
But what is MVC? According to Wikipedia:

> *Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.*

In other words, MVC separates the application into 3 components or layers -- the **model**, **view**, and **controller**. Returning to Wikipedia:

> *[T]he model captures the behavior of the application in terms of its problem domain, independent of the user interface. The model directly manages the data, logic and rules of the application.*
>
> *[The] view can be any output representation of information, such as a chart or a diagram; multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.*
>
> *[T]he controller accepts input and converts it to commands for the model or view.*

Here's a basic diagram of the typical MVC components and process flows.



By decoupling the form views from the data model and the business logic controls, you can make changes to each piece more efficiently.

To use a simple wardrobe analogy, let's imagine that instead of being separate, your favorite socks are actually sewn into your favorite shoes. How often would you clean or replace the combination? Weekly for your socks? Yearly for your shoes? To make it even more messy, let's imagine that your favorite pants are sewn onto your favorite socks too. How much would it cost to clean or replace the whole thing? Weekly? Yearly?

So you see, by decoupling the components, you can react to changing business requirements and update components more quickly and easily. Not to mention, more cleanly and cost-effectively. Aren't your socks glad?

What about the existing framework?

Members

At this point, you might be asking: *"How does the existing TRIRIGA framework deal with MVC today?"* The short answer is "not too bad".

For the **model**, we already cover this MVC component to a certain degree with our business objects in Data Modeler. The business objects, along with their fields and relationships to other business objects, do a pretty good job of modeling real-world entities and their relationships. But business objects aren't flexible enough by themselves. Ideally, the model could contain additional data sources like queries and integration data.

Next, for the **view**, we cover this component moderately well with our forms and queries. But they aren't so great when we're dealing with (1) usability, (2) page-rendering performance, and (3) responsive design. So untying these restrictions on visual interactions or user experience is a key goal. Later, we'll dig deeper into what these restrictions are.

Finally, for the **controller**, we already cover this component really well with our workflows and state families. To be clear, our workflows can fit into both the model and controller. Either way, our workflows do a good job of representing the business rules, while our state families do a nice job of establishing the data lifecycles. So no problems there.

But here's the thing. Although our existing framework applies a partial MVC pattern that's "not too bad", the main problem is in its *separation of responsibilities*. Or lack thereof, as we'll see next.

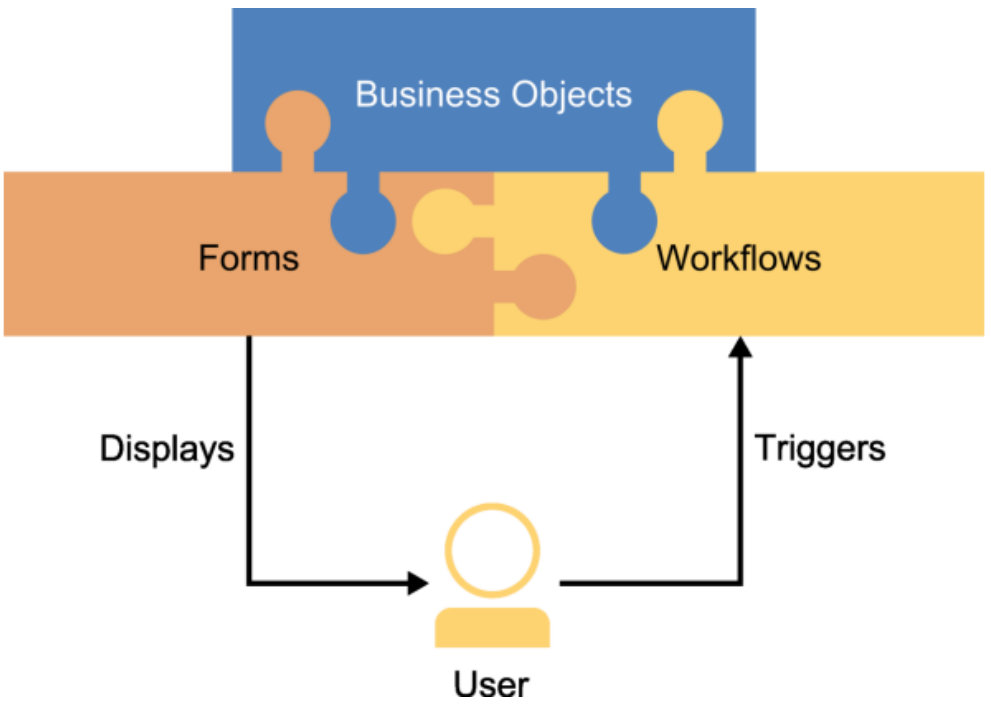
What are the key challenges?

