Metawidget

“UI Generation done right”

http://metawidget.org

What we will cover

- A common requirement
- Current practices
- A better way

Common Requirement

An everyday problem

Most enterprise applications require many different data entry forms, either for collecting or displaying data

Current practices

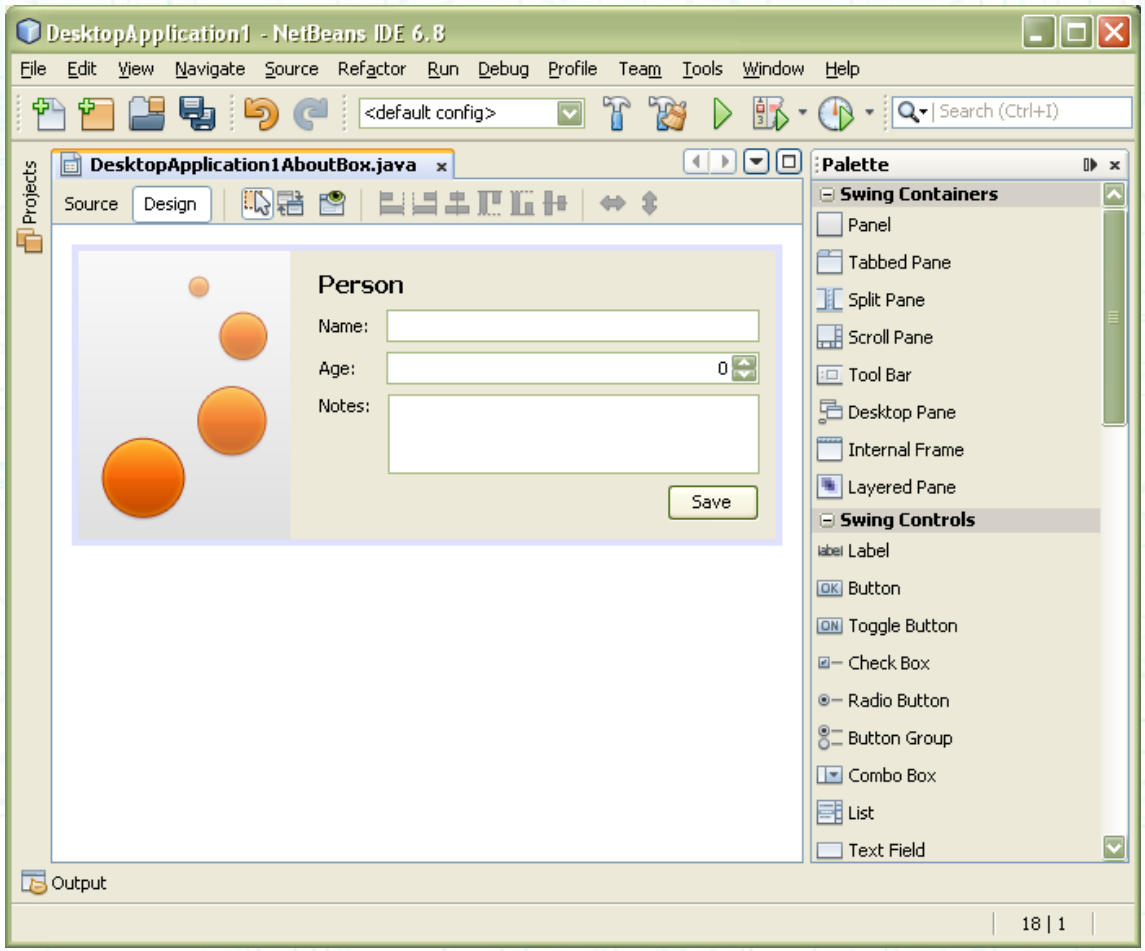
Which one are you using?

- Visual Form Designers
- UI Languages
- Code Generators

Current practices

Visual Form Designers

- Matisse
- JBoss Visual Page Editor
- etc



Current practices

UI Languages

- HTML/CSS
- Java Server Faces
- etc

```
<h:form>

    <h:inputText value="#{foo.name}"/>
    <rich:inputSpinner value="#{foo.age}"/>

</h:form>
```

Current practices

Is this you?

- ✘ Time consuming
- ✘ Duplicating definitions, error prone:

```
public String getName();
public int getAge();

<h:inputText value="#{foo.name}"/>
<rich:inputSpinner value="#{foo.age}"/>
```

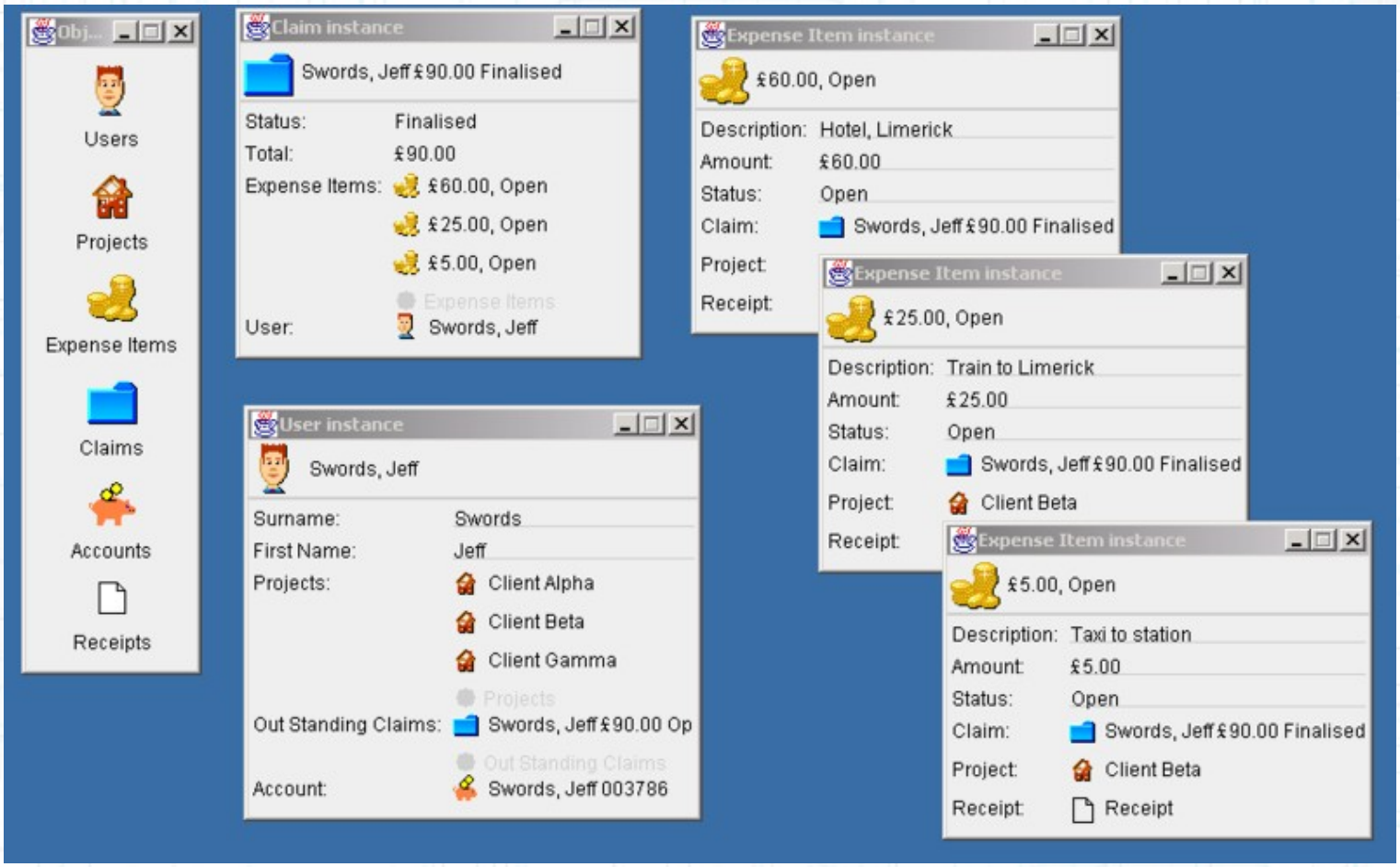
- ✘ Too laborious to do properly:

```
<h:inputText value="#{foo.name}" maxlength="30"/>
```

Current practices

Code Generators

- Naked Objects
- seam-gen
- etc



Current practices

Is this you?

- ✘ Static code generation
  - doesn't help much beyond early stages of development
- ✘ Generic UI
  - basic CRUD
  - isn't enough metadata to do as good a job as a human designer
- ✘ Dictate the architecture
  - if you build your app our way, we'll generate a UI for you

A better way

Metawidget

Designed to address each of these shortcomings

A better way

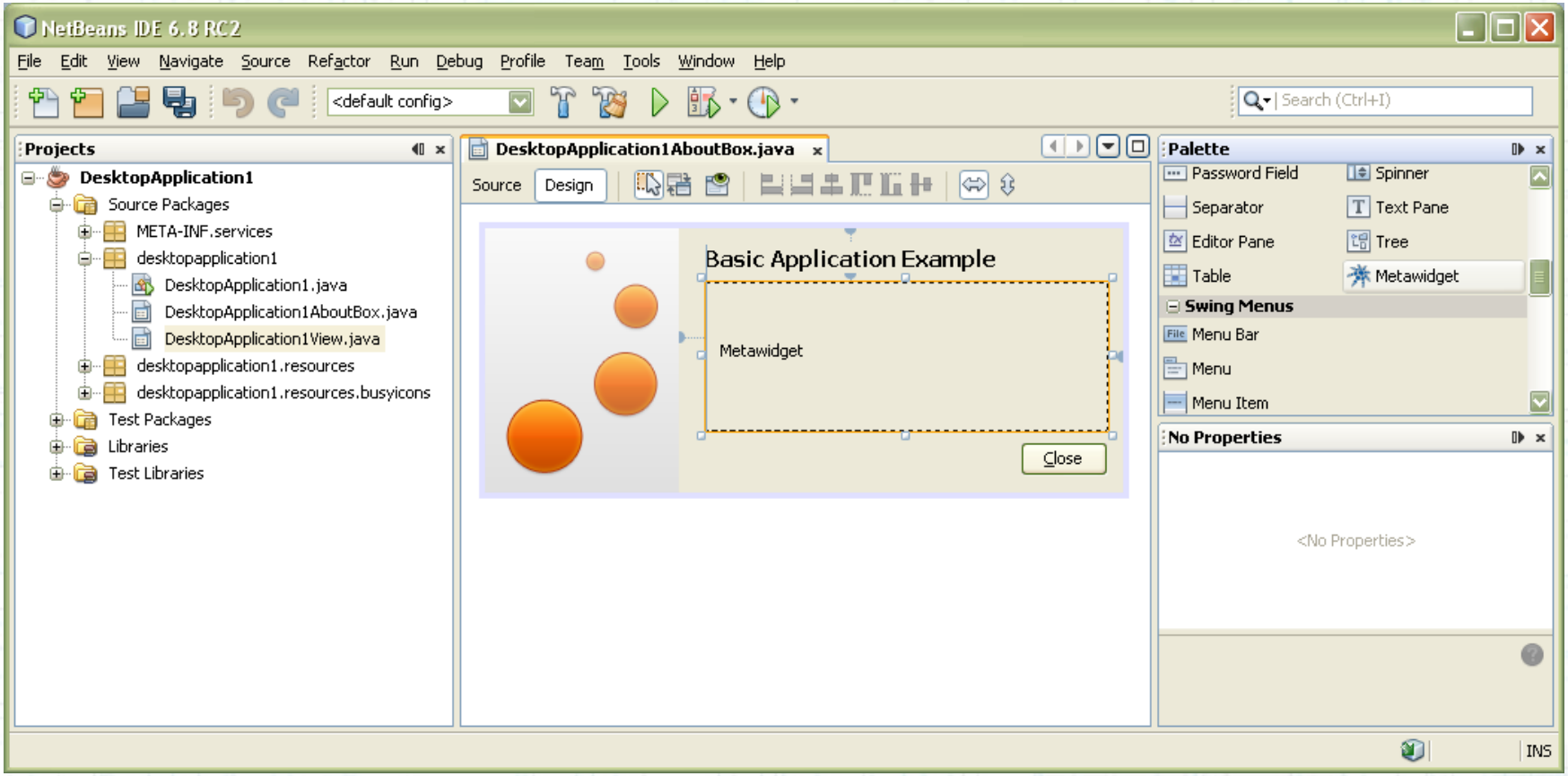
- ✔ Uses your existing architecture
  - *your existing annotations, XML files, business rules*
  - *your existing UI toolkit, third-party libraries, custom components*
  - *easy to mix technologies, or plug-in your own*
- ✔ Doesn't try and 'own' the entire UI
  - *only tries to generate the 'inside' of forms*
  - *doesn't hide your existing UI toolkit*
  - *just another widget in your toolbox*
- ✔ No static code generation
  - *inspects business objects at runtime*

A better way

- ✔ Automatically applies constraints
  - existing validation libraries, easy to add your own
- ✔ No duplicated definitions
  - reads names, types, constraints already defined in your architecture
- ✔ No time at all
  - once configured, changes to screens are free

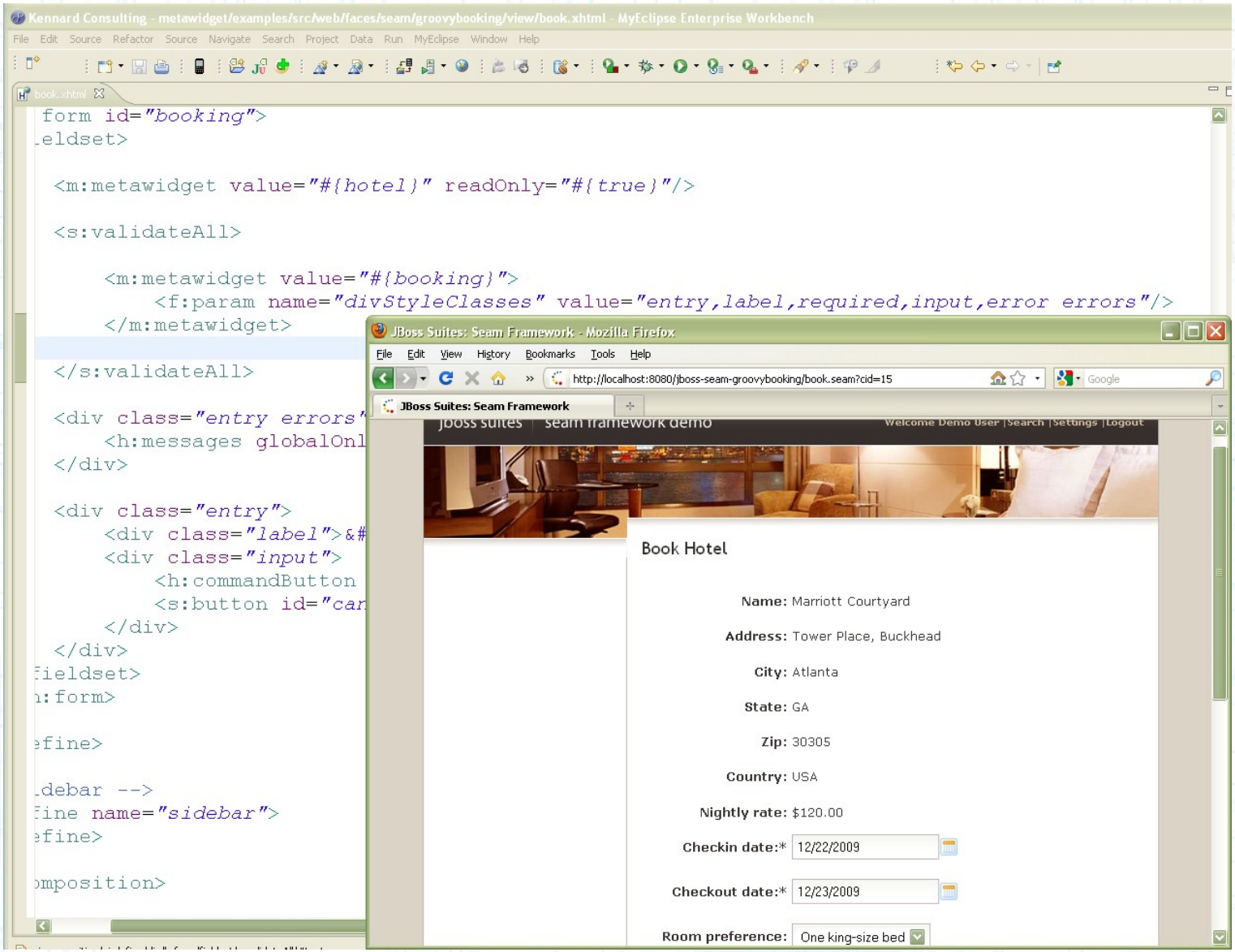
A better way

Doesn't 'own' the UI



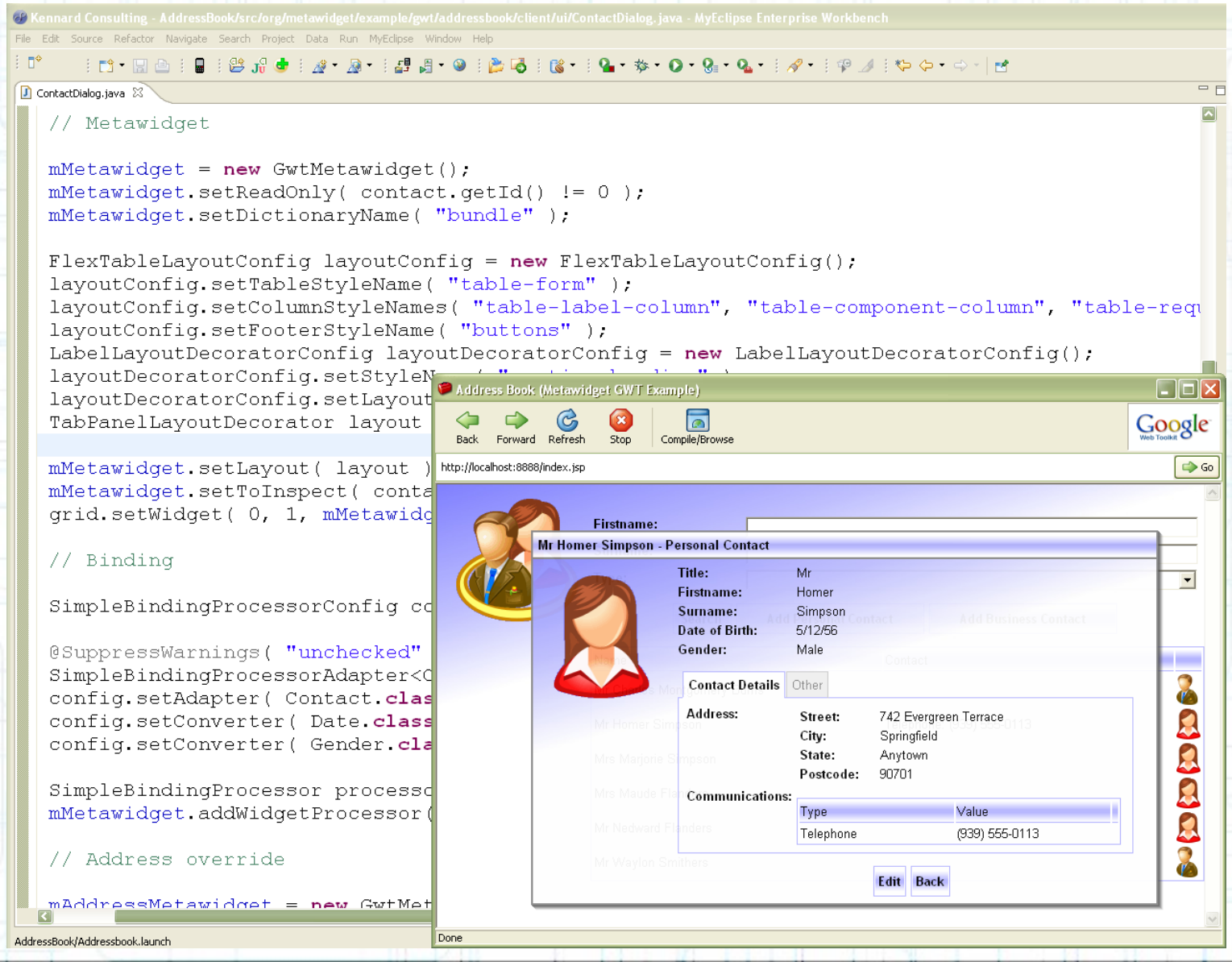
A better way

Doesn't 'own' the UI



A better way

Doesn't 'own' the UI



Conclusion

- Everyday requirement
- Unsatisfactory current practices
- A better way

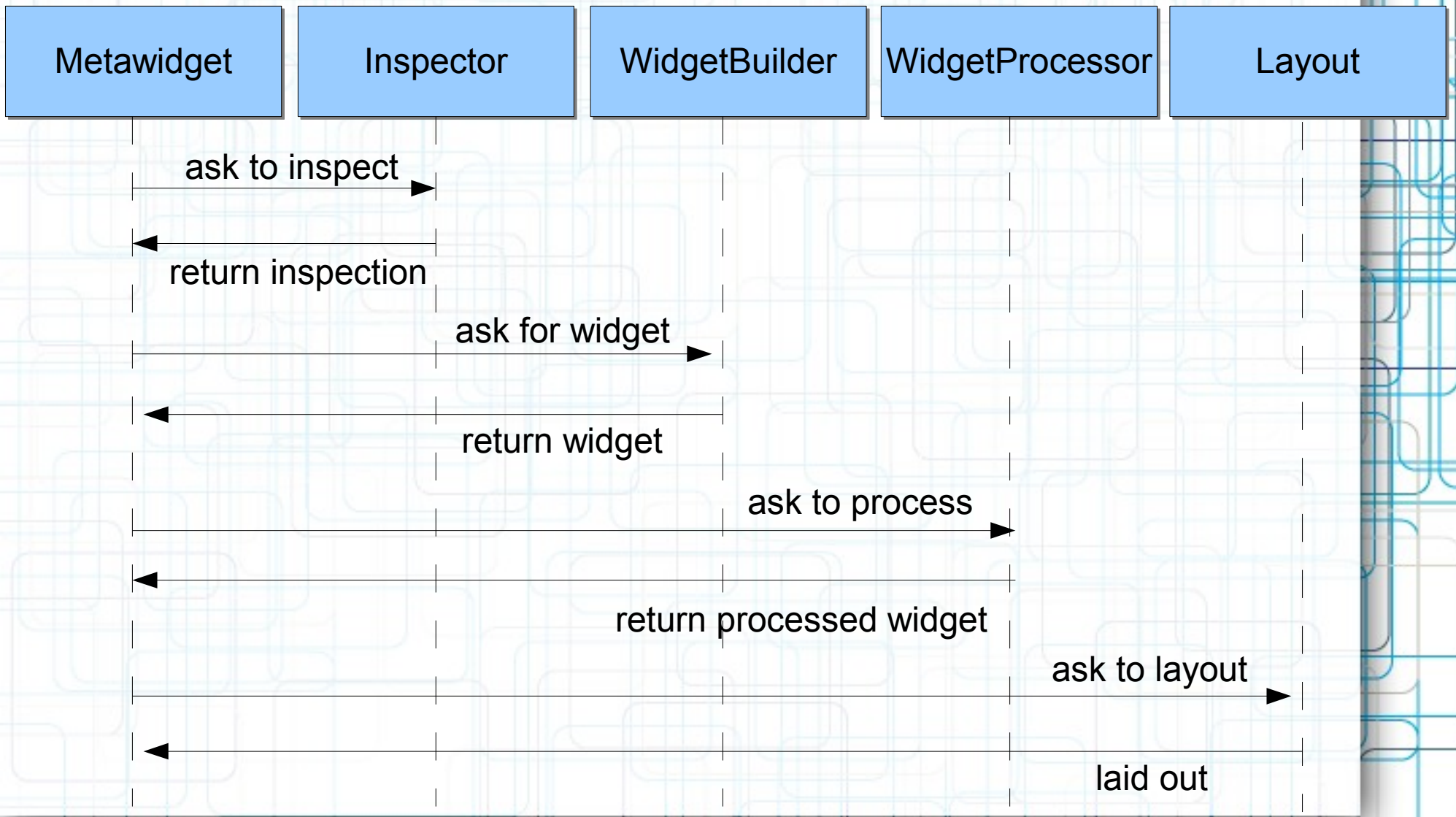
Thank You!