In other words, MVC components are too tightly coupled to each other. We have too many elements in the **model** (business objects) and **controller** (workflows) that directly impact the **view** (forms) and vice versa.

Here are a few examples:

- Records are bound to a single specific form.
- Form sections and fields are tied to BO sections and fields.
- Forms cannot be replaced without breaking workflows.
- The Modify Metadata task in the workflow is tied to a single form.

Now, if we redraw the basic MVC diagram with this lack of separation, our existing TRIRIGA framework might look a little like this. As you can see, the key challenges lie in decoupling these interconnected pieces into their separate MVC components. This is where UX comes in!



What exactly is TRIRIGA UX?

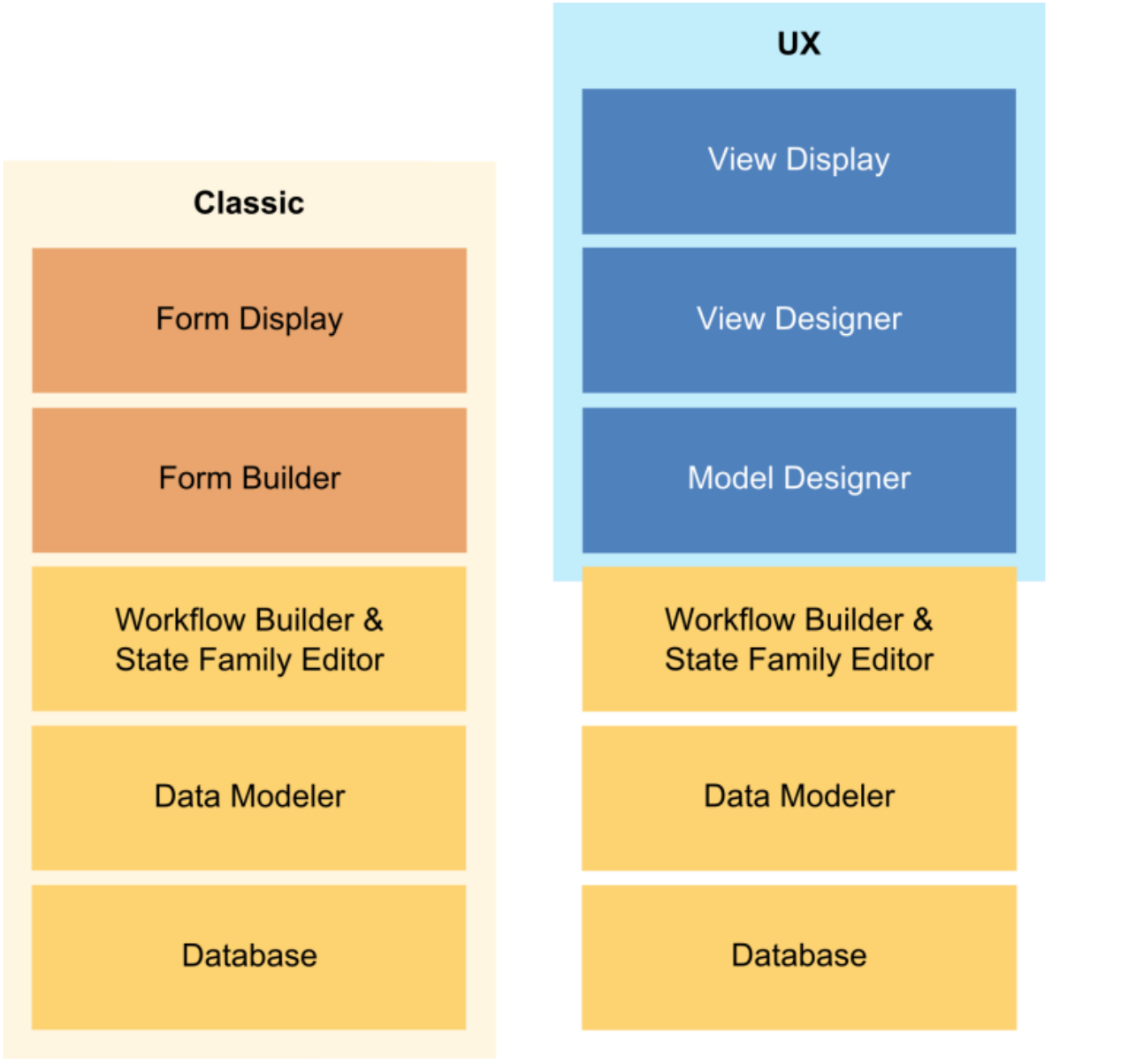
Mike describes it in a nutshell: *"[TRIRIGA UX] is a new framework to support designing and building user experiences."* Adding his earlier words, the UX framework *"implements an MVC architecture for TRIRIGA applications."* In turn, MVC decouples many of our existing pieces. But what exactly does that mean? Will components be rebuilt? Or replaced?