Questions?

Appendix A

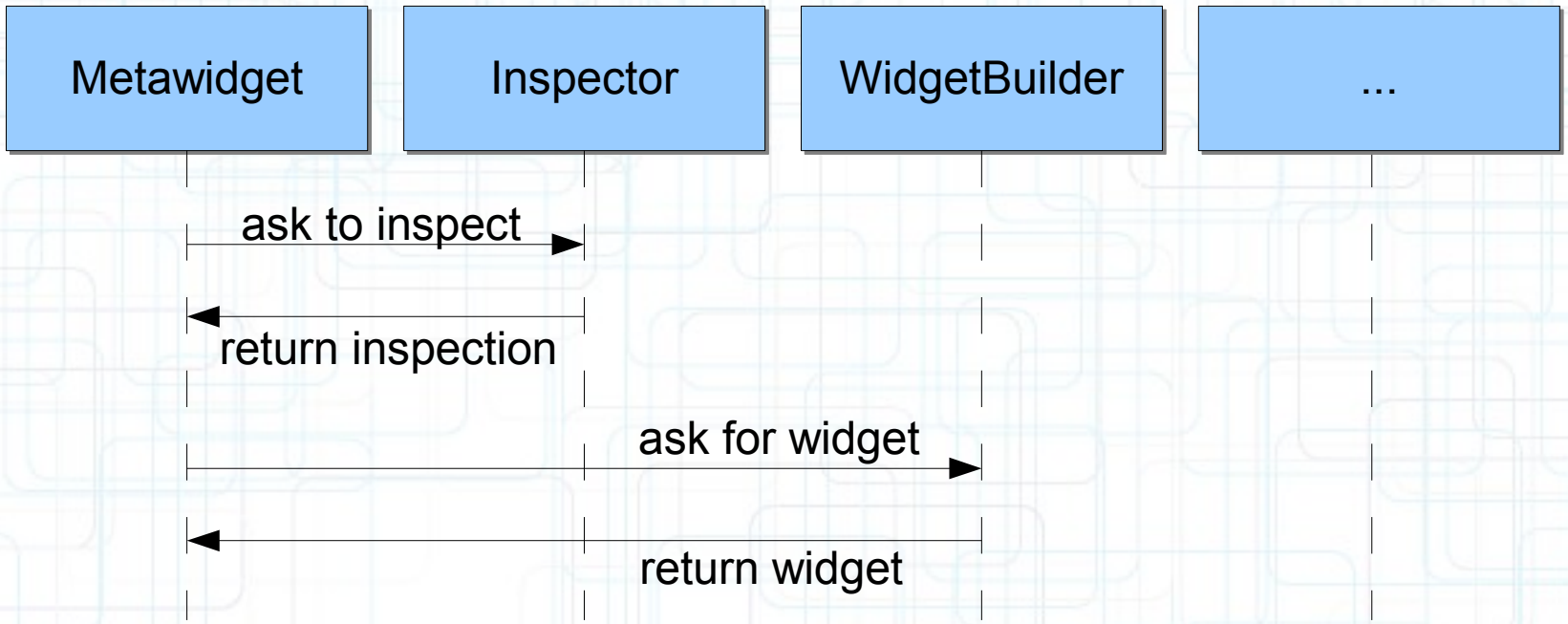
A better way

Uses your existing architecture



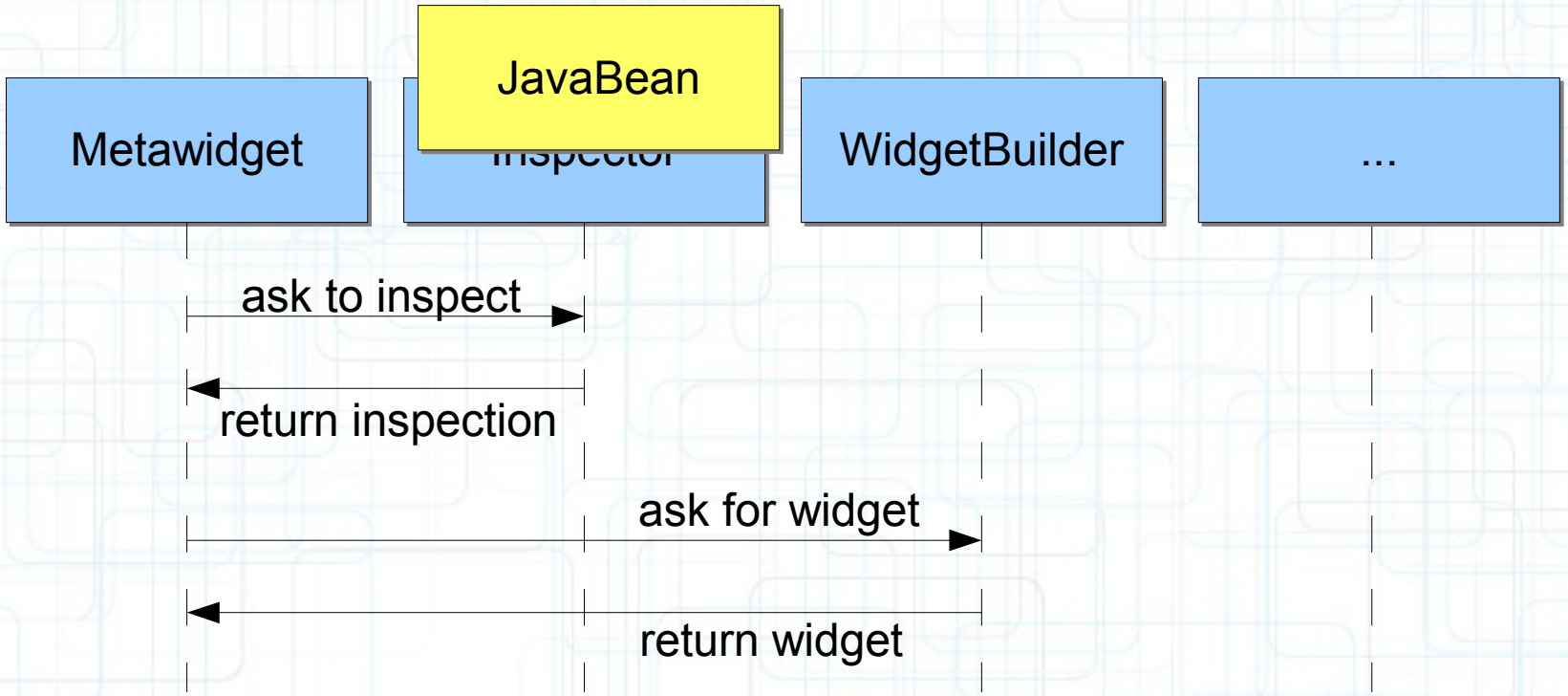
A better way

Uses your existing architecture



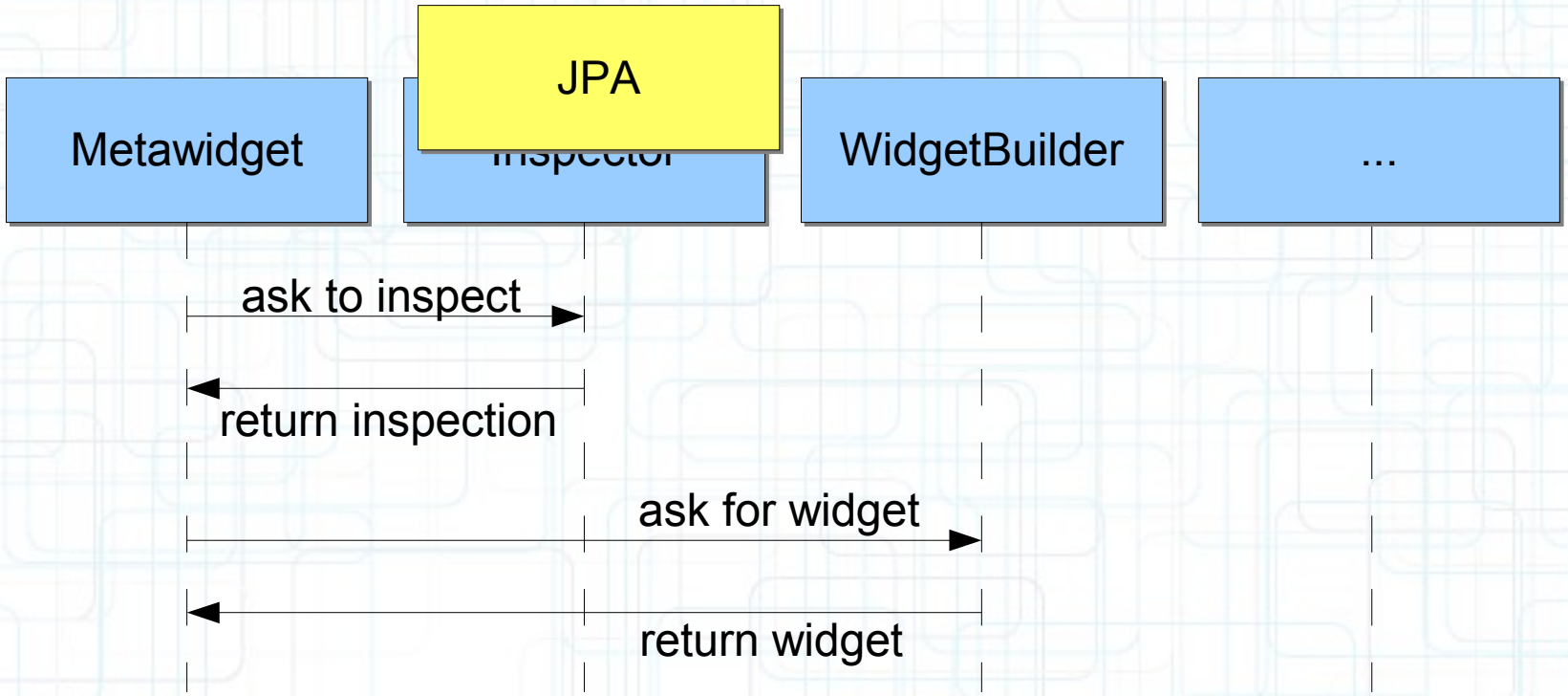
A better way

Uses your existing architecture



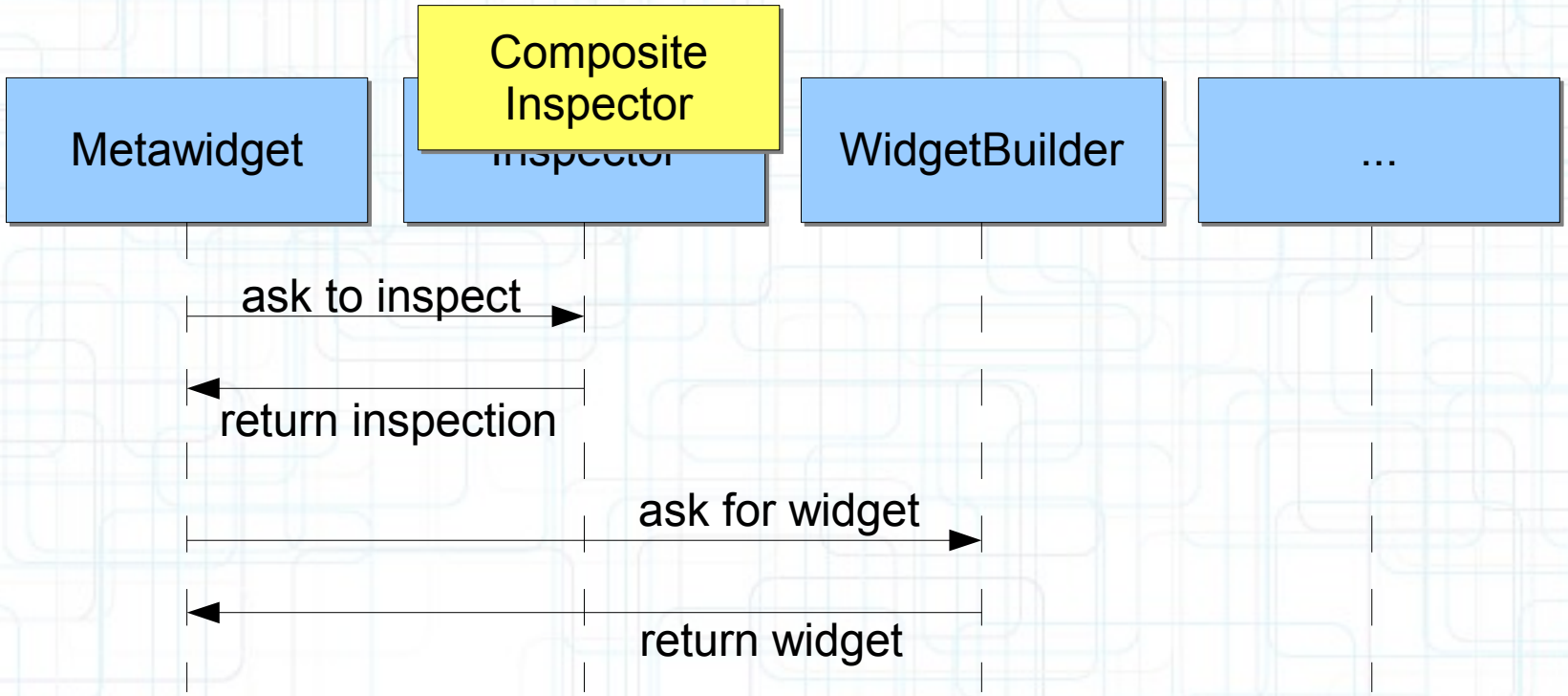
A better way

Uses your existing architecture



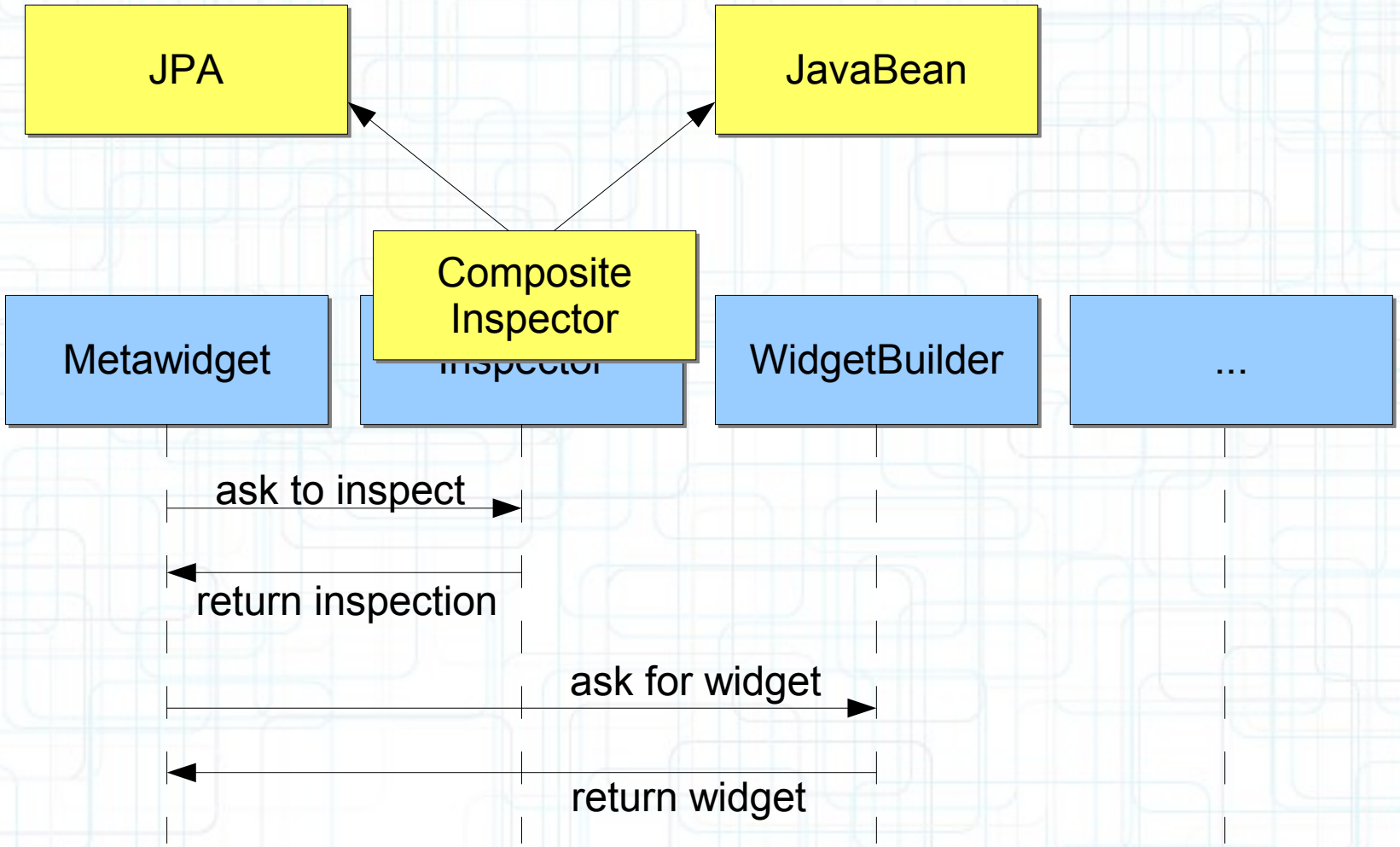
A better way

Uses your existing architecture



A better way

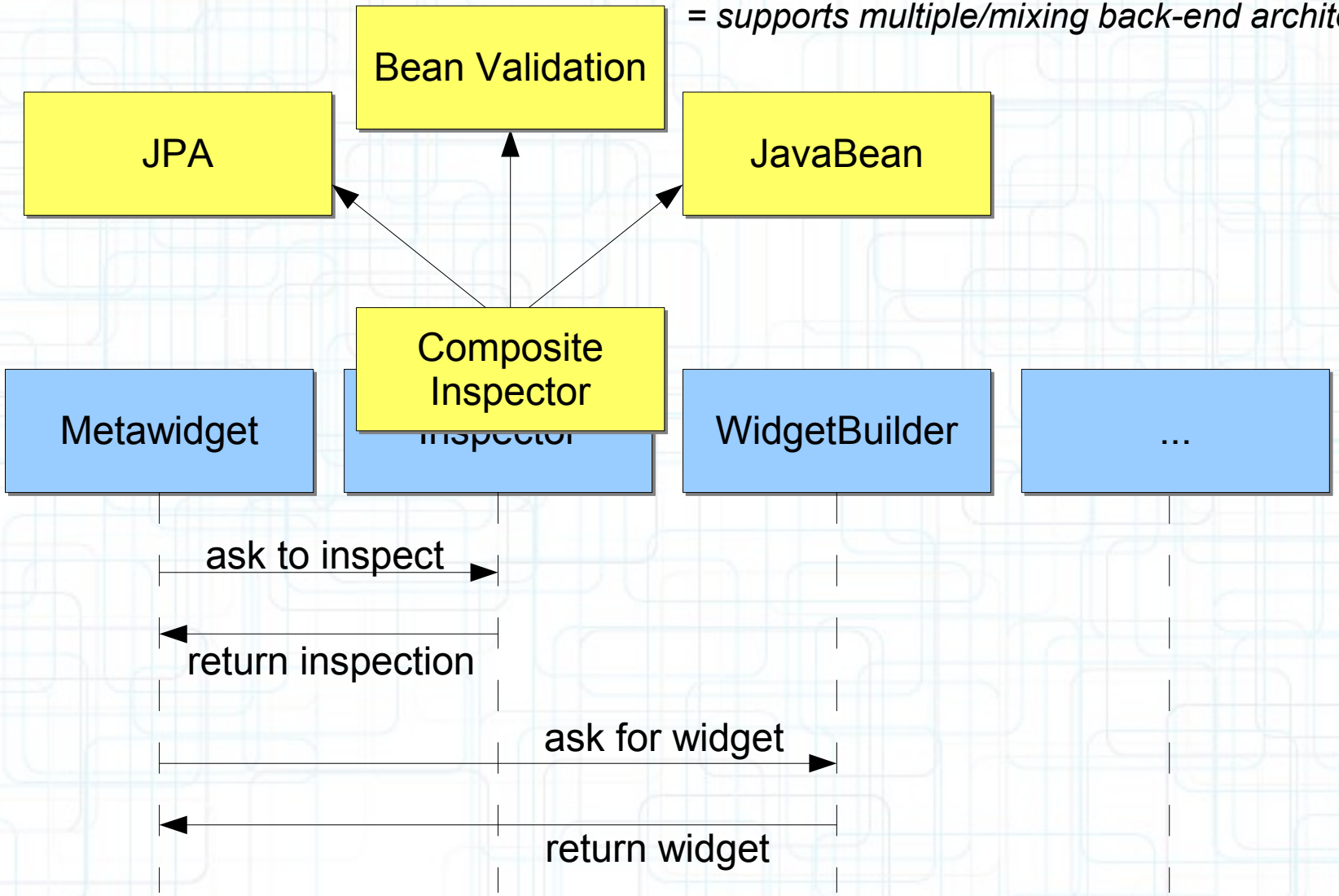
Uses your existing architecture



A better way

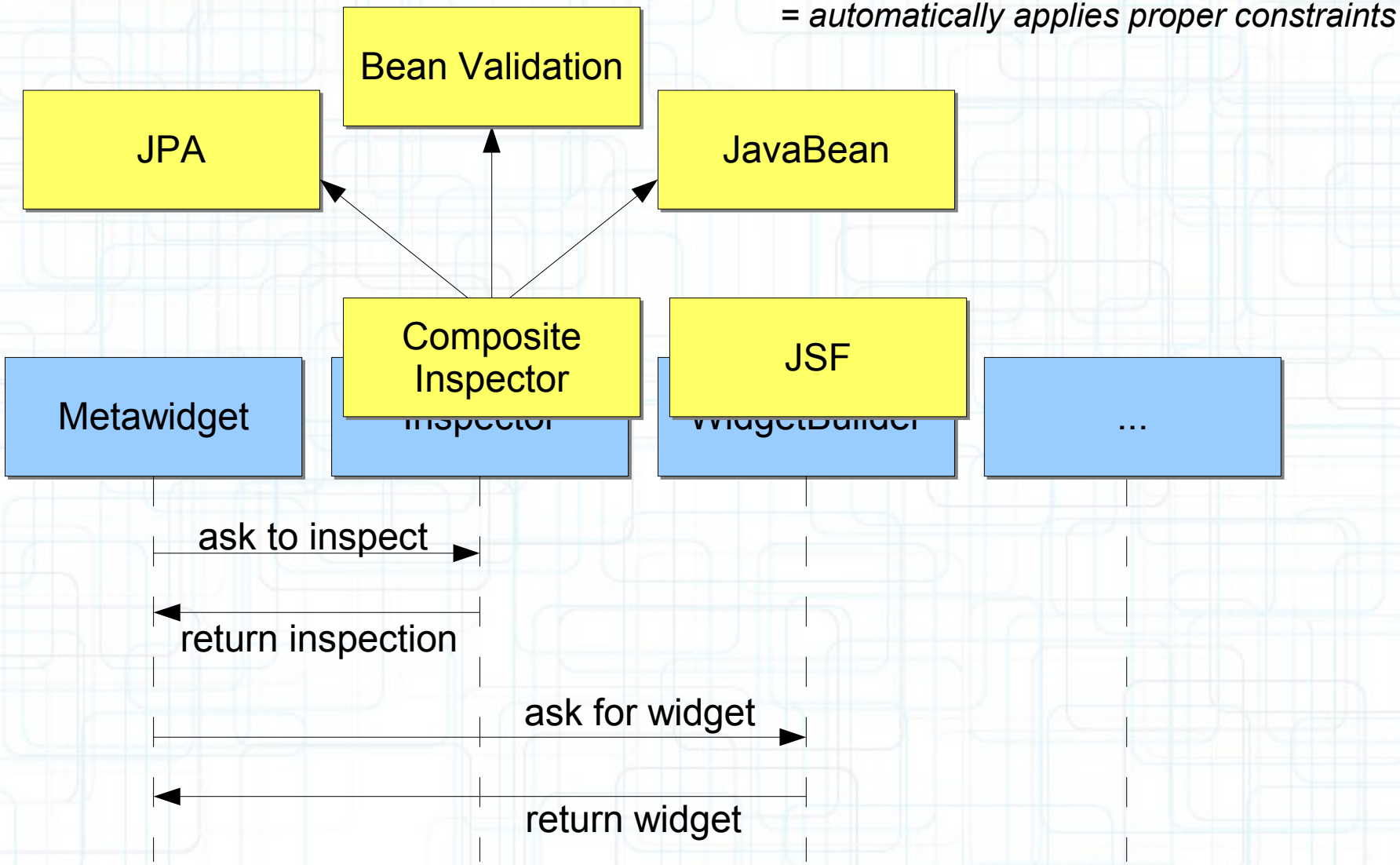
Uses your existing architecture

= no duplicate definitions from other layers
= supports multiple/mixing back-end architectures



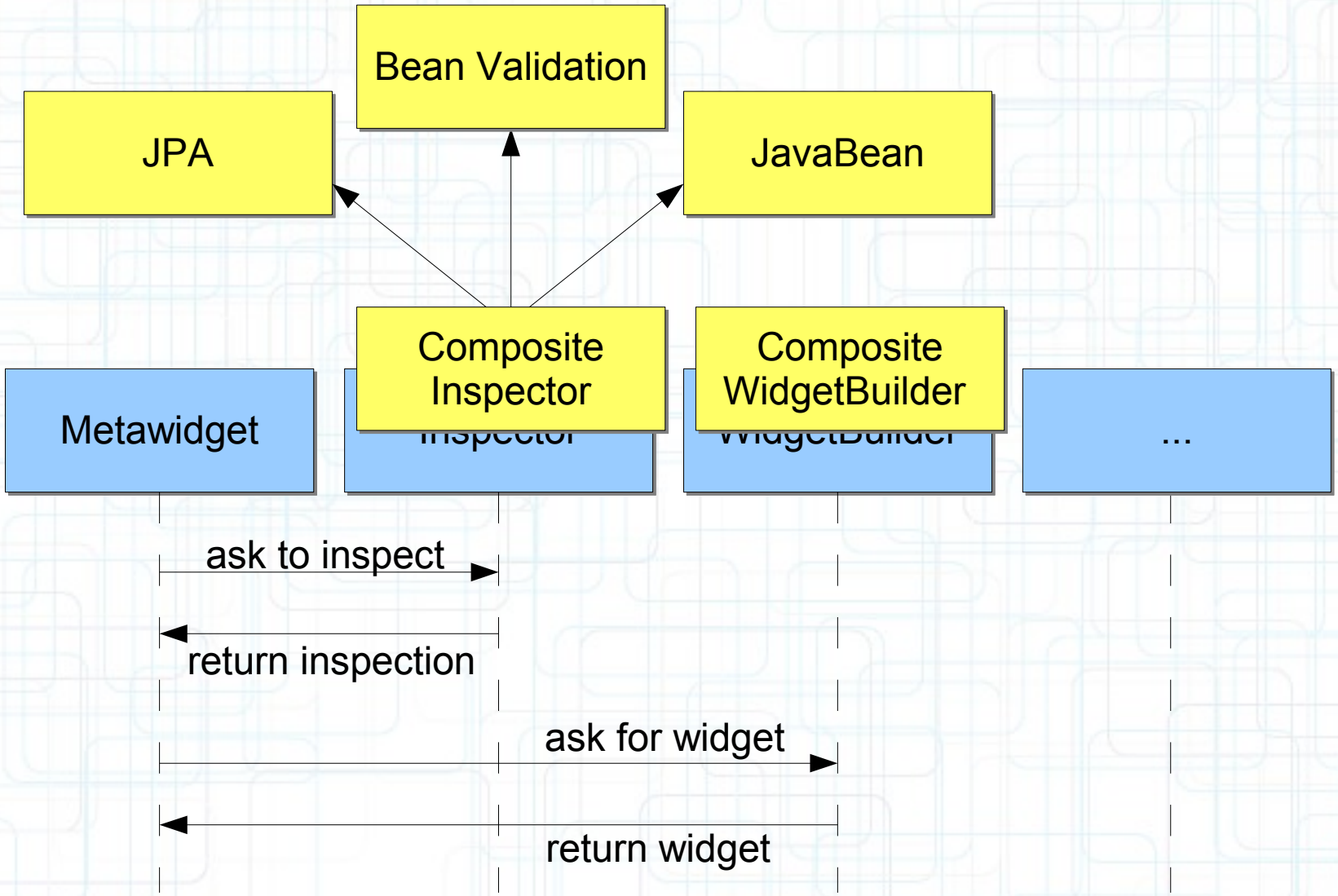
A better way

Uses your existing architecture



A better way

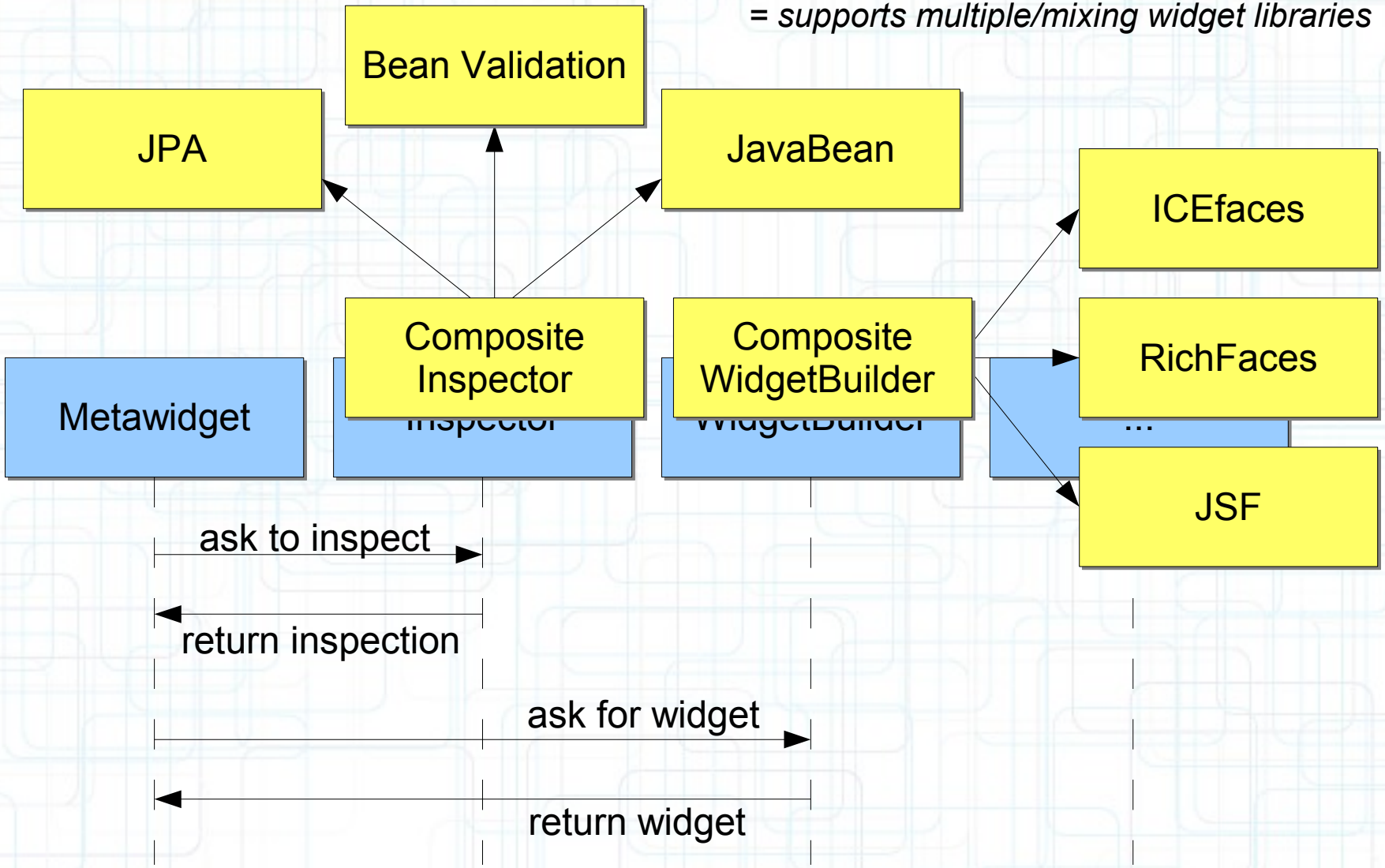
Uses your existing architecture



A better way

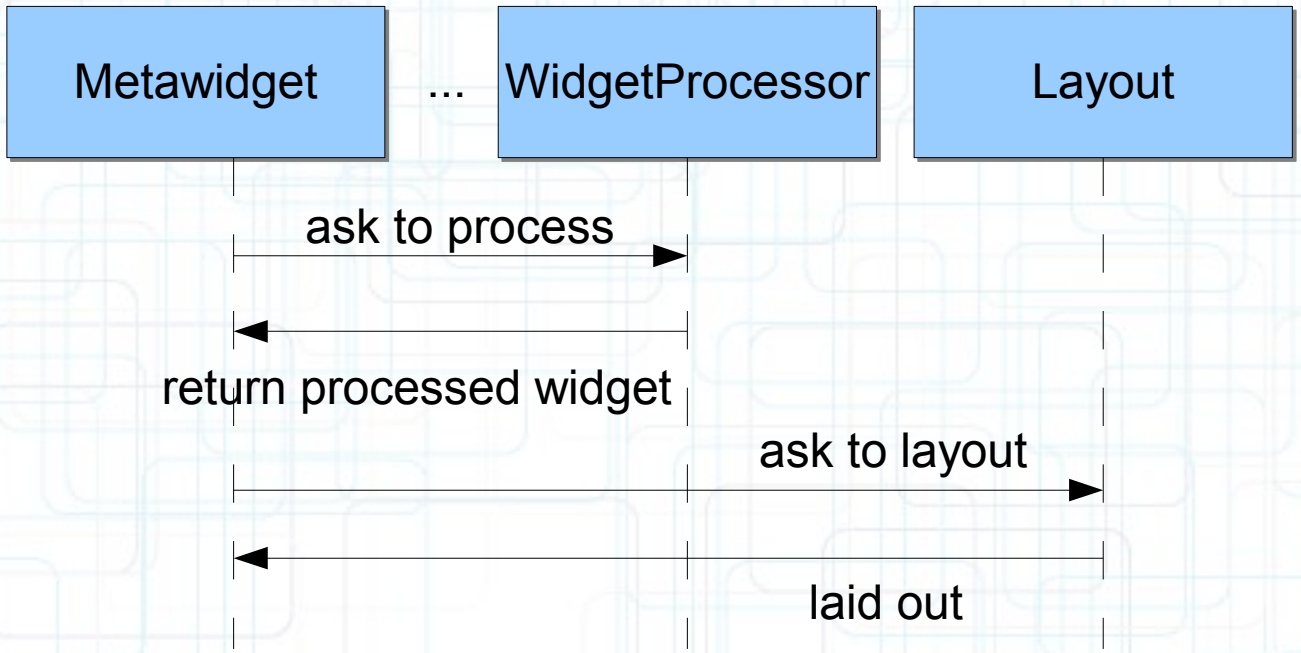
Uses your existing architecture

= new widgets can be swapped in en masse
= supports multiple/mixing widget libraries