Luckily, despite its partial MVC pattern, the core design of our existing framework is pretty solid. Which means that many of the elements can be reused without any modification. Or at worst, with minimal modifications.

Building on this foundation, the UX framework introduces two new metadata concepts: (1) the **model** to retrieve the data and trigger the business logic, and (2) the **view** to render the interfaces or forms. The new renders will be "bolt-on" views that can be quickly added or removed, and will still use our existing application data and workflows.

As the UX framework has matured, we developed a new "model designer" metadata construct to support the model. Similarly, we developed a new "web view designer" metadata construct to support the view. We didn't need to develop a new metadata construct for the controller since existing workflows and state families can already serve this function.

To visualize these metadata concepts, here's a high-level illustration that compares our classic builder tools to our maturing UX designer tools.



Will you be required to use UX?

Not at all! If you've decided that our existing classic framework already satisfies your business needs, then you're good to go. You don't have to implement UX if you don't want to. But still, aren't you a little curious? Going back to my wardrobe analogy, wouldn't you want to untangle your favorite socks from your favorite shoes? To see how it feels?

What are our future plans?

While we can't promise any specific dates for any advanced features, I think we have an exciting vision. As outlined by Ryan Koppelman, our (former) manager of TRIRIGA platform development, here are many of the goals we've set by building a more-flexible UX framework on top of our solid but less-flexible classic framework. We hope you can join the ride.

Development Goals.

Strategy	Classic	UX
SaaS	Applications are robust but difficult to rearrange into digestible chunks, resulting in an intimidating interface.	Role-based or task-based approach with a simplified UI makes it easier to create intuitive applications.
Mobile	Form-UI code is established and reliable but difficult to adapt to responsive display for mobile applications.	View-UI code that leverages built-in features of Google Polymer elements is “mobile responsive” out-of-the-box.
User Experience	Application and platform components are diverse but have grown inefficient from years of adding features.	Outside-in approach gives insights into streamlining the application design, look-and-feel, and performance.
Upgradeability	Customizations are flexible but difficult to track, making application upgrades costly.	Built-in capability for simpler customizations makes it easier to track and upgrade.
Continuous Delivery	Core design is durable but difficult to enhance, and prone to regression with each addition or change.	MVC design pattern makes it more agile to update, easier to add features, and quicker to build applications.

Still confused or curious?

If you have any questions about UX that weren't answered in this article, feel free to reach out to your IBM TRIRIGA representative or business partner. In the meantime, here are some more background questions and answers that might help to fill in the gaps or give you a better idea of what we're trying to do. Or if you want, I'll go ask the team.