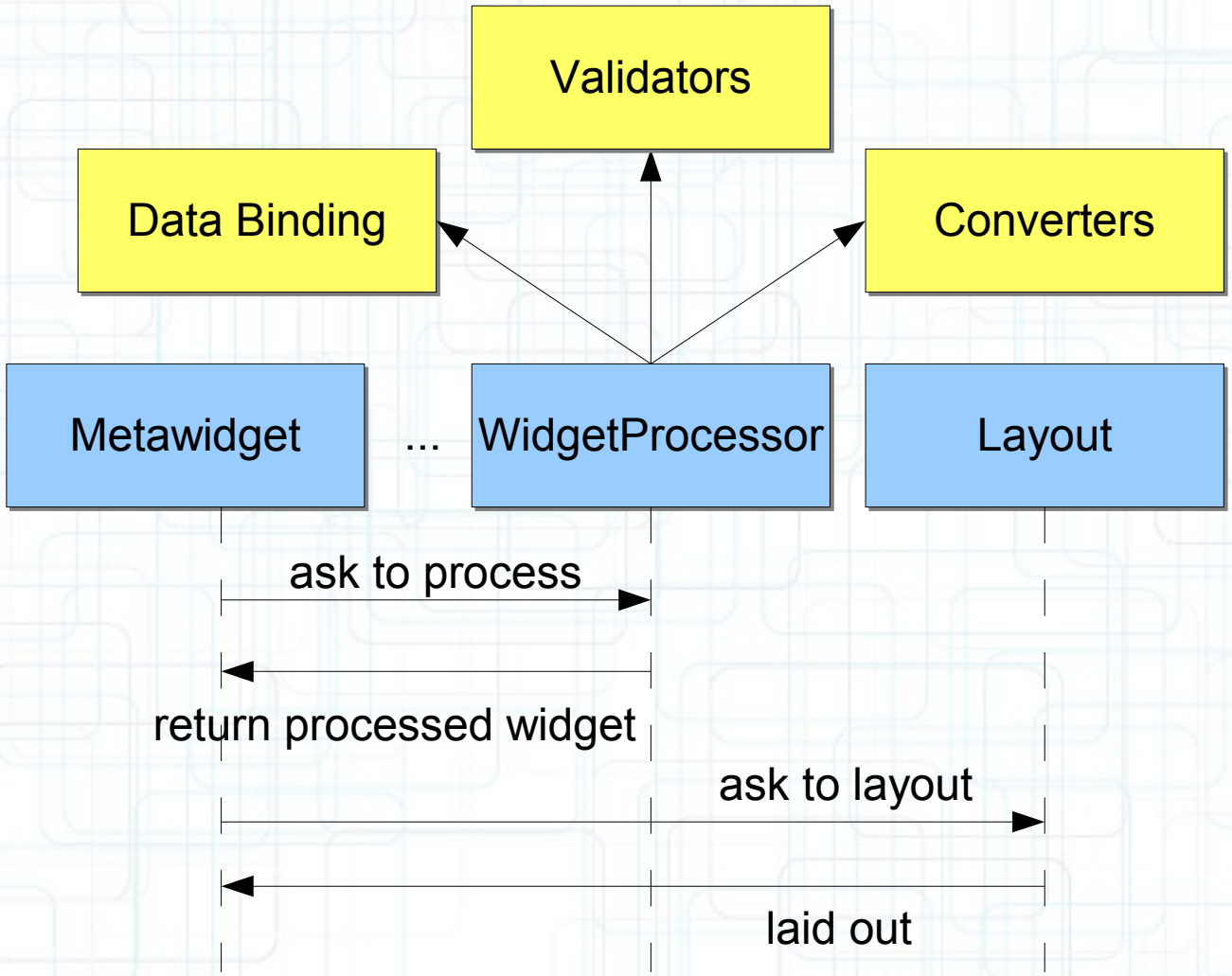
A better way

Uses your existing architecture



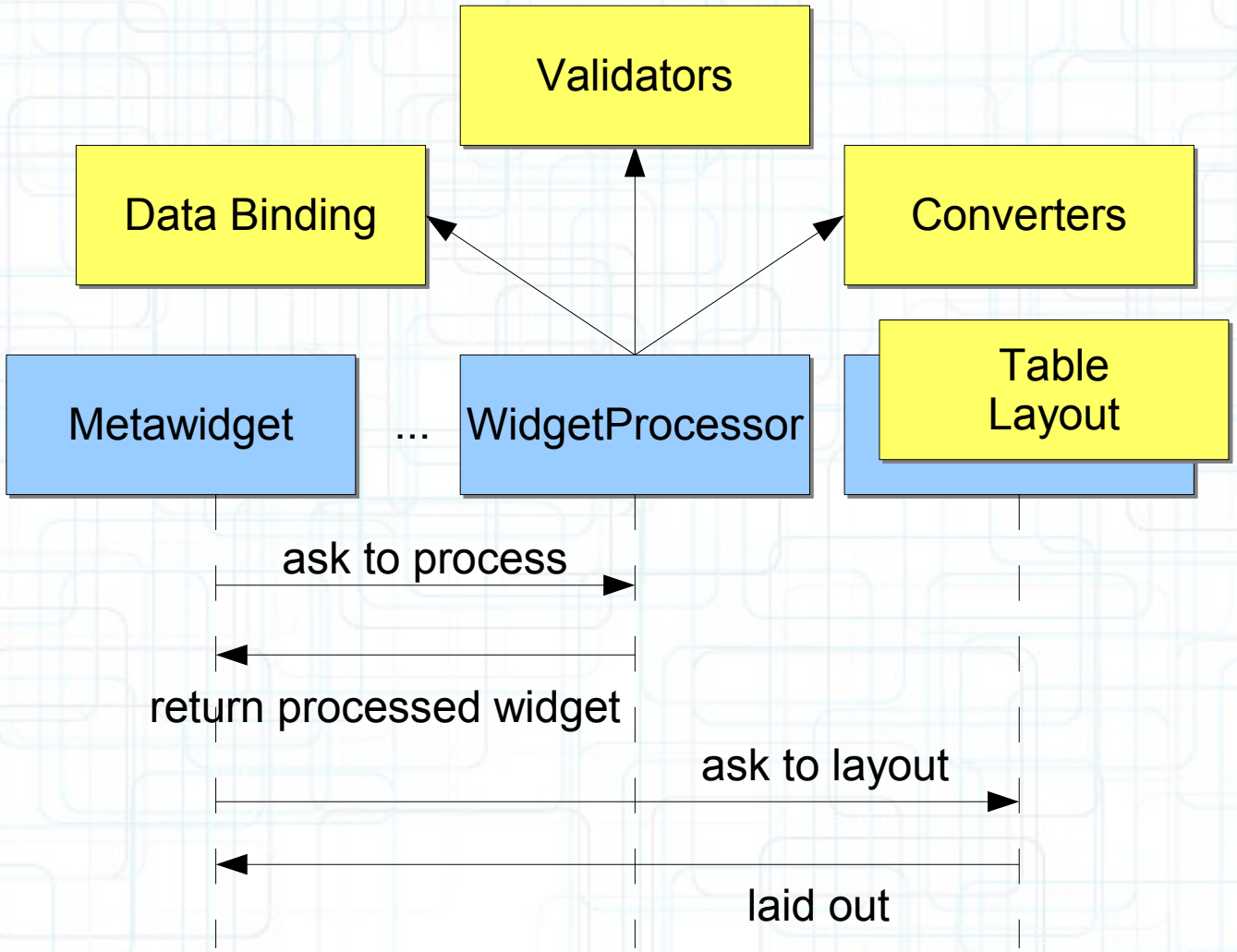
A better way

Uses your existing architecture



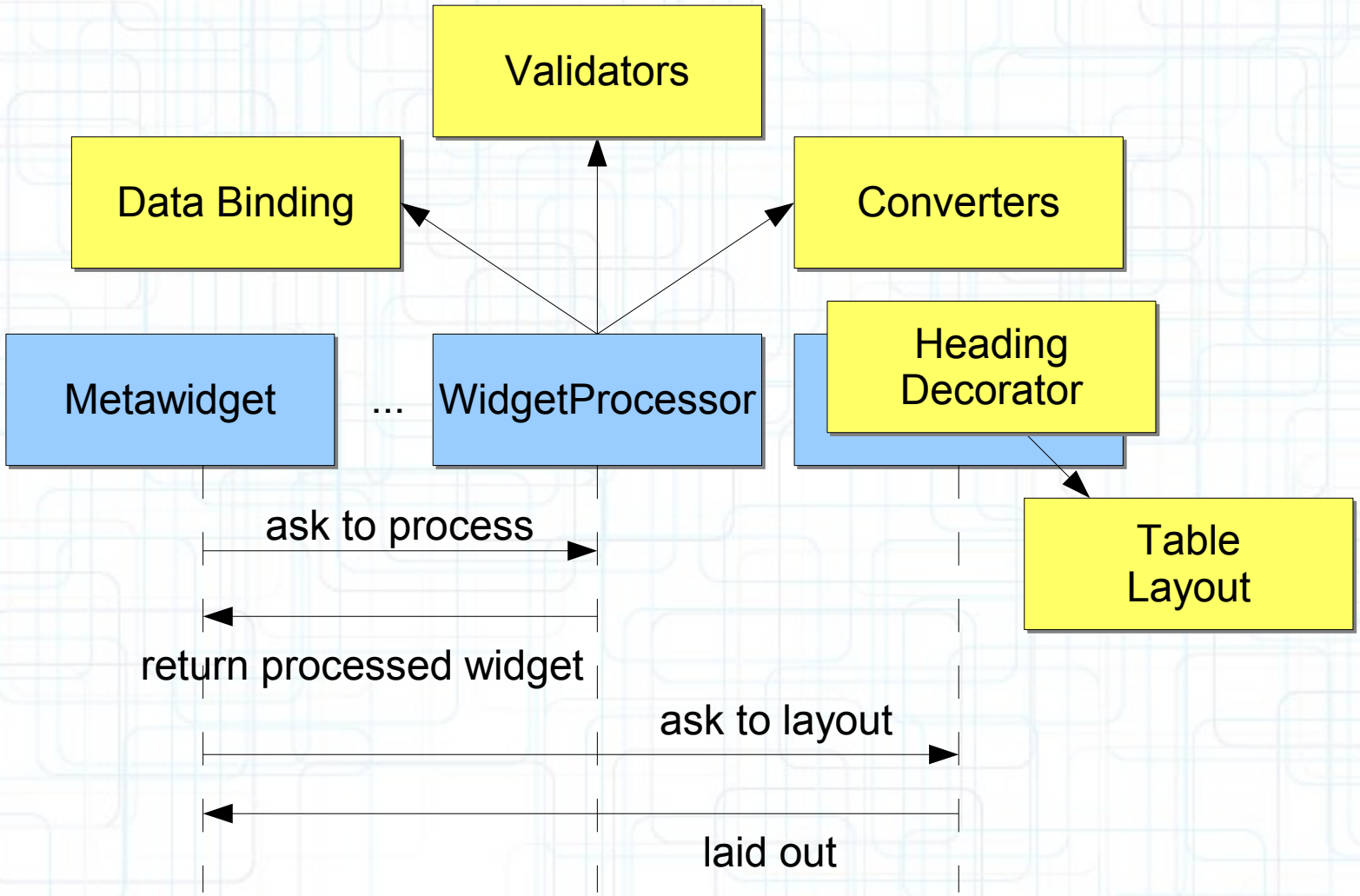
A better way

Uses your existing architecture



A better way

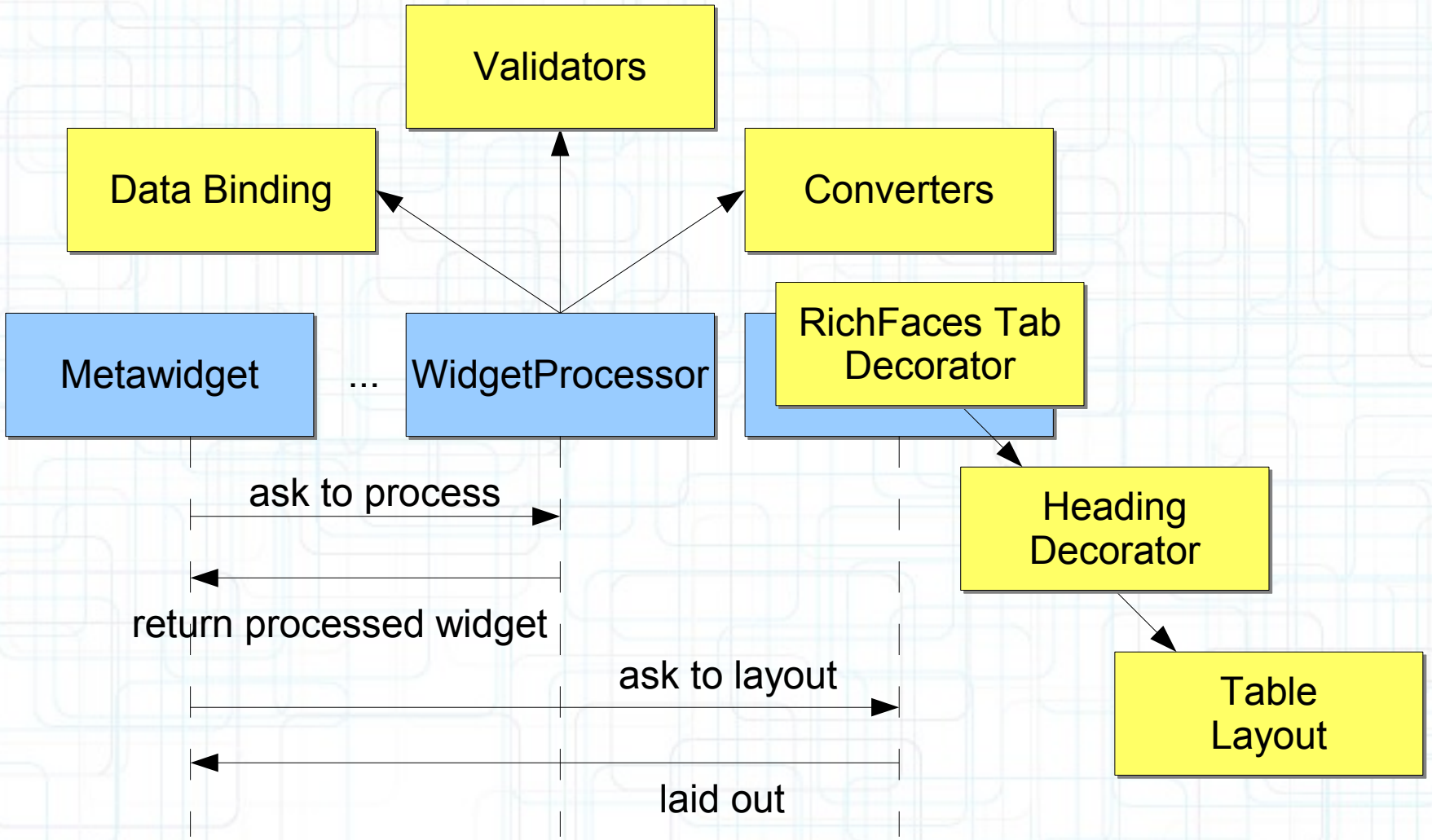
Uses your existing architecture



A better way

Uses your existing architecture

= automatic consistency across forms

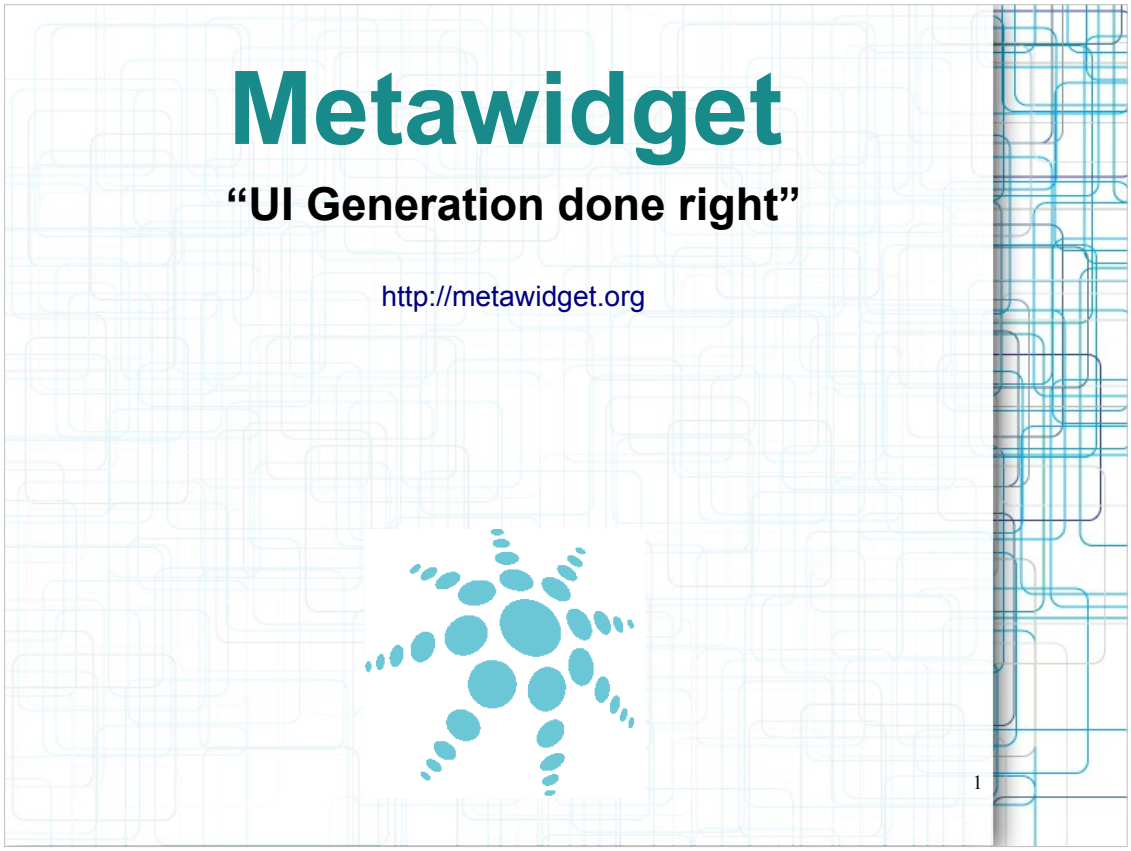


Appendix B

A better way

Acid Test:

Retrofitting
an existing app



Hello everyone and thanks for coming to this presentation about Metawidget.

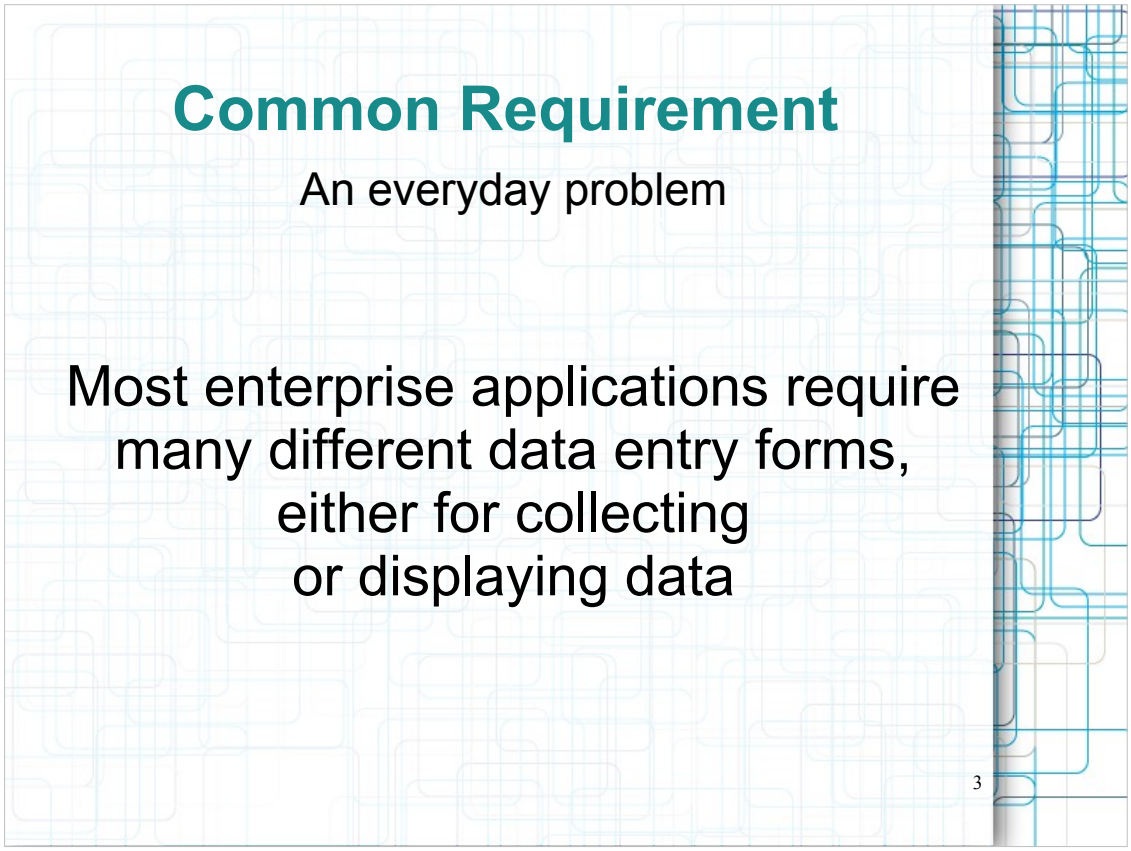
What is Metawidget? Well, as is rather boldly written there, Metawidget is 'UI generation done right'. And I guess over the next 20 minutes I'm going to try and convince you of that.

So, let's get started.

I'm going to briefly state what the problem is we're trying to solve - this common requirement that we all encounter.

Then I'll review what most of us do about it, how none of our existing approaches work very well in practice, and outline some common pain points.

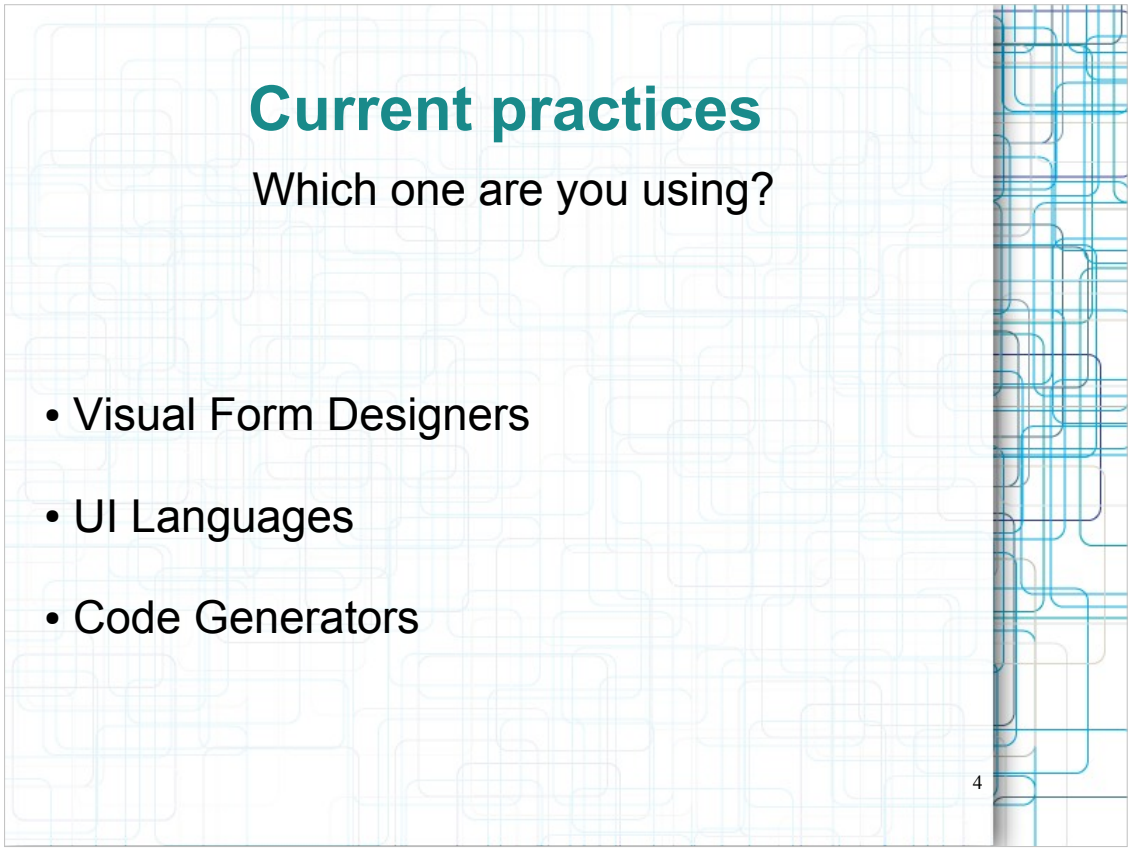
Finally I'll show you what Metawidget is trying to do about each of those pain points, to try and make them all a little bit better.



Most of us building enterprise apps today need to build lots of screens that are basically data entry forms.

They're either collecting data or displaying data, and they're tied to some back-end POJO or other way of storing them.

Even if the heart of your application is not form-based, like Google Maps or something, you've still got all these periphery forms for profile screens, preferences screens, contact us pages, you know the sort of thing.



So how do we build all these forms?

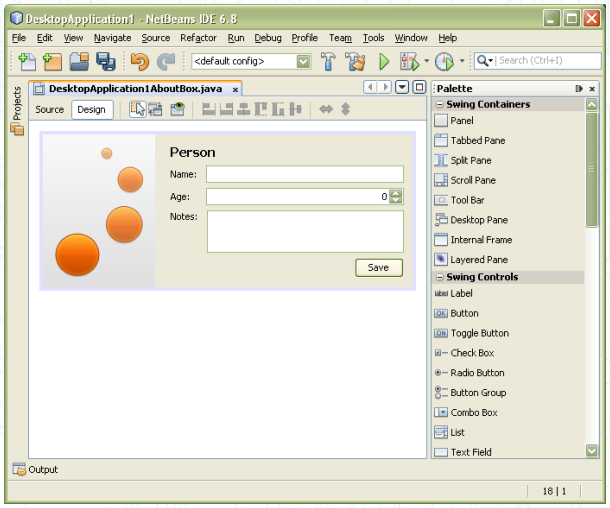
Well, there are three basic approaches that people use today, and all of them have some serious pain points.

Let's review those pain points.