Background Q & A.

Question	Answer
Why should we redesign our forms?	Current IBM initiatives emphasize SaaS and mobile, both of which require a simple and intuitive UX. But in today's market, our existing form UI technology is becoming less nimble, and is designed for desktop, not mobile interfaces. A new UX framework can enable you to more easily meet business requirements with an intuitive UI, compatibility with touch interfaces, and improved performance. With this goal in mind, MVC is the most widely-used pattern for web development. So we're adopting the MVC pattern for our TRIRIGA application-building framework.
Why haven’t we made these improvements already?	Our existing platform metadata components -- forms, workflows, business objects -- are too coupled to each other. This makes it very difficult to make changes quickly. For example: • Records are linked to forms. • Form sections and fields are directly linked to BO sections and fields. • Forms cannot be replaced without breaking workflows or application logic. • The Modify Metadata task forces application logic to assume a single form.

What is MVC?	MVC is a programming pattern that consists of a model, view, and controller. This pattern allows applications to have multiple views that use the same data and business logic. In other words: • The model holds the data and information about the data. • The view is what the user sees. To be rendered, the view only interacts with the model. The view can also make updates to the model. • The controller holds the business logic. The controller creates or manipulates the model and responds to any changes to the model. • By design, the controller and view never directly interact with one another.
What will the new MVC model look like?	You might be wondering if it's enough to reuse business objects from Data Modeler as the model. Unfortunately, business objects are not flexible and would require complex data models to render a rich view. Instead, the model could contain many data sources such as business objects, queries, and even integration data. In addition, the model could also contain "actions" that the user can perform via the controller. These actions could be imported from business objects and queries.
What will the new MVC view look like?	The view could support many types of views such as a form view, mobile form view, and maybe even an API view and custom view. The form view could be rendered from a flexible hierarchy of form-layout components that are sourced by the model. For example, a single field type could be included in several different components and have several different renderings. More complex components could include graphics components and custom components.
How do we decouple the workflow logic from the view?	Our existing technology uses Modify Metadata tasks that couple our business logic to the view and make it difficult to make changes to the view without changing the logic. Instead, to decouple the business logic from the view, we could restrict or remove our dependence on the Modify Metadata task. In its place, we could develop a variety of layout components, field elements, and action-button elements to render a dynamic view.
How do we build applications in the new MVC framework?	While we can't promise any specific dates, we plan to develop a new "model designer" or "model builder" metadata construct that supports the model. Similarly, we plan to develop a new "view designer" or "view builder" metadata construct that supports the view. Fortunately, we don't need to develop a new metadata construct for the controller since existing workflows and state families can already serve this function. Meanwhile, if we store the new platform metadata as records that can be accessed through forms, we can more quickly react to business requirements and add features.
How do we simplify the interface or view?	Our existing technology ties forms to "things" like people and locations. So why not change the pattern so that views are tied to "actions" like creating and submitting requests? This change could be accomplished by designing views that are specific to a user role. Then we could still reuse our existing business objects and workflows to support the new role-based interfaces.
What happens to our existing applications?	Our existing applications and forms will continue to work as they did before MVC. Unlike advanced integrations that require customization, the new renders will be "bolt-on" views that can be quickly added or removed, and will still use our existing application data and workflows.
What happens to our existing user documentation?	Our existing documentation will continue to support our existing applications as they did before MVC. But new MVC-based documentation could be rendered within the same flexible hierarchy of form-layout components as custom "info components". Unlike HTML topics or PDF files in our external IBM Knowledge Center, this content could be accessed within the new framework.
What happens to our existing customers?	Because the new views will be "bolt-on" interfaces that are "bolted onto" existing applications, customers who don't choose the new MVC framework won't be affected. But for customers who choose the new framework, results could vary depending on how new role-based interfaces are applied and how much the application is customized. Fortunately, a flexible MVC model would offer customers a more efficient customization and upgrade strategy. For example, customers could add their own business objects instead of adding fields to our shipped business objects. This scenario would be easier to track during upgrade.

Next >