Current practices

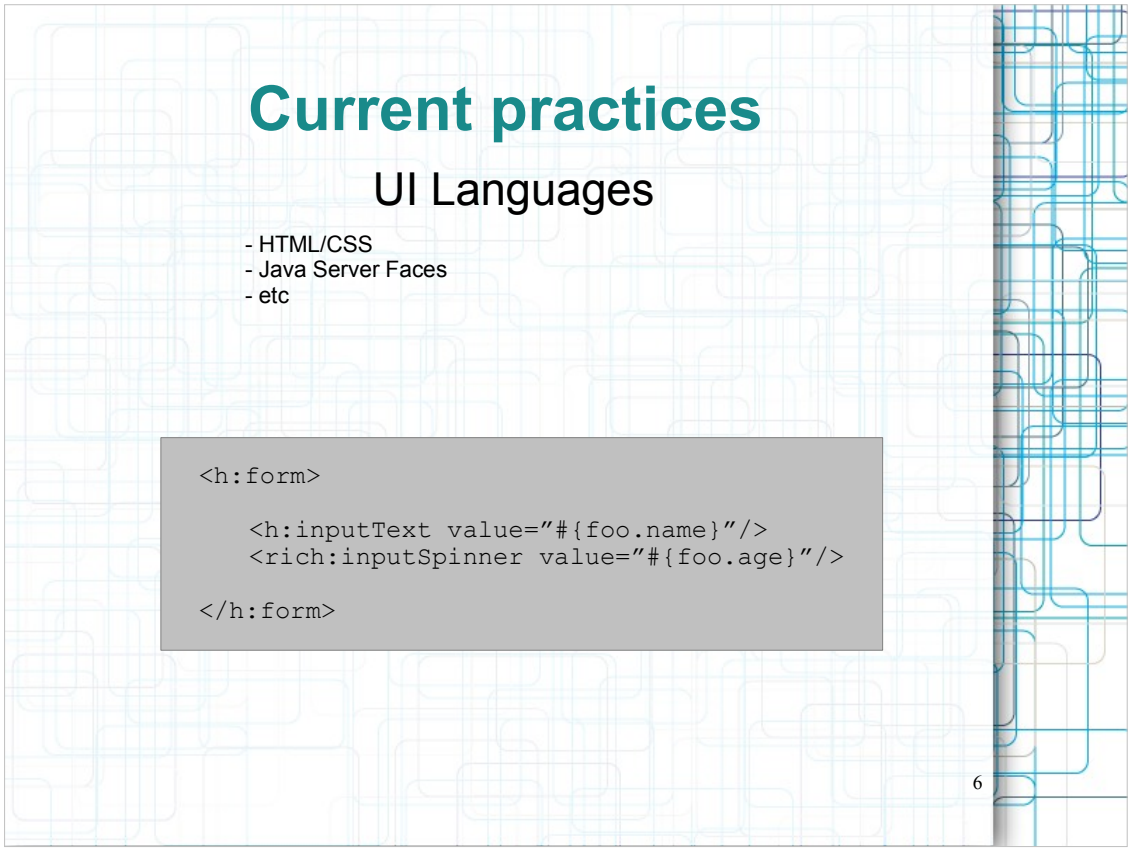
Visual Form Designers

- Matisse
- JBoss Visual Page Editor
- etc



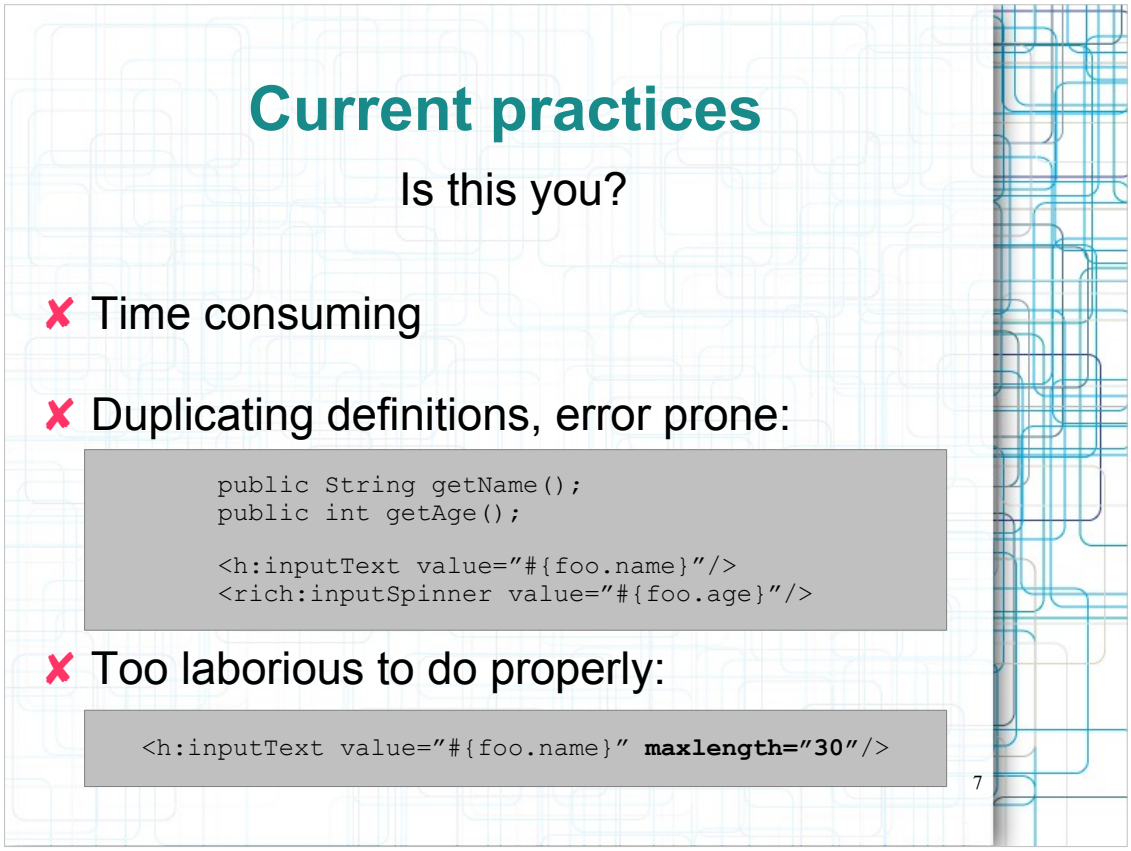
Some people code their forms using 'visual form designers'.

By which I'm referring to something where you have a palette of widgets that you visually drag and drop into place. Things like Matisse for Swing, or the Visual Page Editor for JSF.



Other people use intermediate 'UI languages'. Things like JSP, or Facelets.

These are more 'declarative' than the visual tools – you're not directly positioning things down to the pixel.



But both visual form designers and UI languages share the same pain points.

For a start they are very fiddly, very time consuming. If you have a lot of forms you have to drag-and-drop, or write declarations for, every single widget and label.

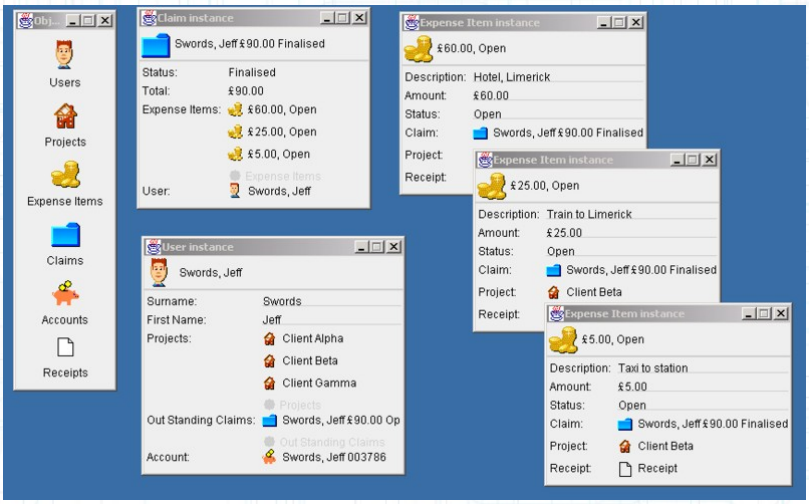
And this is error-prone because each of those widgets and labels has to directly match up with definitions of things in the back-end: either in your POJOs, or in your database schema, or somewhere else

In fact, it's so fiddly and error-prone we generally don't even *try* to do it properly. For example, who here puts 'maxlength' on their text fields?

Current practices

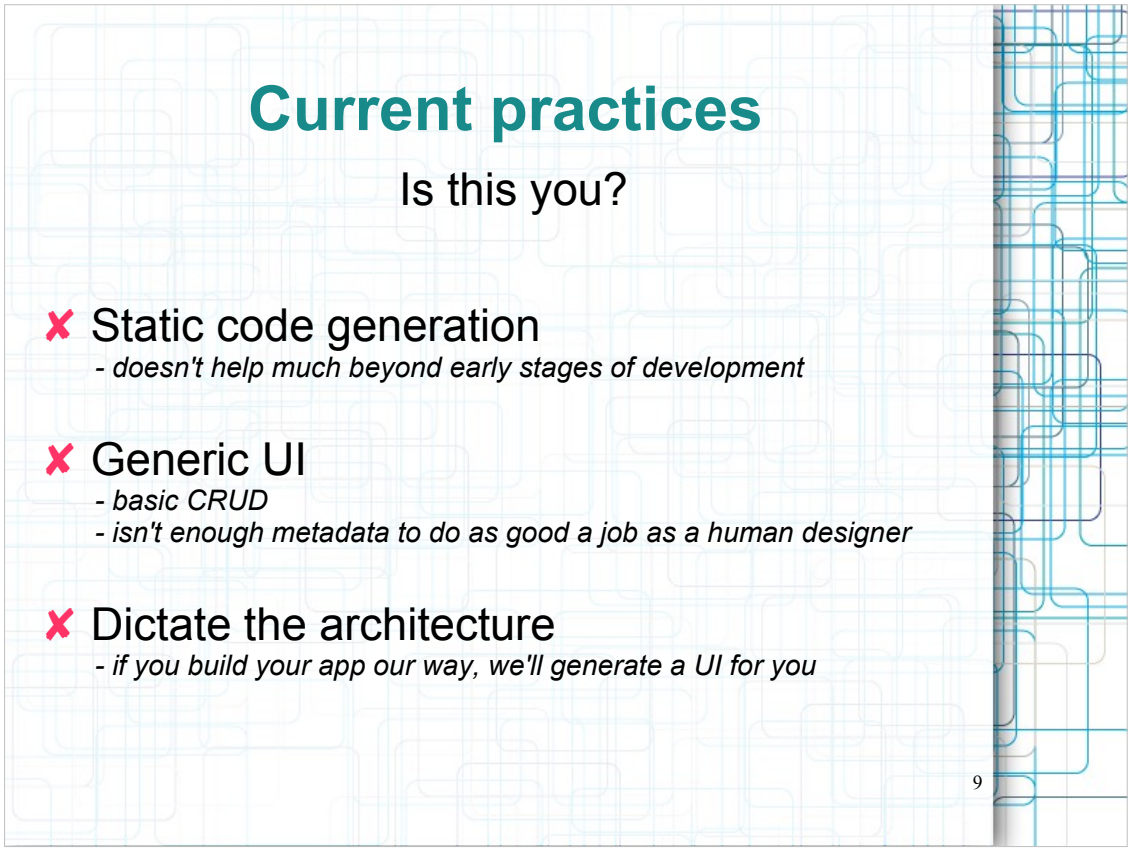
Code Generators

- Naked Objects
- seam-gen
- etc



Other developers use 'code generators' to automatically generate their forms.

By code generators I mean things like seam-gen, Naked Objects, OpenXava.

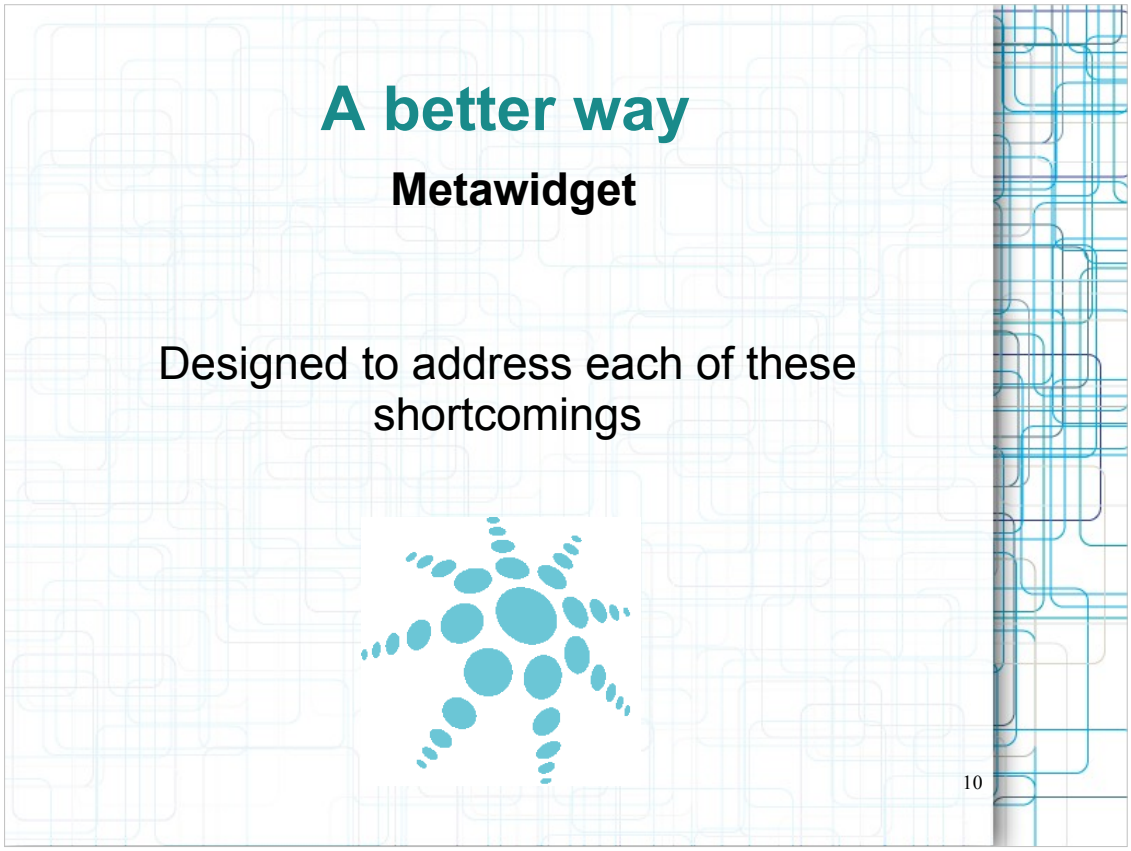


Code generators actually solve many of the problems of visual form designers and UI languages, but have some paint points of their own.

Not all of them use static code generation, but those that do suffer from the problem all static code generators have (not just UI generators): once you start working with the generated code, regeneration is never pretty.

Code generators also tend to generate very generic looking UIs. Too generic for most things beyond basic CRUD apps, and nowhere near as good as a human designer.

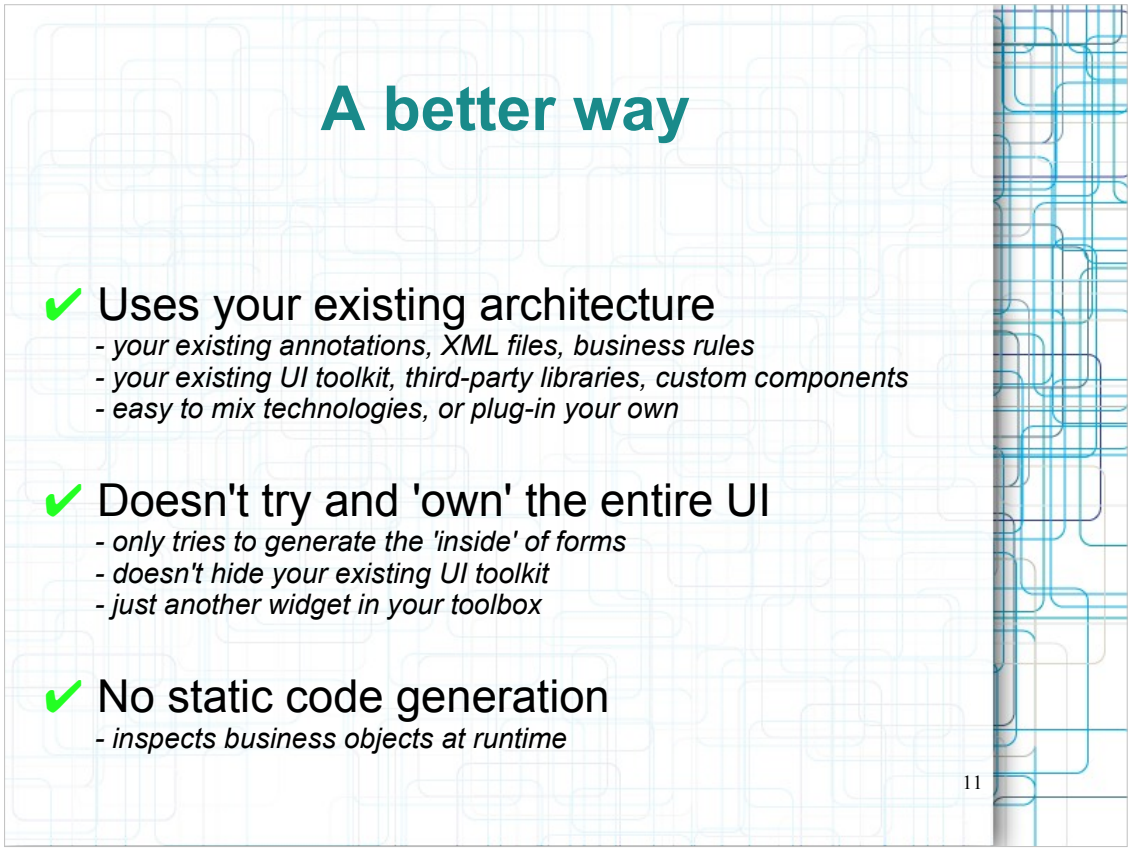
Finally, and really importantly, they tend to constrain the technologies you're allowed to work with. They say 'if you use JPA and Hibernate Validator we'll generate you a Web UI' or 'if you put all your business logic inside your POJOs we'll build you a Swing UI'. And that really limits where you can use them.



But despite all the pain points of our current approaches, this common requirement of building lots of forms isn't going away. So what can we do?

Metawidget is different. It's specifically designed to address each one of the pain points I've outlined. It tries to be 'UI generation done right'.

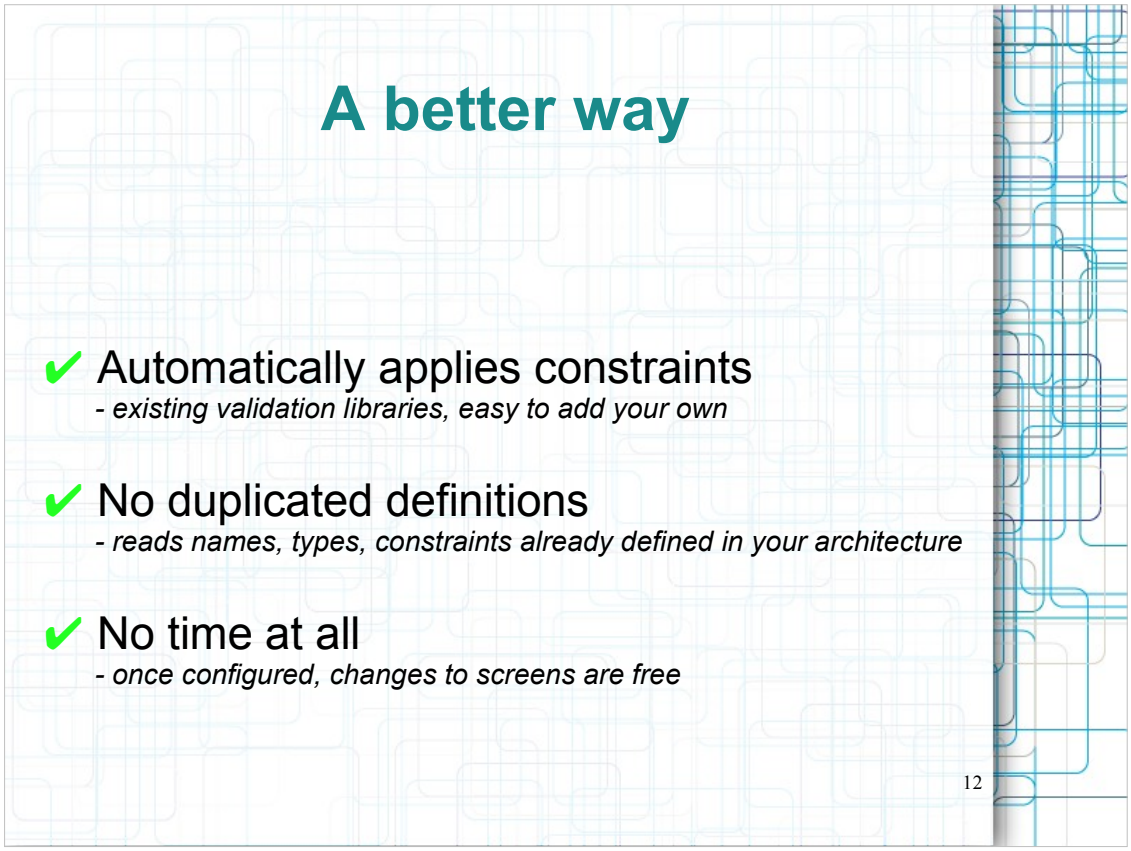
So, if you had your ideal UI generator, what would it look like?



For starters it should work with whatever architecture you chose. *Your* combination of ORM, validation, business rules, app framework. It should have a bunch of plugins you can mix and match, or add your own. And this extends to *your* combination of UI toolkits, third party widget libraries, or custom components. It should be just another part of your EE stack, not try to dictate your whole architecture.

Second, it shouldn't try and generate the entire UI: shouldn't 'guess' some generic CRUD screens. It should stick to just building what is already tightly defined by the back-end, and not hide your existing UI tools or restrict you from developing the rest of your UI the way you normally do. You could develop a Word Processor, say, and use Metawidget just for the preferences screens. It's just another widget in your toolbox.

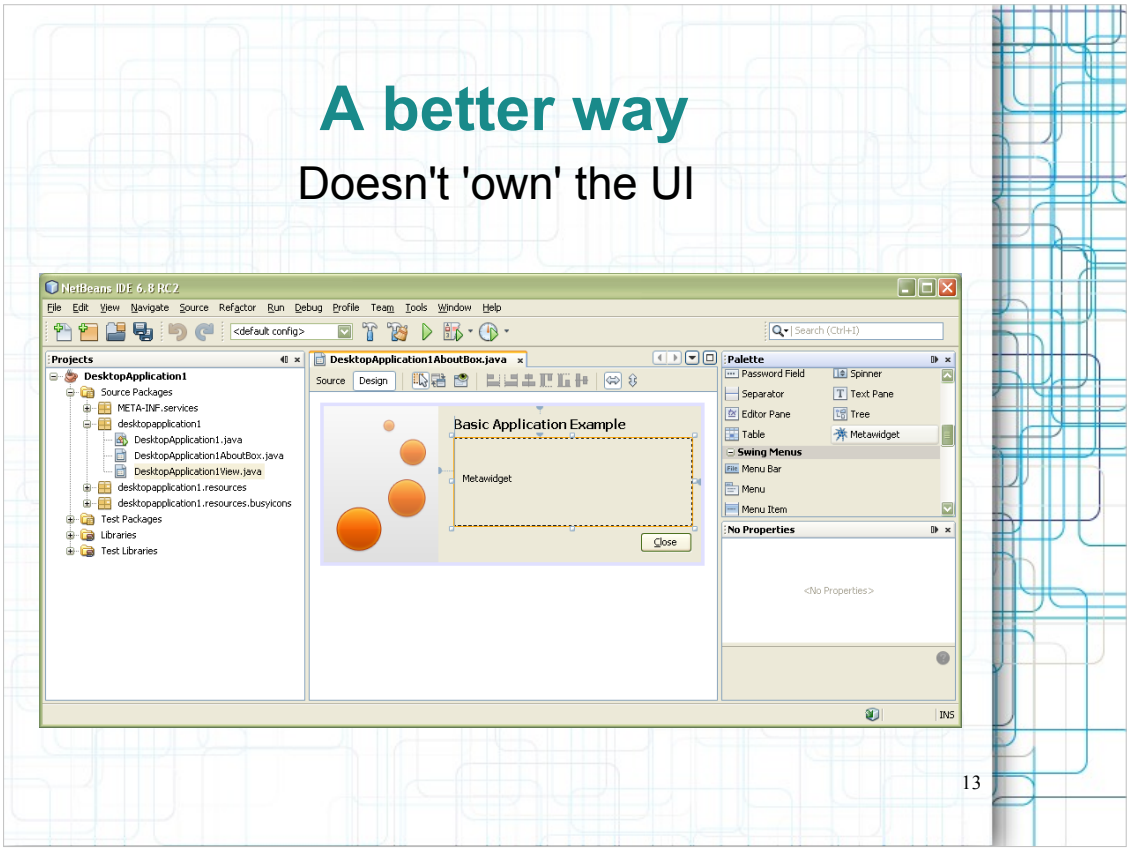
Third, there should no statically generated code. Everything should be runtime based, so it can change as your business objects change (and they change a lot!)



Next, it should give you a *better* UI than you'd do by hand, by applying all those validation constraints that are too fiddly to do yourself.

And in doing this, it should avoid you duplicating any definitions from the rest of your code. If it's in your Bean Validation layer, or in your JPA layer, or in your BPM rules, it's in your UI. Your code is really D.R.Y.

Finally, it should do all this automatically. You can configure and tweak it like crazy initially, but once it's off and generating your forms it should update them for you with no additional work. This even extends to swapping out widget libraries across your entire app, for example changing from ICEfaces to RichFaces.



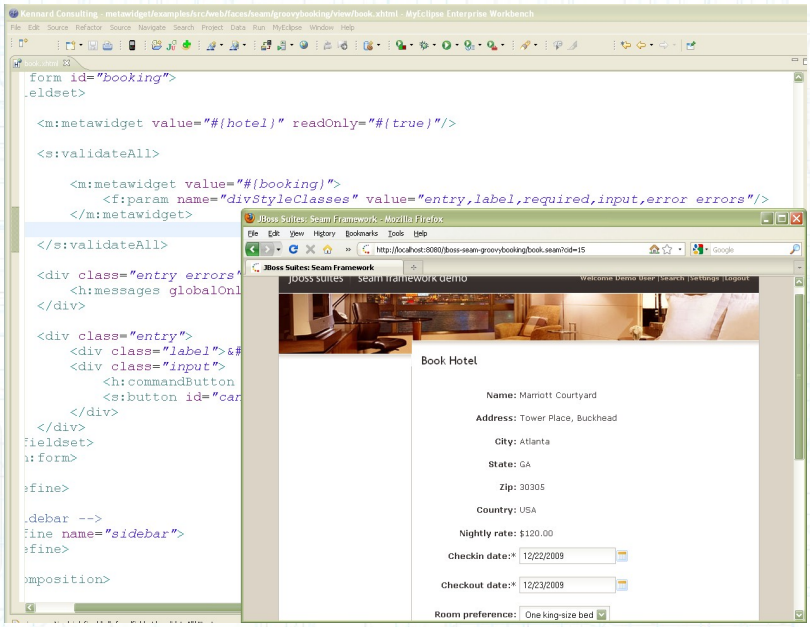
So, just to spend a little bit more time on one of those points.

Metawidget doesn't try and generate your whole UI. It's just a widget, and it presents itself in whatever way you're used to.

If you use a visual form designer, you can drag and drop a Metawidget from your palette just like any other widget. Except at runtime that widget will populate itself automatically.

A better way

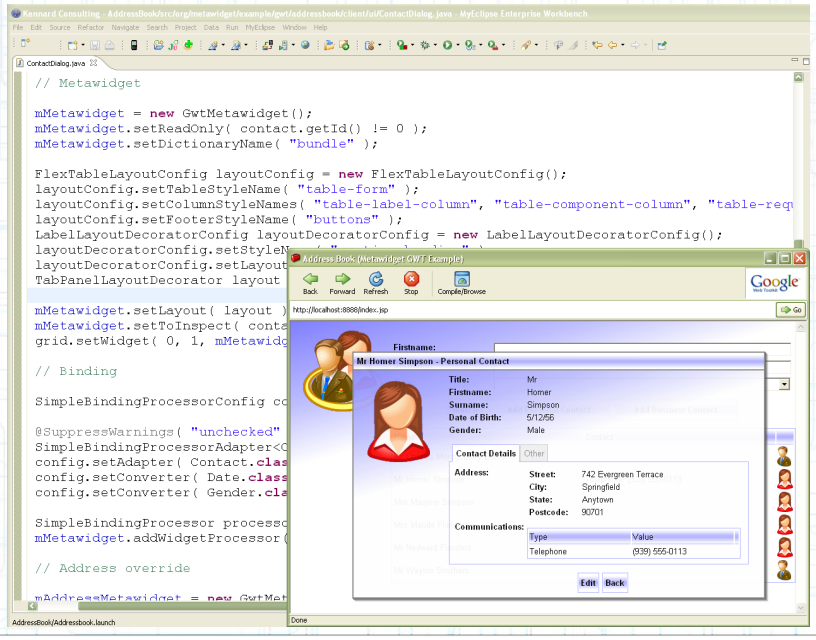
Doesn't 'own' the UI



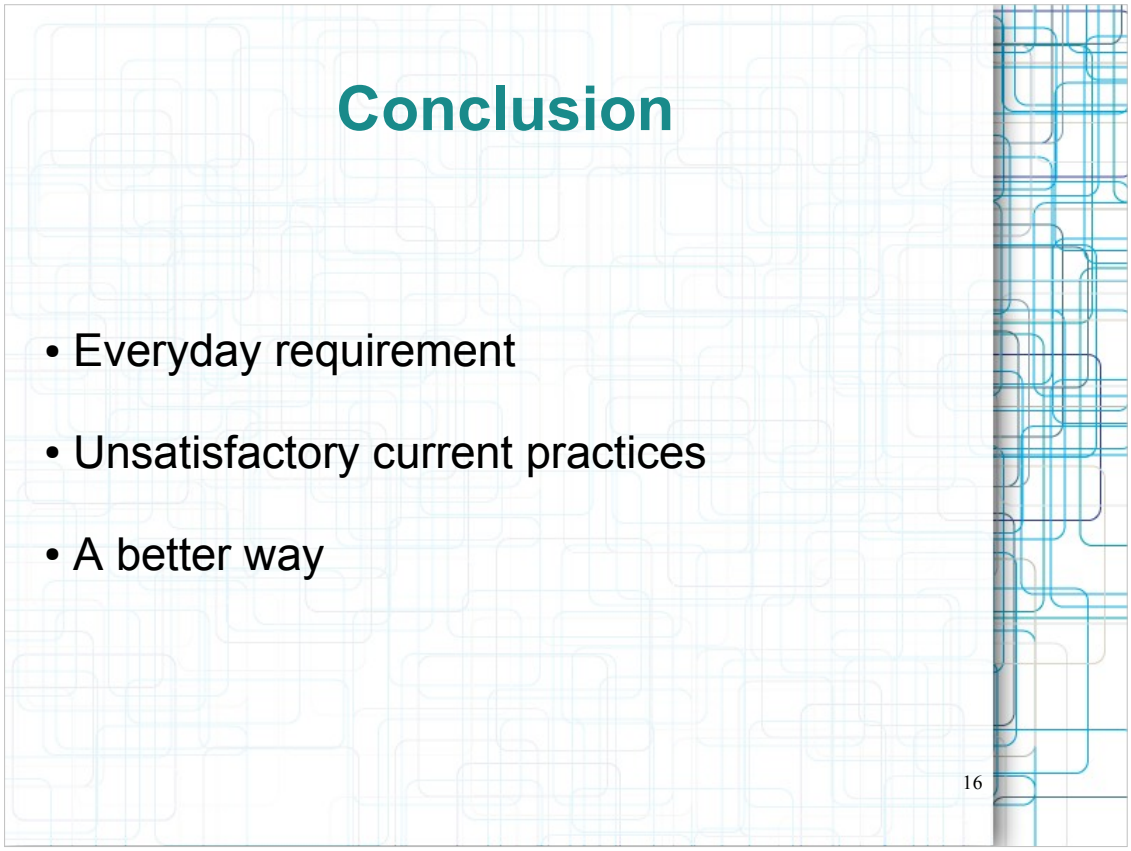
If you use a UI language, Metawidget is just another tag that you mix in amongst all your usual tags.

A better way

Doesn't 'own' the UI



And of course if you build your UIs programmatically, Metawidget fits in that way too, with a nice fluent API.

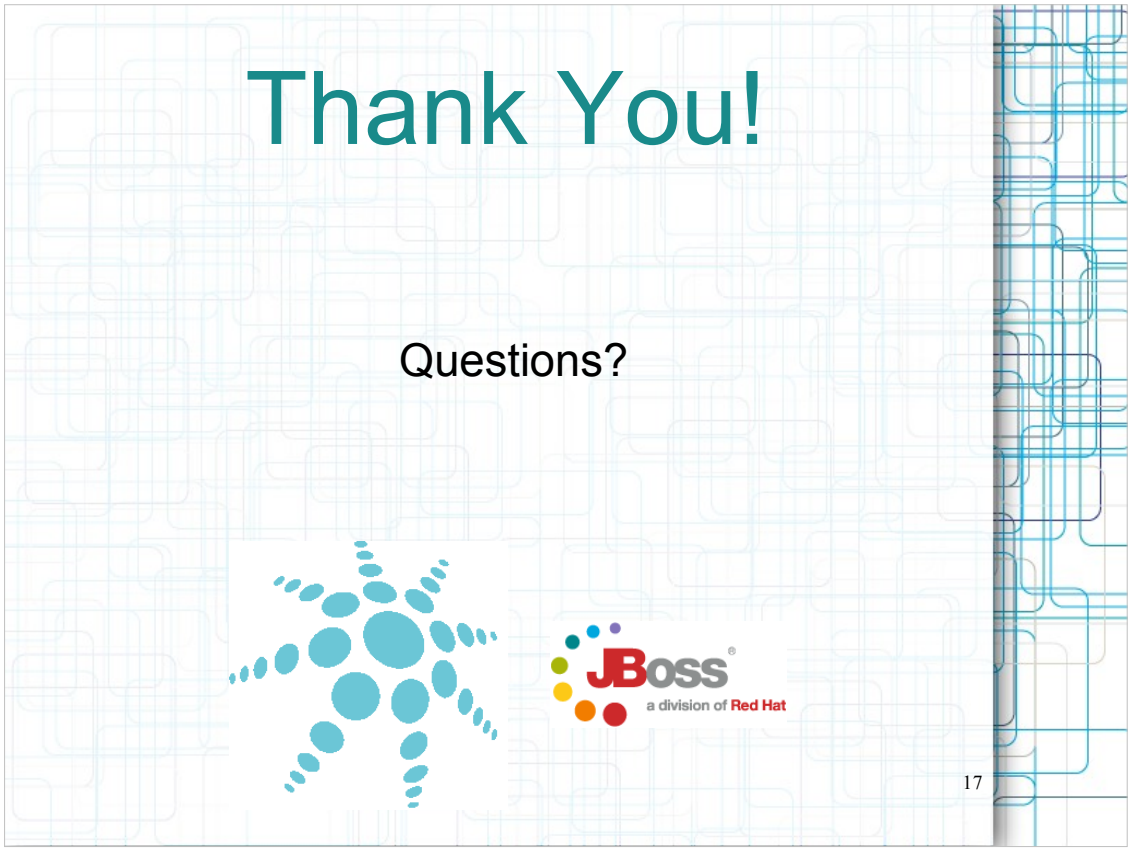


So, in conclusion, we have this common requirement of building forms that most of us wrestle with every day, and none of our existing practices are very satisfactory.

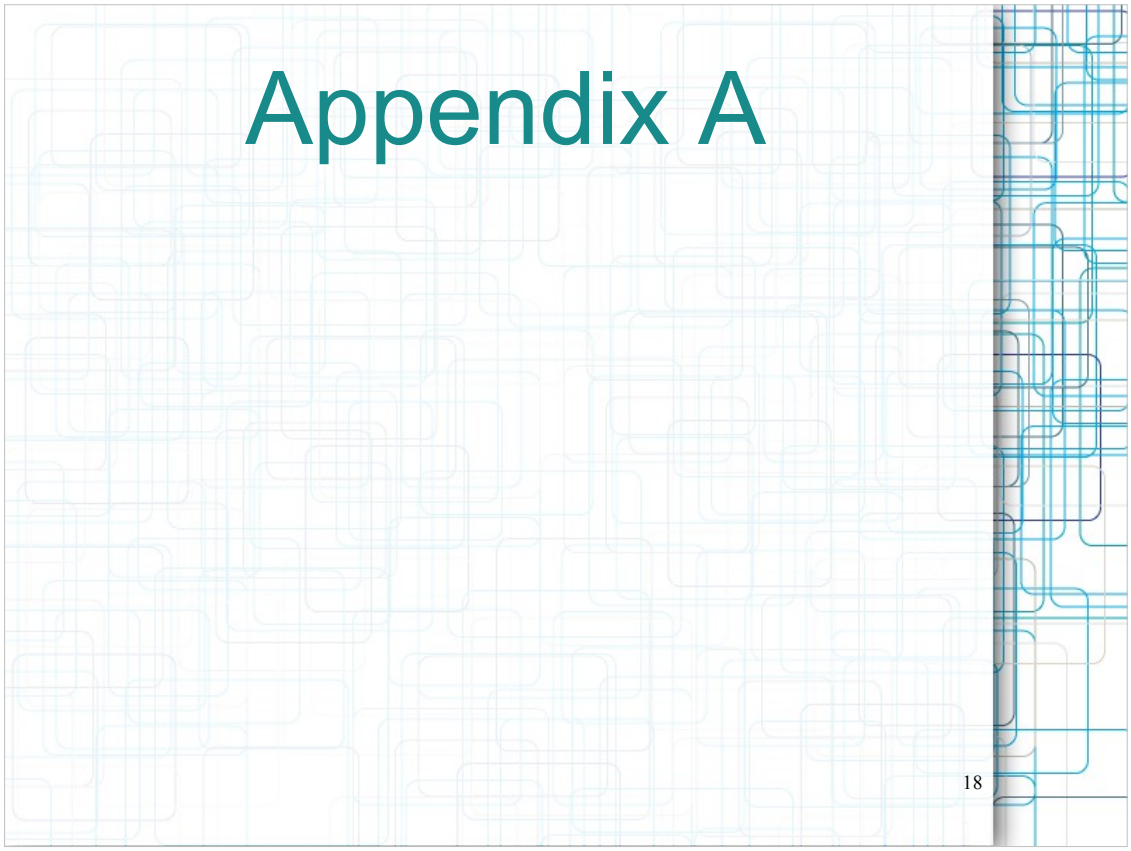
Visual form designers and UI languages are laborious and violate D.R.Y. Code generators tend to dictate how your applications are built, and give you very generic output.

Metawidget is specifically designed to address each of these pain points - to find a 'sweet spot' of generating just enough to be useful, not too much that it constrains you.

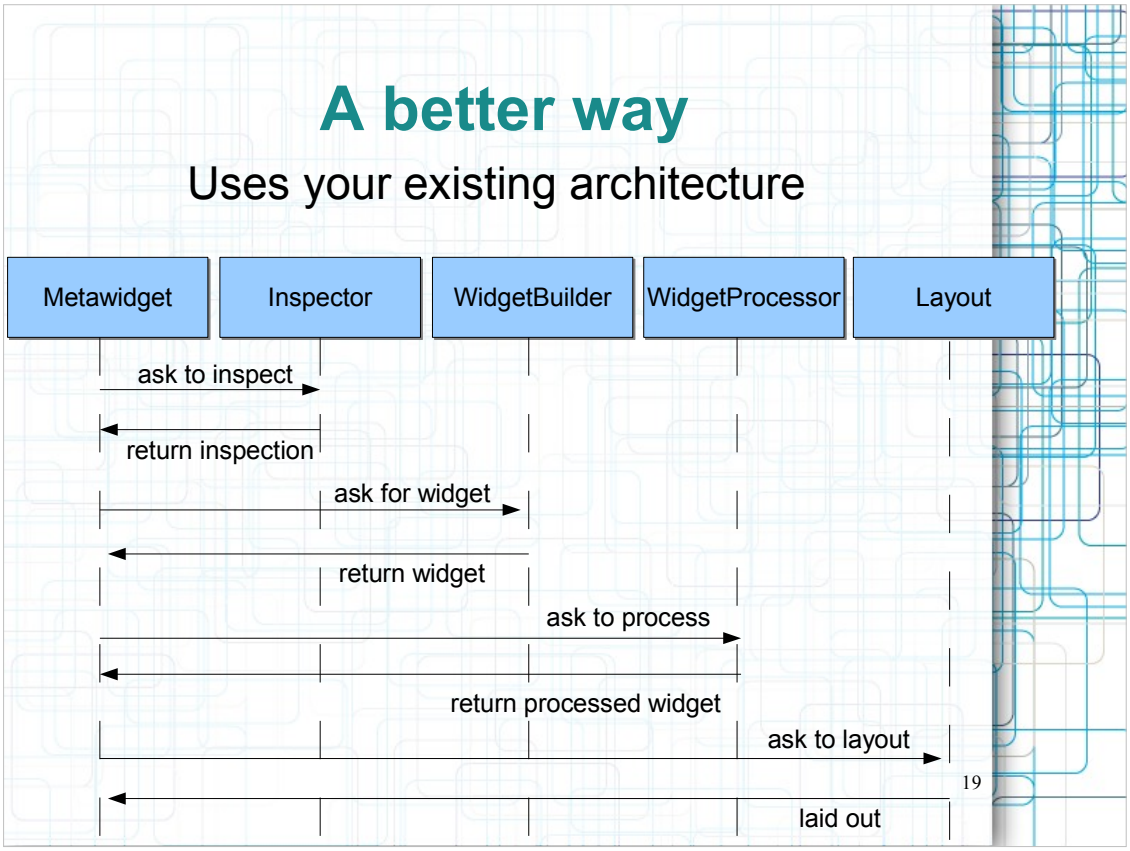
It's trying to make this something that's generally useful, that becomes an everyday part of your software development toolkit, so that we can save everybody a lot of time and a lot of pain.



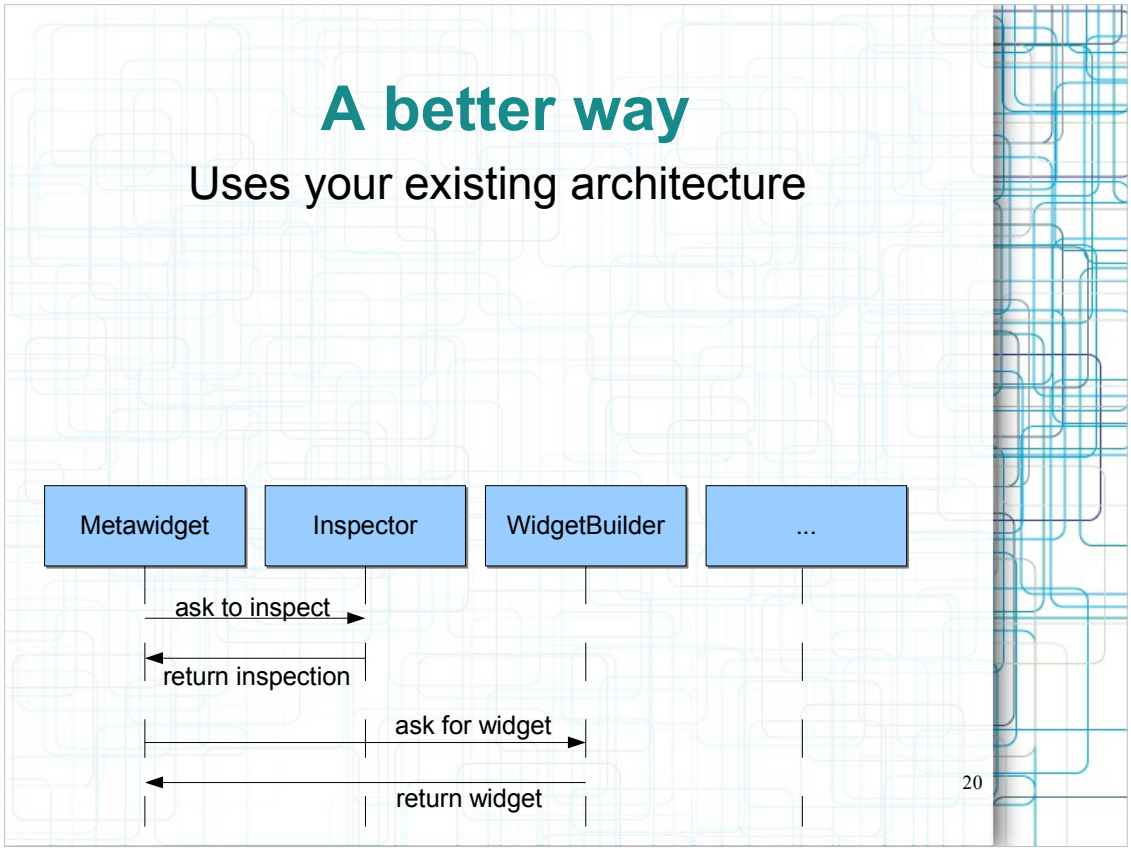
Thanks for listening. I'd now like to throw it out to the floor for questions?



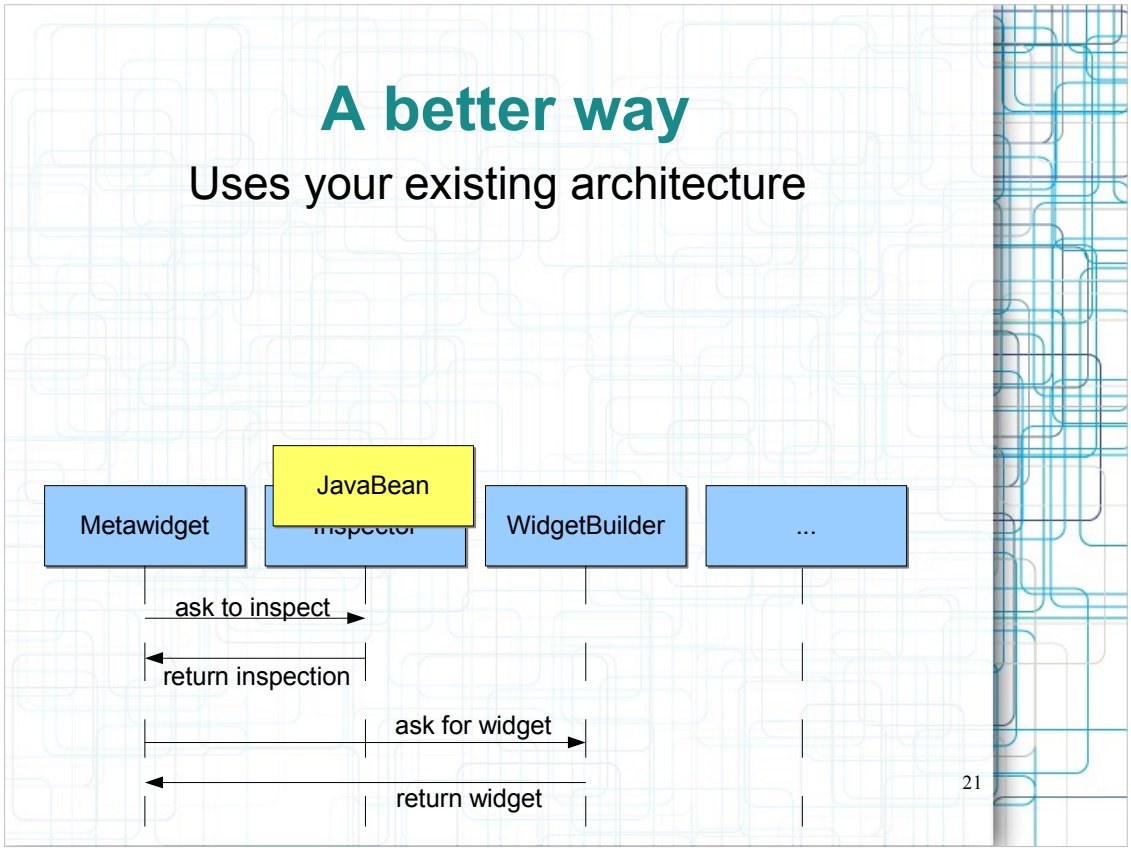
This is an appendix with slides that may be helpful in answering a question.



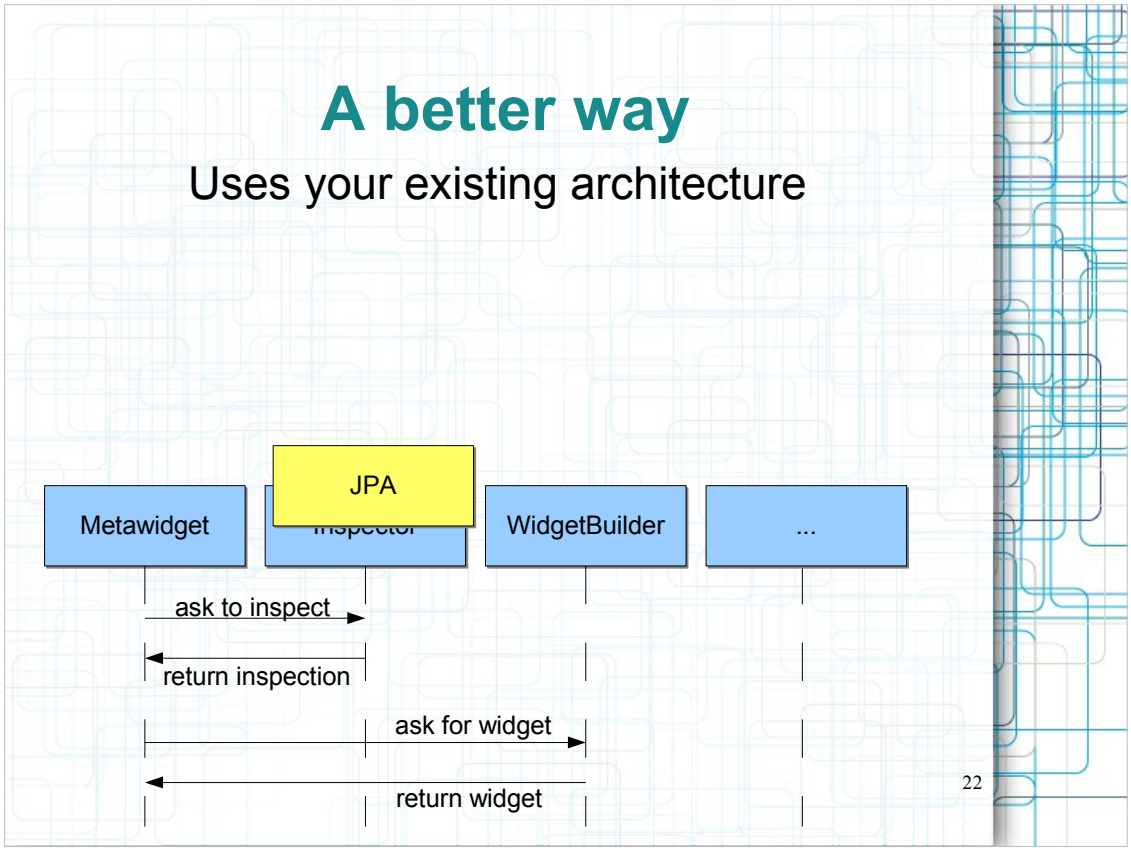
I wanted to talk a little about one of the other points: Metawidget works with whatever existing architecture you choose. It does this by having this a sort of 'pluggable pipeline'.



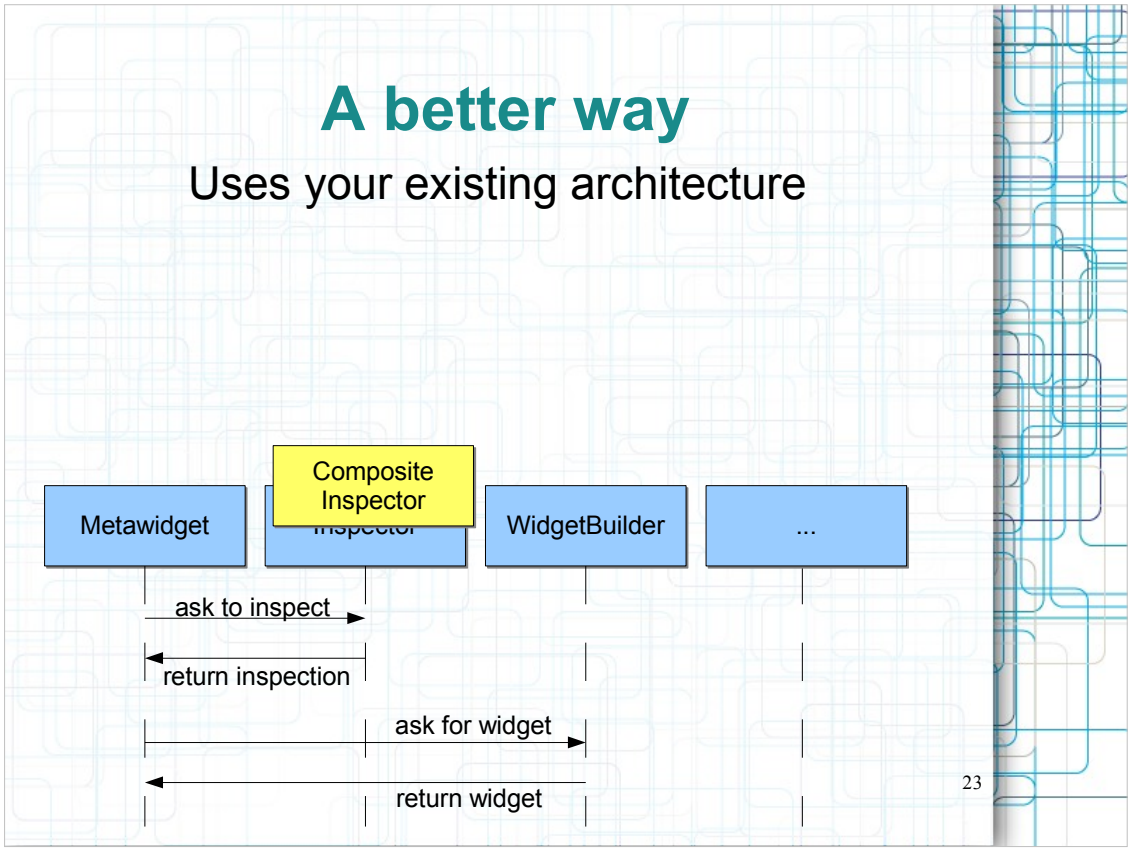
At any point in the pipeline, you can plug in alternate implementations. For example the 'Inspector' is where you plug in what Metawidget inspects in order to determine your business object properties.



So you can plug in a simple JavaBean inspector that will read names and types of properties from a POJO (this is the default)...



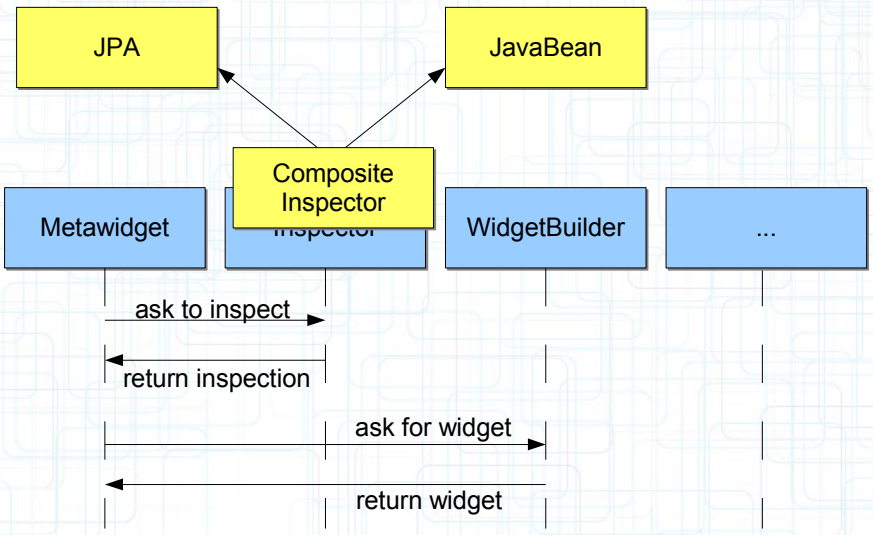
...or a JPA Inspector that will read JPA annotations like @Column(nullable=false) or @Id



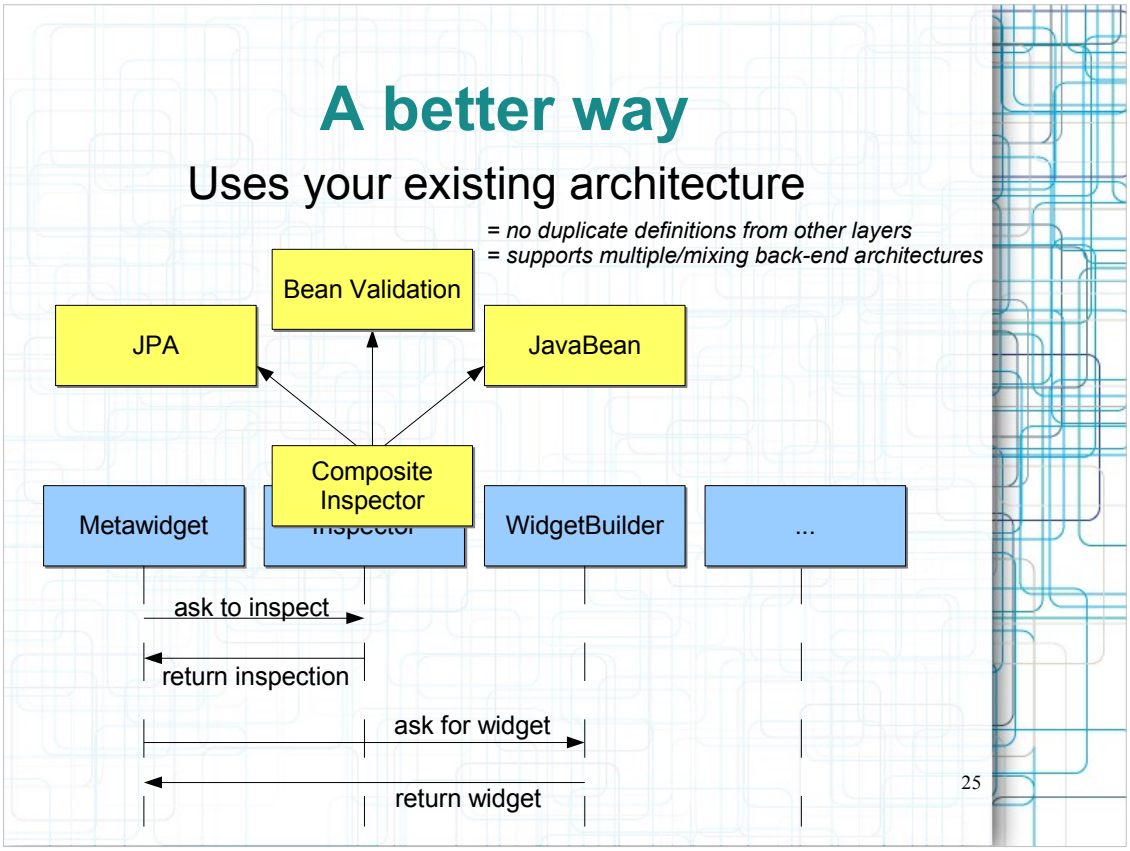
Now of course you can't generate an entire form based on JPA annotations! So you can instead plug in a *composite* Inspector...

A better way

Uses your existing architecture



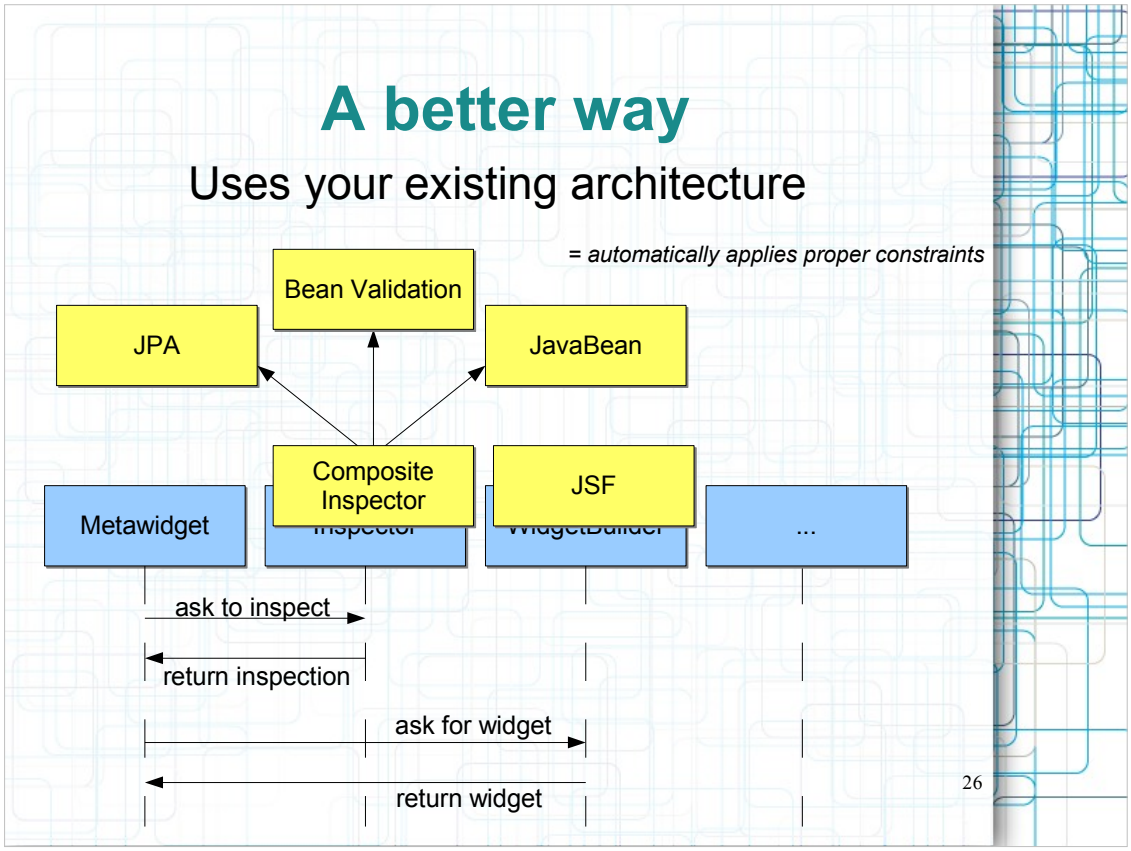
...and use that to *combine* information from both JPA annotations and JavaBean properties.



And now you can start bringing in other sources too: Bean Validation annotations; Hibernate XML mapping files; JBPM rules.

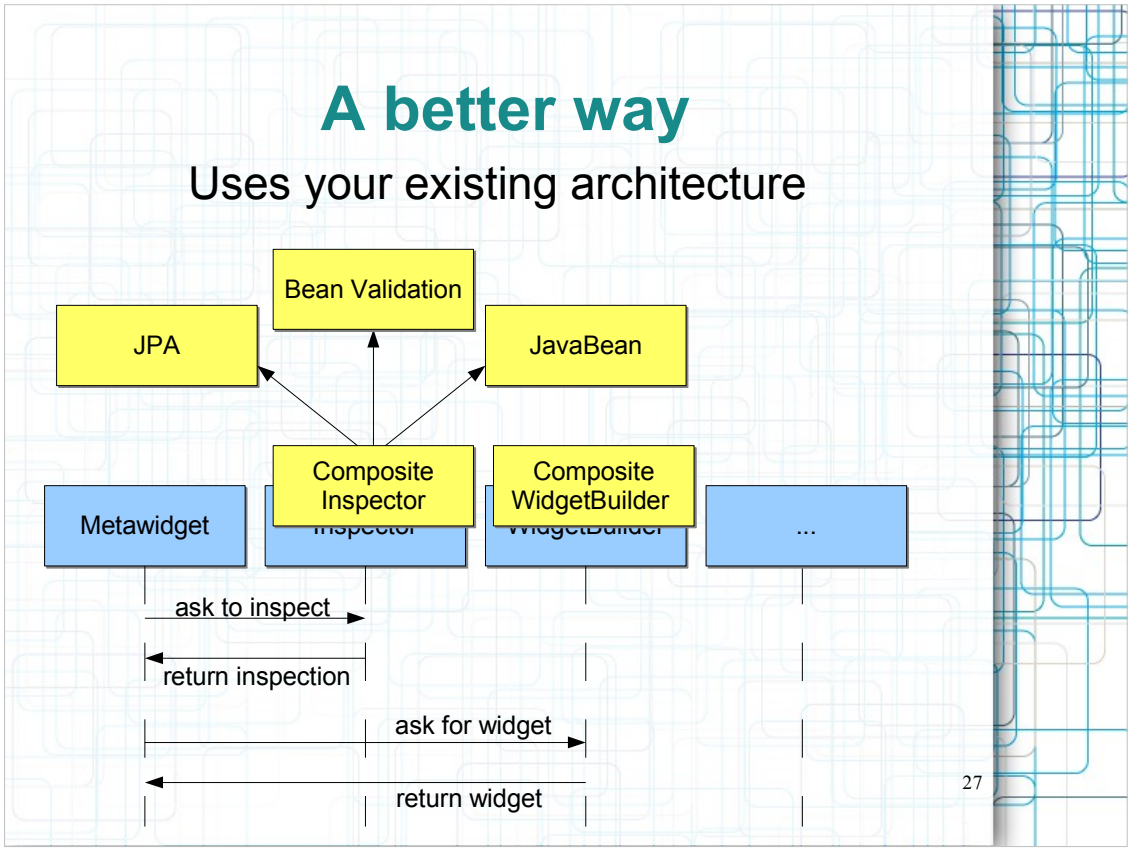
You mine as much useful metadata from as many different existing sources as you can, so that you don't have to duplicate anything from your back-end in your front-end.

It's easy to add your own sources, and they can be really disparate: we've had people use database schemas, WSDL files, properties files, different JVM languages, you name it.

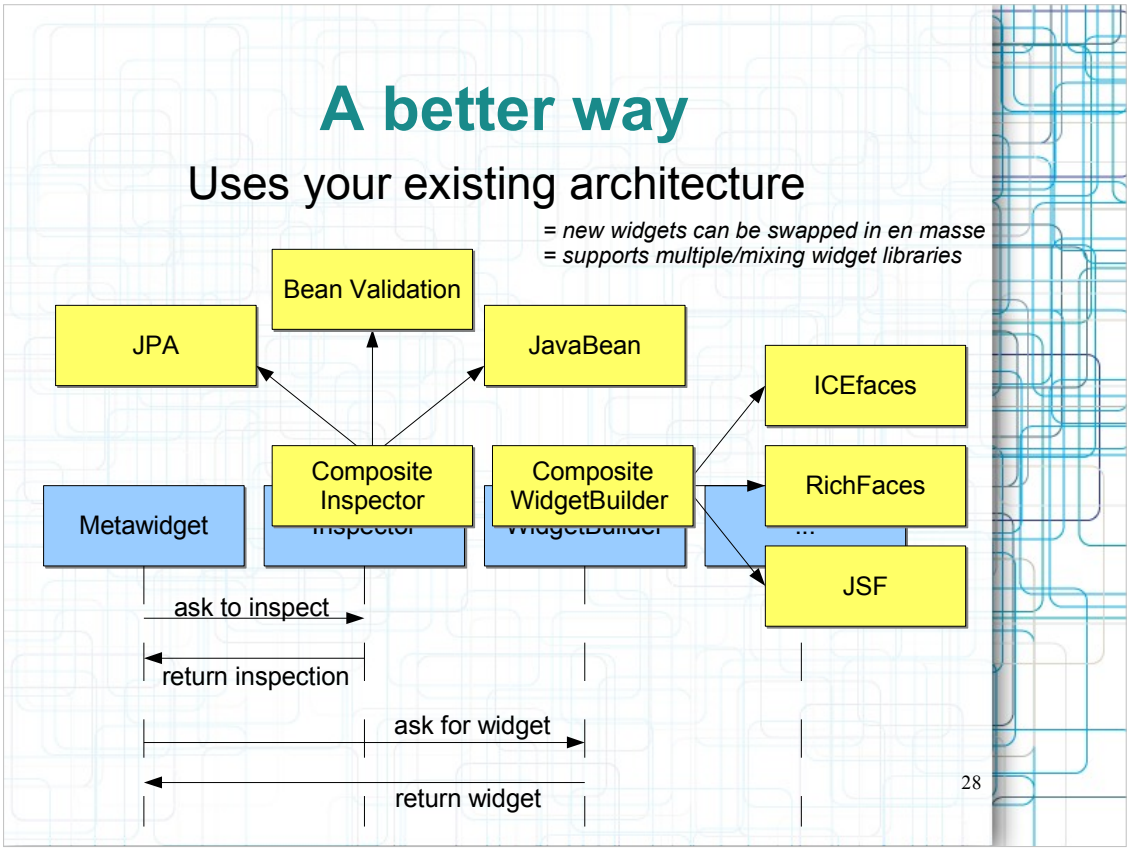


Having gathered together all that useful metadata, we can use it to build our UI widgets.

Here again, Metawidget is really pluggable. So you can plug in a JSF 'widget builder' to build widgets using the standard JSF library...



...or you can plug in a *composite* WidgetBuilder...

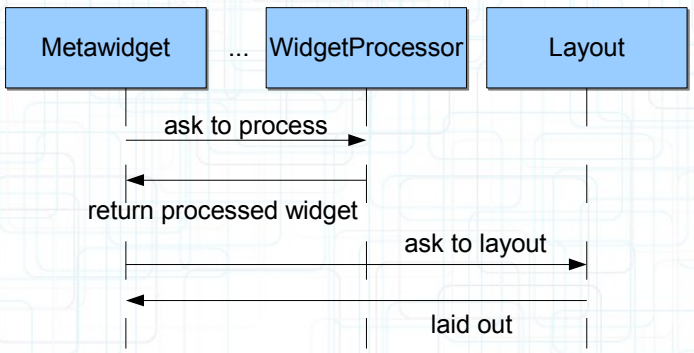


...and use that to *combine* several widget libraries in the same application.

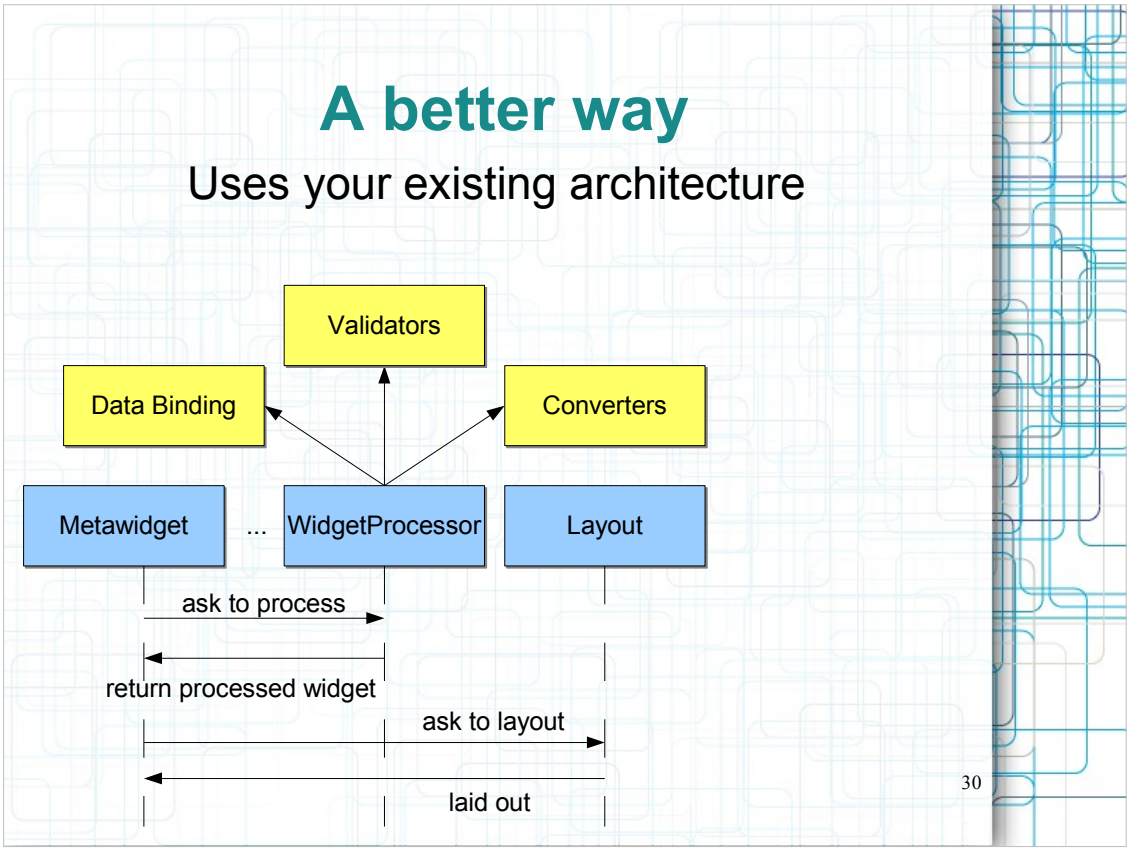
And you can order them too. So, in general, you might prefer ICEfaces components wherever possible. But for something like a colour field, ICEfaces doesn't have a colour picker so we'll fall back to RichFaces which does. And for something like a simple text field, we'll fall back to the standard JSF components.

A better way

Uses your existing architecture

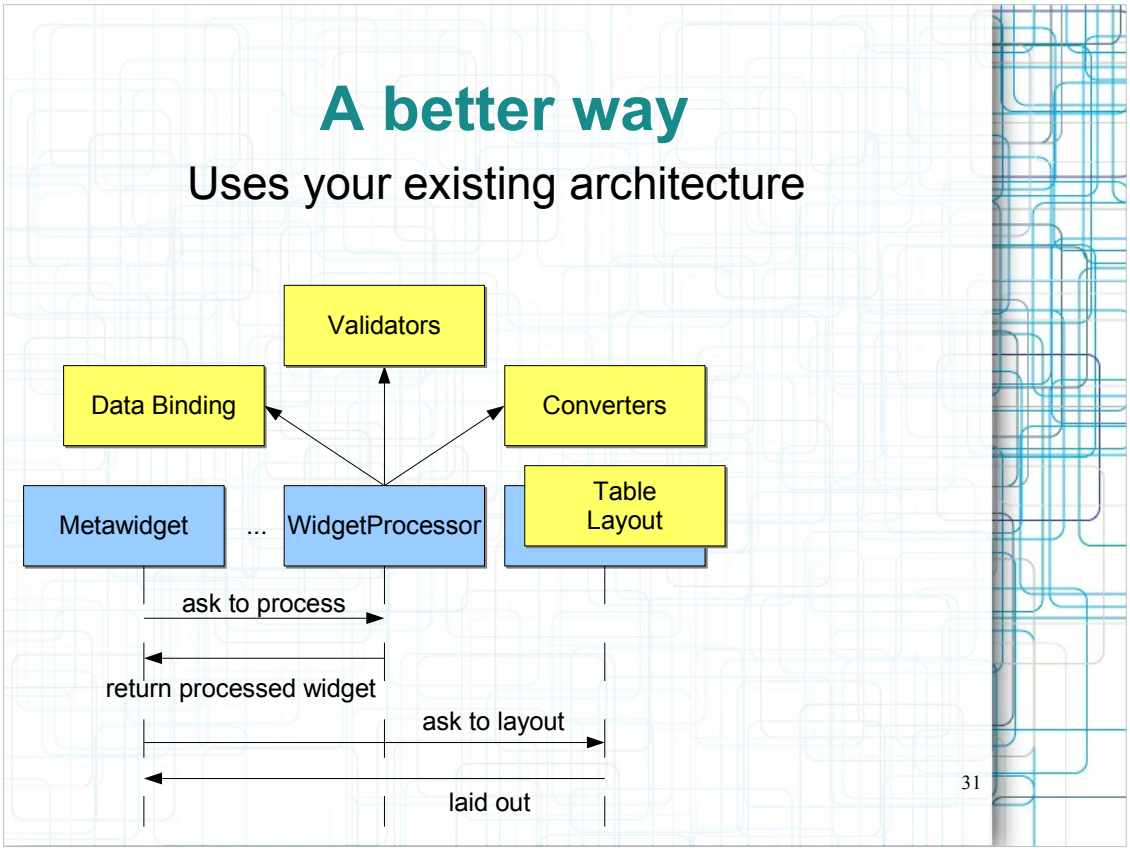


Once we've chosen the best widget for our field, we move on down the pipeline...



...and process the widget to attach validators, converters, data bindings and so on.

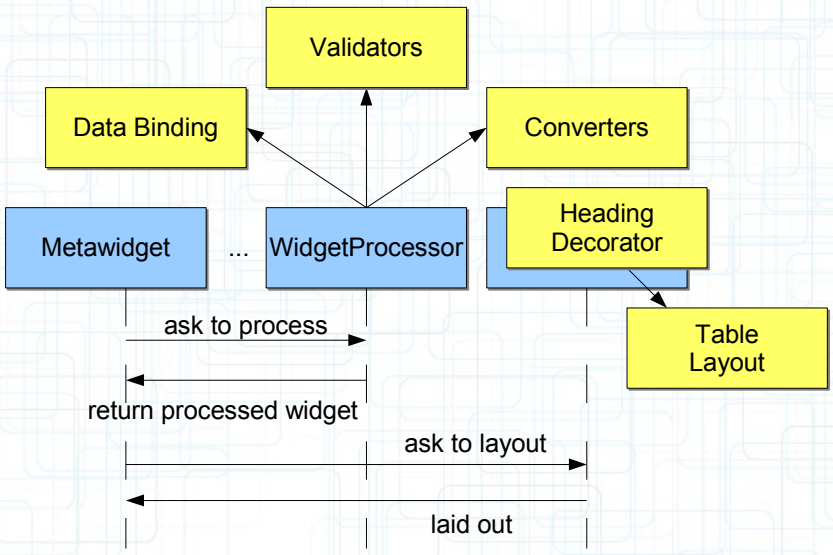
Note that what you attach to a widget can come from a different library than the widget itself. So you can attach a JGoodies validator to a SwingX component, for example.



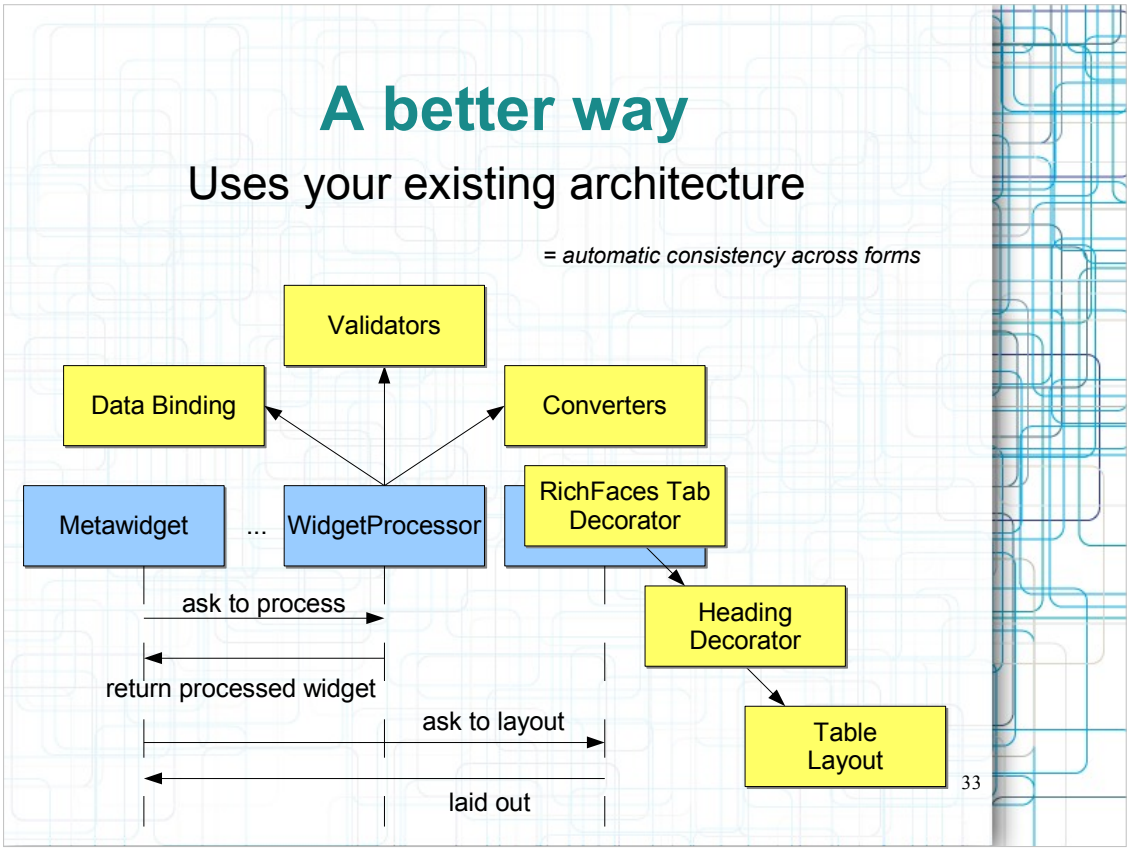
And finally you get to lay the widget out on the screen. Here you can not only plug in different layouts, such as laying the widgets out in a table, or in a row, or some other formation...

A better way

Uses your existing architecture



...but you can also *wrap* one layout with another. So you can wrap a heading around a table...



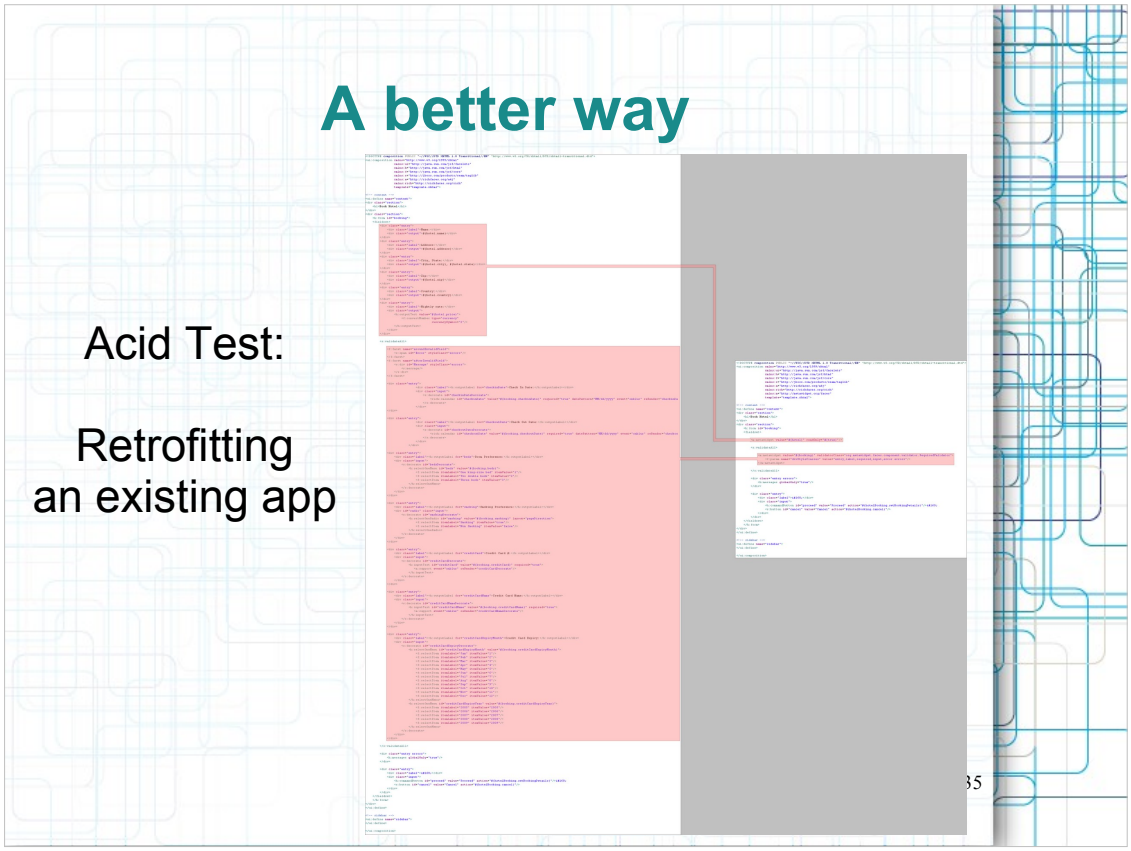
...or a tab strip around the both of them.

Note again that the wrapping layout can come from a different widget library. So you can wrap ICEfaces widgets with a RichFaces tab control.

By being pluggable along all these stages of the pipeline, from inspecting your business objects to choosing your widgets to configuring your layout, you can make Metawidget generate your front-end the way *you* want it to, based on the back-end architecture *you* choose.

Appendix B

This is an appendix with slides that may be helpful in answering a question.



I wanted to just show this slide as a real-world, 'big picture' idea of what Metawidget can do.

On the left is some Facelets code from the Seam Groovy Booking example. On the right is the Metawidget equivalent. The red areas highlight the chunks of boilerplate that have been replaced.

We can replace this boilerplate, which amounts to about 70% of the file, because it's *already* defined in the back-end.

And note what we're doing here is retrofitting an *existing* application. This is not Metawidget telling you how to build your app. This is your app finding savings by using Metawidget.